TI Arm Clang Compiler Tools User's Guide

v4.0

CONTENTS

The TI Arm® code generation tools support development of applications for TI Arm-based plat-forms featuring TI Arm Cortex-M and Cortex-R series devices. There are now two Arm C/C++ compiler toolchains available from TI:

- TI Arm Clang C/C++ Compiler Tools (tiarmclang)

- TI Arm C/C++ Compiler Tools (armcl)

This user's guide documents the TI Arm Clang C/C++ Compiler Tools. The tiarmclang compiler is based on the open source LLVM compiler infrastructure and its Clang front-end. tiarmclang uses the TI Linker and C runtime, which provide additional benefits for stability and reduced code size.
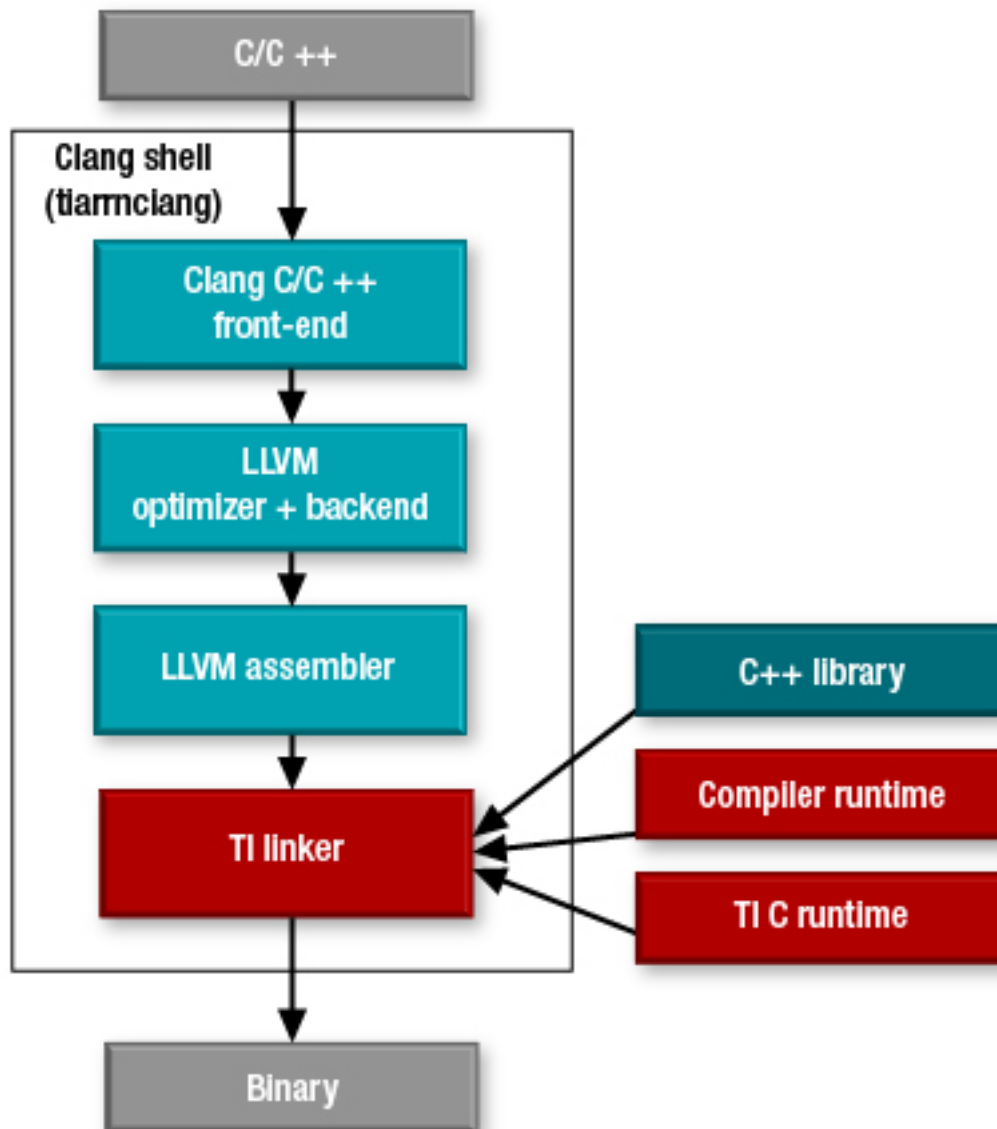
Benefits of tiarmclang include:

- *Excellent C/C++ standards support*

- *Source-based code coverage*

- *Support for migration from armcl*

- Improved code size over armcl/gcc

- Excellent Performance

    - 3.76 CoreMark/MHz (Cortex-R5)

    - 1.58/2.09 DMIPS/MHz (Cortex-R5)

- Ease of use with fast compiles and expressive diagnostic messages

- GCC compatibility

- Supported by CMake (3.29)

- Comprehensive documentation: Getting Started Guide, Migration Guide and Compiler Tools User Manual

- Compiler Qualification Kit to assist qualifying the compiler to functional safety standards such as ISO 26262 and IEC 61508.

Benefits of using the TI linker and C runtime include:

- Stability and flexibility, facilitating ongoing embedded differentiation for TI devices

- Pairs with C runtime library, which is optimized for reduced code size

- Function specialization, minimizing code size on common functions, including *printf*, *memcpy, and memset*

- Compatibility with TI linker command files

- *Support for C preprocessing directives* in TI linker command files, such as #define, #include, and #if/#endif

- *Copy Table support*, allowing automatic copying of code/data during runtime

- *Initialized Data and Copy Table compression*, reducing code size

- Security features such as *ECC*, *CRC*, and *CMSE* (for Cortex-M33)

- Optimized placement of *function call trampolines*

- Segmented memory spaces, allowing section placement into *multiple ranges* as well as *split placement*

While the tiarmclang toolchain serves the same purpose as armcl, there are behavioral differences that need to be accounted for when making the transition from developing C/C++ applications with armcl to tiarmclang. Please refer to the *TI ARM to tiarmclang Migration Guide* for detailed guidance on making a smooth transition from armcl to tiarmclang.

ONE

TIARMCLANG GETTING STARTED GUIDE

This Getting Started Guide provides an introduction to the TI Arm Clang Compiler toolchain and examples to demonstrate how to use the tiarmclang compiler to compile or assemble source files and link a simple application to be run on an Arm Cortex processor.

1.1 The tiarmclang Compiler Toolchain

The TI Arm Clang Compiler Toolchain (tiarmclang) is the next generation TI Arm compiler, replacing the previous TI Arm Compiler Tools (armcl). You can use the tiarmclang compiler toolchain to build applications from C, C++, and/or assembly source files to be loaded and run on one of the Cortex-M or Cortex-R Arm processors that are supported by the toolchain.

1.1.1 Toolchain Components

The tiarmclang compiler toolchain consists of many components. A brief description of the major components is provided in the subsections below.

Essential Tools

- **tiarmclang**

The C/C++ compiler and integrated assembler, tiarmclang, is used to compile C and C++ source files or assemble GNU-syntax Arm assembly language source files. By default, it automatically invokes the linker, tiarmlnk, to combine object files generated by the compiler with object libraries to create an executable program that can be loaded and run on an Arm processor.

The tiarmclang compiler is derived from the open source Clang compiler and its supporting LLVM infrastructure. You can find more details about Clang and LLVM at The LLVM Compiler Infrastructure site.

- **tiarmasm**

The legacy TI-syntax Arm assembler, tiarmasm, should only be used to assemble assembly source files that are written in legacy TI-syntax Arm assembly language. For further information about how to invoke the legacy TI-syntax Arm assembler please refer to the *Assembling Legacy TI-Syntax Arm Assembly Language Source* section of this getting started guide.

- **tiarmlnk**

The linker, tiarmlnk, is the proprietary linker provided by Texas Instruments. It combines object files that are either compiler-generated or have been archived into one or more object libraries to create executable programs that can be loaded and run on an Arm processor. It is typically invoked from the tiarmclang command line so that tiarmclang can implicitly set up the object library search path and implicitly include runtime libraries in the link.

- **tiarmar**

The archiver, tiarmar, can be used to collect object files together into an object library or archive that can be specified as input to the linker to provide definitions of functions or data objects that are not otherwise available in the compiler generated object files that are input to the link. For example, the standard C runtime library is an example of an object library that collects pre-built object files that contain definitions of C runtime functions that are required by the language standard to be provided with a C compiler toolchain like tiarmclang.

The archiver also provides a convenient way to collect logically related object files into an object library that can be distributed as a product to provide capability for use in the development of customer Arm applications.

- **tiarmhex**

The hex conversion utility, tiarmhex, can be used to convert ELF files into other object formats, such as ASCII-Hex, Intel MCS-86, and Motorola-S among others, that can then be downloaded to an EPROM programmer.

Runtime Libraries

- **libc**

The libc library provides an implementation of the C standard runtime features and capabilities that are to be provided as part of a C compiler toolchain.

- **libc++abi** and **libc++**

The libc++ library provides an implementation of the standard C++ library and depends on the libc++abi library to provide implementations of low-level language features.

- **compiler-rt**

The compiler-rt runtime library helps to support the code coverage features in the tiarmclang compiler as well as providing an implementation of low-level target-specific functions that can be used in compiler generated code.

Code Coverage Utilities

- **tiarmcov**

The tiarmcov tool shows code coverage information for programs that have been instrumented to emit profile data.

- **tiarmprofdata**

The profile data tool, tiarmprofdata, is used to merge multiple profile data files generated by profile-guided optimization instrumentation and merges them together into a single indexed profile data file.

Object File Editing and Information Utilities

- **tiarmdem**

The C++ name demangler, tiarmdem, is a debugging aid that converts names that have been mangled by the compiler back to their original names as declared in the C++ source code. The tiarmdem tool can be used on compiler-generated assembly files or linker-generated map files that contain instances of C++ mangled names.

- **tiarmlibinfo**

The tiarmlibinfo command allows you to collect multiple versions of the same object file library, each version built with a different set of command-line options, into a single index library file. This index library file can then be used at link-time as a proxy for the actual object file library.

- **tiarmnm**

The name utility, tiarmnm, prints the list of symbol names defined and referenced in an object file, executable file, or object library. It also prints the symbol values and an indication of each symbol's kind.

- **tiarmobjcopy**

The object copying and editing tool, tiarmobjcopy, can make a semantic copy of an input object file to an output object file, but command-line options are available that allow parts of the input object file to be edited before writing the result of the edit to the output file. For example, the --strip-debug option can be used to remove all debug sections from the output.

- **tiarmobjdump**

The object file dumper utility, tiarmobjdump, can be used to print the contents of an object file. It is commonly used to print out specific parts of the input object file using one of its available options. For example, its -d option disassembles all text sections found in the input object file. For more details about available options use tiarmobjdump's --help option.

- **tiarmofd**

Like tiarmobjdump, the object file display utility, tiarmofd, can be used to print the contents of object files, executable files, and object libraries. The output can be in text format or in XML. There are also tiarmofd options available to alter how text output is displayed and whether DWARF debug information is to be included in the output.

- **tiarmreadelf**

The GNU-style ELF object reader, tiarmreadelf, can be used to display low-level format-specific information about one or more object files. Like tiarmobjdump and tiarmofd, tiarmreadelf provides command-line options to allow you to display certain pieces of information from an object file like relocation entries or section headers.

- **tiarmsize**

The GNU-style size information utility, tiarmsize, prints size information for binary files. The output displayed will show the total size for text sections, for bss sections, and data section as well as a grand total.

- **tiarmstrip**

The tiarmstrip tool can be used to strip sections and symbols from object files.

1.1.2 Software Development Flow

The source code for your application consists of some combination of:

- C and/or C++ source files (.c, .C, or .cpp extension)
- GNU-syntax Arm assembly source files (.s or .S extension)
- Legacy TI-syntax Arm assembly source files (.asm extension)

The compile, assemble, and link part of your development flow will look something like this:
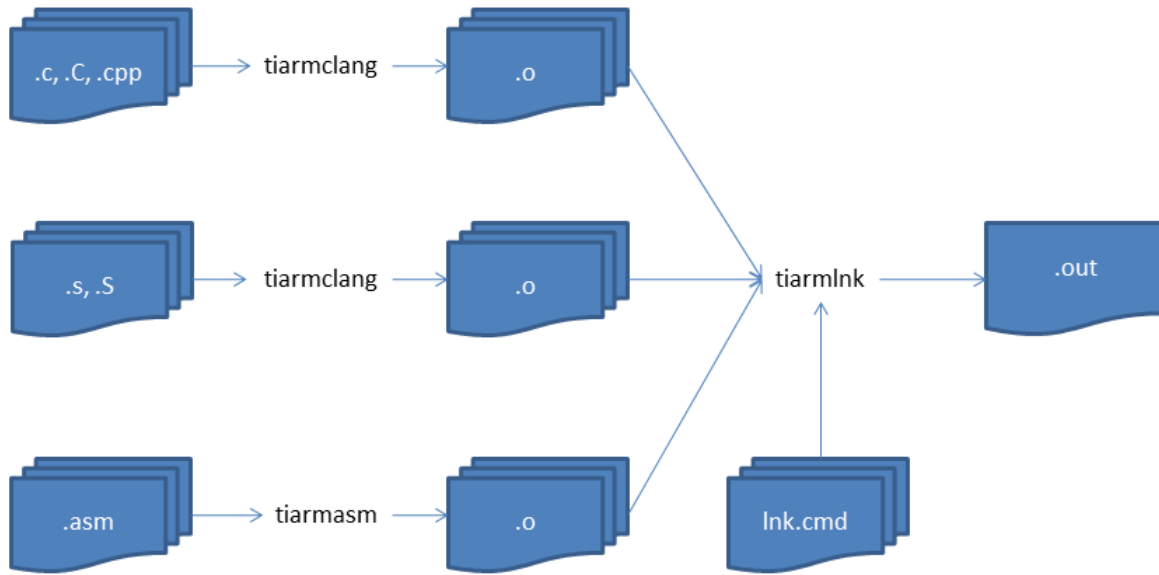
Figure 1.1: Software Development Flow

Note that:

- tiarmclang interprets files with a .c extension as C source, invoking the compiler

- tiarmclang interprets files with a .C or .cpp extension as C++ source, invoking the compiler

- tiarmclang interprets files with a .s extension as GNU-syntax Arm assembly source, invoking the integrated GNU-syntax Arm assembler

- tiarmclang interprets files with a .S extension as GNU-syntax Arm assembly source that needs to be pre-processed, invoking the pre-processor first, and then the integrated GNU-syntax Arm assembler

- files with an .asm extension are usually assembly source files written with legacy TI-syntax Arm assembly language that can be processed by the standalone legacy TI-syntax Arm assembler. You can also invoke the legacy TI-syntax Arm assembler from the tiarmclang command line with the help of tiarmclang's "-x ti-asm" option (see *Invoking the TI-Syntax ARM Assembler from tiarmclang* for details).

All of the object files generated (.o extension) are then combined by the linker and linked against any applicable runtime libraries to create an executable output file that can be loaded and run on a TI Arm processor.

1.2 Using the tiarmclang Compiler and Linker

1.2.1 Using the Compiler and Linker

Both the TI Arm Clang (tiarmclang) Compiler Tools' compiler and linker can be invoked from the **tiarmclang** command-line. The following subsections describe the basics of how to manage the compile and link steps of building an application from the **tiarmclang** command-line.

Compiling and Linking

The default behavior of the tiarmclang compiler is to compile specified C, C++, or assembly source files into temporary object files, and then pass those object files along with any explicitly specified object files and any specified linker options to the linker.

```
tiarmclang [options] [source file names] [object file names] [-
↪Wl,<linker options>]
```

In the following example, assume that the C code in **file1.c** references a data object that is defined in an object file named **file2.o**. The specified **tiarmclang** command compiles **file1.c** into a temporary object file. That object file, along with **file2.o** and a linker command file, **link_test.cmd**, are sent to the linker and linked with applicable object files from the tiarmclang runtime libraries to create an executable output file named **test.out**:

```
tiarmclang -mcpu=cortex-m0 file1.c file2.o -o test.out -Wl,link_
↪test.cmd
```

Note that there is no mention of the tiarmclang runtime libraries on the **tiarmclang** command-line or inside the **link_test.cmd** linker command file. When the linker is invoked from the **tiarmclang** command-line, the tiarmclang compiler implicitly tells the linker where to find applicable runtime libraries, such as the C runtime library (libc.a).

In the above **tiarmclang** command-line, the **-Wl,** prefix in front of the specification of the **link_test.cmd** file name indicates to the compiler that the **link_test.cmd** file should be sent directly to the TI linker (you can also use the **-Xlinker** prefix for this purpose).

Compiling and Linking with Verbose Linker Output

If you add the verbose (**-v**) option to the above **tiarmclang** command, you will see exactly how the linker (**tiarmlnk**) is invoked and with what options. For example, this command:

```
tiarmclang -mcpu=cortex-m0 -v file1.c file2.o -o test.out -Wl,
↪link_test.cmd
```

shows the following with regards to how the **tiarmlnk** command is invoked by the tiarmclang compiler:

```
<install directory>/bin/tiarmlnk -I<install directory>/lib
-o test.out /tmp/file1-98472f.o file2.o link_test.cmd
--start-group -llibc++.a -llibc++abi.a -llibc.a -llibsys.a
-llibsysbm.a -llibclang_rt.builtins.a -llibclang_rt.profile.a --
↪end-group
```

In the above invocation of the linker, the compiler inserts a *-I<install directory>/lib* option, which tells the linker where to find the tiarmclang runtime libraries. The compiler also inserts the *--start_group/--end_group* option list, which specifies exactly which runtime libraries to incorporate into the link.

Compiling Only

You can avoid invoking the linker by specifying the **-c** option on the **tiarmclang** command-line.

```
tiarmclang -c [options] [source file names]
```

The following example generates object files **file1.o** and **file2.o** from the C files **file1.c** and **file2.c**, respectively:

```
tiarmclang -c -mcpu=cortex-m0 file1.c file2.c
```

Using the **-S** option causes the compiler to generate assembly files from the C or C++ source files that are specified on the command-line. When **-S** is specified on the command-line, compilation stops after the assembly files are emitted, preventing the compiler from generating an object file or invoking the linker.

```
tiarmclang -S [options] [source file names]
```

The following example generates assembly files, **file1.s** and **file2.s**, each containing GNU-syntax Arm assembly language directives and instructions:

```
tiarmclang -S -mcpu=cortex-m0 file1.c file2.c
```

Link-Only Using tiarmclang

When only object files are specified as input to the tiarmclang compiler command, the compiler automatically passes those files to the linker along with any other specified options that are applicable to the link.

```
tiarmclang [options] [object file names] [-Wl,<linker options>]
```

As in the default case of "Compiling and Linking" described above, a **-Wl,** or **-Xlinker** prefix must be specified in front of options that are intended for the linker. This example **tiarmclang** command:

```
tiarmclang -mcpu=cortex-m0 file1.o file2.o -o test.out -Wl,link_
↪test.cmd
```

invokes the linker as follows:

```
<install directory>/bin/tiarmlnk -I<install directory>/lib
-o test.out file1.o file2.o link_test.cmd
--start-group -llibc++.a -llibc++abi.a -llibc.a -llibsys.a
-llibsysbm.a -llibclang_rt.builtins.a -llibclang_rt.profile.a --
↪end-group
```

As in the "Compiling and Linking" case, the compiler inserts a *-I<install directory>/lib* option that tells the linker where to find the tiarmclang runtime libraries. The compiler also inserts the *--start_group/--end_group* option list that specifies exactly which runtime libraries are incorporated into the link.

1.2.2 Useful Compiler Options

The commonly used options listed in the subsections below are available on the tiarmclang compiler command-line.

Processor Options

- **-mcpu** - select the target processor version

The tiarmclang compiler supports the following Arm Cortex-M processor variants which support 16-bit and T32 THUMB instructions (as indicated):

- **-mcpu=cortex-m0** - 16-bit THUMB
- **-mcpu=cortex-m0plus** - 16-bit THUMB
- **-mcpu=cortex-m3** - T32 THUMB

- **-mcpu=cortex-m4** - T32 THUMB

The tiarmclang compiler also supports Arm Cortex-R type processor variants, Cortex-R4 and Cortex-R5, which can execute ARM and T32 THUMB instructions. By default, the tiarmclang compiler assumes ARM instruction mode, but the user can instruct the compiler to assume the use of T32 THUMB instructions for a given compilation by specifying the **-mthumb** option.

- **-mcpu=cortex-r4** - ARM (default) and/or T32 THUMB

- **-mcpu=cortex-r5** - ARM (default) and/or T32 THUMB

If an **-mcpu** variant is not specified on the **tiarmclang** command-line, then the compiler assumes a default of **-mcpu=cortex-m4**.

Instruction Mode Options

The Cortex-M type Arm processors only support execution of THUMB mode instructions, so **-mthumb** is selected by default when using the **-mcpu=cortex-m0**, **-mcpu=cortex-m0plus**, **-mcpu=cortex-m3**, or **-mcpu=cortex-m4** processor options.

- **-mthumb** - select THUMB mode instructions (16-bit THUMB or T32 THUMB depending on which processor variant is selected) for current compilation; default for Cortex-M type architectures

Cortex-R type Arm processors can execute both ARM and T32 THUMB mode instructions. When either the **-mcpu=cortex-r4** or **-mcpu=cortex-r5** processor option is specified on the command-line, the tiarmclang compiler assumes ARM instruction mode by default. This can be overridden for a given compilation by specifying the **-mthumb** option in combination with **-mcpu=cortex-r4** or **-mcpu=cortex-r5** to cause the compiler to generate T32 THUMB instructions.

- **-marm** - select ARM mode instructions for current compilation; default for Cortex-R type processors

Endianness

All of the Arm Cortex-M type processor variants supported by the tiarmclang compiler are little-endian. The Arm Cortex-R type processor variants supported by the tiarmclang compiler may be big-endian or little-endian. The following options can be used to select the endian-ness to be assumed for a given compilation:

- **-mlittle-endian** - select little-endian; default

- **-mbig-endian** - select big-endian; only applicable for Cortex-R

Floating-Point Support Options

Some Arm processor variants can be used in conjunction with floating-point hardware to perform floating-point operations more efficiently. To enable this support, specify the **-mfloat-abi=hard** option in combination with the appropriate **-mfpu** option depending on what Arm processor variant is in use.

- **-mfloat-abi=** - select floating point ABI

  - **hard** - floating-point hardware is available; select appropriate hardware with **-mfpu** option

  - **soft** - unless **-mfloat-abi=hard** is selected, floating-point operations are emulated in software

- **-mfpu=** - select floating-point hardware

  Only certain Arm processor variants can be used in conjunction with floating-point hardware processors. The **cortex-m0**, **cortex-m0plus**, and **cortex-m3** Arm processor variants are not compatible with any floating-point hardware processor.

  - **vfpv3-d16** - available in combination with **-mcpu=cortex-r4** or **-mcpu=cortex-r5**

  - **fpv4-sp-d16** - available in combination with **-mcpu=cortex-m4**

Include Options

The tiarmclang compiler utilizes the include file directory search path to locate header files that are included by a C/C++ source file via **#include** preprocessor directives. The tiarmclang compiler implicitly defines an initial include file directory search path to contain directories relative to the tools installation area where C/C++ standard header files can be found. These C/C++ standard header files are considered part of the tiarmclang compiler package and should be used in combination with linker and the runtime libraries that are included in the tiarmclang compiler tools installation.

- **-I<dir>**

  The **-I** option allows you to add your own directories to the include file directory path, allowing user-created header files to be easily accessible during compilation.

Predefined Symbol Options

In addition to the pre-defined macro symbols that the tiarmclang compiler defines depending on which processor options are selected, you can also manage your own symbols at compile-time using the **-D** and **-U** options. These options are useful when the source code is configured to behave differently based on whether a compile-time symbol is defined and/or what value it has.

- **-D<name>[=<value>]**

A user-created pre-defined compile symbol can be defined and given a value using the **-D** option. In the following example, **MySym** is defined and given a value 123 at compile-time. **MySym** will then be available for use during the compilation of the **test.c** source file.

```
tiarmclang -mcpu=cortex-m0 -DMySym=123 -c test.c
```

- **-U<name>**

  The **-U** option can be used to cancel a previous definition of a specified **<name>** whether it was pre-defined implicitly by the compiler or with a prior **-D** option.

Optimization Options

To enable optimization passes in the tiarmclang compiler, select a level of optimization from among the following **-O[0|1|2|3|fast|g|s|z]** options. In general, the options below represent various levels of optimization with some options designed to favor smaller compiler-generated code size over performance, while others favor performance at the cost of increased compiler-generated code size.

Among the options listed below, **-Oz** is recommended as the optimization option to use if small compiler-generated code size is a priority for an application. Using **-Oz** retains performance gains from many of the **-O2** level optimizations that are performed.

- **-O0** - No optimization. This setting is not recommended, because it can make debugging difficult.

- **-O1** or **-O** - Restricted optimizations, providing a good trade-off between code size and debuggability.

- **-O2** - Most optimizations enabled; some optimizations that require significant additional compile time are disabled.

- **-O3** - All optimizations available at **-O2** plus others that require additional compile time to perform.

- **-Ofast** - All optimizations available at **-O3** plus additional aggressive optimizations with potential for additional performance gains, but also not guaranteed to be in strict compliance with language standards.

- **-Og** - Restricted optimizations while preserving debuggability. All optimizations available at **-O1** are performed with the addition of some optimizations from **-O2**.

- **-Os** - All optimizations available at **-O2** plus additional optimizations that are designed to reduce code size while mitigating negative impacts on performance.

- **-Oz** - All optimizations available at **-O2** plus additional optimizations to further reduce code size with the risk of sacrificing performance.

Note: **Optimization Option Recommendations:**

- The **-O1** option is recommended for maximum debuggability.

- The **-Oz** option is recommended for optimizing code size.

- The **-O3** option is recommended for optimizing performance, but it is likely to increase compiler-generated code size.

Debug Options

The tiarmclang compiler generates DWARF debug information when the **-g** or **-gdwarf-3** option is selected.

- **-g** or **-gdwarf-3** - emit DWARF version 3 debug information

Control Options

Some tiarmclang compiler options can be used to halt compilation at different stages:

- **-c** - stop compilation after emitting compiler-generated object files; do not call linker

- **-E** - stop compilation after the pre-processing phase of the compiler; this option can be used in conjunction with several other options that provide further control over the pre-processor output:

    - **-dD** - print macro definitions in addition to normal preprocessor output

    - **-dI** - print include directives in addition to normal preprocessor output

    - **-dM** - print macro symbol definitions *instead of* normal preprocessor output

- **-S** - stop compilations after emitting compiler-generated assembly files; do not call assembler or linker

Compiler Output Option

- **-o<file>**

    The **-o** option names the output file that results from a **tiarmclang** command. If tiarmclang is used to compile and link an executable output file, then the **-o** option's **<file>** argument names that output file. If no **-o** option is specified in a compile and link invocation of tiarmclang then the linker will produce an executable output file named **a.out**.

    If the compiler is used to process a single source file, then the **-o** option will name the output of the compilation. This is sometimes useful in case there is a need to name the output file from the compiler something other than what the compiler will produce by default. In the following example, the output object file from the compilation of C source file **task_42.c** is

named **task.o** by the **-o** option, replacing the **task_42.o** that would normally be generated by the compiler:

```
tiarmclang -mcpu=cortex-m0 -c task_42.c -o task.o
```

Source File Interpretation Option

The tiarmclang compiler interprets source files with a recognized file extension in a predictable manner. The recognized file extensions include:

- **.c** - C source file

- **.C** or **.cpp** - C++ source file

- **.s** - GNU-syntax Arm assembly source file

- **.S** - GNU-syntax Arm assembly source file to be preprocessed by the compiler

The tiarmclang compiler also supports a **-x <language>** option that permits you to dictate how subsequent input files on the command-line are to be treated by the compiler. This can be used to override default file extension interpretations or to instruct the compiler how to interpret a file extension that is not automatically recognized by the compiler. The following **<language>** types are available with the **-x** option:

- **-x none** - reset compiler to default file extension interpretation

- **-x c** - interpret subsequent input files as C source files

- **-x c++** - interpret subsequent input files as C++ source files

- **-x assembler** - interpret subsequent files as GNU-syntax Arm assembly source files

- **-x assembler-with-cpp** - interpret subsequent files as GNU-syntax Arm assembly source files that will be preprocessed by the compiler

- **-x ti-asm** - interpret subsequent source files as legacy TI-syntax assembly source files causing the compiler to invoke the legacy TI-syntax Arm assembler to process them. For more information about handling legacy TI-syntax Arm assembly source files, please see *Invoking the TI-Syntax ARM Assembler from tiarmclang*.

Note: The **-x<language>** option is position-dependent. A given **-x** option on the **tiarmclang** command-line will be in effect until the end of the command-line *or* until a subsequent **-x** option is encountered on the command-line.

1.2.3 Linker Options

Link-Step File Search Path Options

Similar to the way that the tiarmclang compiler utilizes the include file directory search path to locate a header files during compilation, the linker uses the object file directory search path to help locate object libraries and object files that are input to the link step. As mentioned above, the tiarmclang compiler implicitly defines an initial object file directory search path to contain directories relative to the tools installation area where runtime libraries can be found. The following options can be used to help users manage where and how user-created object files and libraries are managed in the link step:

- **--search_path=<dir>** or **-I<dir>** - add specified directory path to the object file directory search path

- **--library=<file>** or **-l<file>** - use object file directory search path to locate specified object library or object file

Basic Linker Options

Listed below are some of the basic options that are commonly used when invoking the linker. They can be specified on the command-line or inside of a linker command file. The tiarmclang tool's linker is nearly identical to the linker in the legacy TI compiler toolchain. You can find more information about linker options in *Linker Options*.

- **--map_file=<file>** or **-m<file>** - emit information about the result of a link into the specified map **<file>**

- **--output_file=<file>** or **-o<file>** - emit linked output to specified **<file>**

- **--args_size=<size>** or **--args=<size>** - reserve **<size>** bytes of space to store command-line arguments that are passed to the linked application

- **--heap_size=<size>** or **--heap=<size>** - reserve **<size>** bytes of heap space to be used for dynamically allocated memory

- **--stack_size=<size>** or **--stack=<size>** - reserve **<size>** bytes of stack space for the run-time execution of the linked application

Specifying Linker Options on the tiarmclang Command-Line

As noted in a few of the above examples, when invoking the linker from the **tiarmclang** command, options that are to be passed directly to the linker must be preceded with a **-Wl,** (note that the comma is required) or **-Xlinker** prefix. In this example, the tiarmclang compiler passes the **link_test.cmd** linker command file directly to the linker:

```
tiarmclang -mcpu=cortex-m0 file1.c file2.o -o test.out -Wl,link_
↪test.cmd
```

The **tiarmclang** command line provides the following ways to pass options to the linker:

- The **-Wl,** option passes a comma-separated list of options to the linker. (A comma after -Wl is required.)

- The **-Xlinker** option passes a single option to the linker and can be used multiple times on the same command line.

- A linker command file can specify options to pass to the linker.

For example, the following command line passes several linker options using the **-Wl,** option:

```
tiarmclang -mcpu=cortex-m0 hello.c -o a.out -Wl,-stack=0x8000,--
↪ram_model,link_test.cmd
```

The following command line passes the same linker options using the **-Xlinker** option instead:

```
tiarmclang -mcpu=cortex-m0 hello.c -o a.out -Wl,-stack=0x8000,--
↪ram_model,link_test.cmd
```

The following lines from a linker command file, pass the same linker options to the linker:

```
/
↪********************************************************************/
↪
/* Example Linker Command File                                     ␣
↪    */
/
↪********************************************************************/
↪
-stack  0x8000                        /* SOFTWARE STACK SIZE       ␣
↪    */
--ram_model                           /* INITIALIZE VARIABLES AT LOAD␣
↪TIME */
```

1.2.4 Runtime Support

Predefined Macro Symbols

The tiarmclang compiler pre-defines compile-time macro symbols for use in source to help distin-
guish code written particularly for Arm, for a specific Arm processor variant, or to be compiled by
the tiarmclang compiler (as opposed to other Arm compilers) from other source code.

The tiarmclang compiler complies with the Arm C Language Extensions (ACLE) in that it will pre-
define appropriate ACLE macro symbols based on the processor options specified on the **tiarm-
clang** command-line. Consider the following **tiarmclang** command:

```
tiarmclang -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16 -c
↪test.c
```

When the tiarmclang compiler is invoked with the above command, these are some of the ACLE
pre-defined macro symbols that will be defined:

```
__ARM_ACLE              200  - indicates compliance with ACLE
                                 specification 2.0
__ARM_ARCH              7    - identifies Arm architecture level
__ARM_ARCH_ISA_THUMB    2    - indicates Thumb instruction set
↪present
__ARM_ARCH_PROFILE      'M'  - identifies a Cortex-M type Arm
↪architecture
__ARM_FP                0x6  - indicates floating-point hardware
↪is available
__ARM_FP16_FORMAT_IEEE  1    - indicates 16-bit floating-point,
↪IEEE format
__ARM_PCS               1    - using Arm procedure call standard
↪(PCS)
__ARM_PCS_VFP           1    - indicates floating-point arguments
↪passed in

                                 floating-point hardware registers
```

For more information about ACLE pre-defined macro symbols, please see the Arm C Language
Extensions - Release ACLE Q2 2018 specification.

In addition to the ACLE pre-defined macro symbols, the tiarmclang compiler also pre-defines sev-
eral TI-specific pre-defined macro symbols that can be used to distinguish the use of the tiarmclang
compiler from other Arm compilers. These include:

For a complete list of pre-defined macro symbols that are defined by the tiarmclang compiler for
a given compilation, the processor options can be combined with the **-E -dM** preprocessor option
combination. This will instruct the compiler to run only the preprocessor pass of the compilation
and emit the list of pre-defined macro symbols that are defined along with their values to stdout.

Header Files

The header files provided with the installation of the tiarmclang compiler tools must be used when using functions from any of the runtime libraries provided with the tools installation. These include the C and C++ standard header files as well as additional header files such as arm_acle.h that help the compiler to support the Arm C Language Extensions. The tiarmclang compiler implicitly defines the initial include file directory search path so that these header files are accessible during a compilation.

Runtime Libraries

When linking an application containing object files that were generated by the tiarmclang compiler, the appropriate tiarmclang runtime libraries must be included in the link so that references to functions and data objects that are defined in the runtime libraries can be properly resolved at link time.

When the **tiarmclang** command is used to invoke the linker, the compiler implicitly defines the initial object file directory search path to contain directories relative to the tools installation area where runtime libraries can be found. The tiarmclang compiler also implicitly adds the following **--start-group/--end-group** option list to the linker invocation:

```
--start-group -llibc++.a -llibc++abi.a -llibc.a -llibsys.a
-llibsysbm.a -llibclang_rt.builtins.a -llibclang_rt.profile.a --
↪end-group
```

This option list instructs the linker to search among the list of specified runtime libraries for definitions of unresolved symbol references. When a definition of a function or data object that resolves a previously unresolved reference is encountered, the section containing the definition is pulled into the link from the runtime library where it is defined. If new unresolved symbol references are introduced while this process is in progress, the libraries are re-read until no further needed definitions can be found among the **--start_group/--end_group** list of runtime libraries.

There are several different runtime library configurations supported in the tiarmclang compiler toolchain. An application built using the tiarmclang compiler tools must use a combination of target options that is compatible with one of the following configurations:

Note: C++ Exceptions

Support for C++ exceptions is off by default. You can specify the *-fexceptions* option on the **tiarmclang** command-line to enable exceptions. If *-fexceptions* is specified on the **tiarmclang** command-line, then the linker will link the application with a runtime support library that was built with C++ exceptions enabled.

Cortex-M Configurations

- *armv6m-ti-none-eabi*: (cortex-m0 or cortex-m0plus), little-endian

  The tiarmclang compiler supports generating execute-only code for Cortex-M0/M0+ Arm processor variants. This feature is off by default, but can be enabled using the *-mexecute-only* compiler option. If the *-mexecute-only* option is specified on the **tiarmclang** command-line, then the linker will link the application with a Cortex-M0/M0+ runtime library that was built with the execute-only feature enabled.

    - execute-only off (default)

    - execute-only on

- *armv7m-ti-none-eabi*: cortex-m3, little-endian

- *armv7em-ti-none-eabi*: cortex-m4, little-endian

- *armv7em-ti-none-eabihf*: cortex-m4, fpv4-sp-d16 fpu (default), little-endian

- *armv8m.main-ti-none-eabi*: cortex-m33, little-endian

- *armv8m.main-ti-none-eabihf*: cortex-m33, fpv5-sp-d16 fpu (default), little-endian

Cortex-R Configurations

Note: The Cortex-R runtime libraries included with the tiarmclang toolchain are built in ARM mode, but are compatible with Cortex-R object files that have been built with THUMB mode T32 instructions.

- *armv7r-ti-none-eabi*: (cortex-r4 or cortex-r5), ARM mode, little-endian (default)

- *armv7r-ti-none-eabihf*: (cortex-r4 or cortex-r5), ARM mode, vfpv3-d16 (default for cortex-r5), little-endian (default)

- *armv7reb-ti-none-eabi*: (cortex-r4 or cortex-r5), ARM mode, big-endian

- *armv7reb-ti-none-eabhf*: (cortex-r4 or cortex-r5), ARM mode, vfpv3-d16 (default for cortex-r5), big-endian

Note: If you are building an application with target options that are not compatible with any of the above configurations, you will encounter a link-time error in which the linker is unable to locate a compatible version of a runtime library in order to successfully complete the link. If this happens, and the configuration is valid for a TI device, please report the issue on the TI E2E™ design community and select the device being used.

For the compile options used in each of these configurations, refer to *Processor Variants Supported by tiarmclang*.

1.3 Creating a Simple Application with the tiarmclang Compiler Tools

This section of the Getting Started Guide provides an example of how to build a simple application using the tiarmclang command-line interface. In addition, it provides a walk-through of how to build a simple application in a Code Composer Studio project that uses the tiarmclang compiler.

The example on this page compiles to run on a TI Arm Cortex-M4.

1.3.1 Source Files

The subsections below describe how to build a simple "Hello World!" program using either the tiarmclang command-line interface or the Code Composer Studio (CCS) development environment. For the purposes of these tutorial examples, it is assumed that you have a C source file containing the following C code:

```c
#include <stdio.h>

int main() {
  printf("Hello World\n");
  return 0;
}
```

It is also assumed that you have at your disposal a linker command file that provides a specification of the available memory and how to place compiler/linker generated output sections in that memory. For example, in the tutorials below, the following linker command file, named *lnkme.cmd*, could be used:

```
/
↪*****************************************************************************
↪
/* Example Linker Command File                                         ␣
↪          */
/
↪*****************************************************************************
↪
-c                                               /* LINK USING C␣
↪CONVENTIONS      */
-stack  0x8000                                   /* SOFTWARE STACK␣
↪SIZE          */
```
(continues on next page)

```
--heap  0x2000                                  /* HEAP AREA SIZE
↪              */
--args  0x1000

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
    P_MEM    : org = 0x00000020   len = 0x20000000  /* PROGRAM
↪MEMORY (ROM) */
    D_MEM    : org = 0x20000020   len = 0x20000000  /* DATA
↪MEMORY   (RAM) */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    .intvecs   : {} > 0x0                  /* INTERRUPT VECTORS
↪           */
    .bss       : {} > D_MEM                /* GLOBAL & STATIC VARS
↪           */
    .data      : {} > D_MEM
    .sysmem    : {} > D_MEM                /* DYNAMIC MEMORY
↪ALLOCATION AREA    */
    .stack     : {} > D_MEM                /* SOFTWARE SYSTEM STACK
↪           */

    .text      : {} > P_MEM                /* CODE
↪           */
    .cinit     : {} > P_MEM                /* INITIALIZATION TABLES
↪           */
    .const     : {} > P_MEM                /* CONSTANT DATA
↪           */
    .rodata    : {} > P_MEM
    .ARM.exidx : {} > P_MEM
    .init_array : {} > P_MEM               /* C++ CONSTRUCTOR TABLES
↪           */
}
```

1.3.2 Compile and Link Using Command-Line

You can use a single command line to compile your source files and link against C/C++ runtime libraries to create an executable file. By default, the tiarmclang compiler will compile and attempt to link compiler generated object files with runtime libraries. If you only want to compile your source files into object files without linking, the tiarmclang -c option can be added to the command line.

If you were building the "Hello World!" example program, you could use the a command like the following:

```
%> tiarmclang -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16␣
↪hello.c -o hello_world.out -Xlinker -llnkme.cmd -Xlinker -␣
↪mhello_world.map
```

When the tiarmclang compiler runs, it implicitly adds the directories where the C/C++ runtime header files are installed to the include file directory search path. Likewise, when the linker is invoked by tiarmclang, it implicitly adds the directories where the C/C++ runtime libraries are installed to your library file directory search path. In addition, the linker implicitly includes the list of applicable C/C++ runtime libraries into a link to resolve references to C/C++ runtime and built-in functions and data objects.

The above command produces an executable file, hello_world.out, which can be loaded and run on the appropriate Arm processor.

1.3.3 Compile and Link Using Build Automation Tools

You can use build systems to automate the command line compilation and linking steps. Supported build systems include GNU Make and CMake (v3.29 or higher).

For example, the following CMakeLists.txt file contains the directives required to use CMake to build a sample tiarmclang application:

```
cmake_minimum_required(VERSION 3.29) # Minimum version for␣
↪tiarmclang support

#----------------------------------------------------------------
↪-------------
# Set up cross compiling with TI clang compiler
#----------------------------------------------------------------
↪-------------
set(CMAKE_SYSTEM_NAME Generic) # Inform cmake of cross-compiling

# find tiarmclang in execution path
find_program(CMAKE_C_COMPILER tiarmclang)
```

```
# or set path to tiarmclang explicity
#set(CMAKE_C_COMPILER  /Users/ti/ti-cgt-armllvm_3.2.0.LTS/bin/
 ↪tiarmclang)
#-----------------------------------------------------------------
 ↪-------------
project(DDREyeFirmware C)

add_executable(app
    main.c
    lib/uart_lib/src/uartConsole.c
    lib/uart_lib/src/uartStdio.c
    lib/uart_lib/src/uart.c
    lib/jacinto7_ddrss_tools_lib/src/jacinto7_ddrss_tools_lib.c
    lib/jacinto7_ddrss_tools_lib/src/mmr.c
    lib/pattern_gen_lib/src/pattern_gen_lib.c)

add_compile_options(-mcpu=cortex-r5 -mfloat-abi=hard -mfpu=vfpv3-
 ↪d16 -mlittle-endian -mthumb)
add_compile_options(-Oz -g)

add_compile_definitions(J784S4)

include_directories(lib/uart_lib/inc)
include_directories(lib/uart_lib/src)
include_directories(lib/jacinto7_ddrss_tools_lib/inc)
include_directories(lib/jacinto7_ddrss_tools_lib/src)
include_directories(lib/pattern_gen_lib/inc)
include_directories(lib/pattern_gen_lib/src)

add_link_options(LINKER:--ram_model)
add_link_options(LINKER:--warn_sections)
add_link_options(${CMAKE_SOURCE_DIR}/linker.cmd)
```

1.3.4 Compile and Link Using Code Composer Studio

If you use CCS as your development environment, the compiler and linker option are automatically set for you when you create a project. The build settings that are created when the project is created determine which compiler and linker command-line options are used to build the project and can be adjusted as needed.

To create and build the "Hello World!" example as a CCS project, follow these steps:

1) Create a project

>1.1) Choose **File > New > CCS Project** from the "File" tab

>1.2) In the "New CCS Project" wizard, if you are compiling for a specific TI Arm processor, then you can select the processor from the **Target** drop-down menus. For the purposes of this tutorial, the "Generic ARM7 Device" setting was selected in the right-hand side **Target** drop-down menu.

>1.3) In the **Project name** field, type a name for the project. For the purposes of this tutorial, we'll refer to the project name "hello_world".

>1.4) In the **Compiler version** drop-down menu, select the tiarmclang compiler that you have installed.

>1.5) Expand the **Project type and tool-chain** section, then select the device endianness. For this tutorial, a little-endian device is assumed.

>1.6) Expand the **Project templates and examples** section, then select a template for your project. For this tutorial, the "Empty Project" template is assumed.

>1.7) Click **Finish**. A new project, "hello_world", will be added to your current workspace.

2) Add source files

>2.1) Left click on the "hello_world" project in the current workspace to make it the active project.

>2.2) Right-click on the "hello_world" project, then select **Add Files…"** from the drop-down menu. You can then browse to find a C source file containing the code described in "Source Files" subsection above. When the source file is found and selected in the **Add files to hello_world** pop-up browser, clock on **Open** and then copy the file into the project. For this tutorial, the source file name is assumed to be "hello.c".

>2.3) Repeat step 2.3 to find and copy an appropriate linker command file to be used in the project. For this tutorial, a linker command file named "lnkme.cmd" is assumed.

3) Open and adjust project build settings as needed

>3.1) Right-click on the "hello_world" project, then select **Show Build Settings…** or **Properties** from the drop-down menu.

>3.2) In the **Properties for hello_world** pop-up dialog box, walk through the categories along the left-hand side of the dialog and make necessary adjustments in each category:

>>3.2.1) In the **General** category, check that the proper **compiler version**, **Device endianness**, and **Linker command file** are selected

3.2.2) In the **Arm Compiler > Processor Options** category, select the appropriate **-march**, **-mcpu**, **-mfloat-abi**, **-mfpu**, and **arm/thumb** options from each of the drop-down menus in the **Processor Options** window. For this tutorial, **-march** is set to **thumbv7em**, **-mcpu** is set to **cortex-m4**, **-mfloat-abi** is set to **hard**, **-mfpu** is set to **fpv4-sp-d16**, and **arm/thumb** is set to **-mthumb**.

3.2.3) Further adjustments to other categories are not necessary for the purposes of this tutorial.

3.2.4) When adjustments to the **Properties for hello_world** are complete, click on **Apply and Close**

4) Build the project

4.1) Right-click on the "hello_world" project, then select **Build Project** from the drop-down menu.

4.2) As CCS runs the compiler and linker commands, the project build progress will appear in the CCS **Console** window, with a resulting output file named "hello_world.out". This .out file can then be loaded and run on an appropriate TI Arm processor. The resulting .out file can be loaded and run on the appropriate TI Arm processor.

1.4 Assembling GNU-Syntax tiarmclang Assembly Language Source

GNU-syntax Arm assembly language source files can be assembled using the tiarmclang compiler's integrated assembler. When the compiler encounters a file name with an .s or .S extension on the command-line, it will interpret the contents of that file as GNU-syntax Arm assembly language source.

For example, in the following tiarmclang command:

```
%> tiarmclang -mcpu=cortex-m4 -c file1.c file2.s file3.S
```

"file1.c" is interpreted as a C source file and both "file2.s" and "file3.S" are interpreted as GNU-syntax Arm assembly language source files. The tiarmclang compiler will pre-process a GNU-syntax Arm assembly file with a ".S" extension. Whereas, it will not pre-process a GNU-syntax Arm assembly file with a ".s" extension.

1.5 Assembling Legacy TI-Syntax Arm Assembly Language Source

In addition to the tiarmclang compiler's integrated assembler used to assemble GNU-syntax Arm assembly language source files, the tiarmclang compiler toolchain also includes a stand-alone legacy TI-syntax Arm assembler, tiarmasm, that can be used to assemble legacy TI-syntax Arm assembly source files.

The use of the legacy TI-syntax Arm assembler can be useful when you are migrating a project built with the armcl compiler to use the tiarmclang compiler, but there are legacy TI-syntax Arm assembly source files that will not require maintenance. For such files, you may choose to assemble the source file with the legacy TI-syntax Arm assembler rather than convert the source file into GNU-syntax Arm assembly language. Please see the *Migrating Assembly Language Source Code* for more information.

You can invoke the legacy TI-syntax Arm assembler to create an object file from a legacy TI-syntax Arm assembly source file like so:

```
%> tiarmasm <options> file.asm
```

With the help of the tiarmclang's -x option, the legacy TI-syntax Arm assembler can also be invoked from the tiarmclang compiler command-line as follows:

```
%> tiarmclang <options> -x ti-asm file1.asm -x none file2.c
```

The "-x ti-asm" option tells the compiler to interpret any source files that follow it as legacy TI-syntax Arm assembly files, for which the compiler will invoke tiarmasm from inside the compiler. If other file types are included on the command-line, then the "-x none" option can be used to instruct the compiler to interpret subsequent files based on their extension. IN the above example, "file1.asm" will be interpreted as a legacy TI-syntax Arm assembly file and "file2.c" will be interpreted as a C file. Please see *Invoking the TI-Syntax ARM Assembler from tiarmclang* for more information.

1.6 Processor Variants Supported by tiarmclang

The tiarmclang compiler toolchain supports development of applications that are to be loaded and run on one of the following Arm Cortex processor variants (applicable -mcpu and floating-point support options are listed for each):

- **Cortex-m0**

  ```
  -mcpu=cortex-m0
  ```

- **Cortex-m0plus**

```
-mcpu=cortex-m0plus
```

- **Cortex-m3**

```
-mcpu=cortex-m3
```

- **Cortex-m4**

  With FPv4SPD16 support:

```
-mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16
```

  Without FPv4SPD16 support:

```
-mcpu=cortex-m4 -mfloat-abi=soft
```

- **Cortex-m33**

  With FPv5SPD16 support:

```
-mcpu=cortex-m33 -mfloat-abi=hard -mfpu=fpv5-sp-d16
```

  Without FPv5SPD16 support:

```
-mcpu=cortex-m33 -mfloat-abi=soft
```

Note: Enabling the Use of Custom Datapath Extension (CDE) Intrinsics

A TI Arm Cortex-M33 device may be equipped with a coprocessor that is able to execute custom datapath extension (CDE) instructions via intrinsics that are defined in arm_cde.h, which is included in the tiarmclang compiler tools installation.

To enable the use of CDE intrinsics in a C/C++ source file, you must include the arm_cde.h header file in your compilation unit prior to the first reference to a CDE intrinsic. You must also specify one of the following *-march* compiler options on the tiarmclang command-line:

```
-march=armv8.1-m.main+cdecp0
```

or

```
-march=thumbv8.1-m.main+cdecp0
```

For more information about the CDE intrinsics that are supported, please see *Custom Datapath Extension (CDE) Intrinsics*.

- **Cortex-r4**

Thumb mode with VFPv3D16

```
-mcpu=cortex-r4 -mthumb -mfloat-abi=hard -mfpu=vfpv3-
↪d16
```

Thumb mode without VFPv3D16

```
-mcpu=cortex-r4 -mthumb -mfloat-abi=soft
```

Arm mode with VFPv3D16

```
-mcpu=cortex-r4 -marm -mfloat-abi=hard -mfpu=vfpv3-
↪d16
```

Arm mode without VFPv3D16

```
-mcpu=cortex-r4 -marm -mfloat-abi=soft
```

- **Cortex-r5**

  Thumb mode with VFPv3D16

  ```
  -mcpu=cortex-r5 -mthumb -mfloat-abi=hard -mfpu=vfpv3-
  ↪d16
  ```

  Thumb mode without VFPv3D16

  ```
  -mcpu=cortex-r5 -mthumb -mfloat-abi=soft
  ```

  Arm mode with VFPv3D16

  ```
  -mcpu=cortex-r5 -marm -mfloat-abi=hard -mfpu=vfpv3-
  ↪d16
  ```

  Arm mode without VFPv3D16

  ```
  -mcpu=cortex-r5 -marm -mfloat-abi=soft
  ```

The tiarmclang compiler will default to the Cortex-M4 ("-mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16") processor if you do not explicitly specify an -mcpu or -march option on the compiler command-line.

TWO

TI ARM TO TIARMCLANG MIGRATION GUIDE

This *Migration Guide* addresses tasks required and issues encountered when porting your existing TI ARM (armcl) application to tiarmclang.

Among these are:

- Converting build options used in an existing armcl project to appropriate tiarmclang build options,

- Making your C/C++ source code more portable by converting legacy TI instances of pre-defined macro symbols, pragmas, and intrinsics into functionally equivalent ACLE (Arm C Language Extensions) forms, and

- Converting assembly source code written in legacy TI-syntax Arm assembly language to GNU-syntax Arm assembly language.

Available Migration Aids

There are a variety of migration aids available in the tiarmclang compiler tools and in Code Composer Studio (CCS) version 10.1 or later to help you address these issues when converting an existing armcl project to use the tiarmclang compiler:

- **CCS Project Migration** - for help converting the build options for an existing armcl CCS project to use the tiarmclang compiler, please see *Migrating armcl CCS Projects to tiarmclang*.

- **C/C++ Source Code Migration Aid Diagnostics** - the tiarmclang compiler can help you find instances of legacy TI pre-defined macro symbols, pragmas, and intrinsics that need to be converted into ACLE form via the use of migration aid diagnostics. For more information, please see *C/C++ Source Migration Aid Diagnostics*.

- **Invoking the Legacy TI-Syntax Arm Assembler from tiarmclang** - if you have assembly source code written in legacy TI-syntax Arm assembly language that you are unlikely to need to change or maintain in your project, then you may want to use the legacy TI-syntax Arm assembler to process that code. The tiarmclang compiler allows you to invoke the legacy TI-syntax Arm assembler from the **tiarmclang** command-line. For more information about how to do this, please see *Invoking the TI-Syntax ARM Assembler from tiarmclang*.

Contents:

2.1 Main Differences Between armcl and tiarmclang

This section of the *Migration Guide* describes the primary differences between the armcl compiler tools and the TI Arm Clang (tiarmclang) Compiler Tools.

For more information about the differences introduced below, see the following subsections:

2.1.1 C/C++ and Assembly Language Differences

The TI Arm Clang (tiarmclang) Compiler Tools do not support legacy TI-specific C/C++ mechanisms that the TI ARM compiler (armcl) did. You can, however, invoke the legacy Arm assembler from the **tiarmclang** command to process legacy TI-syntax Arm assembly source files. This section provides some further details about the tiarmclang compiler's behavior regarding legacy TI-specific C/C++ mechanisms and Arm assembly source files.

C/C++ Source Code: Macro Symbols, Pragmas, and Intrinsics

The tiarmclang compiler supports pre-defined macro symbols and intrinsics that are included in the Arm C Language Extensions (ACLE).

It is highly recommended that you make use of ACLE-compliant pre-defined macro symbols and intrinsics in your C/C++ source code. You can find details about ACLE pre-defined macro symbols and intrinsics in the Arm C Language Extensions - Release ACLE Q2 2018 specification.

Legacy TI pre-defined macros symbols, pragmas, intrinsics are not supported by tiarmclang.

The tiarmclang compiler does not support many of the legacy TI pre-defined macro symbols, pragmas, or intrinsics that are supported in the TI ARM compiler. Use of legacy TI-specific pre-defined macro symbols and intrinsics can usually be replaced by a functionally equivalent ACLE-compliant pre-defined macro symbol or intrinsic.

Rather than using legacy TI-specific pragmas (such as CODE_SECTION, DATA_SECTION, and LOCATION), you should use function, variable, and type attributes where applicable. For example, instead of defining a function in a specially named section using the CODE_SECTION pragma:

```
#pragma CODE_SECTION(my_func, ".text:myfunc")
void my_func(void) {
    <code>
}
```

you can use a section attribute instead:

```
void my_func(void) __attribute__((section(".text:my_func"))) {
    <code>
}
```

Please see the *C/C++ Source Migration Aid Diagnostics* section in the *Migrating C and C++ Source Code* chapter for details on how instances of these legacy TI mechanisms can be found and converted into a portable form that the tiarmclang compiler does support.

Assembly Source Code

- The tiarmclang compiler assumes the use of GNU-syntax Arm assembly language

  The tiarmclang compiler, when instructed to generate assembly source via the **-S** option, produces GNU-syntax rather than legacy TI-syntax Arm assembly source code. Furthermore, the default behavior of the tiarmclang compiler when presented with an assembly source file (with *.s* or *.S* file extension) is to interpret the source file as a GNU-syntax Arm assembly source file.

  The tiarmclang compiler can be made to process legacy TI-syntax assembly source with its TI-syntax Arm assembler, but you must precede such an input file with the **-x ti-asm** option. For more information on migrating assembly source code from legacy TI-syntax to GNU-syntax, please see the *Migrating Assembly Language Source Code* chapter of this migration guide.

2.1.2 Development Flow Differences

There are a few significant differences in terms of development flow behavior when migrating from the TI ARM compiler to the tiarmclang compiler. These include the following:

- **The linker is invoked automatically by default by the compiler.**

  The tiarmclang compiler invokes the linker automatically by default, whereas the TI ARM compiler must be told to invoke the linker via the armcl's **--run_linker** (**-z**) option. Further details about how to manage the linker invocation from the **tiarmclang** command-line can be found in the *Using the tiarmclang Compiler and Linker* section of the *tiarmclang Getting Started Guide*.

- **The interlist option is not supported on the compiler command line.**

  Unlike the TI ARM compiler, which provides **-s**, **-ss**, and **-os** options to instruct the compiler to generate an interlisted assembly source file, the tiarmclang does *not* support an interlisting option on the compiler command-line. Instead, when a C/C++ source file is compiled with debug enabled, the **tiarmobjdump** utility can be used with its **-S** option on the compiler-generated object file to produce disassembled object code with C/C++ source lines interlisted.

- **Altering the file extension of generated files is not supported on the compiler command line.**

  The tiarmclang compiler does not support options to alter the file extension of compiler-generated files. For more details about which TI ARM options do not have analogous tiarmclang options, please see the *Migrating Command-Line Options* chapter of this migration guide.

- **Compilation stops after generating assembly source if the -S option is specified.**

  The tiarmclang compiler supports a **-S** option that allows you to keep the compiler-generated assembly file, but unlike the armcl's **-k** option, the tiarmclang's **-S** option causes the compiler to halt immediately after generating the assembly file. When **-S** is used, an object file is not created by the compiler.

2.1.3 Binary Utility Differences

There are several differences in the behavior of the asm() statement and binary utilities used for C++ name demangling, symbol name listing, and disassembly. These differences are described here.

Inlining Functions that Contain asm() Statements

The tiarmclang compiler allows a function containing an asm() statement to be considered for inlining. The TI ARM compiler does not allow a function containing an asm() statement to be inlined.

If an asm() statement in a function contains the definition of a symbol, then you should strongly consider applying a *noinline* attribute to the function that contains such an asm() statement.

For example, consider the following function definition:

```
void func_a() {
  ...
  asm("a_label:\n");
  ...
}
```

The above function contains a definition of *a_label*. The TI ARM compiler does not allow any function that contains an asm() statement to be inlined. Thus, in the above example, the TI ARM compiler would not attempt to inline *func_a* in any other function that references *func_a*.

The tiarmclang compiler behavior with respect to functions that contain asm() statements is different from the TI ARM compiler. The tiarmclang compiler allows functions containing asm()

statements to be considered for inlining where those functions are referenced. If a function contains an asm() statement that defines a symbol and is inlined multiple times in the same compilation unit, this can cause the tiarmclang compiler to emit a "symbol multiply defined" error diagnostic.

Consider that the above definition of *func_a* is in the same compilation unit as another function, *func_b*:

```
void func_b() {
  ...
  func_a();
  ...
  func_a();
  ...
}
```

If the tiarmclang compiler decides that it is beneficial to inline *func_a* where that function is referenced in *func_b*, the result is multiple definitions of the label *a_label* and the tiarmclang compiler emits an error diagnostic.

You can prevent the tiarmclang compiler from inlining a function by applying a *noinline* attribute to the function in question. For example, you could rewrite *func_a* as follows:

```
__attribute__((noinline))
void func_a() {
  ...
  asm("a_label:\n");
  ...
}
```

This adjustment to the definition of *func_a* prevents *func_a* from being inlined anywhere where it is referenced and avoid any potential of defining *a_label* multiple times in the same compilation unit.

Updated C++ Name Demangler Utility (tiarmdem)

The TI Arm Clang (tiarmclang) Compiler Tools include an LLVM-based version of the C++ Name Demangler Utility (tiarmdem). While the LLVM-based version of this utility is functionally equivalent to the TI ARM compiler tools' version, the command-line interface for the new version is different from the TI ARM version.

The C++ name demangler (tiarmdem) is a debugging aid that translates C++ mangled names to their original name found in the relevant C++ source code. The tiarmdem utility reads in input, looking for mangled names. All unmangled text is copied to output unaltered. All mangled names are demangled before being copied to output.

The syntax for invoking the C++ name demangler provided with tiarmclang is:

```
tiarmdem [options] <mangled names ...>
```

- **options** - affect how the name demangler behaves. The tiarmdem utility is derived from the LLVM project's **llvm-cxxfilt** tool. To display a list of available options, use the help option (**-h**), **tiarmdem -h**. You can also refer to the LLVM project's llvm-cxxfilt page for more information.

- **mangled names ...** - if no names are specified on the command-line, names are read interactively from the standard input stream.

By default, the C++ name demangler writes output to stdout. You can pipe the output to a file if desired.

Differences between the TI ARM version of the C++ name demangler and the tiarmclang version of the C++ name demangler are as follows.

Processing Text Input

Unlike the TI ARM version of the C++ name demangler, the tiarmclang version of tiarmdem does not process a text file specified as an argument to tiarmdem. Assuming that **test.s** is a compiler generated assembly file, you cannot specify **test.s** as an argument to tiarmdem. Instead, you can pipe the text file as input to the tiarmdem utility as follows:

```
tiarmdem < test.s
```

or

```
cat test.s | tiarmdem
```

Saving Output to a File

The TI ARM version of the C++ name demangler supported an "--output-*file*" option that allowed you to write the output of the tiarmdem utility to a file. The tiarmclang version of tiarmdem does not support a --output option. Instead, the output can be redirected to a file like so:

```
cat test.s | tiarmdem > tiarmdemout
```

No ABI Option Needed

The TI ARM version of the C++ name demangler required that an --abi=eabi option be specified in order to demangle C++ names that are generated by the tiarmclang compiler. The tiarmclang version of tiarmdem assumes EABI and no ABI option is needed to process tiarmclang compiler generated C++ mangled names.

Updated Name Utility (tiarmnm)

The TI Arm Clang (tiarmclang) Compiler Tools include an LLVM-based version of the Name Utility (tiarmnm). While the LLVM-based version of this utility is functionally equivalent to the TI ARM compiler tools' version, the command-line interface for the new version is different from the TI ARM version.

The name utility (tiarmnm) prints the list of symbol names defined and referenced in an object file, executable file, or object library. It also prints the symbol value and an indication of the symbol's kind.

The syntax for invoking the name utility is:

```
tiarmnm [options] <input files>
```

- **options** - affect how the name utility behaves. The tiarmnm utility is derived from the LLVM project's **llvm-nm** tool. To display a list of available options, use the help option (**-h**), **tiarmnm -h**. You can also refer to the LLVM project's llvm-nm page for more information.

- **input files** - an input file can be an object file, an executable file, or an object library

The output of the name utility is written to stdout. You can also elect to pipe the output to a file or as input to the C++ name demangler.

Differences between the TI ARM version of the C++ name utility and the tiarmclang version of the C++ name utility are as follows.

Symbol Kind Annotations

In the output from the tiarmnm utility, symbol names are annotated with an indication of their kind. The tiarmclang version of the tiarmnm utility uses the following list of annotation characters to represent the different symbol kinds:

- **a, A** - absolute symbol

- **b, B** - uninitialized data (bss) object

- **C** - common symbol

- **d, D** - writable data object

- **n** - local symbol from a non-alloc section

- **N** - debug symbol or global symbol from a non-alloc section

- **r, R** - read-only data object

- **t, T** - code (text) object

- **u** - GNU unique symbol

- **U** - named object is undefined in this file

- **v** - undefined weak object symbol

- **V** - defined weak object symbol

- **?** - something unrecognizable

Thumb Function Symbols

In the TI ARM version of the name utility, the value of a thumb function symbol would include a 1 in the least significant bit (the thumb mode bit), but the tiarmclang version of tiarmnm does not report a 1 in the least significant bit position for thumb function symbol values.

Debug Symbol Names

The TI ARM version of the name utility would include debug symbol names in the output. However, to include debug symbols in the output of the tiarmclang version of tiarmnm, you must specify the **tiarmnm**'s **--debug-syms** option on the command-line.

Functionally Equivalent Option Mappings

Several of the options available in the TI ARM version of the name utility now have functionally equivalent options with different syntax in the tiarmclang version of the tiarmnm utility. Below is a list of option mappings where the TI ARM's armnm option syntax is specified first and the tiarmclang's **tiarmnm** option syntax is specified second:

KEY: TI ARM **armnm** option syntax -> TI Arm **tiarmnm** option syntax - description

- **--all** -> **--debug-syms** - print all symbols

- **--prep_fname** -> **--print-file-name** - prepend file name to each symbol

- **--undefined** -> **--undefined-only** - only print undefined symbols

- **--sort:value** -> **--numeric-sort** - sort symbols numerically rather than alphabetically

- **--sort:reverse** -> **--reverse-sort** - sort symbols in reverse order

---

- **--global** -> **--externs-only** - print only global symbols

- **--sort:none** -> **--no-sort** - don't sort any symbols

Options No Longer Supported

The TI ARM version of the name utility supported several command-line options that are no longer supported in the tiarmclang version of the tiarmnm utility. These include:

- **--format:long** - produce detailed listing of symbol information

- **--output** - write output to a specified file

- **--quiet** - suppress banner and progress information

Symbol Kind Annotations

The TI ARM version of the name utility annotates some symbols with kind information differently than the tiarmclang version of the tiarmnm utility. One of the known differences is that the previous version of the name utility uses 'd' to annotate debug symbols, whereas the new version of tiarmnm uses 'N'. There may be other differences. Please consult the above list of symbol kind annotations for the tiarmnm utility for more information.

Saving Output to a File

As indicated above, the TI ARM version of the name utility supports a command-line option to write the output to a specified file, but the tiarmclang version of the tiarmnm utility does not support such a command-line option. Instead, you can elect to pipe the output of tiarmnm to a file:

```
tiarmnm test.o > tiarmnmout
```

or to the C++ name demangler utility, for example:

```
tiarmnm test.o | tiarmdem > tiarmdemout
```

An Example Using the Name Utility (tiarmnm) and the Name Demangler Utility (tiarmdem)

Consider the following source file (test.cpp):

```cpp
int g_my_num;
namespace NS { int ns_my_num = 2; }
```

(continues on next page)

```
int f() { return g_my_num + NS::ns_my_num; }
int main() { return f(); }
```

If the above test.cpp is compiled:

```
tiarmclang -mcpu=cortex-m4 -c test.cpp
```

We can then use the **tiarmnm** utility to write out the symbol names in test.o:

```
%> tiarmnm test.o
00000000 T _Z1fv
00000000 D _ZN2NS9ns_my_numE
00000000 B g_my_num
00000000 T main
```

and we could pass the output of **tiarmnm** to **tiarmdem** to demangle the mangled names that are present in the tiarmnm output:

```
%> tiarmnm test.o | tiarmdem
00000000 T f()
00000000 D NS::ns_my_num
00000000 B g_my_num
00000000 T main
```

Disassembling Object Files

The tiarmclang compiler toolchain provides two utilities that can be used to disassemble TI Arm object files, tiarmobjdump and tiarmdis.

tiarmobjdump

When invoked with the **--disassemble** (or **-d**) command, **tiarmobjdump** emits disassembled output for an Arm object file. If an Arm object file contains C/C++ source debug information, then the **--source** (or **-S**) option can be used to emit interlisted C/C++ source with the disassembled output.

> **tiarmobjdump –disassemble** [*options*] *filename*

See the *tiarmobjdump - Object File Dumper* section for more information about the tiarmobjdump utility.

tiarmdis

The standalone **tiarmdis** disassembler installed as part of the tiarmclang compiler toolchain is identical to the **armdis** utility that is provided with the legacy TI Arm C/C++ compiler toolchain (armcl).

> **tiarmdis** [*options*] *input_file* [*output_file*]

If an *output_file* is not specified, the disassembly output is written to *stdout*.

See the *tiarmdis - Standalone Disassembler Tool* section for more information about the tiarmdis utility.

Language Support For C/C++ and Assembly Differences

- **C/C++:** Discontinued support for legacy TI pre-defined macro symbols, pragmas, and intrinsics.

- **Assembly:** The tiarmclang compiler tools support GNU-syntax Arm assembly language by default as opposed to legacy TI-syntax Arm assembly language.

For more information, see *C/C++ and Assembly Language Differences*.

Development Flow Related Differences

- The tiarmclang compiler invokes the linker by default, whereas the user must specify armcl's **-run_linker** option to invoked the linker from the armcl command-line. However, the **--rom_model** (-c) linker option is not set by default by the tiarmclang compiler when running the linker. Therefore, the **--rom_model** (-c) or **--ram_model** (-cr) option must be passed to the linker on the tiarmclang command line or in the linker command file.

- The tiarmclang compiler does not support a C/C++ interlist option from the compiler command-line.

- The tiarmclang compiler ends compilation after emitting assembly output when using the **-S** option.

- The tiarmclang compiler does not support altering the file extension of compiler generated files.

For more information, see *Development Flow Differences*.

Differences in Behavior of Binary Utilities

- The armcl compiler does not attempt to inline a function that contains an asm() statement, but the tiarmclang compiler inlines a function containing an asm() statement if it is beneficial to do so.

- The command-line interface for the tiarmclang versions of the C++ Name Demangler Utility (**tiarmdem**) and the Name Utility (**tiarmnm**) behave differently than the armcl versions of armdem and armnm.

For more information, see *Binary Utility Differences*.

Differences in Type Aliasing Assumptions

- When optimizing memory accesses, tiarmclang assumes that pointers of different types *cannot* refer to the same memory. This means that if a pointer to an object is cast to a pointer *of a different type*, the compiler will treat the two pointers as not referencing the same memory. A user access of either pointer with the assumption that they refer to the same memory is

undefined behavior. The C standard allows compilers to make a less conservative assumption about strict type aliasing in order to better optimize code to yield better performance. This behavior is consistent with that of other compilers, but it is different from armcl, which makes a more conservative assumption while sacrificing optimization.

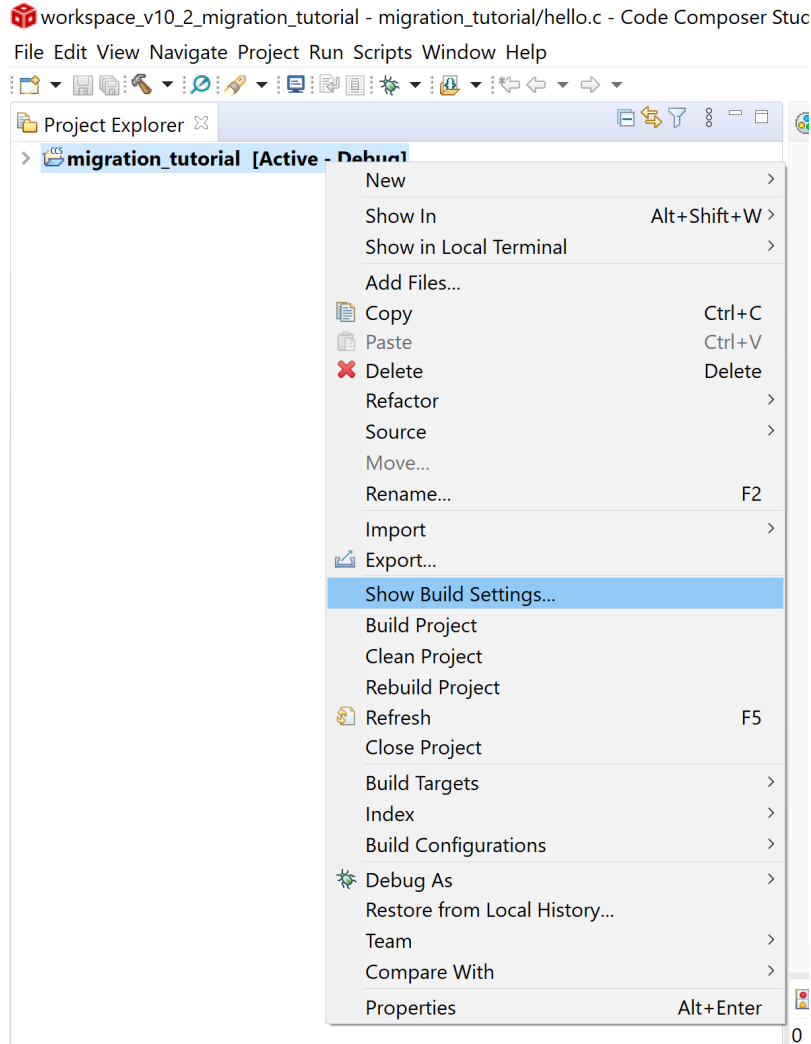- Users with code for which this poses problems should *disable strict type aliasing* by using the **-fno-strict-aliasing** compiler option.

For more information, see *Controlling Optimization*.

2.2 Migrating armcl CCS Projects to tiarmclang

One of the challenges you may face when transitioning an existing TI ARM application in order to build the application with the tiarmclang compiler is in mapping armcl compiler options into their corresponding tiarmclang compiler options. If your TI ARM application exists as a CCS project, then CCS can help with the mapping of armcl compiler options to tiarmclang options.

The process of migrating an armcl CCS project to use the tiarmclang compiler is relatively straight-forward:

1. Import the armcl CCS project into a CCS workspace.

2. In the **Project Explorer** window, left-click on the armcl project name that you want to migrate to make it the active project, then right-click on the armcl project name and select **Show Build Settings**.

workspace_v10_2_migration_tutorial - migration_tutorial/hello.c - Code Composer Stuc

File Edit View Navigate Project Run Scripts Window Help

Project Explorer ✖

> 📁 **migration_tutorial [Active - Debug]**

New	›
Show In	Alt+Shift+W ›
Show in Local Terminal	›
Add Files…	
Copy	Ctrl+C
Paste	Ctrl+V
Delete	Delete
Refactor	›
Source	›
Move…	
Rename…	F2
Import	›
Export…	
Show Build Settings…	
Build Project	
Clean Project	
Rebuild Project	
Refresh	F5
Close Project	
Build Targets	›
Index	›
Build Configurations	›
Debug As	›
Restore from Local History…	
Team	›
Compare With	›
Properties	Alt+Enter

3. You will then see a **Properties** pop-up dialog box for the armcl project.
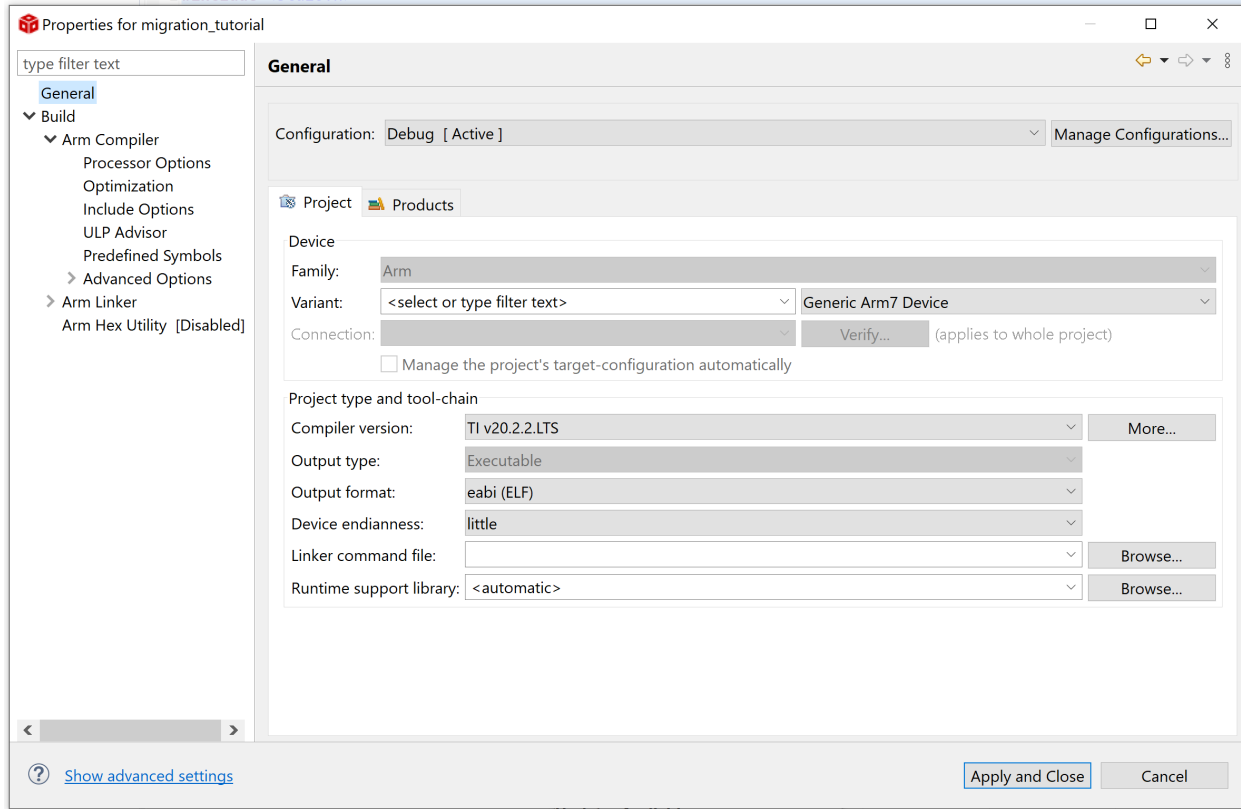
Before migrating the armcl project to use the tiarmclang compiler, you can check the armcl option settings. In this example, the **Processor Options** show that *-mv7m4* and *--float_support=FPV4SPD16* options have been selected:
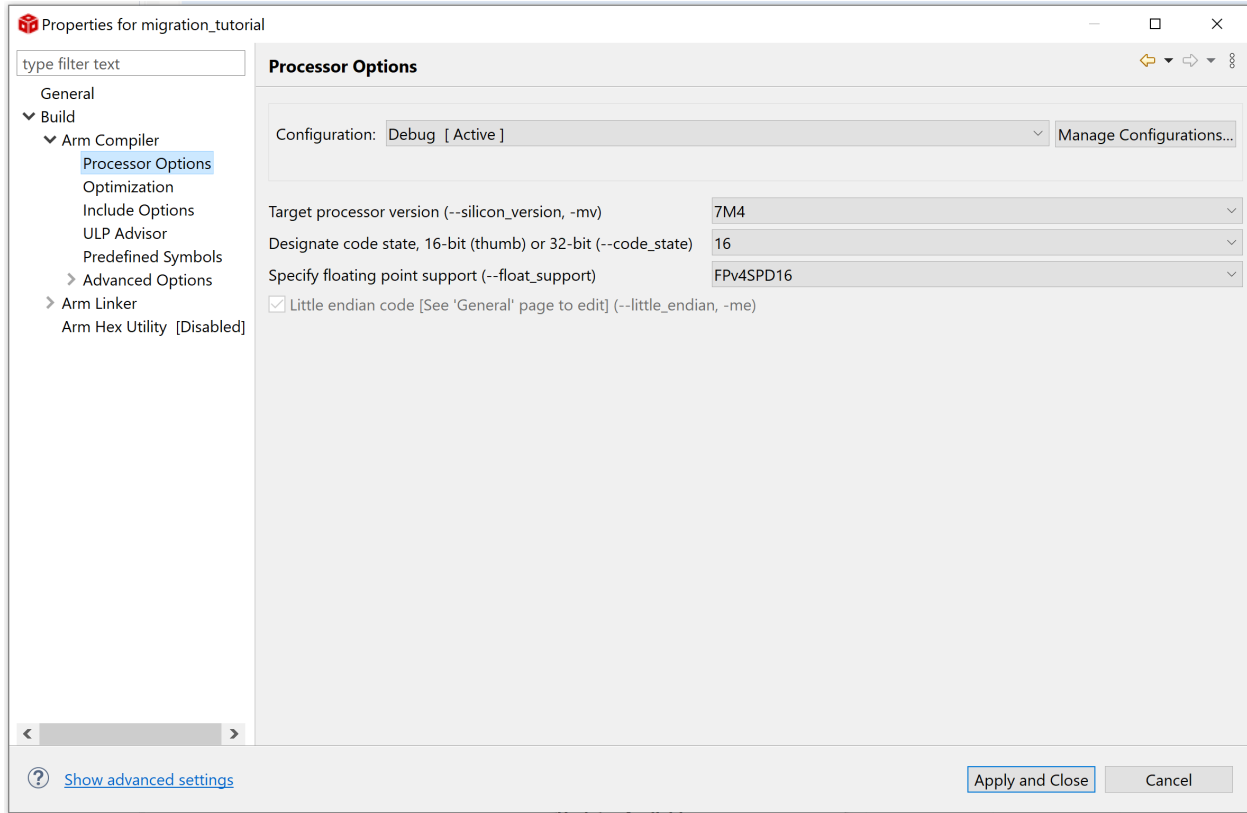
Also, the *-O3* and *-mf0* options have been selected among the **Optimization** options:
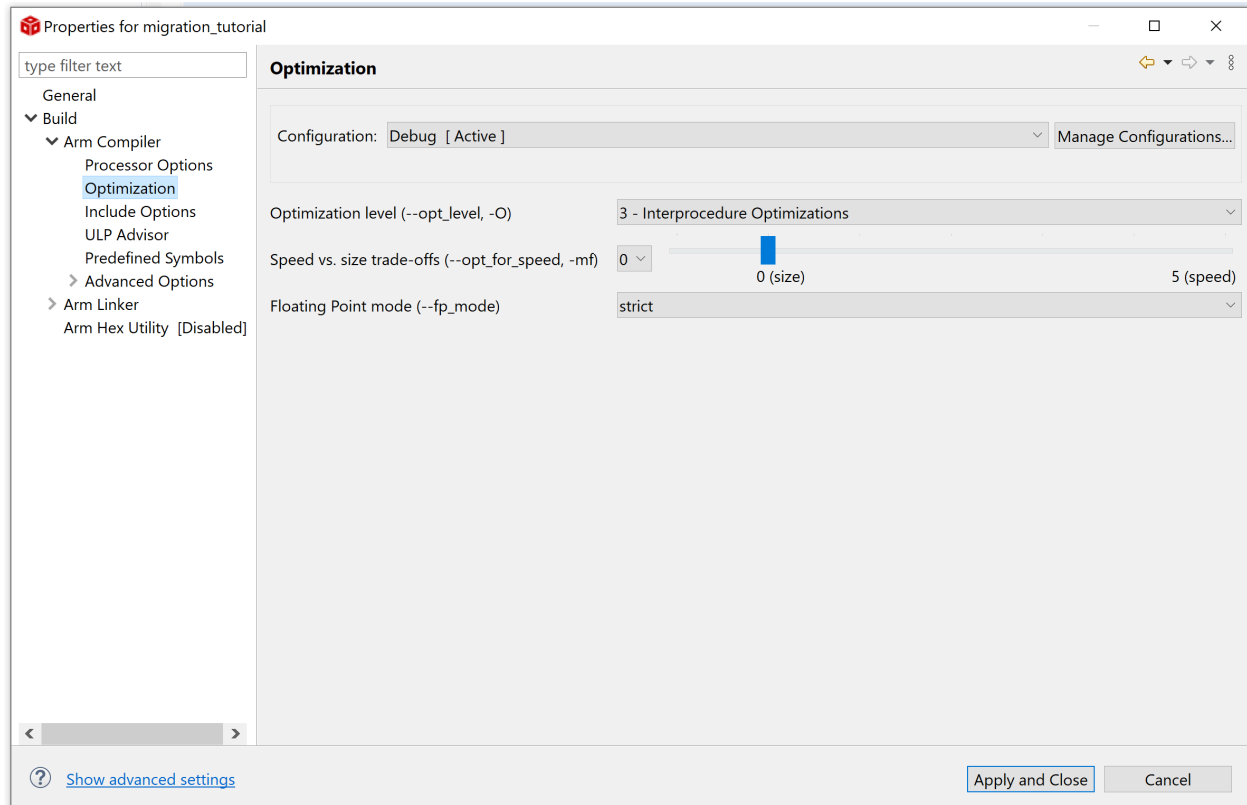
4. Left-click on the **General** category along the left-hand side of the dialog box and find the **Compiler version** drop-down menu near the middle of the dialog box, then Change the compiler selected from the current armcl compiler to the tiarmclang compiler (may be denoted as "TI Clang <version string>"):
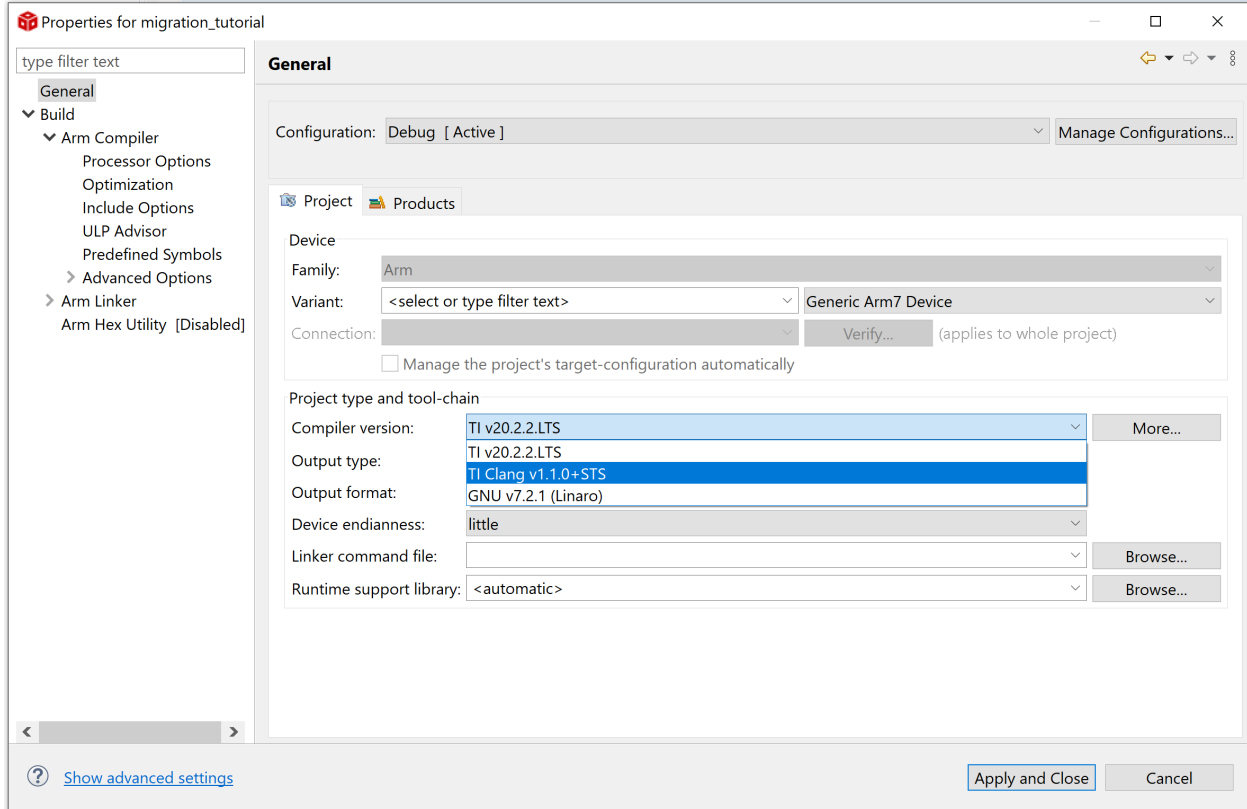
5. Click on **Apply and Close** at the bottom of the dialog box:
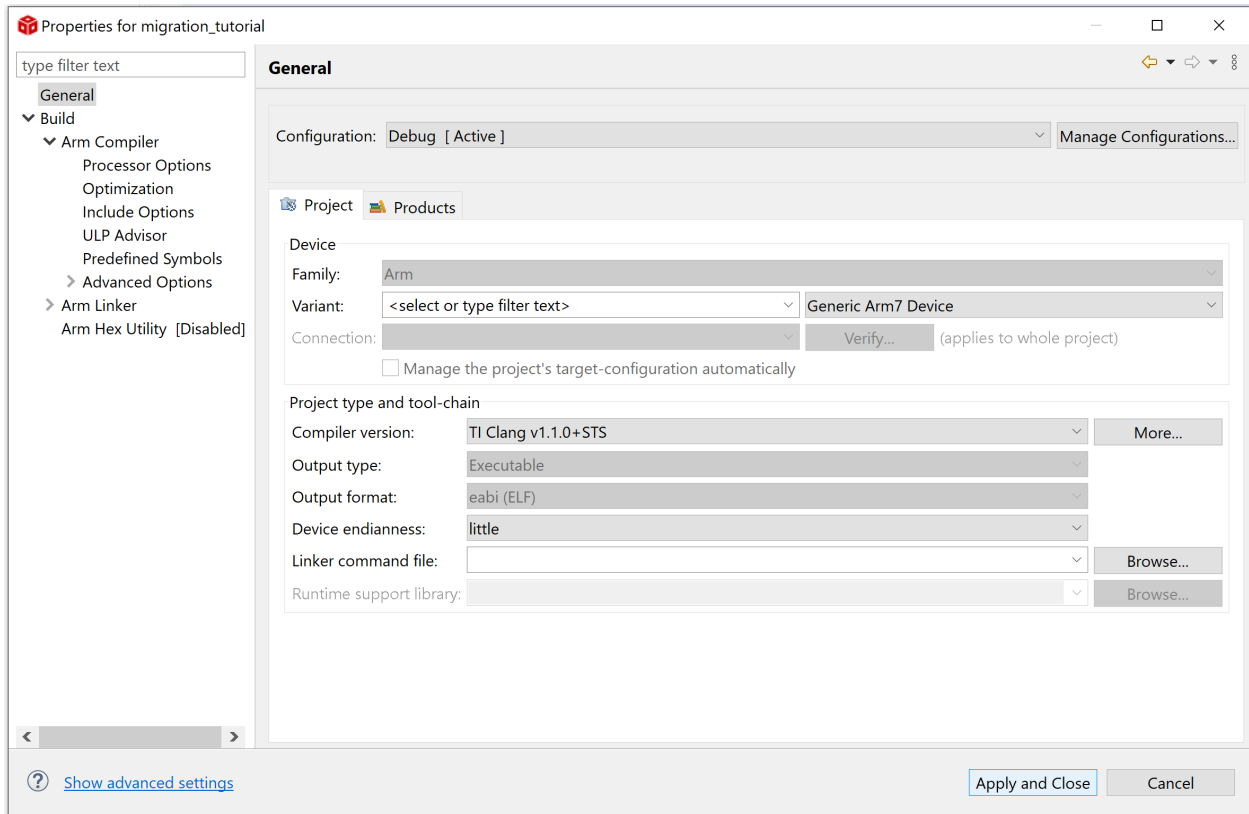
You will see a **Confirmation Needed** pop-up dialog asking if you'd like to proceed with the creation of a new build configuration:
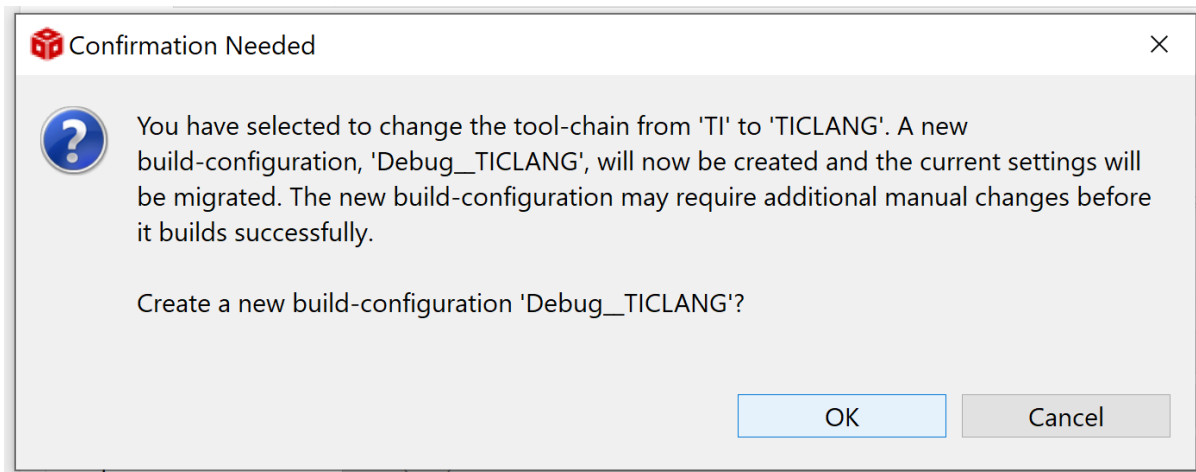
When you click **OK** in the **Confirmation Needed** dialog box, the migration process is started.

Unless all armcl compiler options were migrated flawlessly to their tiarmclang compiler counterparts, you will see a **Warning** pop-up dialog box explaining that some issues were encountered when creating the new build configuration:
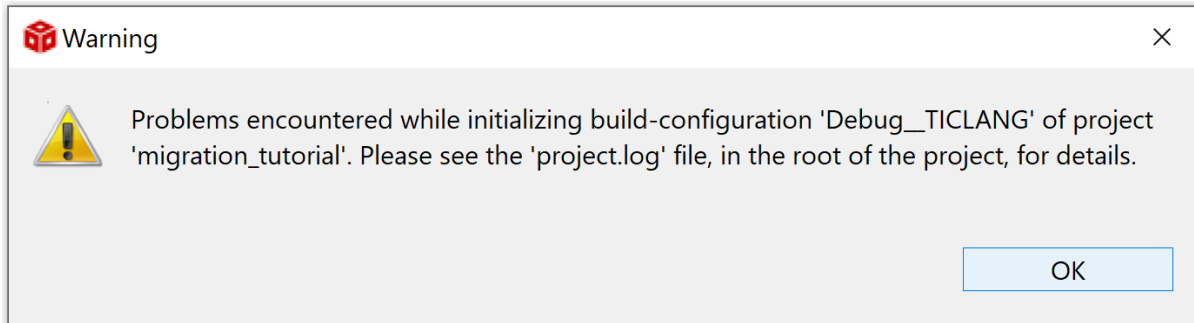
You can then click **OK** and proceed to view a project.log file in the CCS source file window.

The project.log file provides details about each of the armcl to tiarmclang option mappings that were enacted during the migration step. The armcl compiler options that CCS was not able to migrate into a functionally equivalent tiarmclang compiler option will be listed with a **!WARNING** message in the project.log file. You will want to review the mappings listed in the project.log file to ensure that each armcl compiler option was mapped to a tiarmclang option as you expected.

For this tutorial we can see that the armcl *-mv7m4* option was mapped to the *-mcpu=cortex-m4* option.

the armcl *--float_support=FPV4SPD16* option was mapped to the tiarmclang *-mfloat-abi=hard* and *-mfpu=fpv4-sp-d16* options,

and the armcl *-O3* and *-mf0* options were mapped to the tiarmclang *-Oz* option:

To further ensure that the armcl options were properly converted to tiarmclang options, you can check the build settings for the newly created TICLANG configuration of your project. Right-click on the project name and select **Show Build Settings** to bring up the **Properties** dialog box associated with the *TICLANG* configuration of your project.
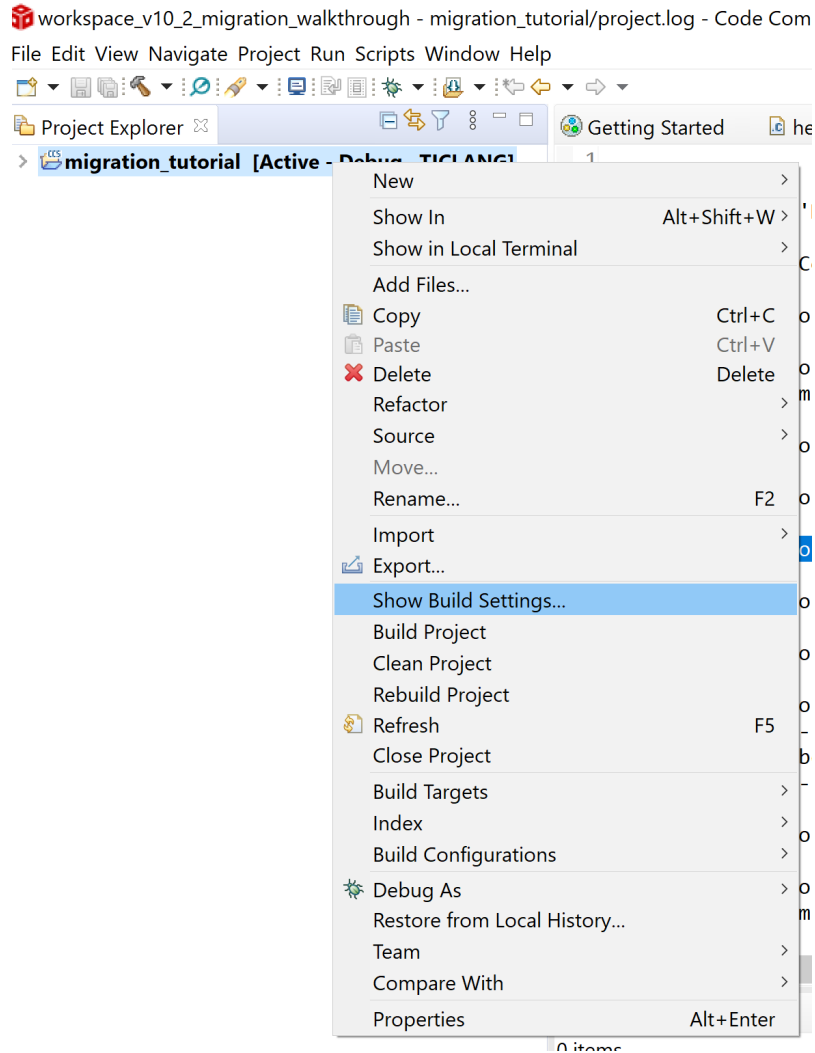
For this tutorial, we can see the **Processor Options** for the tiarmclang compiler show that *-mcpu=cortex-m4*, *-mfloat-abi=hard*, and *-mfpu=fpv4-sp-d16* options have been selected, as expected:
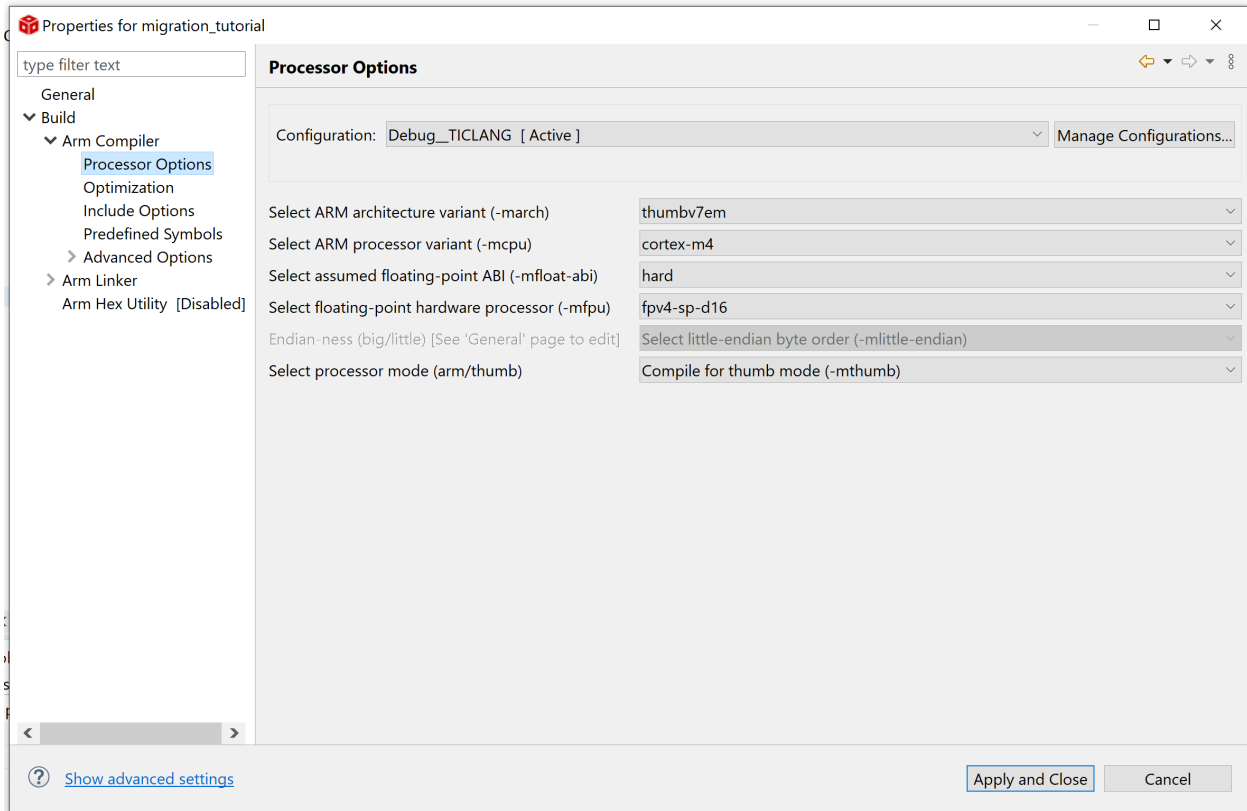
Also, the **Optimization** options view shows that the tiarmclang *-Oz* option has been selected:
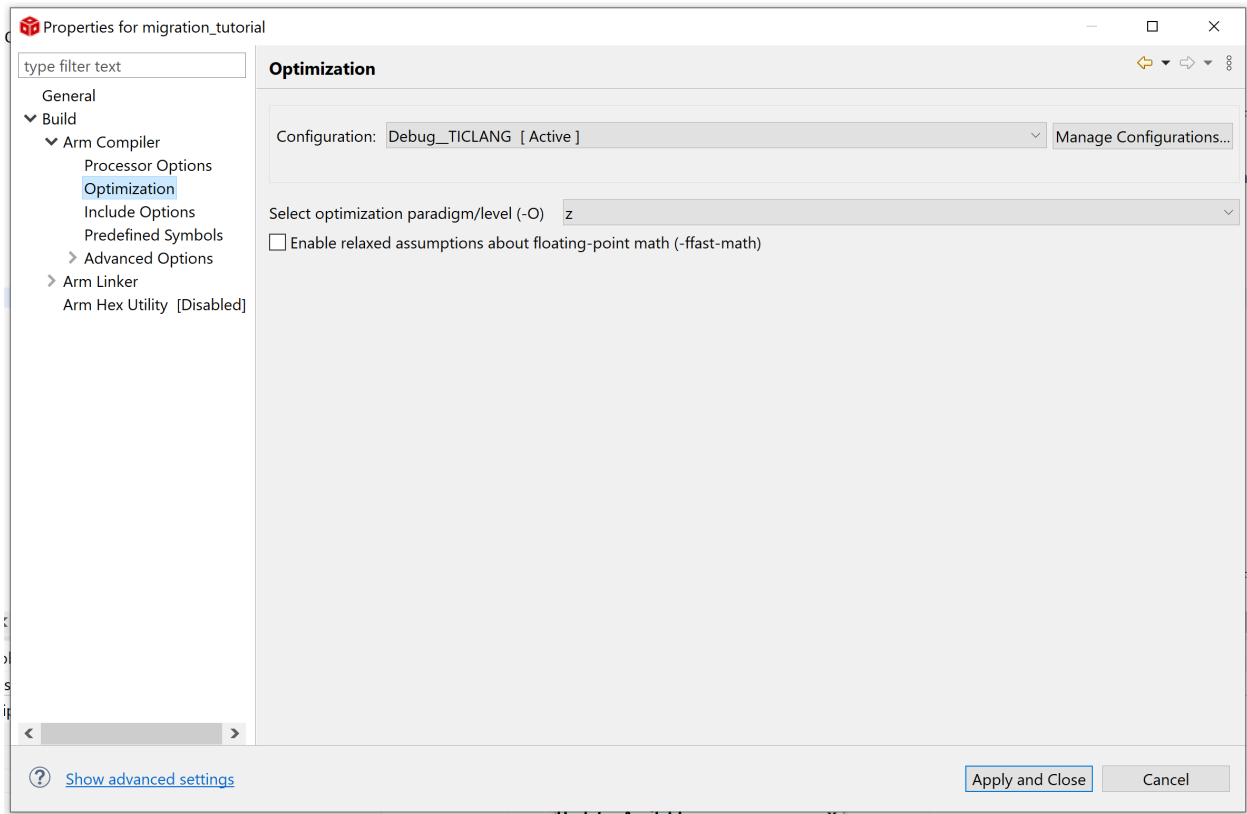
Further details about mapping armcl compiler options to tiarmclang compiler options are provided in the remainder of this chapter.

2.3 Migrating Command-Line Options

In this chapter of the Migration Guide, information is presented to help you map armcl command-line options to an appropriate tiarmclang command-line option, if a mapping is available. There are some armcl command-line options, like the -s interlisting options, that do not have a functional counterpart in the tiarmclang compiler. Such cases are clearly indicated in the tables in this chapter.

In several of this chapter's sub-sections, armcl options are shown side-by-side with one or more functionally relevant tiarmclang options in table form. These tables are often accompanied by a brief commentary discussing further details about the option mapping, including differences in behavior between the armcl and tiarmclang compiler with regards to the options under consideration.

Please note that while this chapter tries to account for all the options provided by the armcl compiler, it does not list all the tiarmclang command-line options that are available. If you cannot find a tiarmclang option that you are looking for in this chapter, refer to the Clang Compiler User's Manual for additional command-line options.

2.3.1 Managing Compiler Build Steps

By default, the tiarmclang compiler performs the following steps:

1. Preprocess the C/C++ source file

2. Compile the C/C++ source file(s) into temporary object file(s)

3. Automatically call the linker to produce an executable image

The following command-line options control the build-process steps performed by tiarmclang.

armcl Option (and alias)	tiarmclang Option
--compile_only (-c)	-c

Preprocess and compile source files, but do not link object files. The output is one object file for each source file.

armcl Option (and alias)	tiarmclang Option
--preproc_only (-ppo)	-E

Run only the preprocessor. The armcl compiler saves the preprocessed output in a .pp file, but the preprocessed output from tiarmclang is streamed to stdout.

armcl Option (and alias)	tiarmclang Option
--skip_assembler (-n)	-S

Halt compilation after code generation. Both the armcl and tiarmclang compilers halt after processing a C/C++ source file and before assembling the generated code into an object file. After halting, an assembly source file containing the generated code will be present in the current working directory. For the armcl compiler, the default file extension for a compiler generated assembly file is '.asm'. For the tiarmclang compiler, the file extension for a compiler generated assembly file is '.s'.

armcl Option (and alias)	tiarmclang Option
--run_linker (-z)	linker is invoked by default

The armcl compiler does not run the linker unless you use the --run_linker (-z) option. By default, the tiarmclang compiler automatically invokes the linker after compiling source files into object files. To see details about what command is used by the tiarmclang compiler to invoke the linker, you can specify the '-v' option on the tiarmclang command- line.

You can prevent tiarmclang from running the linker using one of these options:

- The -S option stops the compiler after generating an assembly file for each C/C++ source file on the command line.

- The -c option stops the compiler after generating an object file for each C/C++ source.

armcl Option (and alias)	tiarmclang Option
-z <linker options>	-Xlinker <linker option> -Wl,<comma-separated list of linker options>

Any options specified after the -z option using armcl are passed to the linker.

To pass options to the linker from the tiarmclang command-line, use either -Xlinker or -Wl. The tiarmclang compiler inserts these options into the list of options used when the linker is invoked after the compilation step. The -Xlinker option can be used to specify a single linker option (with no intervening spaces). The -Wl option accepts a comma-separated list of linker options.

Note that the --rom_model (-c) linker option, which is the default for armcl, is not set by default by the tiarmclang compiler when running the linker. Therefore, either the -rom_model (-c) or --ram_model (-cr) option must be passed to the linker using either -Xlinker or -Wl on the tiarmclang command line (or specified in the linker command file).

armcl Option (and alias)	tiarmclang Option
--help (-h)	-help (-h)

Display list of command-line options available.

2.3.2 Specifying the Compilation Target

The following command-line options specify the device being used and other characteristics about the hardware environment that the compiler should assume during compilation. These options determine which instruction set is used by the compiler and whether or not the compiler can assume the availability of floating-point hardware.

armcl Option (and alias)	commandline_name| Option
--silicon_version=<processor ID> (-mv<processor ID>)	-mcpu=<processor variant> -march=<architecture ID>

The armcl compiler allows you to indicate which architecture and processor variant combination to assume during compilation using the -mv option. The armcl compiler assumes a default of -mv4, but you can also choose -mv5, -mv5e, -mv6M0, -mv7M3, -mv7M4, -mv7R4, or -mv7R5.

The tiarmclang compiler provides two analogous options:

- Use -mcpu to direct the compiler to target a particular processor variant. The tiarmclang compiler requires that a valid -mcpu or -march option be specified (or it will assume cortex-m4 if neither an -mcpu or -march option is provided). You may select one of the following: cortex-m0, cortex-m3, cortex-m4, cortex-r4, or cortex-r5. Each of the available arguments to the -mcpu option imply a specific -march option argument.

- Use -march to direct the compiler to target a particular architecture. The tiarmclang compiler assumes armv7e-m (which corresponds to the cortex-m4 processor variant) if no -mcpu or -march option is specified. You may select one of the following: armv6-m, armv7-m, armv7e-m, and armv7-r.

Please note that the tiarmclang compiler does not provide support for armv4, armv5e, armv6, armv7-a. Likewise, it does not support any of the -march variants that are not mentioned above.

armcl Option	tiarmclang Option
--code_state=<16|32>	-mthumb
	-marm

The armcl compiler uses the --code_state option to select between THUMB (16) and ARM (32) mode. For targets such as Cortex-r4 and Cortex-r5that support both the

ARM and T32 (THUMB) instruction sets, the default is the Arm instruction set.

Use tiarmclang's -mthumb option to select the T32 instruction set. If a cortex-m3 or cortex-m4 argument is specified for the -mcpu option or an armv7-m or armv7e-m argument is specified for the -march option, then the tiarmclang compiler will assume that the -mthumb option is on.

armcl Option	tiarmclang Option
--float_support=<float hardware ID>	-mfloat-abi=<soft|hard>
	-mfpu=<float hardware ID>

armcl Option (and alias)	tiarmclang Option
--endian=little (-me)	-mlittle-endian (default)
--endian=big (default)	-mbig-endian

By default, tiarmclang generates little-endian code. This is the opposite of armcl, which defaults to big-endian code.

2.3.3 Specifying Source Language and Specific Language Characteristics

The following command-line options specify the language standards the compiler should expect C/C++ source code to comply with and also what assumptions to make regarding particular data types.

armcl Option	tiarmclang Option
--c89	-std=<C standard identification>
--c99	
--c11	

The tiarmclang -std option can be used to instruct the compiler to process C files in accordance with the indicated ANSI/ISO C language standard. For the tiarmclang compiler, the available -std option arguments for C are:

- `c89, c90, iso9899:1990` (ISO C 1990)

- `c99, c9x, iso9899:1999` (ISO C 1999)

- `c11, c1x, iso9899:2011` (ISO C 2011)

- `c17, c18, iso9899:2017` (ISO C 2017)

- `iso9899:199409` (ISO C 1990 with amendment 1)

- `gnu89, gnu90` (ISO C 1990 with GNU extensions)

- `gnu99`, `gnu9x` (ISO C 1999 with GNU extensions)

- `gnu11`, `gnu1x` (ISO C 2011 with GNU extensions)

- `gnu17`, `gnu18` (ISO C 2017 with GNU extensions)

If no -std option is specified when compiling a C source file, `gnu17` is assumed by default.

By default, source files with a '.c' extension are interpreted as C source files. The `-x c` option can be specified on the command-line to force a source file that does not have a '.c' extension to be interpreted as a C source file.

The tiarmclang compiler can handle a mix of C and C++ source files on a single invocation of the compiler. To interpret a file, regardless of its extension, as a C file, the `-x c` option should be specified before that file on the tiarmclang command. All source files that follow the `-x c` option are interpreted as C source files until another `-x` option is encountered on the command-line.

armcl Option	tiarmclang Option
--c++03 --c++11 --c++14 --c++17	-std=<C++ standard identification>

The tiarmclang -std option can (also) be used to instruct the compiler to process C++ source files in accordance with the indicated ANSI/ISO C++ language standard. For the tiarmclang compiler, the available supported -std option arguments for C++ are:

- `c++98`, `c++03` (IS0 C++ 1998 with amendments)

- `c++11` (ISO C++ 2011 with amendments)

- `c++14` (ISO C++ 2014 with amendments)

- `c++17` (ISO C++ 2017 with amendments)

- `gnu++98`, `gnu++03` (ISO C++ 1998 with amendments and GNU extensions)

- `gnu++11` (ISO 2011 with amendments and GNU extensions)

- `gnu++14` (ISO C++ 2014 with amendments and GNU extensions)

- `gnu++17` (ISA C++ 2017 with amendments and GNU extensions)

If no -std option is specified when compiling a C++ source file, `gnu++17` is assumed by default.

By default, source files with a '.cpp' extension are interpreted as C++ source files. The `-x c++` option can be specified on the command-line to force a source file that does not have a '.cpp' extension to be interpreted as a C++ source file.

The tiarmclang compiler can handle a mix of C and C++ source files on a single invocation of the compiler. To interpret a file, regardless of its extension, as a C++ file, the `-x c++` option should be specified before that file on the tiarmclang command. All source files that follow the `-x c++` option are interpreted as C++ source files until another `-x` option is encountered on the command-line.

armcl Option (and alias)	tiarmclang Option
--cpp_default (-fg)	-x c++
	--language=c++

The armcl compiler provides a --cpp_default option that tells the compiler to process all source files with a '.c' extension as C++ source files.

The tiarmclang compiler's `-x c++` option provides similar support, but whereas armcl's -fg option applies only to source files with '.c' extensions, the tiarmclang compiler's `-x c++` option applies to any source file that is specified after the `-x c++` option on the tiarmclang command-line up until another `-x` option is encountered.

armcl Option	tiarmclang Option
	-x <language type>
	--language=<language type>

The tiarmclang compiler's -x or --language option provides a way to indicate how source files that are specified after the option are to be interpreted. There are four valid option arguments:

- **c++** - source files specified after the `-x c++` option are interpreted as C++ source files

- **c** - source files specified after the `-x c` option are interpreted as C source files

- **assembler** - source files specified after the `-x assembler` option are interpreted as GNU-style Arm assembly language source files

- **ti-assembler** - source files specified after the `-x ti-assembler` option are interpreted as TI-style Arm assembly language source files

For example, suppose you have three source files, t1.cpp, t2.c, and t3.S. Assuming t2.c is C++ compatible and t3.S is a GNU-style Arm assembly language source file, one could then invoke the tiarmclang compiler with:

```
%> tiarmclang ... t1.cpp -x c++ t2.c -x assembler t3.S ... -o t.
→out ...
```

to ensure that t2.c is interpreted as a C++ source file so that its object is compatible with t1.o, and also to interpret t3.S as a GNU-style Arm assembly language source file so that t3.o can be generated properly.

armcl Option	tiarmclang Option
--enum_type=<packed|unpacked|int>	-fshort-enums (default)
	-fno-short-enums

The armcl compiler assumes that enum type data objects are "packed" by default. That is, the size of a given enum type is the number of bytes needed to represent the full range of values in that enum multiplied by the number of bits per byte.

Similarly, the tiarmclang compiler's will also assume that enum type data objects are packed by default.

armcl Option	tiarmclang Option
--exceptions	-fexceptions
--extern_c_can_throw	on by default

By default, the tiarmclang compiler provides no C++ exception handling support. To enable exception handling, use the compiler's -fexceptions option. See *C++ Exception Handling* for details.

Note: Support for C++ exceptions has now been added. Prior to version 3.1, C++ exceptions were not supported in tiarmclang.

By default, the commandline_name\ compiler allows extern functions to propagate exceptions.

(The C library provides no built-in support for C exception handling. There is no equivalent to the --extern_c_can_throw option for C. See *C Exception Handler Calling Convention*.)

armcl Option (and alias)	tiarmclang Option
--gen_cross_reference_listing (-px)	not supported

The armcl compiler supports the -px option, which causes the compiler to emit a .crl file, which contains a listing of where symbols are referenced and defined.

The tiarmclang compiler does not support an analogous option.

armcl Option	tiarmclang Option
--gen_preprocessor_listing	not supported

The armcl compiler supports the --gen_preprocessor_listing option, which causes the compiler to emit a listing of the pre-processing output to an .rl file.

The tiarmclang compiler does not support an analogous option.

---

armcl Option (and alias)	tiarmclang Option
--plain_char=unsigned (default)	-funsigned-char (default)
--plain_char=signed (-mc)	-fsigned-char

Both the armcl and the tiarmclang compiler assume a "plain" char type is unsigned by default. Both compilers also support an option to interpret a "plain" char type as signed. For the armcl compiler, this option is --plain_char=signed, which maps to the tiarmclang's -fsigned-char option.

armcl Option (and alias)	tiarmclang Option
--relaxed_ansi (-pr)	-std=gnu<90|99|11|17>

The armcl compiler supports GNU extensions to the C language if its -pr option is selected. To enable support of GNU extensions in the tiarmclang compiler, use one of the GNU settings (gnu90, gnu99, gnu11, gnu17) as the argument to tiarmclang's -std option.

armcl Option	tiarmclang Option
--rtti	-fno-rtti (default)
	-frtti

The armcl compiler does not allow the inclusion of Run-Time Type Information (RTTI) to be disabled. RTTI is included if a C++ application may need to refer to a class' type_info object.

The tiarmclang compiler has RTTI support disabled by default. Use the -frtti option to allow C++ RTTI to be generated.

armcl Option (and alias)	tiarmclang Option
--strict_ansi (-ps)	-std=c<90|99|11|17>

The armcl compiler provides the -ps option to allow you to disable support for GNU C extensions to the C standard. To disable support for GNU extensions in the tiarmclang compiler and approximate the behavior of armcl's -ps option, use one of the non-GNU language settings (c90, c99, c11, c17) as the argument to tiarmclang's -std option.

All tiarmclang "-std=c<XX>" C language variants define the __STRICT_ANSI__ predefined macro symbol.

You can combine "-std=c<XX>" with tiarmclang's -pedantic option, which causes warnings to be issued for any conflicts with ISO C and ISO C++. The tiarmclang compiler's -pedantic-errors option causes errors instead of warnings to be issued for such conflicts.

armcl Option	tiarmclang Option
--wchar_t=16 (default)	-fshort-wchar
--wchar_t=32	-fno-short-wchar (default)

The armcl compiler uses a default size of 16-bits for the wchar_t type, but the tiarm-clang compiler uses a default size of 32-bits for the wchar_t type. Use -fshort-wchar to set the size of the wchar_t type to 16 bits.

If your application uses wchar_t type data objects and you are trying to interlink object files that were generated with the armcl compiler with object files that were generated by the tiarmclang compiler, then you might see errors generated at link-time to indicate that there is a disagreement in the compiler generated object files with regards to build attributes associated with the assumed wchar_t type size. You may need to recompile some source files with the desired wchar_t type size option in order to resolve these disagreements.

2.3.4 Controlling Optimization

The following command-line options control optimization behavior.

armcl Option (and alias)	tiarmclang Option
--opt_level=<off|0|1|2|3> (-O<off|0|1|2|3>	-O<0|1|2|3|fast|g|s|z>
--opt_level=4 (-O4)	-flto -O<1|2|3|fast|s|z>

armcl Option (and alias)	tiarmclang Option
--opt_for_speed=<0|1|2|3|4|5> (-mf=<0|1|2|3|4|5>)	-O<z|s|3|fast>
--opt_level=4 --opt_for_speed=<0|1|2|3|4|5> (-mf=<0|1|2|3|4|5>)	-flto -O<z|s|3|fast>

armcl Option	tiarmclang Option
--sat_reassoc=off (default) --sat_reassoc=on	not supported

The armcl compiler provides a --sat_reassoc option to enable or disable reassociation of saturating arithmetic. It is off by default.

The tiarmclang compiler does not support an analogous option.

armcl Option (and alias)	tiarmclang Option
--auto_inline=<size> (-oi<size>	-finline-limit=<size>

The armcl compiler provides the --auto_inline option, which, when used in combination with --opt_level=3, allows you to specify a size threshold for automatic inlining of functions that are not explicitly declared as "inline."

The tiarmclang compiler supports an analogous option, -finline-limit, which allows you to specify a size threshold for functions that can be inlined, where <size> is the number of pseudo instructions.

The tiarmclang compiler also supports the always_inline ("__attribute__((always_inline))") and noinline ("__attribute__((noinline))") function attributes that provide a means for you to control inlining on a function-specific basis. The tiarmclang compiler's -fno-inline-functions option can be used to disable all inlining.

armcl Option (and alias)	tiarmclang Option
--disable_inlining	-fno-inline-functions

The armcl compiler provides the --disable_inlining option, which allows you prevent any inlining from being performed.

To prevent inlining with the tiarmclang compiler, use the -fno-inline-functions option.

armcl Option (and alias)	tiarmclang Option
--call_assumptions=<n> (-op<n>)	not supported

The armcl compiler provides the --call_assumptions option, which, when used in combination with --program_level_compile and --opt_level=3, allows you to provide additional information to the compiler about whether the functions defined in a given module are called from other modules and whether global variable definitions in a given module are referenced from other modules.

armcl Option (and alias)	tiarmclang Option
--gen_opt_info=<0|1|2> (-on=<0|1|2>	-fsave-optimization-record -foptimization-record-file=<filename> -Rpass=<expr> -Rpass-missed=<expr> -Rpass-analysis=<expr>

The armcl compiler provides the --gen_opt_info option, which, when used in combination with --opt_level=3, causes the compiler to emit a human-readable optimization information file. The higher the value of the argument specified, the more verbose the optimization information provided will be.

The tiarmclang compiler does not provide an option that matches the exact behavior of armcl's --gen_opt_info, but tiarmclang reports optimization information via the

following available options:

- **-fsave-optimization-record** - writes optimization remarks to a YAML file

- **-foptimization-record-file** - identifies the name of the YAML file written when using the -fsave-optimization-record option

- **-Rpass** - given a regular expression string argument to identify the optimization pass(es) that you want information about, the -Rpass option writes informative remarks to stdout during compilation about when a specified optimization pass makes a transformation

- **-Rpass-missed** - given a regular expression string argument to identify the optimization pass(es) that you want information about, the -Rpass-missed option writes informative remarks to stdout during compilation about when a specified optimization pass fails to make a transformation

- **-Rpass-analysis** - given a regular expression string argument to identify the optimization pass(es) that you want information about, the -Rpass-analysis option writes informative remarks to stdout during compilation about why a specified optimization pass does or doesn't perform a transformation

armcl Option (and alias)	tiarmclang Option
--optimizer_interlist (-os)	not supported

The armcl compiler provides the --optimizer_interlist option, which tells the compiler to keep an compiler-generated intermediate assembly source file that is annotated with interlisted comments corresponding C/C++ source code optimizations to the assembly code generated by the compiler.

The tiarmclang compiler does not provide an analogous option. However, you can use tiarmclang's -Rpass, -Rpass-missed, and -Rpass-analysis options to gain more insight into which optimizations were performed and potential optimizations that were ruled out during compilation.

armcl Option (and alias)	tiarmclang Option
--program_level_compile (-pm)	-flto

The armcl compiler's --program_level_compile option combines source files into a single compilation unit to enable the compiler's program-level optimizations.

The tiarmclang -flto option enables inter-module optimizations via link-time optimization. The tiarmclang -flto option can be combined with the tiarmclang -O<1|2|3|fast|s|z> optimization level option to instruct the compiler whether to prioritize improving performance over reducing code size or vice versa.

armcl Option (and alias)	tiarmclang Option
--aliased_variables (-ma)	not supported

The armcl compiler's -aliased_variables option instructs the compiler to assume that called functions are capable of creating hidden aliases. As a result, the compiler must assume worst-case aliasing. For example, the optimizer cannot assume that it knows the value stored in a local object if that local object might be accessed via a separate pointer.

The tiarmclang compiler does not provide an analogous option. However, tiarmclang's -fstrict-aliasing and -fno-strict-aliasing options can be used to enable or disable optimizations based on type based alias analysis, but they don't allow the compiler to violate the aliasing rules of C. Some aliasing behavior can also be controlled via tiarmclang's optimization options.

2.3.5 Managing Floating Point Support

Assuming that a floating-point ABI has been selected with the -mfloat-abi option and, if floating-point hardware is available, it has been made known to the compiler via the -mfpu option, then the following command-line options can be used to further refine the compiler's assumptions about what floating-point characteristics are enabled.

armcl Option	tiarmclang Option
--float_operations_allowed=<all|none|32|64>	not supported

The armcl compiler supports a --float_operations_allowed option, which allows you to indicate to the compiler the maximum floating-point precision that can be assumed for a floating-point type data object. For example, you can specify '--float_operations_allowed=32', causing the armcl compiler to flag an error if an attempt is made to use a floating-point type whose size is greater than 32-bits.

The tiarmclang compiler does not provide a mechanism to restrict the use of floating-point types by type size. The tiarmclang compiler assumes that all legal floating-point types are supported. This matches the armcl compiler's default behavior (e.g. --float_operations_allowed=all).

armcl Option	tiarmclang Option
--fp_mode=<relaxed|strict>	-ffp-model=<precise|strict|fast>
	-ffast-math
	-fno-fast-math
	-ffp_contract=[on|off]
	-frounding-math
	-fno-rounding-math
	-ffp-exception-behavior=<ignore|maytrap|strict>

By default, the armcl compiler supports 'strict' conformance to the IEEE-754 floating-point standard, but you can also use the '--fp_mode=relaxed' option to allow the

compiler to be more aggressive about using floating-point hardware instructions, al-
low floating-point arithmetic reassociation, and aggressively convert double-precision
floating-point terms in an expression to single-precision when the result type is single-
precision.

The tiarmclang compiler provides an '-ffp-model' option that allows you to instruct
the compiler to assume a general set of rules for generating code that implements
floating-point math. Each of the available arguments to the '-ffp-model' option will
effect the settings for other, single-purpose, floating-point options.

The available arguments to the -ffp-model option are:

- **precise** - If no '-ffp-model' option is explicitly specified on the tiarmclang
  command-line, then the compiler will assume the 'precise' floating-point model
  by default.

  '-ffp-model=precise' is the recommended tiarmclang compiler option to be used
  in place of armcl's '--fp_mode=strict' option.

  When the 'precise' floating-point model is in effect, all optimizations that are
  not value-safe on floating-point data are disabled. However, if the indicated Arm
  processor's floating-point unit (FPU) supports a fused multiply and add (FMA)
  instruction, then the compiler will assume that any floating-point contraction op-
  timizations are safe (*-ffp-contract=on*).

- **strict** - Specifying 'strict' as the argument to the -ffp-model option enables
  floating-point rounding (*-frounding-math*) and sets the floating-point exception
  behavior option to strict (*-ffp-exception-behavior=strict*). However, floating-
  point contraction optimizations are disabled (*-ffp-contract=off*). Also, no 'fast-
  math' optimizations are enabled (*-fno-fast-math*).

- **fast** - Specifying 'fast' as the argument to the -ffp_model option will enable
  all 'fast math' optimizations (*-ffast-math*) and it will enable more aggressive
  floating-point contraction optimizations (*-fp-contract=fast*).

  The tiarmclang '-ffp-model=fast' option is the most functionally similar to the
  armcl '--fp_mode=relaxed' option among the available arguments to the '-ffp-
  model' option.

For more detailed information about the separate, single purpose floating-point options
mentioned here, please see the *Floating-Point Arithmetic* section of the *Optimization
Options* chapter.

armcl Option	tiarmclang Option
--fp_reassoc	not supported

The tiarmclang compiler does not provide an option that controls whether reassocia-
tion is allowed in floating-point operations. Instead, the tiarmclang's -ffp-mode=std

option can be used to disallow reassociation (the default), and tiarmclang's -ffp-mode=fast option can be used to allow reassociation.

2.3.6 Controlling the Runtime Model

The following options can be used to dictate compiler behavior with regards to how code and data is organized in compiler generated object files, generating code to monitor stack usage at run-time, constraints on pointer alignment when generating code to access memory, and enabling a link-time dead-code removal optimization.

armcl Option	tiarmclang Option
--common=on (default)	-fcommon (default)
--common=off	-fno-common

The tiarmclang compiler's -fcommon and -fno-common options mirror the armcl's --common option. The tiarmclang compiler's -fcommon option indicates to the compiler that variables without initializers will get common linkage. This can be disabled with tiarmclang's -fno-common option.

By default, the armcl --common option is on. Likewise, the default behavior for tiarmclang is -fcommon.

armcl Option	tiarmclang Option
--embedded_constants=on (default)	not supported
--embedded_constants=off	

The armcl compiler embeds constant definitions in functions by default. One consequence of this is that the compiler may generate accesses to data that is defined in code sections that will conflict with attempts to put the code section in an execute-only region of memory.

Whether or not the tiarmclang compiler will embed constant definitions in code may vary depending on the optimization level specified on the tiarmclang command-line. When optimizing for code size in preference to performance with the -Oz or -Oz option, the tiarmclang compiler is more likely to generate embedded constants in code.

armcl Option	tiarmclang Option
--gen_func_subsections=on (default)	-ffunction-sections (default)
--gen_func_subsections=off	-fno-function-sections

The armcl compiler's --gen_func_subsections option, which is on by default, places each function definition into a separate subsection.

The analogous tiarmclang option is -ffunction-sections, also on by default, which generates each function definition into its own section. You can further control sec-

tion placement with the section function attribute, __attribute__((section("<section name>"))), which can be added to the definition of a function to place it in a particular section.

Placing every function definition in its own section will enable the linker to remove unreferenced functions from the linked output file. As this can prove to have a significant impact on reducing the code size of a program without any impact on run-time performance, it is recommended that the default setting of -ffunction-sections be left intact.

armcl Option	tiarmclang Option
--gen_data_subsections=on (default)	-fdata-sections (default)
--gen_data_subsections=off	-fno-data-sections

The armcl compiler's --gen_data_subsections option, which is on by default, places aggregate data (arrays, structs, and unions) into separate subsections.

The analogous tiarmclang option is -fdata-sections, also on by default, which generates a separate section for each variable, including non-aggregate variables. You can further control section placement with the section variable attribute, __attribute__((section("<section name>"))), which can be added to the definition of a particular data object to place it in a particular section.

Placing every variable definition in its own section will enable the linker to remove unreferenced data objects from the linked output file. As this can prove to have a significant impact on reducing the amount of space allocated for variables in a program without impacting run-time performance, it is recommended that the default setting of -fdata-sections be left intact.

armcl Option	tiarmclang Option
--global_register=<r5|r6|r9>	not supported

The armcl compiler allows you to prohibit the compiler from using a register indicated by the --global_register option.

The tiarmclang compiler does not provide an analogous option.

armcl Option	tiarmclang Option
--printf_support=<nofloat|full|minimal>	automatic

The armcl's --printf_support option allows you to limit the printf and scanf support required in the standard C runtime library. For example, if the you know that an application will never pass a floating-point value to be formatted by a printf- or scanf-family function, then using the --printf_support=nofloat option instructs the compiler to use a customized version of the printf- or scanf-family C/C++ runtime function that

does not support floating-point and is therefore much smaller than a printf- or scanf-family function that provides full support for floating-point.

The tiarmclang compiler tools embed metadata in compiler-generated object code to help the linker automatically determine whether a smaller implementation of the printf support function can be used in the link step of an application build.

See *Printf Support Optimization (no option)* for additional information about this behavior.

armcl Option	tiarmclang Option
--ramfunc=off (default)	not supported
--ramfunc=on	

The armcl compiler's --ramfunc option, when it is on (default behavior is off), instructs the compiler to generate the code for the functions defined in a compilation unit into a special ".TI.ramfunc" section, which can then be placed in RAM memory at link time.

The tiarmclang compiler does not provide an analogous option. However, the code generated for a function can be directed into a specific section using a section function attribute, such as __attribute__((section("<section name>"))), attached to the function definition in the C/C++ source file where it is defined.

armcl Option (and alias)	tiarmclang Option
--stack_overflow_check (-mo)	not supported

The armcl compiler's --stack_overflow_check option enables dynamic stack overflow checking.

The tiarmclang compiler does not provide an analogous option. However, the tiarmclang compiler does provide support for dynamically detecting when a function may be writing past the bounds of the local stack space that has been allocated for it. This support is enabled by the -fstack-protector, -fstack-protector-strong, or the -fstack-protector-all compiler option. For more information about detecting stack smashing at run time, please see the *Instrumentation Options* section of the *tiarmclang Compiler User Manual*.

To debug issues related to the stack size, we recommend using the CCS Stack Usage view to see the static stack usage of each function in the application. See Stack Usage View in CCS for more information. Using the Stack Usage View requires that source code be built with *debug enabled*. This feature relies on the *–call_graph* capability provided by the *tiarmofd - Object File Display Utility*.

armcl Option	tiarmclang Option
--unaligned_access=on	-munaligned-access (default)
--unaligned_access=off	-mno-unaligned-access

When the armcl compiler's --unaligned_access option is on (the default behavior for all Cortex processor variants), the compiler assumes that it is safe to generate load and store instructions that access data that falls on an unaligned boundary (i.e. 32-bit data on a 16-bit boundary).

Since the tiarmclang compiler only supports generating code for Cortex processor variants, the default behavior for the tiarmclang compiler is -munaligned-access. The tiarmclang compiler's -mno-unaligned-access option can be specified to disable unaligned accesses.

armcl Option		tiarmclang Option
--generate_dead_funcs_list=<filename> use_dead_funcs_list=<filename>	--	not supported

The armcl tools package provides the --generate_dead_funcs_list and --use_dead_funcs_list options, which allow you to first generate a list of dead functions at link-time, and then use that list as input to a subsequent compile to remove the definition of the dead function from the compilation unit.

The tiarmclang compiler does not provide an analogous pair of options. However, the tiarmclang compiler generates the definition of a given function into its own subsection by default, which enables the linker to remove the function definition from the application if it is not referenced.

2.3.7 Defining the Include File Directory Search Path

In C/C++, an #include preprocessor directive tells the compiler to read C/C++ source statements from another file. When specifying the file, you can enclose the filename in double quotes or in angle brackets. The filename can be a complete pathname, a relative pathname, or a filename with no path information.

When searching for the specified include file, the compiler will incorporate the notion of an include file directory search path into the search.

armcl Include File Directory Search Path

The armcl compiler supports the notion of an include file search path directory. It can make use of environment variables to help extend the include file directory search path.

Using the armcl compiler, the include file directory search path is defined in one of two ways:

- If you enclose the file specification in double quotes (" "), the compiler searches for the file in the following directories in this order:

    1. The directory of the file that contains the #include preprocessor directive

2. Directories named in one or more --include_path options in the order in which the options are specified in the compiler invocation

3. Directories listed in the TI_ARM_C_DIR environment variable definition

- If you enclose the file specification in angle brackets (< >), the compiler searches for the file in the following directories in this order:

   1. Directories named in one or more --include_file options in the order in which the options are specified in the compiler invocation

   2. Directories listed in the TI_ARM_C_DIR environment variable definition

By default, the armcl compiler begins with an empty include file directory search path. The recommended environment variable that serves as a sort of baseline definition of the include file directory search path is TI_ARM_C_DIR, but the armcl compiler will also honor legacy environment variables TMS470_C_DIR and C_DIR. In addition, the armcl compiler allows the TI_ARM_C_OPTION environment variable to define a set of compiler options to be used as if they were on the command line.

tiarmclang Include File Directory Search Path

The tiarmclang compiler also has a notion of an include file search path directory and it can also make use of environment variables to help extend the include file directory search path. However, the tiarmclang's definition of the include file directory search path incorporates a builtin clang include file directory and standard system directories that contain C and C++ runtime header files.

When using the tiarmclang compiler to process an #include preprocessor directive, the include file directory search path is defined in one of two ways:

- If you enclose the file specification in double quotes (" "), the compiler searches for the file in the following directories in this order:

   1. The directory of the file that contains the #include preprocessor directive

   2. Directories named in one or more -I options in the order in which the options are specified in the compiler invocation

   3. Directories listed in an applicable environment variable definition

   4. C++ runtime header file directory (if compiling a C++ source file)

   5. Compiler builtin include directory

   6. Standard system include directories

- If you enclose the file specification in angle brackets (< >), the compiler searches for the file in the following directories in this order:

   1. Directories named in one or more -I options in the order in which the options are specified in the compiler invocation

2. Directories listed in an applicable environment variable definition

3. C++ runtime header file directory (if compiling a C++ source file)

4. Compiler builtin include directory

5. Standard system include directories

By default, the tiarmclang compiler populates the include file directory search path with the builtin include directory and the standard system include directories that are set up when the tiarmclang compiler tools are installed. For example, the following include file directories are installed with the tiarmclang 1.0.0-alpha.1 compiler tools:

- C++ runtime header file directory: <install area>/lib/generic/include/c++/v1

- Compiler builtin include directory: <install area>/lib/clang/10.0.0/include

- Standard system include directory: <install area>/lib/generic/include/c

The following command-line options are available in the tiarmclang compiler to manage whether or not these installed include directories are incorporated into a given compilation:

- **-nostdinc** - do not incorporate the C++ runtime header file directory, the compiler builtin include directory, or the standard system include directory in the default definition of the include file directory search path

- **-nostdlibinc** - do not incorporate the C++ runtime header file directory or the standard system include directory into the include file directory search path, but do incorporate the compiler's builtin include directory

- **-nobuiltininc** - do not incorporate the compiler's builtin include directory into the include file directory search path, but do incorporate the C++ runtime header file directory and the standard system include directory

You may also control the include file directory search path through definitions of the CPATH, C_INCLUDE_PATH (C source files only), or CPLUS_INCLUDE_PATH (C++ source files only) environment variables.

The TI_ARM_C_DIR, C_DIR, and TI_ARM_C_OPTION environment variables are not supported by the tiarmclang compiler, and should be migrated as appropriate.

Adding to the Include File Directory Search Paths with Command-Line Options

As indicated above, the include file directory search path can be controlled using the compiler command-line.

armcl Option (and alias)	tiarmclang Option
--include_path=<dir> (-I=<dir>) (-i=<dir>)	-I

When using the armcl compiler, the --include_path option allows a user to specify a semi-colon separated list of one or more directory paths in which the compiler will search for an include file in accordance with the rules indicated in the above "armcl Include File Directory Search Path" section.

Likewise, using the tiarmclang compiler, the -I option allows a user to specify a semi-colon separated list of one or more directory paths in which the compiler will search for an include file in accordance with the rules indicated in the above "tiarmclang Include File Directory Search Path" section.

Also see migration of the --preinclude option in *Specifying Source Files and File Extensions*.

2.3.8 Defining the Object/Library File Directory Search Path

At link time, a collection of object files, that either have been freshly generated by the compiler or reside in an existing object library, are combined together to form a linked output file. In typical cases, the linked output file is a static executable, but the linker is also capable of generating a partially linked output file in which relocation entries are preserved and which may be combined with other object files in a subsequent link.

Object/Library File Directory Search Path

Within a given linker invocation, an object or library file can be specified explicitly or with a '-l' prefix. Such a specification can be indicated on a command-line invocation of the linker or within a linker command file that is incorporated into a link. If a '-l' prefix is used in an object or library file specification, then the linker will locate the specified file or library using an object/library file directory search path.

The concept is similar to the notion of an include file directory search path in the case where the include file is enclosed in angle brackets (< >).

Specifically, when a '-l' prefix is indicated in an object or library file specification, the linker will search for the specified file in the following locations in this order:

1. Directories named in -L compiler options in the order in which the options are specified in the tiarmclang invocation.

2. Standard system lib directory.

3. Directories named in -Xlinker --search_path options in the order in which the options are specified in the tiarmclang invocation.

4. Directories listed in the TI_ARM_C_DIR environment variable definition.

By default, the armcl compiler begins with an empty object/library file directory search path. The recommended environment variable that serves as a sort of baseline definition of the object/library file directory search path is TI_ARM_C_DIR, but the armcl compiler also honors

---

legacy environment variables TMS470_C_DIR and C_DIR. In addition, the armcl compiler allows the TI_ARM_C_OPTION environment variable to define a set of compiler options to be used as if they were on the command line.

Unlike the armcl compiler, the tiarmclang compiler populates the object/library file directory search path with the standard system lib directory, <install area>/lib/generic, that is set up when the tiarmclang compiler tools are installed.

You may also control the object/library file directory search path using the TI_ARM_C_DIR environment variable when the linker is invoked with either the armcl or tiarmclang compiler. The linker also honors legacy environment variables TMS470_C_DIR and C_DIR. Use of the TI_ARM_C_OPTION environment variable should be migrated to command line options and environment variables that are supported by the tiarmclang compiler.

Please note that while the tiarmlnk linker that is provided with the tiarmclang compiler tools is identical to the armlnk linker that is provided with the armcl compiler tools with respect to these environment variables, the two linker executables are not functionally equivalent in other ways. Some of the significant differences between the executables are discussed in the *Main Differences Between armcl and tiarmclang* chapter.

Adding to the Object/Library File Directory Search Path with Compiler Command-Line Options

As indicated above, the definition of the object/library file directory search path can be controlled using the compiler command-line.

armcl Option (and alias)	tiarmclang Option
--include_path=<dir list> (-I=<dir list>) (-i=<dir list>)	-L <dir>

When using the armcl compiler, the --include_path option allows a user to specify a semi-colon separated list of one or more directory paths that is converted into a --search_path linker option when the armcl compiler is made to invoke the linker. The linker will then follow the rules indicated in the above "Object/Library File Directory Search Path" section when searching for an object or library file that is specified as a linker option on the armcl compiler command-line or within a linker command file using a '-l' prefix.

The tiarmclang compiler's -L option is functionally equivalent to the armcl compiler's --include_path option except that the -L option allows only a single directory path to be specified for each -L option on the tiarmclang command-line. However, you can specify more than one -L option on the tiarmclang command-line to add additional directories to the object/library file directory search path. When multiple -L options are specified, they will be translated into multiple --search_path linker options in the order in which they are specified.

You can also pass explicit --search_path options directly to the linker from the compiler command-line.

armcl Option	tiarmclang Option
-z . . . --search_path=<dir list>	-Xlinker --search_path=<dir> -Wl,–search_path,<dir>

When the linker is invoked from the armcl compiler command-line, all options that are specified after the -z (or --run_linker) option are passed directly to the linker. In this manner, a user can add one or more directories to the object/library file directory search path.

When invoking the linker from the tiarmclang compiler command-line, options that are intended as input to the linker invocation should be preceded by -Xlinker. You may also pass an option to the linker using the -Wl, option mechanism. For example, if you have a directory called "myobj" under the work directory that you are invoking the compiler from, you can append the ./myobj sub-directory to the object/ library file directory search path with the following option:

-Wl,–search_path,./myobj

As is the case with the tiarmclang compiler's -L option, only one directory path may be specified to the --search_path option when passed to the linker from the tiarmclang compiler command-line.

2.3.9 Specifying Temp Directories

The armcl compiler provides several options that allow you to control where a temporary file is written during a given compilation. However, the tiarmclang compiler does not provide an analogous capability for controlling the location of the temporary files that are generated during a compilation.

In most, if not all, cases, the tiarmclang compiler places temporary files in the current working directory (that is, whichever directory the tiarmclang executable was invoked from). Ordinarily, temporary files are removed when they are no longer needed by the tiarmclang compiler. However, like the armcl compiler, the tiarmclang compiler does support command-line options that keep one or more of the temporary files that are generated during a given compilation.

armcl Option (and alias)	tiarmclang Option
--abs_directory=<dir> (-fb)	not supported

The armcl compiler supports an absolute listing capability (-abs), which is not provided in the tiarmclang toolset. Thus the tiarmclang compiler does not provide an option to control where an absolute listing file would be written.

armcl Option (and alias)	tiarmclang Option
--asm_directory=<dir> (-fs)	-S
	-save-temps

The armcl compiler allows you to indicate where a temporary compiler-generated assembly file should be written, but the tiarmclang compiler does not provide this capability. The tiarmclang compiler's -S option instructs the compiler to write the compiler-generated assembly file to the current working directory and then stop the compiler before actually assembling the file into an object file.

The tiarmclang compiler's -save-temps option keeps all temporary files generated during compilation and linking without halting either the compiler or the linker. The typical temporary files that are generated during compilation and linking include: an intermediate file (.i extension), a bitcode intermediate file (non-readable with .bc extension), a compiler-generated assembly file (.s extension), and an object file generated from the tiarmclang assembler (.o extension).

armcl Option (and alias)	tiarmclang Option
--list_directory=<dir> (-ff)	not supported

The armcl tools support the capability to generate an assembly listing file, which displays the encoded object code alongside the assembly language source that was either generated by the compiler or was provided in an assembly language source file. The armcl's --list_directory option allows you to indicate where to write the assembly listing file during the compilation.

The tiarmclang compiler does not provide the analogous capability to generate an assembly listing file. Thus, there is no need for an option to direct where an assembly listing file is to be written.

armcl Option (and alias)	tiarmclang Option
--obj_directory=<dir> (-fr)	-c
	-save-temps

The armcl compiler allows you to indicate where a temporary assembler-generated object file should be written, but the tiarmclang compiler does not provide this capability. The tiarmclang compiler's -c option instructs the compiler to write the compiler-generated object file to the current working directory and then stop the compiler before actually linking the file into an application.

The tiarmclang compiler's -save-temps option keeps all temporary files generated during compilation and linking without halting either the compiler or the linker. The typical temporary files that are generated during compilation and linking include: an intermediate file (.i extension), a bitcode intermediate file (non-readable with .bc extension), a compiler-generated assembly file (.s extension), and an object file generated

from the tiarmclang assembler (.o extension).

armcl Option (and alias)	tiarmclang Option
--output_file=<file> (-o)	-o <file>

Both the armcl and tiarmclang compilers support a -o option, which allows you to specify the name and location of the linked output file.

armcl Option	tiarmclang Option
--pp_directory=<dir>	-E

The armcl compiler supports generating a pre-processor file that is emitted after the parser portion of the compiler completes processing of all pre-processing directives (using the -ppo option, for example). The armcl's --pp_directory option allows you to specify where to write the pre-processor file (.pp extension) during a given compilation.

Whereas the armcl compiler can be made to generate a pre-processor file with a .pp extension, the tiarmclang compiler supports the -E option, which writes the pre-processor output to stdout. You can direct tiarmclang's pre-processor output to a file using the appropriate UNIX or MS-DOS "pipe" command notation.

armcl Option (and alias)	tiarmclang Option
--temp_directory=<dir> (-ft)	not supported

The armcl compiler provides the --temp_directory option to allow you to specify an alternate directory (from the current work directory) where temporary files are to be written.

The tiarmclang compiler writes temporary files to the current working directory (where tiarmclang is invoked from). Normally, temporary files are automatically removed during the compilation process when the compiler no longer needs a given temporary file, but you can keep all of the temporary files generated during a given compilation by specifying tiarmclang's -save-temps option. The tiarmclang compiler does not provide an option to write temporary files to an alternate directory.

2.3.10 Specifying Source Files and File Extensions

The following command-line options specify source file type treatment and extensions.

armcl Option (and alias)	tiarmclang Option
--asm_file=<file> (-fa=<file>)	-x assembler
	-x assembler-with-cpp
	-x ti-asm

The armcl compiler provides the --asm_file option to identify a specific file as an assembly source file regardless of its extension.

The tiarmclang compiler processes source files specified on the command line after the '-x <type>' option as source files of an indicated type. For assembly source files, there are 3 different <type> arguments that can be specified for the -x option:

- **assembler** - assume that source files that follow the '-x assembler' option contain GNU-style assembly source.

- **assembler-with-cpp** - assume that source files that follow the '-x assembler-with-cpp' option contain GNU-style assembly source that contains pre-processing directives that must be processed before the GNU-style assembler is invoked on the GNU-style assembly source.

- **ti-asm** - assume that source files that follow the '-x ti-asm' option contain legacy TI-style assembly source. Instead of calling the default GNU-style assembler to process the source file, the legacy TI-style assembler, tiarmasm, will be used to process the source file. For more details on using tiarmclang to invoke the legacy TI-style assembler, please see "Invoking the Legacy TI- Style Assembler from tiarmclang".

Note that like other instances of the tiarmclang compiler's -x option, the source file type indicated by a given -x option will determine how source files which follow that -x option are interpreted until another -x option that specifies a different source file type is encountered.

armcl Option (and alias)	tiarmclang Option
--c_file=<file> (-fc=<file>)	-x c

The armcl compiler provides the --c_file option to identify a specific file as a C source file regardless of its extension.

The tiarmclang compiler processes source files specified on the command line after the '-x c' option as C source files. Like other instances of the tiarmclang compiler's -x option, the source file type indicated by a given -x option will determine how source files which follow that -x option are interpreted until another -x option that specifies a different source file type is encountered.

armcl Option (and alias)	tiarmclang Option
--cpp_default (-fg)	-x c++
--cpp_file=<file> (-fp=<file>)	

The armcl compiler provides the --cpp_default file option to indicate that C files (with '.c' file extension) should be interpreted as C++ source files. The armcl compiler also provides the --cpp_file option to identify a specific file as a C++ source file regardless of its extension.

The tiarmclang compiler processes source files specified on the command line after the '-x c++' option as C source files. Like other instances of the tiarmclang compiler's -x option, the source file type indicated by a given -x option will determine how source files which follow that -x option are interpreted until another -x option that specifies a different source file type is encountered.

armcl Option (and alias)	tiarmclang Option
--obj_file=<file> (-fo=<file>)	not supported

The armcl compiler provides the --obj_file option to identify a specific file as an object file regardless of its extension.

The tiarmclang compiler does not provide an explicit option to tell the compiler to interpret a given file as an object file, regardless of extension. However, provided there are no -x options preceding an object file specification on the tiarmclang command-line, the tiarmclang compiler will detect that the specified file contains object code and pass the file along to be included in the link step.

Alternatively, if there are -x options on the tiarmclang command- line that would interfere with the proper interpretation of an object files specification, you may precede the object file specification with a -Xlinker option to indicate that the object file is intended as input to the linker.

armcl Option	tiarmclang Option
--preinclude=<file>	-include <file>

The armcl compiler provides the --preinclude option to include a source file at the beginning of compilation.

The tiarmclang compiler's -include option provides the same functionality.

armcl Option (and alias)	tiarmclang Option
--asm_extension=<ext> (-ea=<ext>)	not supported

The armcl compiler provides the --asm_extension option to indicate that files with the specified extension (<ext>) should be interpreted as assembly source files. In addition, assembly files that are generated by the compiler will have the specified extension.

The tiarmclang compiler does not provide support for changing default file extensions. Files with an .s (lower-case) extension are treated as GNU-style assembly source. The .S (upper-case) extension indicates that a GNU-style assembly source file requires preprocessing.

armcl Option (and alias)	tiarmclang Option
--c_extension=<ext> (-ec=<ext>)	not supported

The armcl compiler provides the --c_extension option to indicate that files with the specified extension (<ext>) should be interpreted as C source files.

The tiarmclang compiler does not provide support for changing default file extensions. Files with the .c extension are compiled as C.

armcl Option (and alias)	tiarmclang Option
--cpp_extension=<ext> (-ep=<ext>)	not supported

The armcl compiler provides the --cpp_extension option to indicate that files with the specified extension (<ext>) should be interpreted as C++ source files.

The tiarmclang compiler does not provide support for changing default file extensions. Files with the .cpp, .cxx, .c+, .cc, and .CC extensions are compiled as C++. You may also use the -x c++ option to indicate that any source file that follows the -x c++ option should be interpreted as a C++ source file (until another -x option is encountered on the tiarmclang command-line).

armcl Option (and alias)	tiarmclang Option
--listing_extension=<ext> (-es=<ext>)	not supported

The armcl compiler provides the --listing_extension option to indicate that assembly listing files that are generated by the compiler will have the specified extension (<ext>).

The tiarmclang compiler does not provide support for generating assembly listing files. Instead you may choose to use one of the available binary utilities to display the content of an object file.

armcl Option (and alias)	tiarmclang Option
--obj_extension=<ext> (-eo=<ext>)	not supported

The armcl compiler provides the --obj_extension option to indicate that files with the specified extension (<ext>) should be interpreted as object files. In addition, object files that are generated by the compiler will have the specified extension.

The tiarmclang compiler does not provide support for changing default file extensions. The tiarmclang compiler will attach a '.o' extension to regular object files. The -o option can be used to specify the name of the linked output file.

There are some exceptions to this. For example, the tiarmclang compiler will interpret '.c' file extensions as C source files, '.cpp' file extensions as C++ source files, and '.s' file extensions as GNU-style assembly source files.

2.3.11 Preprocessor Options

The following command-line options control the preprocessor.

armcl Option (and alias)	tiarmclang Option
--define=<name>[=<value>] (-D=<name>[=<value>])	-D<name>[=<value>]

The armcl compiler provides the --define option to predefine a preprocessor macro. The macro symbol can be given a value if an assignment to the optional value argument is included.

The tiarmclang compiler's -D option provides the same functionality. The tiarmclang compiler also provides the ability to append a macro parameter list in order to define function-style macros.

For both compilers, if the assignment to value argument is omitted, then the predefined symbol's value is set to 1.

armcl Option (and alias)	tiarmclang Option
--undefine=<name> (-U=<name>)	-U<name>

The armcl compiler provides the --undefine option to undefine a preprocessor macro.

The tiarmclang compiler's -U option provides the same functionality.

armcl Option (and alias)	tiarmclang Option
--preproc_dependency[=<file>] (-ppd[=<file>])	-M

The armcl compiler provides the --preproc_dependency option, which produces a list of dependency rules for use by a make utility. This list includes both system and user header files. When this option is used, the compiler executes only the preprocessor step of the compilation. Unless a filename is specified, the preprocessed output is sent to a file with an extension of .pp.

The tiarmclang compiler's -M option provides the same functionality. However, the preprocessed output is sent to stdout by default.

armcl Option (and alias)	tiarmclang Option
--preproc_includes[=<file>] (-ppi[=<file>])	-MM

The armcl compiler provides the --preproc_includes option, which produces a list of dependency rules for use by a make utility. This list includes only user header files. When this option is used, the compiler executes only the preprocessor step of the compilation. Unless a filename is specified, the preprocessed output is sent to a file with an extension of .pp.

The tiarmclang compiler's -MM option provides the same functionality. However, the preprocessed output is sent to stdout by default.

armcl Option (and alias)	tiarmclang Option
--preproc_only (-ppo)	-E

The armcl compiler provides the --preproc_only option, which causes the compiler to execute only the preprocessor step of the compilation. The preprocessed output is sent to a file with an extension of .pp.

The tiarmclang compiler's -E option provides the same functionality. However, the preprocessed output is sent to stdout by default.

armcl Option (and alias)	tiarmclang Option
--preproc_macros{=<file>] (-ppm{=<file>])	-E -dM

The armcl compiler provides the --preproc_macros option, which produces a list of predefined and user-defined macros. When this option is used, the compiler executes only the preprocessor step of the compilation. Unless a filename is specified, the preprocessed output is sent to a file with an extension of .pp.

The tiarmclang compiler's -E and -dM options used together provide the same functionality. However, the preprocessed output is sent to stdout by default.

armcl Option (and alias)	tiarmclang Option
--preproc_with_comment (-ppc)	-E -C

The armcl compiler provides the --preproc_with_comment option, which causes the compiler to execute only the preprocessor step of the compilation. It keeps the comments instead of discarding them as is done with the --preproc_only option. The preprocessed output is sent to a file with an extension of .pp.

The tiarmclang compiler's -E and -C options used together provide the same functionality. However, output is sent to stdout by default.

armcl Option (and alias)	tiarmclang Option
--preproc_with_compile (-ppa)	not supported

The armcl compiler provides the --preproc_with_compile option, which causes the compiler to continue after executing one of the --preproc_* options that produce preprocessor output files.

The tiarmclang compiler does not support continuing compilation after generating preprocessor output with the -E option.

armcl Option (and alias)	tiarmclang Option
--preproc_with_line (-ppl)	-E

The armcl compiler provides the --preproc_with_line option, which causes the compiler to execute only the preprocessor step of the compilation. It adds line-control information (#line directives) to the output. Output is sent to a file with an extension of .pp.

The tiarmclang compiler's -E option stops the compiler after the preprocessing stage. The preprocessed source code is emitted to stdout containing line-control information. The tiarmclang does not provide an option to disable the output of line-control information in the preprocessed output.

2.3.12 Controlling Entry/Exit Hooks

The tiarmclang compiler tools do not support entry/exit hooks in the same way as the armcl compiler. However, tiarmclang does support an -finstrument-functions option, which inserts calls to __cyg_profile_func_enter() and __cyg_profile_func_exit() at the entry and exit of each function. This feature has not been adequately tested and may be problematic for C++ applications.

armcl Option	tiarmclang Option
--entry_hook=<func>	-finstrument_functions

The armcl compiler's *--entry_hook* option allows you to specify the name of a function to be called on entry.

The tiarmclang compiler provides the capability to instrument functions via its -*finstrument-functions* option, inserting a call to *__cyg_profile_func_enter()* at the entry of each function defined in a compilation unit. While the tiarmclang compiler does not provide an option to allow you to name the entry function, the definition of *__cyg_profile_func_enter()* can be customized to serve as an entry hook function.

The tiarmclang compiler tools package does not provide an implementation of the *__cyg_profile_func_enter()*. If you specify the *-finstrument_functions* option on the command-line, you will need to supply a definition of the *__cyg_profile_func_enter()* function to be linked with your application.

See *Function Entry/Exit Hook Options* for more information about tiarmclang's -*finstrument-functions* option.

armcl Option	tiarmclang Option
--entry_parm=<none|name|address>	not supported

When using entry hook functions with the armcl compiler, you can pass the name or the address of the calling function as an argument to the entry hook function.

The tiarmclang compiler does not support an analogous capability.

armcl Option	tiarmclang Option
--exit_hook=<func>	-finstrument_functions

The armcl compiler's *--exit_hook* option allows you to specify the name of a function to be called before exiting.

The tiarmclang compiler provides the capability to instrument functions via its *-finstrument-functions* option, inserting a call to *__cyg_profile_func_exit()* prior to exiting from each function defined in a compilation unit. While the tiarmclang compiler does not provide an option to allow you to name the entry function, the definition of *__cyg_profile_func_exit()* can be customized.

The tiarmclang compiler tools package does not provide an implementation of the *__cyg_profile_func_exit()*. If you specify the *-finstrument_functions* option on the command-line, you will need to supply a definition of the *__cyg_profile_func_exit()* function to be linked with your application.

See *Function Entry/Exit Hook Options* for more information about tiarmclang's *-finstrument-functions* option.

armcl Option	tiarmclang Option
--exit_parm=<none|name|address>	not supported

When using exit hook functions with the armcl compiler, you can pass the name or the address of the calling function as an argument to the exit hook function.

The tiarmclang compiler does not support an analogous capability.

armcl Option	tiarmclang Option
--remove_hooks_when_inlining	not supported

If the armcl compiler inlines a function, the *--remove_hooks_when_inlining* option can be used to remove entry/exit hook function calls from the inlined function.

The tiarmclang compiler does not provide an analogous option.

2.3.13 Controlling DWARF Debug Information

The following command-line options control what form of debug information, if any, is generated by the compiler and is propagated to a linked executable file.

armcl Option (and alias)	tiarmclang Option
--symdebug:dwarf (-g)	-g

The -g option causes both the armcl and tiarmclang compilers generate debug information for a compilation unit in accordance with the DWARF standard. When the -g option is specified, the armcl compiler generates DWARF version 3 debug information by default. The tiarmclang compiler also generates DWARF version 3 debug information by default when the -g option is specified on the compiler command-line.

However, please note that in the tiarmclang 1.0.0+sts compiler tools, the use of -gdwarf-4 may introduce debug information discontinuities with the CCS debugger. It is recommended that until these issues are addressed that you should use -gdwarf-3 for debugging.

armcl Option	tiarmclang Option
--symdebug:dwarf_version=<version>	-gdwarf-<version> -gdwarf-3

The armcl compiler provides the --symdebug:dwarf_version option to allow you to select what version of DWARF debug information will be generated by the compiler. The tiarmclang compiler currently only generates DWARF version 3 debug information. Support for generating DWARF version 4 and version 5 will be added in a future release of the tiarmclang compiler tools.

The tiarmclang compiler provides the analogous -gdwarf-<version> option, allowing you to select between DWARF versions 2, 3 (the tiarmclang default), or 4.

armcl Option (and alias)	tiarmclang Option
--symdebug:none	(default)

Even if the -g option is not specified on the command-line, the armcl compiler still generates DWARF version 3 debug information by default. The armcl compiler's --symdebug:none option allows you to instruct the compiler to avoid generating any debug information for a compilation unit.

The default behavior for the tiarmclang compiler is to not generate any DWARF debug information unless the -g or the -gdwarf-<version> option is specified on the tiarmclang command-line.

2.3.14 Diagnostic Message Options

Whereas the armcl compiler identifies diagnostics by number, the tiarmclang compiler identifies diagnostics by name. The following table explains how armcl diagnostics are managed via the armcl compiler and how certain diagnostic-related functionality in the armcl compiler might translate into a relevant tiarmclang option.

armcl Option	tiarmclang Option
--compiler_revision	--version-string

The armcl compiler supports a hidden option, --compiler_revision that prints only the version number string itself as opposed to the additional information that is emitted with the -version option.

Likewise, the tiarmclang compiler's --version-string option emits only a string representation of the compiler version number without the additional information that is emitted when the --version option is specified.

armcl Option	tiarmclang Option
--tool_version (-version)	--version

Both the armcl and tiarmclang compilers support an option to print out version information about the compiler to stdout. The armcl compiler also supports a -version option, which lists the version information associated with each of the executable components in the armcl compiler tools package.

The tiarmclang compiler's --version option prints the compiler version number and some additional information, including:

- identity of source branches used to build compiler,

- the version of the LLVM open source repository that compiler's source code base is derived from,

- the target "triple" identifier,

- the relevant thread model, and

- the location where the compiler is installed.

armcl Option (and alias)	tiarmclang Option
--diag_error=<number> (-pdse=<number>)	-Weverything
	-Werror=<category>
--diag_remark=<number> (-pdsr=<number>)	
	-W<category>
--diag_suppress=<number> (-pds=<number>)	-Wno-<category>
--diag_warning=<number> (-pdsw=<number>)	

The armcl compiler provides options that allow a diagnostic identified by a specific number (<number>) to be treated as an error, warning, or remark using the --diag_[error|warning|remark] options. You can also suppress a specified diagnostic from being emitted by the compiler using the --diag_suppress option.

The tiarmclang compiler provides several options that are similar to the armcl --diag_[error|remark|warning|suppress] options, but there are subtle differences in functionality:

- -Weverything - enables all warning diagnostics

- -Werror=category - indicates that a specific category of warning diagnostics is to be interpreted as errors

- -Wcategory - enables a specific category of warning diagnostics

- -Wno-category - disables a specific category of warning diagnostics

armcl Option	tiarmclang Option
--diag_wrap=<on|off>	not supported

The armcl's --diag_wrap option, which is on by default, tells the compiler to wrap diagnostic messages at 79 columns.

The tiarmclang compiler does not provide this capability.

armcl Option (and alias)	tiarmclang Option
--display_error_number (-pden)	-fdiagnostics-show-option (default)
	-fno-diagnostics-show-option

In order to determine the identity of a particular diagnostic, the armcl compiler provides the --display_error_number option. Once the identity of a diagnostic has been determined, you can then specify the number associated with the diagnostic to one of armcl's diagnostic control options such as --diag_suppress, for example.

Similarly, the tiarmclang compiler enables you to discover the category name associated with a given diagnostic by using the -fdiagnostics-show-option (which is on by default). Once a warning category name has been identified, you can specify the category name as an argument to one of tiarmclang's diagnostic control options (like -Werror=<category>, for example, which treats warnings that are flagged by the specified <category> as errors).

armcl Option (and alias)	tiarmclang Option
--emit_warnings_as_errors (-pdew)	-Werror[=<category>]
	-Wno-error=<category>

The armcl compiler's option --emit_warnings_as_errors functionally maps to tiarmclang's -Werror option. The use of this option instructs the compiler to interpret all warning diagnostics as errors.

An optional <category> argument can also be specified with the -Werror option to indicate that only warnings in the specified category should be treated as errors.

The tiarmclang compiler's -Wno-error=<category> option provides a mechanism by which you can identify a particular category of warning to continue being interpreted as a warning even if the -Werror option is used on the same command-line.

armcl Option (and alias)	tiarmclang Option
--issue_remarks (-pdr)	not supported

The armcl compiler can be made to emit remark diagnostics (non-serious warnings) during a compilation when the --issue_remarks option is specified.

While the tiarmclang compiler does not explicitly support issuing remarks in general, it does provide capability through other options (like the -Rpass option, for example) to enable the compiler to emit remarks related to a specific topic (like optimization transformations that are performed during compilation in the case of -Rpass).

armcl Option (and alias)	tiarmclang Option
--no_warnings (-pdw)	-w

Both the armcl and tiarmclang compilers support an option to disable the reporting of all warning diagnostics. On the armcl compiler, this option is --no_warnings (or -pdw). On tiarmclang, it is simply -w. Lower case 'w' is essentially the opposite of upper case 'W', which enables all diagnostic warnings.)

armcl Option (and alias)	tiarmclang Option
--quiet (-q)	(default)

The armcl compiler emits nominal progress and status information by default when compiling more than one source file during an invocation, but this can be suppressed with armcl's -q option.

The tiarmclang compiler does not generate progress or status information even while compiling more than one file, so there is no need for a -q option.

armcl Option	tiarmclang Option
--section_sizes=<on|off>	not supported

The armcl compiler reports information about code and constant sections at compile time if the --section-sizes option is turned 'on'. If the option is not specified on the command-line or if the 'off' argument is specified to the option, then this capability is disabled.

The tiarmclang compiler does not provide an analogous option. However, section size information is readily available with the use of binary utilities like tiarmreadelf or tiarmobjdump, which are provided with the tiarmclang tools package.

armcl Option (and alias)	tiarmclang Option
--set_error_limit=<number> (-pdel=<number>)	-ferror-limit=<number>

Both the armcl and tiarmclang compilers provide an option that allows you to indicate

the number of errors to be detected / reported before a compilation attempt is aborted. The armcl compiler uses the --set_error_limit option for this purpose. The tiarmclang compiler's -ferror-limit serves the same purpose.

By default, the armcl compiler abandons compilation after 100 errors are detected / reported. The default error limit for tiarmclang is 20. You can disable the error limit by specifying a <number> of 0 as the option argument.

armcl Option	tiarmclang Option
--verbose	-v

Both armcl and tiarmclang support an option to display verbose progress and status information during the compilation of one or more source files. The tiarmclang compiler's -v option emits information about the include file directory search path as well as details about how different executables are invoked during a compilation.

armcl Option (and alias)	tiarmclang Option
--verbose_diagnostics (-pdv)	-fdiagnostics-...

If the --verbose_diagnostics option is specified on the armcl command-line, the compiler provides a more verbose diagnostic message with a given error, warning, or remark if a more verbose message is available.

The tiarmclang compiler can be made to annotate diagnostics with extra information that is gathered by the compiler during a given compilation. For example, tiarmclang's -fdiagnostics-fixit-info, which is on by default, allows the compiler to annotate diagnostics with information about how to resolve a problem if the fix is known to the compiler.

More details about available -fdiagnostics-... options can be found in the online Clang Compiler User's Manual.

armcl Option (and alias)	tiarmclang Option
--write_diagnostics_file (-pdf)	not supported

You can redirect the diagnostics reported by the armcl compiler to a file using the --write_diagnostics_file option. The name of the generated diagnostics file will be the name of the source file provided to the compiler with its file extension replaced by an '.err' extension.

The tiarmclang compiler does not provide an analogous option.

2.3.15 Compiler Feedback Options

The following table provides information about feedback directed optimization options that are available in the armcl compiler. The tiarmclang compiler does not currently support profile guided optimizations, although the tiarmclang compiler does provide some support for generating code coverage information. Support for profile guided optimizations may be considered in future version of the tiarmclang compiler tools.

armcl Option	tiarmclang Option
--analyze=codecov	-fprofile-instr-generate
	-fcoverage-mapping

The armcl compiler can be made to generate code coverage analysis information from profile data. This option must be used in combination with --use_profile_info.

The tiarmclang compiler can be made to generate linked output files that have been instrumented with code coverage information using the -fprofile-instr-generate option in combination with the -fcoverage-mapping option. Please see the *Source-Based Code Coverage in tiarmclang* section in the *tiarmclang Compiler User Manual* for more information on what code coverage capabilities are available in the tiarmclang compiler tools.

armcl Option	tiarmclang Option
--analyze_only	-fprofile-instr-generate
	-fcoverage-mapping

The armcl compiler can be made to generate only a code coverage information file. This option must be used in combination with --use_profile_info. To instruct the compiler to perform code coverage analysis of an instrumented application, you must specify --analyze=codecov, --analyze_only, and --use_profile_info.

Please refer to the *Source-Based Code Coverage in tiarmclang* section in the *tiarmclang Compiler User Manual* for more information on what code coverage capabilities are available in the tiarmclang compiler tools.

armcl Option	tiarmclang Option
--gen_profile_info	-fprofile-instr-generate
	-fcoverage-mapping

The armcl compiler can be made to append compiled code with instrumentation that can collect profile data information when the instrumented application is run.

Please refer to the *Source-Based Code Coverage in tiarmclang* section in the *tiarmclang Compiler User Manual* for more information on what code coverage capabilities are available in the tiarmclang compiler tools.

armcl Option	tiarmclang Option
--use_profile_info=<file1>[,<file2>,. . .]	not supported

The armcl compiler can be made to read profile data information from the list of files that are specified as arguments to the --use_profile_info option and use this information to inform optimization choices and/or the generation of code coverage information.

The tiarmclang compiler tools include executables such as tiarmprofdata and tiarmcov to help view code coverage information that has been generated when an instrumented linked output file is run.

Please refer to the *Source-Based Code Coverage in tiarmclang* section in the *tiarmclang Compiler User Manual* for more information on what code coverage capabilities are available in the tiarmclang compiler tools.

2.3.16 Assembler Options

The transformation from generated code to encoded object is part of the tiarmclang compiler. In fact, you may present an assembly source file written in GNU-syntax Arm assembly language (typically with an .s extension) to the compiler and it will produce an encoded object file for that assembly source file.

If you are processing a GNU-syntax assembly source file with the tiarmclang compiler, then the normal tiarmclang command-line options are applicable. However, if you have a TI-syntax Arm assembly language source file that you want to assemble and include in an application where all of the C/C++ source files are compiled with tiarmclang, then you can use the -x ti-asm option to tell tiarmclang to invoke the TI-syntax assembler (tiarmasm) on the TI-syntax assembly source file. The TI-syntax assembler only recognizes and parses TI-syntax Arm assembly language source to produce a valid Arm object file.

Please see *Invoking the TI-Syntax ARM Assembler from tiarmclang* in the *Migrating Assembly Language Source Code* chapter in this migration guide for more information.

None of the options listed below are applicable to processing GNU-syntax Arm assembly source code with the tiarmclang compiler, but those that the armcl compiler uses to manage the behavior of its assembler (armasm) during compilation are listed below. Where applicable, it will be indicated how the armcl assembler option can be passed to the TI-syntax assembler via use of the tiarmclang compiler's "-Xti-assembler" or "-Wti-a," options.

armcl Option (and alias)	tiarmclang Option
--asm_listing (-al)	-Xti-assembler -l

The armcl compiler's --asm_listing option instructs its standalone assembler to generate an assembly listing file.

The tiarmclang's TI-syntax assembler can be made to generate an assembly listing file
by passing the -l option as an argument to the -Xti-assembler option.

armcl Option (and alias)	tiarmclang Option
--c_src_interlist (-ss)	not applicable

The armcl compiler's --c_src_interlist option instructs the compiler to generate comments interlisted with compiler-generated assembly that displays the correspondence between C/C++ statements and the assembly code that the compiler generated for those statements.

The tiarmclang compiler does not support interlisting C/C++ source code comments among compiler generated assembly.

armcl Option (and alias)	tiarmclang Option
--src_interlist (-s)	not applicable

The armcl compiler's --src_interlist option instructs the compiler to generate comments interlisted with compiler-generated assembly that displays the correspondence between C/C++ source statements or optimized C/C++ source (if the optimizer is run) and the assembly code that the compiler generated for those statements.

The tiarmclang compiler does not support interlisting C/C++ source code comments among compiler generated assembly.

armcl Option (and alias)	tiarmclang Option
--absolute_listing (-aa)	not applicable

When the --absolute_listing option is specified to the armcl compiler, the compiler will invoke its absolute lister (armabs) during compilation to produce a listing annotated with absolute addresses rather than section-relative offsets.

The tiarmclang compiler does not have an absolute lister executable at its disposal, so there no equivalent functionality available in the tiarmclang compiler tools.

armcl Option (and alias)	tiarmclang Option
--asm_define=<name>[=<value>] (-ad=<name>[=<value>])	-Xti-assembler --define=<name>[=<value>]

The armcl compiler's --asm_define option creates a pre-defined symbol for the assembler with an initial value indicated by the optional <value> argument. If no <value> argument is specified, then the symbol indicated by <name> gets a value of 1.

If the pre-defined symbol is intended for use by the TI-syntax assembler, then you can create the pre-defined symbol using the -Xti-assembler option.

armcl Option (and alias)	tiarmclang Option
--asm_dependency=<file> (-apd=<file>	-Xti-assembler --depend=<file>

The armcl compiler's --asm_dependency option will cause the TI-syntax assembler
(armasm) to pre-process a TI-syntax assembly source files and generate a list of de-
pendencies (suitable for input to a standard make utility).

If invoking the tiarmclang TI-syntax assembler, then you can pass the --depend op-
tion as an argument to the tiarmclang -Xti-assembler option to cause the TI-syntax
assembler to pre-process a TI-syntax assembly source file and generate a list of depen-
dencies.

armcl Option (and alias)	tiarmclang Option
--asm_includes (-api)	-Xti-assembler --includes

The armcl compiler's --asm_includes option will cause its TI-syntax standalone as-
sembler (armasm) to pre-process an assembly source file and generate a list of included
files.

If invoking the TI-syntax assembler from tiarmclang, then you can pass the --includes
option as an argument to the tiarmclang -Xti-assembler option to cause the TI-syntax
assembler to pre-process a TI-syntax assembly source file and generate a list of in-
cluded files.

armcl Option (and alias)	tiarmclang Option
--asm_undefine=<name> (-au=<name>)	-Xti-assembler --undefine=<name>

You can undefine a symbol that is used in a TI-syntax assembly source file with the
armcl compiler's --asm_undefine option.

If invoking the tiarmclang's TI-syntax assembler from tiarmclang, then you can pass
the --undefine option as an argument to the tiarmclang -Xti-assembler option to unde-
fine a symbol that may be referenced in a TI-syntax assembly source file.

armcl Option (and alias)	tiarmclang Option
--asm_cross_reference_listing (-ax)	-Xti-assembler -x

If the --asm_cross_reference_listing option is specified to the armcl compiler when
processing a TI-syntax assembly source file, then the armcl's standalone assembler
(armasm) will be instructed to generate a cross-reference assembly listing file.

When invoking the TI-syntax assembler (tiarmasm) from the tiarmclang compiler, you
can pass the -x option as an argument to the tiarmclang Xti-assembler option to cause
the tiarmclang's TI-syntax assembler to generate a cross reference listing file.

armcl Option (and alias)	tiarmclang Option
--include_file=<file> (-ahi=<file>)	-Xti-assembler -hi=<file>

The armcl compiler's --include_file option indicates that the specified <file> should be prepended to the TI-syntax assembly source file that is being process as if a .include directive were specified on the first line of the assembly source file.

If invoking the TI-syntax assembler from tiarmclang, then you can pass the -hi option as an argument to the tiarmclang -Xti-assembler option to include the specified <file> at the top of the TI-syntax assembly source file that is being processed.

2.3.17 Command File Option

Sometimes the list of command-line options that are used during the invocation of a compiler can become unwieldy. The @ option described below provides a mechanism for collecting command-line options and input file specifications into a text file that can be fed into the compiler as a sort of extension of the compiler invocation command.

armcl Option (and alias)	tiarmclang Option
--cmd_file=<file> (-@=<file>)	@<file>

Both the armcl and tiarmclang compilers provide an option that allows the use of a the specified <file>'s contents as an extension for the command-line used to invoke the compiler. The tiarmclang version of the option should not be preceded with a hyphen.

2.3.18 ULP Advisor Options

The tiarmclang compiler does not support the ULP Advisor options.

armcl Option	tiarmclang Option
--advice:power=<all|none|<rule>>	not supported
--advice:power_severity=<error|warning|remark|suppress>	not supported

2.3.19 MISRA-C 2004 Options

The tiarmclang compiler does not support MISRA checking.

armcl Option	tiarmclang Option
--check_misra=<all|required|advisory|none|<rule>>	not supported
--misra_advisory=<error|warning|remark|suppress>	not supported
--misra_required=<error|warning|remark|suppress>	not supported

2.4 Migrating C and C++ Source Code

The process of converting the C/C++ source code for an existing TI ARM application that is built with the armcl compiler so that it can be built using the tiarmclang compiler can be thought of as the process of making your C/C++ source code portable. The armcl compiler supports the use of legacy TI-specific versions of many predefined macro symbols, intrinsics, and pragmas that are not supported by other compilers. The use of such legacy TI mechanisms will render a program unbuildable by other compilers, including the tiarmclang compiler.

2.4.1 Arm C Language Extensions - ACLE

The Arm C Language Extensions - Release ACLE Q2 2018 is a specification provided by Arm Ltd. that describes extensions to the C/C++ language that are Arm architecture-specific and can be leveraged to make effective use of the features of the Arm architecture. Source code that is written in accordance with the ACLE specification will tend to be portable between Arm compilers that support the pre-defined macro symbols, attributes, and intrinsics that are documented in the ACLE specification.

The tiarmclang compiler provides support for Arm C Language Extensions that are applicable to the Cortex-M and Cortex-R versions of the Arm architecture. Please refer to the link above for more details about specific language extensions.

2.4.2 C/C++ Source Migration Aid Diagnostics

When migrating an Arm C/C++ application project from using the TI ARM compiler to using the tiarmclang compiler you may have instances of TI-specific pragmas, pre-defined macro symbols, or intrinsics that are supported by the armcl compiler, but not the tiarmclang compiler.

To make your C/C++ source code more portable, you will need to locate instances of TI-specific pragmas, pre-defined macro symbols, and intrinsics in your source code and convert them into their ACLE (Arm C Language Extensions) counterparts.

Please refer to the Arm C Language Extensions - Release ACLE Q2 2018 specification for details about ACLE.

To help with this process, the tiarmclang compiler emits a diagnostic when it encounters the use of a legacy TI pre-defined macro symbol, pragma, or intrinsic and provides information about how that use can be safely transformed into a functionally equivalent alternative, if one exists. In cases where there is no functionally equivalent alternative to replace an instance of a legacy TI pre-defined macro symbol, pragma, or intrinsic, the tiarmclang compiler emits a diagnostic to inform you about the presence of that legacy TI mechanism.

Let's consider a couple of examples ...

Legacy TI Pragmas

The legacy TI pragma **FUNC_CANNOT_INLINE** has a valid alternative, so if the tiarmclang compiler encounters the following line of code:

```
#pragma FUNC_CANNOT_INLINE
```

The tiarmclang compiler will emit the following diagnostic:

```
warning: pragma FUNC_CANNOT_INLINE is a legacy TI pragma and not
supported in clang compilers. use '__attribute__((always_inline))
 ↪'
instead
```

For more information about how many of the commonly occurring legacy TI pragmas can be converted into attribute form, please see the *Pragmas and Attributes* section.

Legacy TI Pre-Defined Macro Symbols

The legacy TI pre-defined macro symbol __TI_ARM_V7M4__ is an example of a pre-defined macro symbol that can be used to configure Cortex-m4 specific code in an application, but this pre-defined macro symbol is not supported in tiarmclang and needs to be replaced by a functionally equivalent ACLE expression. Specifically, when the following line of code is encountered by the tiarmclang compiler:

```
#if defined(__TI_ARM_V7M4__)
...
#endif
```

the tiarmclang compiler will emit the following diagnostic:

```
warning: __TI_ARM_V7M4__ is a legacy TI macro that is not defined
in clang compilers and will evaluate to 0, use '(__ARM_ARCH ==␣
 ↪7) &&
(__ARM_ARCH_PROFILE == 'M') && defined(__ARM_FEATURE_SIMD32)'
instead [-Wti-macros]
```

The warning can then be averted by replacing the __TI_ARM_V7M4__ symbol reference with the following:

```
#if (__ARM_ARCH == 7) && (__ARM_ARCH_PROFILE == 'M') && defined(_
 ↪_ARM_FEATURE_SIMD32)
```

However, there are other legacy TI pre-defined macro symbols, like __TI_ARM_V4__, that do not have a viable ACLE alternative, so the following code:

```
#if defined(__TI_ARM_V4__)
```

will yield the following diagnostic when encountered by the tiarmclang compiler:

```
warning: '__TI_ARM_V4__' is a legacy TI macro and not supported
 ↪in clang
compilers
```

For more information about how many of the legacy TI pre-defined macro symbols can be converted into their functionally equivalent ACLE form, please refer to the *Pre-Defined Macro Symbols* section.

Legacy TI Intrinsics

The legacy TI intrinsic "_smulbt" is an example of an armcl compiler intrinsic that has a viable alternative form, so when the tiarmclang compiler encounters the following function:

```
int foo(int x, int y) {
  return _smulbt(x,y);
}
```

the tiarmclang compiler emits the following diagnostic:

```
warning: _smulbt is a legacy TI intrinsic and is not supported
 ↪in clang
compilers, use '__smulbt' declared in arm_acle.h [-Wti-
 ↪intrinsics]
```

Note that while the armcl version of the "_smulbt" is prefixed with a single underscore, the ACLE version of the same intrinsic is prefixed by two underscores, "__smulbt", so transforming the armcl version of the intrinsic into its ACLE form is as easy as adding an extra underscore.

Note also that unlike the armcl intrinsics, which are declared implicitly within the armcl compiler, the tiarmclang compiler requires that the ACLE intrinsics be explicitly declared by including the arm_acle.h include file in your source before the first use of an ACLE intrinsic.

Not all armcl intrinsics are as easy as "_smulbt" to migrate to a functionally equivalent ACLE form. For more details on how specific armcl intrinsics can be migrated, please refer to the *Intrinsics and Built-in Functions* section.

Turning Off the Migration Aid Diagnostics

The migration aid diagnostics for use of legacy TI macro symbols, pragmas, and intrinsics are enabled by default in the tiarmclang compiler. The following tiarmclang compiler options can be specified to selectively turn off the migration aid diagnostic categories:

- **-Wno-ti-pragmas** : to suppress migration aid diagnostics for legacy TI pragmas

- **-Wno-ti-macros** : to suppress migration aid diagnostics for legacy TI pre-defined macro symbols

- **-Wno-ti-intrinsics** : to suppress migration aid diagnostics for legacy TI intrinsics

2.4.3 Pre-Defined Macro Symbols

Many applications support a variety of configurations that are often administered via the use of pre-defined macro symbols.

For example, an application may want to initialize a particular global variable differently based on which Arm processor variant the application is being built for. armcl supports several TI-specific pre-defined macro names that help with this.

While several pre-defined macro symbols supported by the armcl compiler are also supported by the tiarmclang compiler, many are not. For a given armcl pre-defined macro symbol that is not supported by the tiarmclang compiler, there are ways one can successfully transition the use of such a pre-defined symbol to be compatible with the tiarmclang compiler.

If there exists a functionally equivalent means of representing the symbol's use with an Arm C Language Extension (ACLE) pre-defined symbol, then the developer is encouraged to perform this conversion on their original source code, since both the armcl and tiarmclang compilers fully support ACLE pre-defined macro symbols. Using ACLE pre-defined macro symbols in your C/C++ source code improves the portability of your code.

A way to "transition" C/C++ source code that makes use of armcl pre-defined macro symbols that are not supported in the tiarmclang compiler is to include the ti_compatibility.h header file in the source file before any of the relevant armcl pre-defined macro symbols are referenced.

This section of the "Migrating C and C++ Source Code" chapter of the migration guide lists each of the pre-defined macro symbols that are supported in the armcl compiler and, if conversion is needed, explains how to modify the C/C++ source to make it compatible with the tiarmclang compiler.

For details about macro symbols that are pre-defined by the tiarmclang compiler, see *Generic Compiler Pre-Defined Macro Symbols*.

For more details on the Arm C Language Extensions, these documents may prove useful:

- Arm C Language Extensions - Release ACLE Q2 2018

- Arm Optimizing C/C++ Compiler User's Guide - v18.12.0.LTS or later of literature number spnu151 - Summary of ACLE Pre-Defined Macros - Table 2-30

Pre-Defined Macro Symbols that are Available in Both armcl and tiarmclang

Some pre-defined macro symbols supported by the armcl compiler are also supported by the tiarmclang compiler. The following table identifies those pre-defined macro symbols that are supported by both compilers and require no conversion when migrating an application from armcl to tiarmclang:

Macro Symbol	Description / Comments
__COUNTER__	References to the __COUNTER__ macro symbol expand to an integer value starting from 0. This symbol can be used in conjunction with a "##" operator in C/C++ source code to create unique symbol names.
__cplusplus	The __cplusplus symbol is defined if the armcl or tiarmclang compiler is invoked to process a C++ source file. If the source file in question is an obvious C++ source file with a .cpp extension, then both compilers define __cplusplus when processing such a file. The user can also force __cplusplus to be defined via the armcl -fg option or the tiarmclang -x c++ option. These instruct the compiler to process a source file, whether it is a C++ or C file, as a C++ file.
__DATE__	References to the __DATE__ macro symbol expand to a string representing the date on which the compiler was invoked. The date is displayed in the form: mmm dd yyyy.
__ELF__	The __ELF__ macro symbol is defined by both the armcl and tiarmclang compilers, since both generate ELF object format.
__FILE__	References to the __FILE__ macro symbol expand to a string representation of the name of the source file being compiled.
_INLINE	Defined if some level of optimization is specified when the compiler is invoked. The armcl compiler allows C/C++ source code to undefine the _INLINE symbol to disable some optimization while processing C/C++ source. The tiarmclang compiler does not support turning off inlining by undefining macros. The tiarmclang compiler additionally supports the the __GNUC_GNU_INLINE__, __GNUC_STDC_INLINE__, and __NO_INLINE__ macros, so that code can test to see what type of inlining is enabled.
__LINE__	References to the __LINE__ macro symbol expand to an integer constant indicating the current source line in the source file. The value of the integer constant depends on which source line the macro symbol is referenced.
__STDC__	Both the armcl and tiarmclang compilers define the __STDC__ macro symbol to indicate compliance with the ISO C standard. Please refer to the TI ARM Optimizing C/C++ Compiler User's Guide for exceptions to ISO C compliance that apply to the armcl compiler. Exceptions to ISO C compliance in the tiarmclang compiler can be found in the TI Arm Clang Compiler User Guide.
__STDC_HOSTED__	The __STDC_HOSTED__ macro symbol is always defined to 1 to indicate that the target is a hosted environment, meaning the standard C library is available.
__STDC_NO_THREADS__	The __STDC_NO_THREADS__ macro symbol is always defined to 1 to indicate the

2.4. Migrating C and C++ Source Code

Converting armcl Pre-Defined Macro Symbols to tiarmclang Compatible Form

Several armcl pre-defined macro symbols are not supported by tiarmclang, but these macro symbols can often be re-written in terms of ACLE or GCC pre-defined macro symbols that tiarmclang does support. The following table lists armcl pre-defined macro symbols that are not supported by the tiarmclang compiler, and how they may be converted to a form that is supported by tiarmclang:

armcl Macro Symbol	tiarmclang Equivalent
__16bis__	defined(__thumb__)

The armcl __16bis__ macro symbol is defined if the selected instruction set (via the --code_state option) is 16-bit. This is the default for the cortex-m0, cortex-m3, and cortex-m4 processor variants. However, it is also possible to specify --code_state=16 in combination with the cortex-r4 and cortex-r5 processor variants, in which case, __16bis__ will also be defined.

tiarmclang supports the __thumb__ macro symbol that can be used as indicated, checking that __thumb__ is defined, in place of references to __16bis__ in C/C++ source code. To explicitly select the 16-bit instruction set from the tiarmclang command-line, use the -mthumb option.

armcl Macro Symbol	tiarmclang Equivalent
__32bis__	!defined(__thumb__)

The armcl __32bis__ macro symbol is defined if the selected instruction set (via the --code_state option) is 32-bit. This is the default when the cortex-r4 or cortex-r5 processor variant is selected.

tiarmclang supports the __thumb__ macro symbol that can be used as indicated, checking that __thumb__ is not defined, in place of references to __32bis__ in C/C++ source code.

armcl Macro Symbol	tiarmclang Equivalent
__big_endian__	defined(__ARM_BIG_ENDIAN)

The armcl compiler defines the __big_endian__ macro symbol to indicate that the compiler assumes big-endian mode for the current compilation. References to __big_endian__ can safely be replaced with a test that the ACLE's __ARM_BIG_ENDIAN macro symbol is defined.

The armcl compiler assumes big-endian mode by default. The tiarmclang compiler assumes little-endian mode by default. The tiarmclang's -mbig-endian option can be used to make the tiarmclang compiler assume big-endian mode during compilation.

armcl Macro Symbol	tiarmclang Equivalent
__little_endian__	!defined(__ARM_BIG_ENDIAN)

The armcl compiler defines the __little_endian__ macro symbol to indicate that the compiler assumes little-endian mode for the current compilation. References to __little_endian__ can safely be replaced with a test that the ACLE's __ARM_BIG_ENDIAN macro symbol is not defined.

While the armcl compiler assumes big-endian mode by default, the armcl -me option can be used to select little-endian mode during compilation. The tiarmclang compiler assumes little-endian mode by default.

armcl Macro Symbol	tiarmclang Equivalent
__signed_chars__	!defined(__CHAR_UNSIGNED__)

The armcl __signed_chars__ macro symbol is defined if the plain char type is interpreted as signed. That is, if the char type name is specified without a "signed" or "unsigned" qualifier is interpreted as signed, then __signed_chars__ is defined.

tiarmclang supports the GCC macro symbol __CHAR_UNSIGNED__ which can be used in place of references to __signed_chars__ using the transformation: "__signed_chars__" -> "!defined(__CHAR_UNSIGNED)"

The default behavior for both armcl and tiarmclang is that the plain char type is interpreted as unsigned. To change this default behavior, the armcl compiler supports the --plain_char=signed option and the tiarmclang compiler supports the -fsigned-char option.

armcl Macro Symbol	tiarmclang Equivalent
__unsigned_chars__	defined(__CHAR_UNSIGNED__)

The armcl __unsigned_chars__ macro symbol is defined if the plain char type is interpreted as unsigned. That is, if the char type name is specified without a "signed" or "unsigned" qualifier is interpreted as unsigned, then __unsigned_chars__ is defined.

tiarmclang supports the GCC pre-defined macro symbol __CHAR_UNSIGNED__ which has the same meaning as the armcl __unsigned_chars__ macro symbol. All references to __unsigned_chars__ can be replaced by __CHAR_UNSIGNED__ in the C/C++ source that is to be built with tiarmclang.

The default behavior for both armcl and tiarmclang is that the plain char type is interpreted as unsigned.

armcl Macro Symbol	tiarmclang Equivalent
__PTRDIFF_T_TYPE__	__PTRDIFF_TYPE__

The armcl compiler defines the __PTRDIFF_T_TYPE__ macro symbol to indicate the equivalent base type associated with the ptrdiff_t type.

The tiarmclang compiler defines __PTRDIFF_TYPE__ to reflect the underlying type

for the ptrdiff_t typedef.

armcl Macro Symbol	tiarmclang Equivalent
__SIZE_T_TYPE__	__SIZE_TYPE__

The armcl compiler defines the __SIZE_T_TYPE__ macro symbol to indicate the equivalent base type associated with the size_t type.

The tiarmclang compiler defines __SIZE_TYPE__ to reflect the underlying type for the size_t typedef.

armcl Macro Symbol	tiarmclang Equivalent
__TI_ARM__	defined(__ARM_ARCH)

The armcl __TI_ARM__ macro symbol is defined for all Arm architectures.

References to __TI_ARM__ can be safely replaced by __ARM_ARCH in C/C++ source code that can then be compiled with either the armcl or tiarmclang compiler, since both support the ACLE macro symbol. The values of __ARM_ARCH are described in *TI Arm-Specific Pre-Defined Macro Symbols*.

armcl Macro Symbol	tiarmclang Equivalent
__TI_ARM_V6__	(__ARM_ARCH == 6)

The armcl __TI_ARM_V6__ macro symbol is defined when the -mv6 or -mv6M0 is specified on the armcl command-line. This symbol is analogous to a check of ACLE's __ARM_ARCH macro symbol. References to __TI_ARM_V6__ can be safely replaced by "(__ARM_ARCH == 6)" in C/C++ source code that can then be compiled with either the armcl or tiarmclang compiler, since both support the ACLE macro symbol.

When compiling with tiarmclang, only the cortex-m0 argument to the -mcpu option selects v6 of the Arm architecture, setting the value of __ARM_ARCH to 6.

armcl Macro Symbol	tiarmclang Equivalent
__TI_ARM_V6M0__	(__ARM_ARCH == 6) && (__ARM_ARCH_PROFILE == 'M')

When the armcl __TI_ARM_V6M0__ macro symbol is defined, it indicates that the compiler has been invoked to generate code for the cortex-m0 processor variant. References to __TI_ARM_V6M0__ can be replaced with the expression "(__ARM_ARCH == 6) && (__ARM_ARCH_PROFILE == 'M')" in C/C++ source that can then be compiled with either the armcl or tiarmclang compiler.

armcl Macro Symbol	tiarmclang Equivalent
__TI_ARM_V7__	(__ARM_ARCH == 7)

2.4. Migrating C and C++ Source Code 100

The armcl __TI_ARM_V7__ macro symbol is defined when the -mv7m3, -mv7m4, -mv7a8, -mv7r4, or -mv7r5 options are specified on the armcl command-line. This symbol is analogous to a check of ACLE's __ARM_ARCH macro symbol. References to __TI_ARM_V7__ can be safely replaced by "(__ARM_ARCH == 7)" in C/C++ source code that can then be compiled with either the armcl or tiarmclang compiler, since both support the ACLE macro symbol.

When compiling with tiarmclang, specifying cortex-m3, cortex-m4, cortex-r4, or cortex-r5 as the argument for the -mcpu option selects v7 of the Arm architecture, setting the value of __ARM_ARCH to 7.

armcl Macro Symbol	tiarmclang Equivalent
__TI_ARM_V7M3__	(__ARM_ARCH == 7) && (__ARM_ARCH_PROFILE == 'M') && !defined(__ARM_FEATURE_SIMD32)

References to the armcl __TI_ARM_V7M3__ macro symbol can be replaced with a combination of the ACLE's __ARM_ARCH, __ARM_ARCH_PROFILE, and __ARM_FEATURE_SIMD32 macro symbols. The __ARM_FEATURE_SIMD32 macro symbol indicates that the processor that the source code is being compiled for supports 32-bit SIMD instructions. This feature is supported on the cortex-m4, but not on the cortex-m3. Thus, checking that __ARM_FEATURE_SIMD32 is not defined can be used to distinguish the cortex-m3 from the cortex-m4 processor variant.

armcl Macro Symbol	tiarmclang Equivalent
__TI_ARM_V7M4__	(__ARM_ARCH == 7) && (__ARM_ARCH_PROFILE == 'M') && defined(__ARM_FEATURE_SIMD32)

References to the armcl __TI_ARM_V7M4__ macro symbol can be replaced with a combination of the ACLE's __ARM_ARCH, __ARM_ARCH_PROFILE, and __ARM_FEATURE_SIMD32 macro symbols as indicated. The __ARM_FEATURE_SIMD32 macro symbol indicates that the processor that the source code is being compiled for supports 32-bit SIMD instructions. This feature is supported on the cortex-m4, but not on the cortex-m3. Thus, checking that __ARM_FEATURE_SIMD32 is defined can be used to distinguish the cortex-m4 from the cortex-m3 processor variant.

armcl Macro Symbol	tiarmclang Equivalent
__TI_ARM_V7R4__	(__ARM_ARCH == 7) && (__ARM_ARCH_PROFILE == 'R') && !defined(__ARM_FEATURE_IDIV)

References to the armcl __TI_ARM_V7R4__ macro symbol can be replaced with a combination of the ACLE's __ARM_ARCH, __ARM_ARCH_PROFILE, and __ARM_FEATURE_IDIV macro symbols as indicated. The __ARM_FEATURE_IDIV macro symbol indicates that hardware support for 32-bit integer divide is available. This is not the case for cortex-r4, but it is true for cortex-r5. Thus, checking that __ARM_FEATURE_IDIV is not defined can be used to distinguish the cortex-r4 from the cortex-r5 processor.

armcl Macro Symbol	tiarmclang Equivalent
__TI_ARM_V7R5	(__ARM_ARCH == 7) && (__ARM_ARCH_PROFILE == 'R') && defined(__ARM_FEATURE_IDIV)

References to the armcl __TI_ARM_V7R4__ macro symbol can be replaced with a combination of the ACLE's __ARM_ARCH, __ARM_ARCH_PROFILE, and __ARM_FEATURE_IDIV macro symbols as indicated. The __ARM_FEATURE_IDIV macro symbol indicates that hardware support for 32-bit integer divide is available. This is the case for cortex-r5, but not for cortex-r4. Thus, checking that __ARM_FEATURE_IDIV is defined can be used to distinguish the cortex-r5 from the cortex-r4 processor.

armcl Macro Symbol	tiarmclang Equivalent
__TI_COMPILER_VERSION__	__ti_version__

The armcl compiler defines the __TI_COMPILER_VERSION__ macro symbol to a 7-9 digit integer, depending on if X has 1, 2, or 3 digits. The number does not contain a decimal. For example, version 3.2.1 is represented as 3002001. The leading zeros are dropped to prevent the number being interpreted as an octal.

The tiarmclang compiler defines __ti_version__ to encode the major, minor, and patch version number values associated with the current release, where:

```
<encoding> = <major> * 10000
             <minor> * 100
             <patch>
```

For 3.2.1.LTS, for example, the value of <encoding> would be 30201.

armcl Macro Symbol	tiarmclang Equivalent
__TI_EABI_SUPPORT__	defined(__ELF__)

Both the armcl and tiarmclang compilers support the generation of ELF object format code only. Consequently, the armcl __TI_EABI_SUPPORT__ macro symbol is always defined when the armcl compiler is invoked.

2.4. Migrating C and C++ Source Code 102

tiarmclang does not support __TI_EABI_SUPPORT__, but it does support the
__ELF__ macro symbol which is also supported by armcl. Therefore, references to
__TI_EABI_SUPPORT__ in the C/C++ source can be safely replaced by __ELF__.

armcl Macro Symbol	tiarmclang Equivalent
__TI_FPV4SPD16_SUPPORT__	defined(__ARM_FP) &&
	(__ARM_FP == 6) &&
	(__ARM_ARCH_PROFILE == 'M')

References to the armcl __TI_FPV4SPD16_SUPPORT__ macro symbol can be re-
placed with a combination test of the ACLE's __ARM_FP macro symbol as indicated.
A definition of __ARM_FP indicates the availability of floating-point hardware in-
structions, and then the value assigned to __ARM_FP can be used to distinguish one
variant of floating-point hardware from another.

In the case of the FPv4SPD16 floating-point hardware that is available when cortex-m4
is the selected core processor variant, the value for __ARM_FP is 6. Please see section
6.5.1 of the Arm C Language Extensions - Release 2.1 (3/24/2016) specification for
more details on values that may be assigned to the __ARM_FP macro symbol.

armcl Macro Symbol	tiarmclang Equivalent
__TI_GNU_ATTRIBUTE_SUPPORT__	defined(__clang__)

The armcl compiler defines the __TI_GNU_ATTRIBUTE_SUPPORT__ macro sym-
bol to indicate that a C/C++ dialect mode where generic attributes is supported. tiarm-
clang does not support an analogous macro symbol, but generic attributes are sup-
ported by tiarmclang, nonetheless.

armcl Macro Symbol	tiarmclang Equivalent
__TI_STRICT_ANSI_MODE__	__STRICT_ANSI__

The armcl compiler defines the __TI_STRICT_ANSI_MODE__ macro symbol to 1
if the armcl compiler is invoked with the --strict_ansi option. The armcl compiler
defines __TI_STRICT_ANSI_MODE__ with a value of 0 by default to indicate that
the compiler does not enforce strict conformance to the ANSI C standard.

The tiarmclang compiler defines the __STRICT_ANSI__ macro symbol if any of the
--std=<spec> options are specified on the tiarmclang command-line (where spec indi-
cates the identity of a C or C++ language standard).

armcl Macro Symbol	tiarmclang Equivalent
__TI_VFP_SUPPORT__	defined(__ARM_FP)

The armcl compiler defines the __TI_VFP_SUPPORT__ macro symbol to indi-
cate that the compiler assumes that floating-point hardware is available. Refer-

ences to __TI_VFP_SUPPORT__ can safely be replaced with a test that the ACLE's __ARM_FP macro symbol is defined.

armcl Macro Symbol	tiarmclang Equivalent
__TI_VFPLIB_SUPPORT__	!defined(__ARM_FP)

The armcl compiler defines the __TI_VFPLIB_SUPPORT__ macro symbol to indicate that the compiler assumes that floating-point arithmetic will be performed using software and floating-point math helper functions that do not use floating-point hardware instructions. References to __TI_VFPLIB_SUPPORT__ can safely be replaced with a test that the ACLE's __ARM_FP macro symbol is not defined.

armcl Macro Symbol	tiarmclang Equivalent
__TI_VFPV3D16_SUPPORT__	defined(__ARM_FP) && (__ARM_FP == 12) && (__ARM_ARCH_PROFILE == 'R')

References to the armcl __TI_VFPV3D16_SUPPORT__ macro symbol can be replaced with a combination test of the ACLE's __ARM_FP macro symbol as indicated. A definition of __ARM_FP indicates the availability of floating-point hardware instructions, and then the value assigned to __ARM_FP can be used to distinguish one variant of floating-point hardware from another.

In the case of the VFPv3D16 floating-point hardware that is available when cortex-r4 is the selected core processor variant, the value for __ARM_FP is 12. Please see section 6.5.1 of the Arm C Language Extensions - Release 2.1 (3/24/2016) specification for more details on values that may be assigned to the __ARM_FP macro symbol.

armcl Macro Symbol	tiarmclang Equivalent
__TI_WCHAR_T_BITS__	(__ARM_SIZEOF_WCHAR_T << 3)

The armcl compiler defines the __TI_WCHAR_T_BITS__ macro symbol to indicate the size of the wchar_t type. The analogous expression that the tiarmclang compiler supports involves the use of the ACLE __ARM_SIZEOF_WCHAR_T macro symbol as indicated.

Code that involves the use of pre-defined macro symbols that are associated with the wchar_t type may need some special attention when migrating an application from armcl to tiarmclang. The size of the wchar_t type is different on the armcl compiler than it is on the tiarmclang compiler. In armcl, the wchar_t type is 16-bits wide and is equivalent to the uint16_t type (unsigned short). However, in the tiarmclang compiler, the wchar_t type is 32-bits wide and is equivalent to the uint32_t type (unsigned int). Thus, when converting code that refers to __TI_WCHAR_T_BITS__ or __WCHAR_T_TYPE__, the developer must take the difference in compiler assumptions about the size of the wchar_t type into consideration as well as just replacing the

references to the armcl macro symbols.

armcl Macro Symbol	tiarmclang Equivalent
__WCHAR_T_TYPE__	__WCHAR_TYPE__

The armcl compiler defines the __WCHAR_T_TYPE__ macro symbol to indicate the equivalent base type associated with the wchar_t type.

The tiarmclang compiler supports the analogous GCC __WCHAR_TYPE__ macro symbol to indicate the underlying type for the wchar_t typedef.

Pre-Defined Macro Symbols in armcl that are Not Applicable in tiarmclang

There are several pre-defined macro symbols that are supported by the armcl compiler that are either not applicable for tiarmclang or are simply not supported.

For example, the __TI_ARM_V4__ pre-defined macro symbol indicates support for version 4 of the Arm architecture in the armcl compiler, but this version of the Arm architecture is not supported by the tiarmclang compiler.

armcl Macro Symbol	Description / Comments
__TI_ARM_V4__	The armcl __TI_ARM_V4__ macro symbol is not applicable since tiarmclang does not support the Arm v4 architecture.
__TI_ARM_V5__	The armcl __TI_ARM_V5__ macro symbol is not applicable since tiarmclang does not support the Arm v5 architecture.
__TI_ARM_V7A8__	The armcl __TI_ARM_V7A8__ macro symbol is not applicable since tiarmclang does not support the cortex-a8 processor variant.
__TI_FPALIB_SUPPORT__	The --float_support=fpalib option has been deprecated. The __TI_FPALIB_SUPPORT__ macro symbol is no longer defined by the armcl compiler.
__TI_NEON_SUPPORT__	The --float_support=fpalib option has been deprecated. The __TI_FPALIB_SUPPORT__ macro symbol is no longer defined by the armcl compiler.
__TI_STRICT_FP_MODE__	The armcl __TI_STRICT_FP_MODE__ macro symbol is defined to 1 by default to indicate that the compiler is to be strict about floating-point math (adherence to the IEEE-754 standard for floating-point arithmetic). This reflects the default argument for the --fp_mode option (i.e. --fp_mode=strict). To instruct the compiler to be more relaxed about floating-point math, the --fp_mode=relaxed option can be specified, which will cause __TI_STRICT_FP_MODE__ to be defined with a value of 0.
__TI_VFPV3_SUPPORT__	The armcl __TI_VFPV3_SUPPORT__ macro symbol is not applicable since tiarmclang does not support the cortex-a8 processor variant and VFPv3 floating-point hardware is only available with cortex-a8.

Additional Pre-Defined Macro Symbols Supported in tiarmclang

The following pre-defined macro symbols are provided by the tiarmclang but not by the armcl compiler.

The tiarmclang compiler defines specific GNU macro symbols that are included in the table below. These macro symbols are not meant to distinguish GCC as a compiler; instead, they indicate code compatibility with GCC.

tiarmclang Macro Symbol	Description / Comments
__ARM_FP_FAST	The tiarmclang compiler defines the __ARM_FP_FAST macro symbol if the -ffast-math option is specified on the tiarmclang command-line.
__clang__	An invocation of tiarmclang will always define the __clang__ macro symbol.
__EXCEP-TIONS	The tiarmclang compiler defines the __EXCEPTIONS macro symbol if -fexceptions is specified when compiling a C++ source file. Exceptions are disabled by default when compiling with the tiarmclang compiler.
__GNUC__	The __GNUC__ macro symbol indicates that the compiler's C pre-processor is compatible with a major version of the GNU C pre-processor. The tiarmclang compiler's C pre-processor is compatible with version 3 of the GNU C pre-processor.
__GNUC_GNU_INLINE__	The tiarmclang compiler defines the __GNUC_GNU_INLINE__ macro symbol if optimization is turned on and functions declared inline are handled in GCC's traditional gnu90 mode. Object files will contain externally visible definitions of all functions declared inline without extern or static. They will not contain any definitions of any functions declared extern inline.
__GNUC_STDC_INLINE__	The tiarmclang compiler defines the __GNUC_STDC_INLINE__ macro symbol if optimization is turned on and functions that are declared inline are handled according to the ISO C99 (or later) C language standard. Object files will contain externally visible definitions of all functions declared extern inline. They will not contain definitions of any functions declared inline without extern.
__INCLUDE_LEVEL__	The tiarmclang compiler defines the __INCLUDE_LEVEL__ macro symbol as an integer constant indicating the current include level. For example, if file f1.c includes f2.h and f2.h contains a reference to __INCLUDE_LEVEL__, then that reference to __INCLUDE_LEVEL__ would evaluate to 1.
__INTMAX_TYPE__	The tiarmclang compiler defines __INTMAX_TYPE__ to reflect the underlying type for the intmax_t typedef.
__NO_INLINE__	If no optimization level is specified on the tiarmclang command-line, then tiarmclang defines the __NO_INLINE__ macro symbol to indicate that compilation mode.
OP-	If an optimization level is specified via the -O

The _AEABI_PORTABILITY_LEVEL Pre-Defined Macro Symbol

To enable full object file portability when header files are included, the _AE-ABI_PORTABILITY_LEVEL symbol can be defined to 1. If the symbol is defined with a value of 0, then this implies a requirement that the C source code be fully compliant with the C language standard.

For more detailed information about the meaning and purpose of the _AE-ABI_PORTABILITY_LEVEL macro symbol, please see the C Library ABI for the Arm Architecture.

2.4.4 Intrinsics and Built-in Functions

The compiler intrinsics supported by the armcl compiler are fully detailed in the Arm Optimizing C/C++ Compiler User's Guide in section 5.13.

To make the use of compiler intrinsics in your C/C++ source file more portable, it is recommended that they be written in compliance with the Arm C Language Extensions - Release ACLE Q2 2018 specification when at all possible. Both the armcl and tiarmclang compilers are shipped with their own versions of the arm_acle.h header which must be included in a C/C++ compilation unit before the first reference to an ACLE compiler intrinsic in a C/C++ source file.

This section of the Migration Guide lists the compiler intrinsics supported by the armcl compiler, explaining how many can be converted into a form that can be successfully compiled with the tiarmclang compiler. There are several armcl intrinsics that do not have a functionally equivalent form that is supported by the tiarmclang compiler.

ACLE Compiler Intrinsics

This section of the "Migrating C and C++ Source Code" chapter will describe how many of the intrinsics that are supported by the armcl compiler can be converted to their functionally equivalent ACLE form. Some armcl intrinsics are already ACLE compliant and some may not have a functionally equivalent ACLE counterpart.

In general, you should refer to the Arm Optimizing C/C++ Compiler User's Guide (section 5.13) to find the armcl intrinsic that needs to be converted and then consult the Arm C Language Extensions - Release ACLE Q2 2018 specification to find the armcl intrinsic's ACLE counterpart, if one exists.

The remainder of this section will show some examples of armcl intrinsics that can be converted into their ACLE form.

Renaming

Many of the armcl-specific compiler intrinsics can be converted into ACLE compliant form simply by renaming the intrinsic in the C/C++ source file. For example, a C/C++ source file that contains a reference to the armcl's _qaddsubx intrinsic can be made ACLE compliant by including the arm_acle.h header file before the reference to the intrinsic in the compilation unit and renaming _qaddsubx to the ACLE name for the same intrinsic, __qasx.

The following table lists those armcl-specific compiler intrinsics that can be converted simply by including the arm_acle.h header file in the compilation unit and using the ACLE name for the intrinsic in place of its armcl-specific name (note that in many cases, the only difference between the armcl and ACLE names for an intrinsic is that the ACLE name uses an additional underscore in the beginning of the name):

armcl Intrinsic Name	ACLE Intrinsic Name
_smulbb	__smulbb
_smulbt	__smulbt
_smultb	__smultb
_smultt	__smultt
_smulwb	__smulwb
_smulwt	__smulwt
_sadd	__qadd
_ssub	__qsub
_smlabb	__smlabb
_smlabt	__smlabt
_smlatb	__smlatb
_smlatt	__smlatt
_smlawb	__smlawb
_smlawt	__smlawt
_ssat16	__ssat16
_usat16	__usat16
_sel	__sel
_qadd8	__qadd8
_qsub8	__qsub8
_sadd8	__sadd8
_shadd8	__shadd8
_shsub8	__shsub8
_ssub8	__ssub8
_uadd8	__uadd8
_uhadd8	__uhadd8
_uhsub8	__uhsub8
_uqadd8	__uqadd8
_uqsub8	__uqsub8

continues on next page

Table 2.1 – continued from previous page

armcl Intrinsic Name	ACLE Intrinsic Name
_usub8	__usub8
_usad8	__usad8
_qadd16	__qadd16
_qaddsubx	__qasx
_qsubaddx	__qsax
_qsub16	__qsub16
_sadd16	__sadd16
_saddsubx	__sasx
_shadd16	__shadd16
_shaddsubx	__shasx
_shsubaddx	__shsax
_shsub16	__shsub16
_ssubaddx	__ssax
_ssub16	__ssub16
_uadd16	__uadd16
_uaddsubx	__uasx
_uhadd16	__uhadd16
_uhaddsubx	__uhasx
_uhsubaddx	__uhsax
_uhsub16	__uhsub16
_uqadd16	__uqadd16
_uqaddsubx	_uqasx
_uqsubaddx	__uqsax
_uqsub16	__uqsub16
_usubaddx	__usax
_usub16	__usub16
_smuad	__smuad
_smuadx	__smuadx
_smusd	__smusd
_smusdx	__smusdx
_MCR	__arm_mcr
_MCR2	__arm_mcr2
_MCRR	__arm_mcrr
_MCRR2	__arm_mcrr2
_MRC	__arm_mrc
_MRC2	_arm_mrc2
_MRRC	__arm_mrrc
_MRRC2	__arm_mrrc2

Conversions that Require More Attention

Here are a couple of examples of armcl intrinsics that require some more work than just renaming the intrinsic:

- **_dmb** -> **__dmb**

    - armcl intrinsic form

```
void _dmb(void);
```

    - ACLE intrinsic form

```
void __dmb(unsigned int k);
```

    In this case, the ACLE form takes an unsigned int constant argument that is encoded into the 4-bit "option" field of the DMB instruction's opcode. The armcl's _dmb intrinsic implicitly encodes 0xf into this "option" field.

- **_usata** -> **__ssat**

    - armcl intrinsic form

```
uint32_t _usata(int32_t x, unsigned int shift,
→unsigned int k);
```

    - ACLE intrinsic form

```
int32_t __ssat(int32_t x, unsigned int k);
```

    In this case, the conversion can be cleanly made if the shift operand to armcl's _ssata intrinsic is zero. That is, armcl's "_ssata(x,0,k)" is equivalent to the ACLE "__ssat(x,k)". However, if the armcl intrinsic uses a non-zero shift, you will need to define _ssata as a function (see the ti_compatibility.h header file for an example implementation).

Some armcl Intrinsics May Just Need the arm_acle.h Header to be Included

Some armcl intrinsics are already ACLE compliant, but since the tiarmclang compiler does not implicitly declare intrinsics, you must #include the arm_acle.h header file in your source file prior to the first use of an ACLE intrinsic.

Here are a few examples of such intrinsics:

- **__nop**

```
void __nop(void);
```

- **__rev16**

```
uint32_t __rev16(uint32_t x);
```

- **__ror**

```
uint32_t __ror(uint32_t x, uint32_t y);
```

Non-ACLE Compiler Intrinsics

This section will account for non-ACLE intrinsics that are supported by the armcl compiler. Many of these intrinsics can be made available in an application built with the tiarmclang compiler by including the ti_compatibility.h header file in your source file prior to the use of a given non-ACLE intrinsic.

The following is a list of a few of the non-ACLE compiler intrinsics supported by armcl that require conversion to a form that tiarmclang supports. Each item indicates the name of the armcl intrinsic followed by the form of the non-ACLE intrinsic. The definition of each armcl intrinsic listed can be found in the tiarmclang compiler's ti_compatibility.h header file. For a full list of armcl's non-ACLE intrinsics that are supported, please refer to the tiarmclang's ti_compatibility.h header file.

- **_abs_s**

```
int dst = _abs_s(int src);
```

- **__bitband**

```
char *dst = __bitband(void *x, unsigned int y, unsigned int z);
```

- **_call_swi**

```
void _call_swi(unsigned int src);
```

- **__curpc**

```
void *dst = __curpc(void);
```

- **_disable_FIQ**

```
unsigned int cpsr = _disable_FIQ(void);
```

- **_disable_IRQ**

```
unsigned int pri = _disable_IRQ(void);
```

- **_disable_interrupts**

```
unsigned int fmsk = _disable_interrupts(void);
```

- **_enable_FIQ**

```
unsigned int cpsr = _enable_FIQ(void);
```

- **_enable_IRQ**

```
unsigned int pri = _enable_IRQ(void);
```

- **_enable_interrupts**

```
unsigned int fmsk = _enable_interrupts(void);
```

- **__get_PRIMASK**

```
unsigned int dst = __get_PRIMASK(void);
```

- **__ldrex**

```
float dst= _itof(unsigned int src);
```

Non-ACLE armcl Compiler Intrinsics Not Supported in tiarmclang

The following non-ACLE compiler intrinsics are supported in the armcl compiler, but have no viable form that is supported in tiarmclang nor are they supported in the ti_compatibility.h header file:

- **_addc**
- **_delay_cycles**
- **__sqrt**
- **__sqrtf**
- **_subc**

Accessing armcl Compiler Intrinsics Via the ti_compatibility.h Header File

The idea behind the ti_compatibility.h header file that you can find in the /include area of your tiarmclang tools installation is to define armcl-specific compiler intrinsics that are not available in tiarmclang in terms of macros or static functions that are always inlined. Where applicable, the equivalent forms of the armcl compiler intrinsics will be used.

For example, suppose a C/C++ source file references armcl's _smpy compiler intrinsic. The ti_compatibility.h header file can be included in the C/C++ source file before the reference to _smpy. When the C/C++ source file is compiled with tiarmclang, the relevant code from ti_compatibility.h kicks in and the static function named _smpy defined in ti_compatibility.h will be inlined at each point where the _smpy intrinsic is referenced in the C/C++ source code.

All of the armcl compiler intrinsics that are mentioned in the above discussions as having a viable conversion to a form that is compatible with tiarmclang are represented in the ti_compatibility.h header file that is included with the tiarmclang tools installation. That is, you can find a definition of each armcl-specific intrinsic that has a viable conversion in the ti_compatibility.h header file. Each such armcl intrinsic is defined via a macro or a static function that is always inlined. Please see the ti_compatibility.h header file itself for more details about how each conversion is implemented.

2.4.5 Pragmas and Attributes

Pragmas

While the tiarmclang compiler does support some of the same pragma directives that the armcl compiler supports, there are several pragma directives supported by armcl that are not supported by tiarmclang. Some of these can be converted into their functionally equivalent attribute or pragma forms while others may be supported in an indirect way or not supported at all. This section walks through all of the pragma directives that are supported in the armcl compiler, providing guidance on how to transition each pragma directive for a project to be built with the tiarmclang compiler.

In general, if there is a tiarmclang functionally equivalent attribute or pragma form for an armcl pragma directive, these should be converted to attribute or pragma form. The use of GNU-like attributes and pragmas in C/C++ source code is very likely to be portable between armcl and tiarmclang.

armcl Pragmas to be Converted to Attribute or Pragma Form

Listed below are several commonly occurring armcl pragmas that, when converted to attribute form, are supported by the tiarmclang compiler.

- **CODE_SECTION** pragma -> **section** attribute

    armcl pragma:

    ```
    #pragma CODE_SECTION(func_name, "scn_name")
    ```

    tiarmclang functionally equivalent attribute:

    ```
    __attribute__((section("scn_name")))
    ```

    The section attribute can be used to instruct the compiler to generate code associated with a function into a section called scn_name.

- **DATA_ALIGN** pragma -> **aligned** attribute

    armcl pragma:

    ```
    #pragma DATA_ALIGN("sym_name", alignment)
    ```

    tiarmclang functionally equivalent attribute:

    ```
    __attribute__((aligned(alignment)))
    ```

    The aligned attribute instructs the compiler to align the address where the data object that the attribute is associated with is defined to a specified alignment boundary (where alignment is indicated in bytes and must be a power of two).

- **DATA_SECTION** pragma -> **section** attribute

    armcl pragma:

    ```
    #pragma DATA_SECTION(sym_name, "scn_name")
    ```

    tiarmclang functionally equivalent attribute:

    ```
    __attribute__((section("scn_name")))
    ```

    The section attribute can be used to instruct the compiler to generate the definition of a data object into a section called scn_name.

- **FORCEINLINE** pragma -> [[clang::always_inline]] statement attribute

    armcl pragma:

```
#pragma FORCEINLINE
```

tiarmclang functionally equivalent attribute:

```
[[clang::always_inline]] *statement*;
```

The [[clang::always_inline]] statement attribute can be used before a statement to cause any function calls made in that statement to be inlined. It has no effect on other calls to the same functions. It cannot be used as a prefix for a declaration statement, even if that statement calls a function. For example:

```
[[clang::always_inline]] myFunc1(); // attempts to␣
↪inline myFunc1
[[clang::always_inline]] i = myFunc2(); // attempts␣
↪to inline myFunc2
```

The [[clang::always_inline]] attribute is part of the C23 and C++11 standards.

This attribute does not force inline substitution to occur. The compiler only inlines a function if it is legal to inline the function. Functions are never inlined if the compiler is invoked with the -O0 or -fno-inline-functions option. If the -finline-functions or -O2 (or higher) option is used, the compiler attempts to inline functions even if they are not called with the [[clang::always_inline]] attribute. See *Optimization Options* for more about inlining.

- **FUNC_ALWAYS_INLINE** pragma -> **always_inline** function attribute

    armcl pragma:

    ```
    #pragma FUNC_ALWAYS_INLINE(func_name)
    ```

    tiarmclang functionally equivalent attribute:

    ```
    __attribute__((always_inline))
    ```

    The always_inline attribute instructs the compiler to inline the definition of the function the attribute precedes wherever it is referenced in the C/C++ source code for an application.

- **FUNC_CANNOT_INLINE** pragma -> **noinline** attribute

    armcl pragma:

    ```
    #pragma FUNC_CANNOT_INLINE(func_name)
    ```

    tiarmclang functionally equivalent attribute:

```
__attribute__((noinline))
```

The noinline function attribute indicates to the compiler that it should not attempt to inline the function the attribute is associated with (func_name).

- **LOCATION** pragma -> **location** attribute

    armcl pragma:

    ```
    #pragma LOCATION(address)
    ```

    tiarmclang functionally equivalent attribute:

    ```
    __attribute__((location(address)))
    ```

    The location attribute can be used to instruct the compiler to generate information for the linker to dictate the specific memory address where the data object the attribute is associated with (sym_name) is to be placed.

- **NOINIT** pragma -> **noinit** attribute

    armcl pragma:

    ```
    #pragma NOINIT(sym_name)
    ```

    tiarmclang functionally equivalent attribute:

    ```
    __attribute__((noinit))
    ```

    The noinit attribute instructs the compiler to pass instructions to the linker to ensure that a global or static data object that the attribute is associated with (sym_name) does not get initialized at startup or reset.

- **NOINLINE** pragma -> [[clang::noinline]] statement attribute

    armcl pragma:

    ```
    #pragma NOINLINE
    ```

    tiarmclang functionally equivalent attribute:

    ```
    [[clang::noinline]] *statement*;
    ```

    The [[clang::noinline]] statement attribute can be used before a statement to prevent any function calls made in that statement from being inlined. It has no effect on other calls to the same functions. It cannot be used as a prefix for a declaration statement, even if that statement calls a function. For example:

```
[[clang::noinline]] myFunc1(); // prevents inlining␣
↪of myFunc1
[[clang::noinline]] i = myFunc2(); // prevents␣
↪inlining of myFunc2
```

The [[clang::noinline]] attribute is part of the C23 and C++11 standards.

See *Optimization Options* for more about inlining.

- **PERSISTENT** pragma -> **persistent** attribute

    armcl pragma:

    ```
    #pragma PERSISTENT(sym_name)
    ```

    tiarmclang functionally equivalent attribute:

    ```
    __attribute__((persistent))
    ```

    The persistent attribute can be applied to a statically initialized data object to indicate to the compiler that the data object that the attribute is associated with (sym_name) need not be initialized at startup. The data object sym_name will be given an initial value when the application is loaded, but it is never again initialized.

- **RETAIN** pragma -> **retain** or **used** attribute

    armcl pragma:

    ```
    #pragma RETAIN(sym_name)
    ```

    tiarmclang functionally equivalent attribute:

    ```
    __attribute__((retain))
    __attribute__((used))
    ```

    The retain or used attribute, when applied to a function or a data object, indicates to the linker that the section in which the function or data object is defined must be included in the linked application, even if there are no references to the function/data object.

- **SET_CODE_SECTION** pragma -> **clang section text** pragma

    armcl pragma:

    ```
    #pragma SET_CODE_SECTION("scn_name")
    ```

    tiarmclang functionally equivalent pragma:

```
#pragma clang section text="scn_name"
```

This pragma will place enclosed functions within a named section, which can then be placed with the linker using a linker command file. Note that use of the **section** attribute will take priority over this pragma, and using the pragma means that enclosed functions will not be placed in individual subsections.

The pragma can be reset by using

```
#pragma clang section text=""
```

- **SET_DATA_SECTION** pragma -> **clang section data** pragma

    armcl pragma:

    ```
    #pragma SET_DATA_SECTION("scn_name")
    ```

    tiarmclang functionally equivalent pragma:

    ```
    #pragma clang section data="scn_name"
    ```

    This pragma will place enclosed variables within a named section, which can then be placed with the linker using a linker command file. Note that use of the **section** attribute will take priority over this pragma, and using the pragma means that enclosed variables will not be placed in individual subsections.

    The pragma can be reset by using

    ```
    #pragma clang section data=""
    ```

- **UNROLL** pragma -> **clang loop unroll** pragma

    armcl pragma:

    ```
    #pragma UNROLL(n)
    ```

    tiarmclang functionally equivalent pragmas:

    ```
    #pragma clang loop unroll_count(n)
    ```

    The tiarmclang compiler supports a *clang loop unroll_count(n)* pragma, where *n* is a positive integer indicating the number of times to unroll the loop in question. If the specified value for *n* is greater than the loop trip count, then the loop will be fully unrolled.

- **WEAK** pragma -> **weak** attribute

    armcl pragma:

```
#pragma #pragma WEAK(sym_name)
```

tiarmclang functionally equivalent attribute:

```
__attribute__((weak))
```

The weak attribute can be used to mark a symbol definition as having weak binding. If a strong definition of the symbol the attribute is associated with (sym_name) is available from an input object file at link time, it will preempt this weak definition. However, if a strong definition of sym_name is available in a referenced archive file, then the linker will not automatically pull in the strong definition from the archive file to preempt the weak definition.

pack Pragma Directives

The following pack pragma directives are supported in both the armcl and tiarmclang compilers:

- **pack(n)**

```
#pragma pack(n)
```

The pack pragma instructs the compiler to apply an n-byte alignment to the fields within a class, struct, or union type, where n is an integer value that is also a power of 2. This form of the pack pragma applies to all subsequent class, struct, or union types in the same compilation unit. It forces the maximum alignment of each field to be the value specified by n. If the default alignment of a field is less than n, the alignment is not changed.

- **pack(push,n)** and **pack(pop,n)**

```
#pragma pack(push, n)
#pragma pack(pop)
```

The push and pop forms of the pack pragma define a region within the C/C++ source code in which the compiler will apply an n-byte maximum alignment to the fields in a class, struct, or union type that is specified within that region, where n is an integer value that is also a power of 2. If the default alignment of a field is less than n, the alignment is not changed.

- **pack(show)**

```
#pragma pack(show)
```

The show form of the pack pragma instructs the compiler to emit a warning to STDERR that displays the current pack alignment value in effect.

armcl Pragmas That Are Not Available in tiarmclang

The following armcl pragma directives are not supported by the tiarmclang compiler and don't have a functionally equivalent attribute form:

- **CHECK_MISRA**

```
#pragma CHECK_MISRA("{all|required|advisory|none|rulespec}")
```

- **CHECK_ULP**

```
#pragma CHECK_ULP("{all|none|rulespec}")
```

- **CLINK**

```
#pragma CLINK(sym_name)
```

- **DUAL_STATE**

```
#pragma DUAL_STATE(func_name)
```

- **FORCEINLINE_RECURSIVE**

```
#pragma FORCEINLINE_RECURSIVE)
```

- **FUNC_EXT_CALLED**

```
#pragma FUNC_EXT_CALLED(func_name)
```

- **FUNCTION_OPTIONS**

```
#pragma FUNCTION_OPTIONS(func_name, "added_opts")
```

- **MUST_ITERATE**

```
#pragma MUST_ITERATE(min[, max[, multiple]])
```

- **NO_HOOKS**

```
#pragma NO_HOOKS(func_name)
```

- **RESET_MISRA**

```
#pragma RESET_MISRA("{all|required|advisory|rulespec}")
```

- **RESET_ULP**

```
#pragma RESET_ULP("{all|rulespec}"
```

- **SWI_ALIAS**

```
#pragma SWI_ALIAS(func_name, swi_number)
```

For more information about these pragmas and how they function, please refer to the Arm Optimizing C/C++ Compiler User's Guide (Section 5.10).

Attributes

Both the armcl and tiarmclang compilers support the notion of attributes that can be applied to functions, variables, or types. The attributes supported in armcl and tiarmclang follow the guidelines for attributes that can be found in the Extensions to the C Language Family section of the GNU Compiler Collection user guide.

This section provides details of which function, variable, and type attributes are supported in both the armcl and tiarmclang compilers. This section also provides details about which attributes are supported in the armcl compiler, but not the tiarmclang compiler. References to these attributes in your application's source code will need to be addressed in some way before attempting to compile your application with the tiarmclang compiler. Finally, this section provides information on additional attributes that are supported in the tiarmclang compiler, but not the armcl compiler.

Function Attributes

The function attributes listed below are supported in both the armcl and tiarmclang compilers. Please consult the Declaring Attributes of Functions page of the GNU Compiler Collection for more details about what they do.

- **alias**

```
__attribute__((alias("target_fcn")))
```

Declare function to be an alias of "target_fcn".

- **always_inline**

```
__attribute__((always_inline))
```

Compiler should inline the definition of this function wherever it is referenced in the application's source code.

- **const**

```
__attribute__((const))
```

Function has no effect except to compute the return value.

- **constructor**

```
__attribute__((constructor))
```

This function needs to be called/executed before main().

- **format**

```
__attribute__((format(archetype, string_index, first_to_check)))
```

Function takes printf, scanf, strftime, or strfmon style arguments which should be type-checked against a format string. The string_index argument indicates which function argument is the format string. The first_to_check argument indicates the first function argument that is to be checked against the format string.

- **format_arg**

```
__attribute__((format_arg(string_index)))
```

The function argument indicated by string_index is to be interpreted as a format string when passed to a printf, scanf, strftime, or strfmon function that is called from the function that the format_arg attribute is applied to.

- **interrupt**

```
__attribute__((interrupt("int_kind")))
```

In armcl, the available "int_kind"s are: DABT, FIQ, IRQ, PABT, RESET, or UNDEF. In tiarmclang, "int_kind" can be: IRQ, FIQ, SWI, ABORT, or UNDEF.

Note: FPU Registers are Not Automatically Preserved by |commandline_name| for Interrupt Functions

The tiarmclang compiler will not generate code to preserve FPU registers in a function marked with the *interrupt* attribute. If floating-point operations are required in the scope of an interrupt, then the FPU registers must be preserved manually.

- **malloc**

```
__attribute__((malloc))
```

Function may be treated by the compiler as if it were a malloc function.

- **naked**

```
__attribute__((naked))
```

Function is to be treated as an embedded assembly function.

- **noinline**

```
__attribute__((noinline))
```

Compiler should not attempt to inline this function.

- **noreturn**

```
__attribute__((noreturn))
```

Calls to this function will never return to their caller.

- **pure**

```
__attribute__((pure))
```

This function has no effect except to compute the return value which is dependent only on the arguments passed into the function and/or global variables.

- **section**

```
__attribute__((section("scn_name"))
```

Generate code for the definition of this function into a section named "scn_name".

- **unused**

```
__attribute__((unused))
```

This function might not be used by an application. The attribute can be useful in that the compiler knows not to generate a warning when the function is in fact not used.

- **used**

```
__attribute__((used))
__attribute-_((retain))
```

Generate code for this function even if the compiler knows that there are no references to the function. This attribute is also a synonym for tiarmclang's retain attribute which tells the linker to include this function in the link whether or not it is referenced elsewhere in the application.

- **warn_unused_result**

```
__attribute__((warn_unused_result))
```

> Compiler will generate a warning if any callers to this function do not use the function's return value.

- **weak**

```
__attribute__((warn_unused_result))
```

> The definition of this function is considered "weak" meaning that it will be preempted if a strong definition of the function is encountered among the object files specified to the linker. Note, however, that if a strong definition of the function is contained in a referenced archive, it will not automatically be pulled into the link to preempt the weak definition of the function.

The following list of armcl function attributes are not supported in the tiarmclang compiler:

- **aligned**
- **builtin**
- **calls**
- **deprecated**
- **impure**
- **ramfunc**
- **target**

Variable Attributes

The variable attributes listed below are supported in both the armcl and tiarmclang compilers. Please consult the Specifying Attributes of Variables page of the GNU Compiler Collection for more details about what they do.

- **aligned**

```
__attribute__((aligned(alignment)))
```

> Align this data object to a minimum of the specified alignment argument. The alignment argument must be a power of 2.

- **deprecated**

```
__attribute__((deprecated))
```

If there are references to this data object in the current application, then the compiler should generate a warning about remaining references to this data object which has been marked deprecated.

- **location**

```
__attribute__((location(address)))
```

The compiler will instruct the linker to place this data object at a specific address at link time.

- **noinit**

```
__attribute__((noinit))
```

The compiler will not auto-initialize this data object.

- **packed**

```
__attribute__((packed))
```

The packed attribute can be applied to individual fields within a struct or union. This tells the compiler to relax alignment constraints for a struct or union member that may be larger than a byte in size. A packed member of a struct or union will be aligned on a byte boundary and may require an unaligned load or store instruction to be properly accessed.

- **persistent**

```
__attribute__((persistent))
```

This data object is initialized once and is not re-initialized again in the event of a processor reset.

- **section**

```
__attribute__((section("scn_name")))
```

Generate the definition of this data object into a section named "scn_name".

- **transparent_union**

```
__attribute__((transparent_union))
```

The transparent_union attribute may be applied to the specification of a union type. If a function is declared with a parameter of this union type, then the argument type at the call site to the function determines which member of the union is initialized. A transparent union can accept an argument of any type that matches that of one of its members without an explicit cast.

Transparent unions are not supported in C++.

- **unused**

```
__attribute__((unused))
```

Avoid generating a diagnostic at compile time if this data object is not referenced.

- **used**

```
__attribute__((retain))
__attribute__((used))
```

Retain the definition of this static data object even if it is not referenced in the compilation unit where it is defined. In the tiarmclang compiler the used attribute acts as a synonym for the tiarmclang's retain attribute which instructs the linker to include the definition of this data object in the link even if it is not referenced elsewhere in the application.

- **weak**

```
__attribute__((weak))
```

The weak attribute marks the definition of the variable that it is being applied to as a "weak" definition, meaning that if a strong definition of the same variable is provided to the link in another input object file, then that definition will preempt the weak definition of the variable. Note, however, that if a strong definition is present in a referenced archive file, the linker will not automatically pull in the strong definition of the variable from the archive to preempt the weak definition.

The following list of armcl variable attributes are not supported in the tiarmclang compiler:

- **aligned**

- **externally_visible**

- **memread**

- **memwrite**

Type Attributes

- **aligned**

```
__attribute__((aligned(alignment)))
```

The aligned attribute, when applied to a type, instructs the compiler to align the address where a data object is defined of that type to a minimum of the specified alignment argument. The alignment argument must be an integer constant that is a power of 2.

- **deprecated**

```
__attribute__((deprecated))
```

The deprecated attribute instructs the compiler to emit warnings for any references to a type with this attribute. This is useful for finding remaining references to a type that should no longer be used by an application.

- **packed**

```
__attribute__((packed))
```

The packed attribute may be applied to a struct or union type definition. Members of a packed data structure are stored as closely to one another as possible, omitting additional bytes of padding between fields that would have been necessary to preserve alignment of a member within a structure. The packed attribute can only be applied to the original definition of a struct or union type. It cannot be applied with a typedef to a non-packed data structure type that has already been defined, nor can it be applied to the declaration of a struct or union data object.

- **transparent_union**

```
__attribute__((transparent_union))
```

The transparent_union attribute may be applied to the specification of a union type. If a function is declared with a parameter of this union type, then the argument type at the call site to the function determines which member of the union is initialized. A transparent union can accept an argument of any type that matches that of one of its members without an explicit cast.

Transparent unions are not supported in C++.

- **unused**

```
__attribute__((unused))
```

Avoid generating a diagnostic at compile time if this type is not referenced.

The following list of armcl type attributes are not supported in the tiarmclang compiler:

- **unpacked**

For Loop Attributes

- **TI::unroll** for loop attribute -> **clang loop unroll** pragma

    armcl attribute:

```
[[TI::unroll(4)]]
for (...)
{
    ...
}
```

    tiarmclang functionally equivalent pragma:

```
#pragma unroll 4
 for (...)
 {
     ...
 }
```

The following armcl for loop attributes are not supported in the tiarmclang compiler:

- **TI::must_iterate**

2.5 Migrating Assembly Language Source Code

Applications developed with the armcl compiler tools may include some source code written in assembly language. Within the scope of this migration guide, this will be referred to as legacy TI-syntax Arm assembly language. Legacy TI-syntax Arm assembly language is any assembly source that is currently accepted by the armcl compiler tools to produce a valid Arm object file. Such assembly source is likely to be rejected when you attempt to assemble it with the tiarmclang compiler tools.

In contrast, we'll refer to the assembly language syntax accepted by the tiarmclang compiler tools as GNU-syntax Arm assembly source code.

So how does one use legacy TI-syntax Arm assembly source in a project to be built with the tiarmclang compiler tools? There are two choices to be considered:

- **Maintenance planned** - If maintenance may be needed during the project life-cycle or if the existing assembly code is likely to be shared or reused by other projects built with the tiarmclang compiler tools, then we recommend that you translate existing legacy TI-syntax code into GNU-syntax Arm assembly source code. Translating most often occurs in either pure assembly source code files or in C/C++ source files that use embedded asm() statements. *Translating Legacy TI-Syntax Arm Assembly to GNU-Syntax Arm Assembly* provides more

details about how to make your assembly source files tiarmclang-friendly for a project that is likely to be maintained over a long period of time.

- **Maintenance unlikely** - If maintenance is unlikely during the life of the migrated project, then you can assemble the legacy TI-syntax assembly source using the standalone Arm assembler (armasm). This assembler is shipped as a separate executable with the tiarmclang compiler tools. The object files can then be fed into the link step for the application that is being built. You can also invoke the standalone TI-syntax Arm assembler from tiarmclang using the "-x ti-asm" option to identify TI-syntax assembly source files on the tiarmclang command line. The *Invoking the TI-Syntax ARM Assembler from tiarmclang* section provides more information about how to do this.

2.5.1 Translating Legacy TI-Syntax Arm Assembly to GNU-Syntax Arm Assembly

This area of the "Migrating Assembly Language Source Code" section addresses how to convert explicit legacy TI-syntax Arm assembly language in an assembly source file into GNU-syntax Arm assembly language.

First, we will go through a discussion of the anatomy of a given assembly language source line, noting differences between the rules employed for legacy TI-syntax Arm assembly versus GNU-syntax Arm assembly in each field of a line of assembly source.

Second, we'll consider concerns that may surface when converting Arm assembly instructions from legacy TI-syntax to GNU-syntax.

Third, we'll discuss how to deal with converting assembly directives from legacy TI-syntax to GNU-syntax. While many of the legacy TI-syntax assembly directives have a functionally equivalent GNU-syntax counterpart, some may require more than one line of GNU-syntax assembly (the .usect directive, for example) when converted. Other legacy TI-syntax assembly directives may not have a functionally equivalent GNU-syntax counterpart and, similarly, some GNU-syntax assembly directives do not have functionally equivalent legacy TI-syntax counterparts.

Finally, the migration of assembly macro definitions from legacy TI-syntax to GNU-syntax will be taken up in some detail. Assembly macro definitions consist of more than just the .macro and .endm directives that define the beginning and end of the scope of a given macro definition. One must also address the conversion of macro parameter references and the definition of and references to local labels that are used in the scope of a macro definition.

Assembly Source Anatomy

Fields of an Assembly Source Line

While there are many differences in the details between the legacy TI-syntax and GNU-syntax Arm assembly languages, they are generally very similar. Both follow the same general form:

label field: *mnemonic field operand list field*

For example, the following Arm instruction is legal in both legacy TI-syntax and GNU-syntax Arm assembly language:

```
add_me:    add    r0, r1
```

where "add_me" occupies the label field, "add" occupies the mnemonic field, and the operand list field consists of registers r0 and r1 with operands separated by a comma.

Many of the assembly directives supported in legacy TI-syntax will need to be converted into their functionally equivalent GNU-syntax counterparts, and while most Arm instructions will likely assemble successfully with either the legacy TI-syntax assembler or the tiarmclang GNU-syntax assembler, there are different rules governing legal syntax in the label, mnemonic, and operand list fields that one should be aware of when migrating assembly source files:

Labels

An optional label field can be used to associate a value with a symbol. Label symbol names are case sensitive, and a label must begin in the leftmost column of the assembly source line. The rules governing the name of a symbol defined by a label are largely similar for legacy TI-syntax and GNU-syntax assembly source code, but there are some subtle differences:

Legacy TI-Syntax:

- Label symbols must begin with a letter or an underscore

- Label symbols can consist of alphanumeric characters, the dollar sign ("$"), and underscores ("_")

- Label symbol definitions may be delimited by an optional terminating colon (":")

GNU-Syntax:

- Label symbols must start with a letter, an underscore, or a period (".")

- Label symbols can consist of alphanumeric characters, the dollar sign ("$"), an underscore ("_"), or a period (".")

- Label symbol definitions must be delimited with a terminating colon (":"), otherwise the tiarmclang assembler tries to interpret the symbol as a mnemonic identifier

The value assigned to a symbol defined in a label field may vary depending on whether the label occurs within the context of an instruction or a directive. A more detailed discussion of the label field in each of these contexts is provided in the *Converting TI-Syntax Arm Instructions to GNU-Syntax Arm Instructions* and the *Converting TI-Syntax Assembly Directives to GNU-Syntax Assembly Directives* sections.

Local Labels

TI-Syntax Arm Assembler Local Labels

The TI-syntax Arm assembler provides support for *local labels* whose scope and effect are temporary. Local labels cannot be declared with global linkage. There are two forms of local labels supported in the TI-syntax Arm assembler:

- **$n** - where *n* is an integer in the range [0,9]

- **name?** - where *name* is a legal identifier. The TI-syntax Arm assembler replaces the *?* with a period followed by a unique integer.

Normal labels must be unique, but the TI-syntax Arm assembler allows local labels to be undefined and defined again within the same compilation unit. Please see the TI Arm Assembly Language Tools User's Guide for more details about how to use and manage local labels in the TI-syntax Arm assembler.

GNU-Syntax Arm Assembler Local Labels

The GNU-syntax Arm assembler that is integrated into the tiarmclang compiler also supports the notion of local labels with similar limitations on the scope in that GNU-syntax local labels cannot be declared with global linkage. The syntax for defining and referring to GNU-syntax local labels is as follows:

- Local label definitions use the form **N:** in the label field of a line of GNU-syntax assembly code, where *N* is an integer in the range [0,9].

- References to the most recently defined local label use the form **Nb**, where *N* is the ID of the local label (an integer in [0,9]) and *b* indicates a backward reference.

- References to the nest definition of a local label use the form **Nf**, where *N* is the ID of the local label (an integer in [0,9] and *f* indicates a forward reference.

Like TI-syntax local labels, GNU-syntax local labels can be redefined in the same compilation unit. However, unlike TI-syntax local labels, there are no special directives needed to undefine an existing local label. The GNU-syntax assembler will associate a unique ordinal ID for every local label definition so that it is able to distinguish one instance of a local label definition from another that was defined with the same value *N*.

Simple Local Label Example

Consider a snippet of TI-syntax Arm assembly source that implements a simple loop:

```
;* assume external global int "sum_tot"
    .global   sum_tot

;* assume incoming r0 has loop limit
    .global   foo
    .sect     ".text"
    .thumb
foo:
    ...
    MOVS    r1,#0
    CMP     r1, r0
    BLE     $1
$0:
    LDR     r2, C_CON1
    LDR     r3, [r2]
    ADDS    r3, r3, r1
    STR     r3, [r2]
    ADDS    r1, r1, #1
    CMP     r1, r0
    BGT     $0
$1:
    ...
    BX      LR

    .align 4
C_CON1:   .int    sum_tot
```

where *$0* and *$1* are TI-syntax local labels.

This loop example can be converted to GNU-syntax assembly as follows:

```
// assume external global int "sum_tot"
    .global   sum_tot

// assume incoming r0 has loop limit
    .global   foo
    .section  .text
    .thumb
foo:
    ...
    MOVS    r1,#0
    CMP     r1, r0
    BLE     1f
0:
```

```
      LDR     r2, C_CON1
      LDR     r3, [r2]
      ADDS    r3, r3, r1
      STR     r3, [r2]
      ADDS    r1, r1, #1
      CMP     r1, r0
      BGT     0b
1:
      ...
      BX      LR

      .align 4
C_CON1:   .int    sum_tot
```

where the following changes were made to the TI-syntax source to make a functionally equivalent GNU-syntax implementation of the loop:

- The ";*" comment delimiters are replaced with "//".

- The TI-syntax for *.sect* directive is converted to the GNU-syntax *.section* directive.

- The TI-syntax local labels, *$0* and *$1*, are replaced with GNU-syntax local labels *0* and *1*

- The TI-syntax forward reference to *$1* is converted to a GNU-syntax forward reference to local label *1* via the syntax *1f* ('f' indicates a forward reference)

- The TI-syntax backward reference to *$0* is converted to a GNU-syntax backward reference to local label *0* via the syntax *0b* ('b' indicates a backward reference)

Macro Example Use of Local Labels

The following macro is an example of the *name?* form of the TI-syntax local label:

```
;* TI-syntax implementation of trace_pc macro using local labels
trace_pc    .macro
L_?:
    .sect    ".trace_scn"
    .int     L_?
    .sect    ".text"
    .endm

    .sect    ".text"
foo:
    nop
    trace_pc
    nop
```

```
        trace_pc
        nop
        trace_pc
        nop
```

where *L_?* gets converted into *L_0*, *L_1*, and *L_2* with each invocation of the *trace_pc* macro in the above example.

The above example can be converted to GNU-syntax as follows:

```
// GNU-syntax implementation of trace_pc macro using local labels
        .macro    trace_pc
\@:
        .section  .trace_scn,"aw",%progbits
        .int      \@b
        .previous
        .endm

        .section  .text
foo:
        nop
        trace_pc
        nop
        trace_pc
        nop
        trace_pc
        nop
```

where the following changes were made to the TI-syntax source to make a functionally equivalent GNU-syntax implementation of the macro example:

- The ";*" comment delimiter is converted to "//".

- The name of the macro is moved from the label field for the TI-syntax implementation to the first operand field for the GNU-syntax implementation

- The definition of the TI-syntax local label *L_?* is replaced by the use of the special @ operator for GNU-syntax assembly macros used in the label field to auto-generate a local label each time the macro is invoked.

- Likewise, the reference to the TI-syntax local label *L_?* is replaced by *@b*, which the GNU-syntax assembler interprets as a backward reference to the auto-generated local label.

- Finally, the *.previous* directive returns the assembler back to the previous input section that was being assembled into.

If we were to use a normal label like *xyz_@* for the GNU-syntax implementation of the macro, the

assembler would report a duplicate label definition for *xyz_0*. When the GNU-syntax assembler invokes the *trace_pc* macro using a local label definition, then the local label *0* is auto-generated for each invocation. Since the GNU-syntax assembler will assign a unique ordinal ID to each instance of a local label, it is able to avoid a duplicate label definition when local labels are used in the macro definition.

Mnemonics

The mnemonic field of a legal line of assembly code contains a pre-defined textual identifier that indicates whether the source line represents an *instruction* or a *directive*.

For example, the "push" mnemonic in the following line of assembly code is recognized as a valid Arm instruction:

```
        .text
        .thumb
        .global     simple_function

simple_function:
        push {r7,lr}
        ...

The ".text", ".thumb", and ".global" contents in the mnemonic
field are recognized as Arm assembly directives.
```

More information about how to convert legacy TI-syntax Arm instructions and directives into GNU-syntax is provided in both the *Converting TI-Syntax Arm Instructions to GNU-Syntax Arm Instructions* and the *Converting TI-Syntax Assembly Directives to GNU-Syntax Assembly Directives* sections, respectively.

With regards to syntax rules governing the mnemonic field, the only major difference between legacy TI-syntax and GNU-syntax is that GNU-syntax mnemonic identifiers may begin in the leftmost column of the assembly source line. Legacy TI-syntax mnemonic identifiers must not begin in the leftmost column.

Operand List

The syntax rules governing the operand list field is dependent on the identifier specified in the mnemonic field. For example, in the "push" instruction shown earlier in this section, the operand list field contains a list of one or more registers enclosed in braces, whereas the operand field of a .global directive expects a legal symbol identifier.

You can find more information about the Arm instruction set in the Arm Developer's Instruction Set Architecture page. Legacy TI-syntax assembler directives are described in the Arm Assembly

---

Language Tools User's Guide. More information about GNU-syntax Arm assembler directives can be found in an up-to-date description of the GNU as Arm assembler.

Comments

When migrating assembly language source from legacy TI-syntax to GNU-syntax, you'll need to modify the way that comments are delimited in your code. The syntax rules governing the demarcation of comments are significantly different between legacy TI-syntax and GNU-syntax Arm assembly language.

Legacy TI-Syntax Comment Delimiters

In legacy TI-syntax assembly source, comments can be delimited in two ways:

- Text appearing after an asterisk, '*', in column 0 is interpreted as a comment.

- Text appearing after a semi-colon, ';', on any column is interpreted as a comment.

The following snippet of legacy TI-syntax Arm assembly code demonstrates the use of these two methods of delimiting comments:

```
* Loop entry
loop_entry:
        BL      ef1             ; call ext func 1, ef1
        BL      ef2             ; call ext func 1, ef1
        LDR     A1, [SP, #0]
        ADDS    A1, A1, #1      ; I++ (A1)
        STR     A1, [SP, #0]
        LDR     A1, $C$CON1
        LDR     A2, [SP, #0]    ; load I (A2)
        LDR     A1, [A1, #0]    ; load ext var, evar (A1)
        CMP     A1, A2          ; I > evar?
        BGT     loop_entry      ; I < evar, go to loop_entry

* Loop exit
loop_exit:
        MOVS    A1, #0
        POP     {A4, PC}
```

GNU-Syntax Comment Delimiters

In GNU-syntax assembly source, comments can be delimited using:

- C-style comments; text enclosed between "/*" and "*/" which may span multiple lines.

- C++-style comments; text appearing after "//" on a line.

- Text appearing after an at-sign, '@', is interpreted as a comment unless that '@' character appears in a macro definition preceded by a backslash ''. For more details about how to convert macro definitions from legacy TI-syntax to GNU-syntax please see the *Converting TI-syntax Assembly Macros into GNU-syntax Assembly Macros* section.

Now consider the following snippet of GNU-syntax assembly code, which implements the same instructions as the above legacy TI-syntax example. Note the use of C- and C++-style comments compared to the above example:

```
/*
 * Loop entry - comment can span multiple lines
 */
loop_entry:
        bl      ef1                     // call ext func 1, ef1
        bl      ef2                     // call ext func 1, ef1
        ldr     r0, [sp]
        adds    r0, #1                  // I++ (r0)
        str     r0, [sp]
        movw    r1, :lower16:evar
        movt    r1, :upper16:evar
        ldr     r1, [r1]                // load evar (r1)
        cmp     r0, r1                  // I > evar?
        blt     loop_entry              // I < evar, go to loop_entry

/* Loop exit */
loop_exit:
        movs    r0, #0
        pop     {r7, PC}
```

Converting TI-Syntax Arm Instructions to GNU-Syntax Arm Instructions

By and large, Arm assembly language instructions can be assembled and encoded by both the armcl and the tiarmclang compiler toolchains without modifying the way the instruction is written in an assembly source file.

For example, the following sequence of Arm instructions will assemble successfully with either the integrated tiarmclang Arm assembler or the legacy TI-syntax Arm assembler (armasm):

```
my_memcpy:
            PUSH     {r4, lr}
            MOVS     r3, #0
_loop:      CMP      r2, r3
            IT       eq
            POPEQ    {r4, pc}
            LDRB     r4, [r1, r3]
            STRB     r4, [r0, r3]
            ADDS     r3, #1
            B        _loop
```

However, when converting legacy TI-syntax Arm assembly language instructions into GNU-syntax Arm assembly instructions, some minor differences may come into play . . .

Labels

For GNU-syntax Arm assembly language, labels must begin in column 0 and have a colon (':') appended to them when assembled using the tiarmclang integrated assembler.

In legacy TI-syntax Arm assembly language a label must also begin in column 0, but the colon (':') appended to the end of the label name is optional in legacy TI-syntax Arm assembly language.

The value assigned to the label name symbol that is followed by an Arm instruction will be given the value of the address where that instruction is placed at link-time.

Mnemonics

For GNU-syntax Arm assembly language, instruction mnemonic fields may begin in column 0, whereas in legacy TI-syntax Arm assembly language, the instruction mnemonic field cannot begin in column 0. You'll note that since the rule on where the mnemonic field can begin for a legacy TI-syntax Arm instruction is more restrictive, it is safe to assume that an instruction mnemonic from a legacy TI-syntax Arm assembly source file will be portable to GNU-syntax Arm assembly.

Register Names

The legacy TI Arm syntax assembler recognizes some alternative names for common Arm registers. Listed below are the legacy TI-syntax alternative names and the common names for the Arm registers that they map to. When converting legacy TI-syntax Arm assembly into GNU-syntax Arm assembly, you may need to convert references to these alternate names to their common names:

- Legacy TI-syntax: **a1 - a4** -> GNU-syntax: **r0 - r3**

- Legacy TI-syntax: **v1 - v9** -> GNU-syntax: **r4 - r12**

- Legacy TI-syntax: **ap** -> GNU-syntax: **r7**

- Legacy TI-syntax: **ip** -> GNU-syntax: **r12**

Arm Instruction Set

You can find more information about the Arm instruction set in the Arm Developer's Instruction Set Architecture page.

Converting TI-Syntax Assembly Directives to GNU-Syntax Assembly Directives

While the code actually executed by applications is made up of assembly language instructions, an assembly language source file makes extensive use of directives to create code sections and data sections that can be manipulated at link time to bind code and data to specific target memory locations. Directives are also used to define symbols, define data-objects, and communicate debug information via the object file created by the assembler.

As described previously, instructions written in legacy TI-syntax are likely to be relatively portable to GNU-syntax. However, there are extensive differences in the directives used by legacy TI-syntax and GNU-syntax. The majority of the effort required to migrate a legacy TI-syntax assembly language source file to GNU-syntax will likely involve modifying directives.

This section addresses the problem of transforming legacy TI-syntax assembly directives to be GNU-syntax compatible. First, general differences in the fields of a directive are discussed. These differences apply to one or more directives.

Second, each category of directives available in legacy TI-syntax Arm assembly language will be presented with guidance about how to convert a given legacy TI-syntax directive into its functionally equivalent GNU-syntax counterpart, if one exists.

Finally, the last two sections will present legacy TI-syntax directives that do not have a functionally equivalent GNU-syntax counterpart.

Anatomy of a Directive

Labels

For both legacy TI-syntax and GNU-syntax Arm assembly language, the label field begins in the leftmost column and contains a legal identifier that becomes the name of a symbol that will be assigned a value. In the context of a directive, the meaning of the label symbol and the value assigned to it depend upon the semantics of the directive that the label is associated with.

In legacy TI-syntax, symbols specified in the label field are not required to be terminated with a colon (':'). Several directives in the legacy TI-syntax require that a symbol be specified in the label field on the same line as the directive. For example, the ".set" directive assigns a value to a symbol

where the symbol to be defined is specified in the label field and the value that is assigned to the symbol is specified as the single operand in the operand field:

```
define_me    .set    10
```

In this case, the "define_me" symbol specified in the label field cannot be placed on a separate line from the .set directive in legacy TI-syntax.

GNU-syntax supports a directive identified with the ".equ" mnemonic identifier, but with different syntax rules for the operand field. The above ".set" legacy TI-syntax directive is equivalent to the following GNU-syntax directive:

```
.equ    define_me, 10
```

Instead of a single operand indicating the value to be assigned to "define_me", the GNU-syntax ".equ" directive requires two operands in the operand field. The first specifies the symbol name being defined, and the second provides the value to be associated with the symbol.

In GNU-syntax, a mnemonic field that begins in the leftmost column is distinguished from a label field that must begin in the leftmost column of an assembly source line by the requirement that a label field identifier be terminated with a colon (':'). Consequently, in GNU-syntax, the sole purpose of a label is to associate an address value with the symbol name specified in the label field. While GNU-syntax directives do not require a label, some legacy TI-syntax directives do. If a legacy TI-syntax directive is described as having a symbol preceding the mnemonic field, then it can be assumed that the symbol is to be defined in the label field and is required according to legacy TI-syntax rules.

Mnemonics

In both the legacy TI-syntax and GNU-syntax assembly languages, directives can be easily distinguished from instructions because the mnemonic identifier associated with a directive begins with period ('.') and instructions do not.

The major difference between a directive mnemonic in legacy TI-syntax and a directive mnemonic in GNU-syntax is that a mnemonic in legacy TI-syntax may **not** begin in the leftmost column of a line of assembly. GNU-syntax allows a mnemonic field to begin in the leftmost column of an assembly source line as long as the identifier specified in the mnemonic field is not terminated with a colon (':'). If the identifier starts in the leftmost column and is terminated with a colon (':'), then tiarmclang interprets that identifier as a label specification.

Operands

Symbols

In assembly language a symbol represents some value. If the value is known at assembly time, the symbol becomes an alias for that value. This is useful if the value of a symbol is dependent on a configuration. For example, you might use conditional directives to define a symbol whose value is dependent on which variant of the Arm architecture you are assembling for:

```
.if        __ARM_ARCH == 7
.equ       my_coeff, 10
.else
.equ       my_coeff, 1
.endif
```

Note that GNU-syntax Arm assembly language supports the same list of ACLE pre-defined symbols that are supported in C/C++ by the tiarmclang compiler.

Perhaps the most common way that symbols are defined and referenced in assembly source is as representatives for a particular address in memory. In such cases, a symbol is usually defined by a label. The label is associated with a particular location in memory that may not be known until link-time, and so references to the label symbol are typically accompanied by a relocation entry that helps the linker resolve the final value of the symbol at link time and patch each reference to the symbol according to the context in which the label symbol was referenced.

Immediates

Unlike instruction syntax, immediate operands for directives are not prefixed with a hash ('#'). In both legacy TI-syntax and GNU-syntax, the legality of where an immediate value is specified, as an operand or as part of an expression, depends on the directive specification in which the immediate occurs. Some directives, like GNU-syntax's ".balign", require an operand that evaluates at assembly time to be an integer constant. In the following GNU-syntax ".org" directive, an immediate is specified in the context of an expression to allocate space that can be accessed through the label symbol "start_addr":

```
start_addr:
        .org    start_addr + 0x100
```

Built-In Functions and Operators

The legacy TI-syntax supports a rich set of built-in functions and operators detailed in the Arm Assembly Language Tools User's Guide. These are trigonometric and other arithmetic and symbolic functions that can be evaluated by the legacy TI-syntax assembler at assembly time and resolve to a legal value for the context in which they occur.

GNU-syntax does not support the built-in functions and operators that are available to the legacy TI-syntax assembler.

TI-Syntax Assembly Directives and Their GNU-Syntax Counterparts

In the subsections below, a group of commonly occurring legacy TI-syntax directives is listed along with guidance about how each legacy TI-syntax directive can be converted into a functionally equivalent GNU-syntax directive that is compatible with the tiarmclang's integrated assembler.

Section Directives

Anything that requires space - be it code, read-only data, or read-write data - will be contained in the assembly language's notion of a section. While the migration of legacy TI-syntax initialized sections to a GNU-syntax representation is reasonably straightforward, the migration of uninitialized sections in legacy TI-syntax to a functional equivalent in GNU-syntax is more involved.

In legacy TI-syntax assembly source, a user can allocate space and define a label symbol whose value is the starting address of that allocated space all with a single directive. The following assembly source contains a legacy TI-syntax .usect directive:

```
        .text
        nop
XYZ     .usect ".bss:XYZ", 8, 4
        nop
```

This example allocates a space of 8 bytes on a 4-byte boundary within a section called ".bss:XYZ". It also defines a symbol called "XYZ" whose value is the address of the first byte in the ".bss:XYZ" section that is allocated by this directive. Since the space allocated by the .usect directive contains no data, it is called an "uninitialized" section in the legacy TI-syntax. The GNU-syntax functional equivalent of the above .usect directive is a sequence of multiple assembly source lines:

```
        .text
        nop
        .section .bss.XYZ, "aw", %nobits
        .p2align 2
XYZ:
        .zero 8
        .text
        nop
```

There are several interesting characteristics of this migration example to take note of:

- GNU-syntax does not make a distinction between initialized and uninitialized sections via the directive mnemonic name like legacy TI-syntax does (e.g. .sect vs. .usect). Instead, the section type flag operand, "%nobits", indicates to the assembler that what follows the .section directive only represents space that is contained in the ".bss.XYZ" section and implies that the load image of the ".bss.XYZ" section is not initialized with any data.

- The "aw" section flags indicate that the ".bss.XYZ" section is allocatable and writable

- While the legacy TI-syntax's .usect directive uses both a size and alignment operand to indicate the details of the space allocated, this functionality is carried out with the lines after the .section directive in GNU-syntax:

    – The GNU-syntax .p2align directive explicitly aligns the ".bss.XYZ" section's program counter to the next 4-byte boundary.

    – The "XYZ:" label defines a symbol whose value is the newly aligned ".bss.XYZ" section program counter.

    – The .zero 8 directive then marks off 8 bytes of space in the section, the first byte of which is pointed to by XYZ.

- Another characteristic of the legacy TI-syntax's .usect directive is that it acts as a temporary escape from the current section (.text in the example), allocates space into the ".bss:XYZ" section, and then implicitly returns the assembler to the section that it was in when the .usect was encountered. (In this example, the assembler implicitly returns to the .text section after processing the .usect directive.) In contrast, the GNU-syntax's .section directive instructs the assembler to start assembling into the ".bss.XYZ" section, but it requires an explicit .text directive to instruct the assembler to return to assembling instructions into the .text section.

- **.bss** -> **.section**

    legacy TI-syntax

    ```
    .bss <symbol>, <size>, <alignment>, <offset>
    ```

    GNU-syntax

    ```
            .section .bss, "aw", %nobits
            .p2align <log2(<alignment>)> <subsection␣
    ↪number>
    <symbol>:
            .zero    <size>
            <section directive returns assembler to␣
    ↪previous section>
    ```

    Allocate space in an uninitialized section named ".bss" for a specified symbol with a specified size and alignment. In a section directive, the name of the base section can be annotated with a subsection name. For example, you might create a section name ".bss.myvar" for a symbol called "myvar" as follows:

    ```
            .text
            nop
            .section .bss.myvar, "aw", %nobits
            .p2align 4
    ```
    (continues on next page)

```
myvar:
        .zero    8
        .text
        nop
```

Alternate GNU-syntax

```
        .bss <subsection number>
        .p2align <log2<alignment>)> <subsection
→number>
<symbol>:
        .zero    8
        <section directive returns assembler to
→previous section>
```

The ".bss" directive is also supported in GNU-syntax, but is less flexible than the ".section" directive in that the ".bss" directive only allows an integer value to be specified as the ".bss" subsection name. For example,

```
        .text
        nop
        .bss     1
        .p2align 4
myvar:
        .zero    8
        .text
        nop
```

creates a ".bss.1" section containing the definition of "myvar".

- **.common -> .comm**

  legacy TI-syntax

  ```
  .common <symbol>,<size>[,<alignment>]
  ```

  GNU-syntax

  ```
  .comm   <symbol>,<size>[,<alignment>]
  ```

  The .comm directive will allocate space for a variable in a "common" block as opposed to placing the variable in a section. The compiler uses .comm to define an uninitialized file scope variable. If the same variable is defined and uninitialized in another source file in an application, it will also be defined in a common block and the linker will resolve the definitions in multiple common blocks to a single location in memory.

- **.usect -> .section**

    legacy TI-syntax

    ```
    <symbol> .usect "<section name>", <size>, <alignment>
    ↪, <offset>
    ```

    GNU-syntax

    ```
            .section <section name>, "aw", %nobits
            .p2align <log2(<alignment>)>
    <symbol>:
            .zero    <size>
            <section directive returns assembler to␣
    ↪previous section>
    ```

    Allocate space in an uninitialized user-named section containing the definition of a specified data object represented by the specified symbol with a specified size and alignment.

- **.sect -> .section**

    legacy TI-syntax

    ```
    .sect "<section name>"[, <rw flag>[, <alloc flag>]] ␣
    ↪         |
    ```

    GNU-syntax

    ```
    .section <section name>, "<scn_flags>", %progbits
    ```

    GNU-syntax does not distinguish between initialized sections and uninitialized sections via the mnemonic identifier as legacy TI-syntax does (via .sect and .usect). Instead, the GNU-syntax .section directive uses the optional operands after "section name" to indicate characteristics about the section to assemble subsequent lines into.

    scn_flags - is a string that can contain up to 3 distinct characters:

    'a' - section is allocatable. 'w' - section is writable. This is typically used for a section that contains data objects whose values may change during the run of an application. 'x' - section is executable. This simply means that the section contains executable code, the section may also include data.

    scn_type - indicates whether or not the section contains data (needs to be loaded). There are two legal type specifications:

    'progbits' - section contains data. This characterizes the section as "initialized". 'nobits' - section contains no data and only occupies space. This characterizes the

section as "uninitialized".

- **.data**

    legacy TI-syntax and GNU-syntax

    ```
    .data
    ```

    Begin assembling into an initialized section named ".data".

- **.text**

    legacy TI-syntax and GNU-syntax

    ```
    .text
    ```

    Begin assembling into an initialized section named ".text".

Alignment Directives

- **.align** -> **.align** and other alternatives

    legacy TI-syntax and GNU-syntax

    ```
    .align <alignment>
    ```

    Advance the current section program counter to the next specified alignment boundary. The specified alignment boundary operand must resolve to a power of 2.

    Some GNU-syntax alternatives

    ```
    .balign[w|l]  <alignment>[,<fill>[,<max skip count>]]
    .p2align[w|l] <exponent>[,<fill>[,<max skip count>]]
    ```

    The .balign and .p2align directives, like .align, will advance the current section to the next specified alignment boundary (for .palign the alignment boundary is expressed as 2 to the exponent power), but only if the number of bytes to be skipped to get to that boundary is less than or equal to the max skip count. The fill value operand is optional. If specified, it indicates the byte value to be inserted into any padding space that is created by the alignment directive.

Data-Defining and Alignment Directives

- **.[u]byte**, **.[u]char** -> **.byte**

    legacy TI-syntax variants

    ```
    .byte  <value1>[, ... <valueN>]
    .ubyte <value1>[, ... <valueN>]
    .char  <value1>[, ... <valueN>]
    .uchar <value1>[, ... <valueN>]
    ```

    GNU-syntax

    ```
    .byte  <value1>[, ... <valueN>]
    ```

    Place 8-bit integer encodings of each operand into the current section. Specified expression operands must evaluate to an integer in the range [-127, 255].

- **.[u]half**, **.[u]short** -> **.hword**, **.short**

    legacy TI-syntax variants

    ```
    .half  <value1>[, ... <valueN>]
    .uhalf <value1>[, ... <valueN>]
    .short <value1>[, ... <valueN>]
    .ushort <value1>[, ... <valueN>]
    ```

    GNU-syntax

    ```
    .hword <value1>[, ... <valueN>]
    .short <value1>[, ... <valueN>]
    ```

    Place 16-bit integer encodings of each expression operand into the current section. Specified expression operands must evaluate to an integer in the range [-32767, 65535].

- **.[u]int**, **.[u]long**, **.[u]word** -> **.int**, **.long**, **.word**

    legacy TI-syntax variants

    ```
    .int   <value1>[, ... <valueN>]
    .uint  <value1>[, ... <valueN>]
    .long  <value1>[, ... <valueN>]
    .ulong <value1>[, ... <valueN>]
    .word  <value1>[, ... <valueN>]
    .uword <value1>[, ... <valueN>]
    ```

    GNU-syntax

```
.int    <value1>[, ... <valueN>]
.long   <value1>[, ... <valueN>]
.word   <value1>[, ... <valueN>]
```

Place 32-bit integer encodings of each expression operand into the current section. Specified expression operands must evaluate to an integer in the range [-2147483647, 4294967295].

- **.float** and **.double**

  legacy TI-syntax and GNU-syntax

  ```
  .float  <value1>[, ... <valueN>]
  .double <value1>[, ... <valueN>]
  ```

  GNU-syntax also supports an alternative to .float

  ```
  .single <value1>[, ... <valueN>]
  ```

  The .float and .single directives will place 32-bit IEEE-754 single precision encodings of each floating-point constant operand into the current section.

  The .double directive will place 64-bit IEEE-754 double- precision encodings of each floating-point constant operand into the current section.

- **.cstring**, **.string** -> **.ascii**, **.asciz**, **.string**

  legacy TI-syntax variants

  ```
  .cstring <expr1>|"<string1>"[, ... <exprN>|"<stringN>
  ↪"]
  .string  <expr1>|"<string1>"[, ... <exprN>|"<stringN>
  ↪"]
  ```

  GNU-syntax variants

  ```
  .ascii   "<string1>"[, ... "<stringN>"]
  .asciz   "<string1>"[, ... "<stringN>"]
  .string  "<string1>"[, ... "<stringN>"]
  ```

  Place 8-bit ASCII encodings of each character from each string operand into the current section. Legacy TI-syntax also allows an operand to be specified as an 8-bit integer constant value.

  The .cstring/.asciz directives differ from the .string/.ascii directives in that the assembler inserts a terminating NUL character ('0') at the end of each string operand for the .cstring/.asciz directives.

Space Reserving Directives

- **.bes** -> **.space**

  legacy TI-syntax

  ```
  <symbol> .bes <size>
  ```

  GNU-syntax nearly functional equivalent

  ```
          .space <size>, 0x0
  <symbol>:
  ```

  The legacy TI-syntax's .bes directive reserves size bytes in the current initialized section, filling that space with zeros. If a label is specified, the value assigned to the label symbol will be the address of the last byte reserved by the directive.

  Converting the legacy TI-syntax .bes directive involves both a .space directive and a label definition following the .space directive. Please note, however, that in the GNU-syntax conversion, the value assigned to the label will be one byte past the end of the reserved space as opposed to the address of the last byte in the reserved space as is the case with the .bes directive.

- **.space**

  legacy TI-syntax

  ```
  <symbol>   .space <size>
  ```

  GNU-syntax

  ```
  <symbol>: .space <size>,0x0
  ```

  As shown here, both the legacy TI-syntax and the GNU- syntax .space directives are functionally equivalent. The directive reserves size bytes in the current initialized section, filling that space with zeros. If a label is specified, the value assigned to the label symbol will be the address of the first byte reserved by the directive.

  Note that in the GNU-syntax version the colon (':') appended to the symbol in the label field is required to have the symbol interpreted as a label.

Directives that Change the Instruction Type

- **.arm**, **.state32** -> **.arm** + **.p2align 2**

  legacy TI-syntax and GNU-syntax for .arm

  ```
  .arm
  ```

  legacy TI-syntax supports an alias for .arm

  ```
  .state32
  ```

  The legacy TI-syntax version of the .arm and .state32 directives perform two actions:

  1) instruct the assembler to interpret instructions that follow the .arm directive as 32-bit Arm instructions, and

  2) align current section's program counter to a 4-byte boundary

  The GNU-syntax .arm directive does not perform the 4-byte alignment action. Therefore, in order to perform a functionally equivalent migration of the TI-syntax .arm directive, you should map the TI-syntax .arm directive to the following sequence of GNU-sytax directives:

  ```
  .arm
  .p2align 2
  ```

  GNU-syntax also supports an alias for the .arm directive, .code 32. Thus, the above GNU-syntax sequence is equivalent to the following:

  ```
  .code    32
  .p2align 2
  ```

  When used by itself, the GNU-syntax .arm directive performs a single action: instruct the assembler to interpret instructions that follow the .arm directive as 32-bit Arm instructions.

- **.thumb**

  legacy TI-syntax and GNU-syntax for .thumb

  ```
  .thumb
  ```

  GNU-syntax also supports an alias for .thumb

  ```
  .code 16
  ```

The effect of the .thumb directive is to instruct the assembler to interpret instructions that follow the .thumb directive as T32 instructions.

Copy/Include Directives

- **.copy** -> **.include**

  legacy TI-syntax

  ```
  .copy     <file>
  ```

  GNU-syntax nearly functional equivalent

  ```
  .include "<file>"
  ```

  The difference between legacy TI-syntax's .copy directive and its .include directive is that the .copy directive will display the contents of the referenced file in a generated assembly listing file at the location where the .copy directive occurred in the assembly source, the .include directive does not display the contents of the referenced file in an assembly listing file.

  The GNU-syntax .include directive behaves like the legacy TI-syntax directive and is the only viable alternative for translating a legacy TI-syntax .copy directive into something functionally equivalent. Note that the GNU-syntax .include directive requires that the file operand be enclosed in double-quotes.

- **.include**

  legacy TI-syntax

  ```
  .include <file>
  ```

  GNU-syntax

  ```
  .include "<file>"
  ```

  The GNU-syntax .include directive behaves like the legacy TI-syntax directive in that they both allow you to include the specified file at a specific point in the assembly source file. Note that the GNU-syntax .include directive requires that the file operand be enclosed in double-quotes.

Symbol Definition Directives

- **.equ**, **.set**

    legacy TI-syntax variants

    ```
    <symbol> .equ <value>
    <symbol> .set <value>
    ```

    GNU-syntax variants

    ```
    .equ <symbol>, <value>
    .set <symbol>, <value>
    ```

    Both the legacy TI-syntax and GNU-syntax versions of the .equ and .set directives
    are functionally equivalent. They define a symbol and assign it a value. However,
    the syntax is different. In legacy TI-syntax, the symbol is specified in the label
    field, and in GNU-syntax, the symbol is specified as the first operand.

Symbol Linkage/Visibility Directives

- **.global**

    legacy TI-syntax and GNU-syntax

    ```
    .global <symbol1>[, ... <symbolN>]
    ```

    The .global directive identifies one or more symbols to be externally visible (ex-
    ported) if defined in this module, or identifies one or more symbols to be acces-
    sible to this module (imported) if defined in an external module. Each symbol on
    the symbol list will be given STB_GLOBAL binding.

- **.weak**

    legacy TI-syntax and GNU-syntax

    ```
    .weak <symbol1>[, ... <symbolN>]
    ```

    The .weak directive identifies one or more symbols used in the current module that
    if strongly defined in another module will yield to the strong definition. Symbols
    specified in the sym_list operand of the .weak directive will be given STB_WEAK
    binding.

Conditional Assembly Directives

- **.if**

    legacy TI-syntax and GNU-syntax

    ```
    .if <condition>
    <true block>
    .endif
    ```

    Assemble the subsequent block of assembly source lines if the specified condition
    evaluates to a non-zero integer constant at assembly time.

- **.elseif** -> **.elseif**

    legacy TI-syntax

    ```
    .if <condition1>
    <true block1>
    .elseif <condition2>
    <true block2
    .else
    <false block>
    .endif
    ```

    GNU-syntax

    ```
    .if <condition1>
    <true block1>
    .elseif <condition2>
      <true block2>
    .else
      <false block>
    .endif
    ```

    The legacy TI-syntax assembler supports the .elseif directive, which behaves in
    assembly language similarly to "else if" in C/C++ source code. It creates an
    alternate block of assembly lines to be assembled when the condition expression
    associated with the opening .if directive is 0 and the condition associated with the
    .elseif directive evaluates to a non-zero integer constant.

    GNU-syntax also supports the .elseif directive. You can get equivalent function-
    ality using a nested .if/.endif as follows:

    ```
    .if <condition1>
    <true block1>
    ```

```
.else
  .if <condition2>
    <true block2>
  .else
    <false block>
  .endif
.endif
```

- **.else**

    legacy TI-syntax and GNU-syntax

```
.if <condition>
<true block>
.else
<false block>
.endif
```

    When the condition of the preceding conditional directives evaluate to 0, assemble
    the block of assembly source lines that follow the .else directive.

- **.endif**

    legacy TI-syntax and GNU-syntax

```
.if <condition>
...
.endif
```

    The .endif directive marks the end of a sequence of one or more conditionally
    assembled blocks of assembly source lines.

- **.if $defined(<symbol>) -> .ifdef <symbol>**

    legacy TI-syntax

```
.if $defined(<symbol>)
...
.endif
```

    GNU-syntax

```
.ifdef <symbol>
...
.endif
```

Assemble the subsequent block of assembly source lines if the specified symbol is defined.

Note that the legacy TI-syntax equivalent to GNU-syntax's .ifdef directive makes use of the $defined operator that is available in the legacy TI-syntax assembler.

- **.if !$defined(<symbol>) -> .ifndef <symbol>**

  legacy TI-syntax

  ```
  .if !$defined(<symbol>)
  ...
  .endif
  ```

  GNU-syntax

  ```
  .ifndef <symbol>
  ...
  .endif
  ```

  Assemble the subsequent block of assembly source lines if the specified symbol is **not** defined.

  Note that the legacy TI-syntax equivalent to GNU-syntax's .ifndef directive makes use of the $defined operator that is available in the legacy TI-syntax assembler.

- **.loop/.endloop -> .rept/.endr**

  legacy TI-syntax

  ```
  .loop <count>
  ...
  .endloop
  ```

  GNU-syntax

  ```
  .rept <count>
  ...
  .endr
  ```

  The legacy TI-syntax's .loop and the GNU-syntax's .rept directive mark the beginning of a sequence of assembly lines to be repeated count times. The end of the sequence is marked by the legacy TI-syntax's .endloop directive and the GNU-syntax's .endr directive. The count must evaluate to an integer constant value at assembly time.

Assembly Macro Related Directives

Information about how to migrate assembly language macros written in legacy TI-syntax to GNU-syntax can be found in the *Converting TI-syntax Assembly Macros into GNU-syntax Assembly Macros* section.

- **.macro**

    legacy TI-syntax

    ```
    <mname>   .macro [<param1>[, ... <paramN>]]
    ```

    GNU-syntax

    ```
    .macro <mname>[, <param1>[, ... <paramN>]]
    ```

    Mark the start of a macro definition with a specified name (*mname* in above examples). Note that in the legacy TI-syntax, the macro name must be specified in the label field, but in GNU-syntax, the macro name is specified as the first operand of the .macro directive.

- **.mexit -> .exitm**

    legacy TI-syntax

    ```
    .mexit
    ```

    GNU-syntax

    ```
    .exitm
    ```

    The GNU-syntax .exitm directive allows you to exit early from within a current macro definition.

- **.endm**

    legacy TI-syntax and GNU-syntax

    ```
    .endm
    ```

    The .endm directive marks the end of a macro definition.

Linker Information Directives

- **.bound -> .sym_meta_info (for location)**

    legacy TI-syntax

    ```
    .bound "<section name>", <address>
    ```

    GNU-syntax

    ```
    .sym_meta_info <symbol>, "location", <address>
    ```

    Both the armcl and the tiarmclang compilers support the location attribute which allows you, in your C/C++ source file, to indicate the memory address where you would like a function definition or data object to be placed in memory.

    The armcl compiler will generate a .bound directive that will instruct the linker to place a specific section (containing the definition of the function or data object) at a specific address in target memory.

    Similarly, the tiarmclang uses its .sym_meta_info "location" directive to instruct the linker to place a specific symbol, representing a function or data object definition, at a specific address in target memory.

- **.retain -> .sym_meta_info or .no_dead_strip**

    legacy TI-syntax

    ```
    .retain "scn"
    ```

    GNU-syntax

    ```
    .sym_meta_info <sym>,"retain",1
    .no_dead_strip <sym>
    ```

    The tiarmclang integrated assembler supports two directives that can be used to serve the same purpose as the legacy TI-syntax's .retain directive. Instead of instructing the linker to retain a specific section, the .sym_meta_info and .no_dead_strip directives instruct the linker to retain the section containing the definition of the identified symbol (sym).

TI-Syntax Assembly Directives Without a GNU-Syntax Counterpart

The following legacy TI-syntax directives do not have a functionally equivalent counterpart in the GNU-syntax Arm assembly language. For more information about the legacy TI-syntax directives and what they do, please consult the Arm Assembly Language Tools User's Guide.

Absolute Listing Directives

- **.setsect**
- **.setsym**

Assembly Macro Related Directives

- **.mlib**
- **.var**

Assembly Source Debug Directives

- **.asmfunc**
- **.endasmfunc**

Conditional Assembly Directives

- **.break**

Data-Defining Directives

- **.bits**
- **.field**

Embedded Compiler Options Directive

- **.compiler_opts**

Group Directives

- **.group**
- **.gmember**
- **.endgroup**

Linker Information Directives

- **.label**
- **.retainrefs**

Object Listing Format Directives

- **.drlist**
- **.drnolist**
- **.fclist**
- **.fcnolist**
- **.length**
- **.list**
- **.mlist**
- **.mnolist**
- **.nolist**
- **.option**
- **.page**
- **.sslist**
- **.ssnolist**
- **.tab**
- **.title**
- **.width**

Structure/Union Type Definition Directives

- **.cstruct**
- **.struct**
- **.endstruct**
- **.cunion**
- **.union**
- **.endunion**
- **.member**
- **.tag**
- **.enum**

Substitution Symbol Manipulation Directives

- **.asg**
- **.define**
- **.eval**
- **.unasg**
- **.undefine**

Symbol Linkage/Visibility Directives

- **.def**
- **.ref**
- **.symdepend**

User-Defined Diagnostic Directives

- **.emsg**
- **.wmsg**
- **.mmsg**

Miscellaneous Directives

- **.cdecls**

- **.end**

- **.newblock**

Converting TI-syntax Assembly Macros into GNU-syntax Assembly Macros

If you are in the process of converting your TI-syntax Arm assembly source into GNU-syntax Arm assembly source and you utilize assembly language macros in your source code, then you will need to explicitly translate those TI-syntax macros into their GNU-syntax equivalent. This section of the migration guide will provide information about how to perform that translation.

Macro Directive and Macro Definitions

Both the legacy TI-syntax Arm assembler and the tiarmclang GNU-syntax Arm assembler support a .macro directive used to define an assembly macro, but the TI-syntax form of a macro definition is different from the GNU-syntax form.

Specifically, the TI-syntax form of the .macro directive and a macro definition looks like this:

```
<macro name> .macro [<parameter1>[, ..., <parameterN>]]
            <model assembly statements or macro-related␣
↪directives>
            .endm
```

where:

- **<macro name>** - is the name of the macro. After a macro has been defined, its name can then be used in the mnemonic field of an assembly statement to effect an invocation of the macro. For the TI-syntax form, the macro name must be specified in the label field of an assembly source statement prior to the .macro directive mnemonic.

- **.macro** - is the assembly directive that identifies the assembly source statement as the first line of a macro definition. For the TI-syntax form, it must appear in the mnemonic field after the macro name.

- **<parameter(s)>** - are optional identifiers that serve as vehicles for operand arguments that are passed into a macro when a macro is invoked. Each parameter takes on the value of the argument that is specified in the macro invocation at the corresponding operand position.

- **<model assembly statements>** - are instructions or directives that are executed each time the macro is invoked.

- **<macro-related directives>** - are assembly directives that can be used to manipulate the values that are maintained in the macro parameters.

- **.endm** - is an assembler directive that delimits the end of the current macro definition.

The GNU-syntax form of a .macro directive and macro definition looks like this:

```
.macro <macro name>[, <parameter1>[, ..., <parameterN>]]
<model assembly statements>
.endm
```

where:

- **.macro** - is the assembly directive that identifies the assembly source statement as the first line of a macro definition. For the GNU-syntax form, it must appear in the mnemonic field before the macro name.

- **<macro name>** - is the name of the macro. After a macro has been defined, its name can then be used in the mnemonic field of an assembly statement to effect an invocation of the macro. For the GNU-syntax form, the macro name must be specified as the first operand of the .macro directive.

- **<parameter(s)>** - are optional identifiers that serve as vehicles for operand arguments that are passed into a macro when a macro is invoked. Each parameter takes on the value of the argument that is specified in the macro invocation at the corresponding operand position. For the GNU-syntax form, the use of commas to separate operands of the .macro directive are optional as long as there is at least one blank space between each operand and the next.

- **<model assembly statements>** - are instructions or directives that are executed each time the macro is invoked.

- **.endm** - is an assembler directive that delimits the end of the current macro definition.

Referencing Macro Parameters and Local Labels

Within a macro definition, macro parameters can be referenced in the context of an assembly instruction, a directive, or an operand to an instruction or directive. In addition, local labels can be defined and referenced within the scope of a given macro definition.

Consider the definition of an assembly macro named COMPARE64 specified in TI-syntax form:

```
/* COMPARE64 macro definition - legacy TI-syntax
COMPARE64 .macro x_hi, x_lo, y_hi, y_lo
        CMP x_hi, y_hi
        BNE $?
        CMP x_lo, y_lo
$?:
        .endm
```

An invocation of the above COMPARE64 macro:

```
; COMPARE64 macro invocation
COMPARE64 r3, r2, r1, r0
```

will expand into the following sequence of assembly source:

```
; COMPARE64 macro invocation
        CMP r3, r1
        BNE $$1$
        CMP r2, r0
$$1$:
```

Note that the values passed in as argument operand to the macro invocation, in this case register names, get propagated to where the parameters are referenced in the context of the macro definition. Each reference to a parameter in the macro definition will expand to its corresponding argument value when the macro is expanded during invocation. The local label specification, indicated by "$?" within the macro definition expands to "$$1$" in the expansion of the macro.

If the macro definition is written in GNU-syntax form:

```
/* COMPARE64 macro definition - GNU-syntax
        .macro COMPARE64, x_hi, x_lo, y_hi, y_lo
        CMP \x_hi, \y_hi
        BNE .L_\@
        CMP \x_lo, \y_lo
.L_\@:
        .endm
```

you'll notice that parameter references must be delimited with a leading backslash ('') character in the context of the macro definition. Similarly, the local label is indicated with ".L_@" in the macro definition where the "@" part of the label name instructs the assembler to generate a unique ID in place of the "@" when it is used in a label definition. The COMPARE64 macro can be invoked in the same way in both TI-syntax and GNU-syntax Arm assembly:

```
; COMPARE64 macro invocation
COMPARE64 r3, r2, r1, r0
```

and the GNU-syntax form of the macro expands into the following sequence of assembly source:

```
; COMPARE64 macro invocation
        CMP r3, r1
        BNE .L_0
        CMP r2, r0
.L_0:
```

2.5. Migrating Assembly Language Source Code

Summary

To summarize, there are three major steps for converting a TI-syntax macro definition into a Gnu-syntax macro definition:

(1) Update .macro directive usage, moving the macro name from the label field to the first operand of the .macro directive

(2) Insert a leading backslash ('') in front of all parameter references within the scope of the macro definition

(3) Update local label references within the scope of the macro definition, replacing each '?' with "@" and making sure that the prefix to the "@" forms a legal GNU-syntax label name

Predefined Assembler Symbols

The legacy TI-syntax Arm assembler provides several types of predefined symbols, many of which will need to be mapped to their GNU-syntax Arm assembler's functional equivalent when migrating TI-syntax Arm assembly source to GNU-syntax Arm assembly source.

This section of the migration guide will provide guidance on which TI-syntax Arm assembler predefined symbols can be mapped to their GNU-syntax equivalents and how to map them, also noting which of the TI-syntax Arm assembler predefined symbols to not have a functionally equivalent predefined symbol in GNU-syntax Arm assembly source.

Section Location Counter

The TI-Syntax '$' Symbol

The TI-syntax Arm assembler provides a special **$** symbol that can be used to represent the current value of the section location counter (also known as the *section program counter* or *SPC*). The **$** symbol can come in handy in TI-syntax Arm assembly source when computing a section-relative offset at assembly time.

The GNU-Syntax '.' Symbol

The tiarmclang's integrated GNU-syntax Arm assembler provides a special dot symbol (**.**) that is functionally equivalent to the TI-syntax Arm assembler's **$** symbol. The dot symbol refers to the current address of the section that tiarmclang is assembling into.

An Example Migration

In the following example, a section location counter is used to compute the size of a block of data.

```
* TI-syntax block of data
      .data
data_start:
```

```
        .int        data_end - $
        <data defining directives>
data_end:
```

We can simply replace the **$** symbol with the dot symbol to convert the block of data from TI-syntax to GNU-syntax Arm assembly:

```
* GNU-syntax block of data
        .data
data_start:
        .int        data_end - .
        <data defining directives>
data_end:
```

Predefined Symbolic Constants

The TI-syntax Arm assembler predefines a number of symbolic constants. The value of many of these symbolic constants is determined by the command-line arguments with which the assembler is invoked. For example, the *__TI_ARM_V6M0__* symbol is predefined with a value of 1 if the TI-syntax Arm assembler is invoked for a Cortex-M0 target (-mv6M0 option).

The vast majority of references to TI-syntax Arm assembler symbolic constants are made in the context of conditional directives. For example, a Cortex-M0 specific implementation of a function written in TI-syntax Arm assembly might be guarded by an ".if" conditional directive:

```
;* Cortex-M0 implementation of my_func (TI-syntax)

    .if __TI_ARM_V6M0__

    ...
    .global my_func

my_func:
    ...
    <assembly source implementation>
    ...

    .endif
```

Convert TI-Syntax Symbolic Constant References to Preprocessor Directives

While the tiarmclang's integrated GNU-syntax Arm assembler does support conditional directives like those supported in the TI-syntax Arm assembler, the GNU-syntax Arm assembler does *not*

---

support the same predefined symbolic constants as the TI-syntax Arm assembler. However, a given reference to a TI-syntax symbolic constant can be converted into a C/C++ preprocessing directive that can be embedded in a GNU-syntax Arm assembly source file that will be processed by the tiarmclang's C/C++ preprocessor before invoking the integrated GNU-syntax assembler.

Thus, the above snippet of TI-syntax Arm assembly source can be converted to GNU-syntax using a preprocessor directive in place of the conditional directives like so:

```
;* Cortex-M0 implementation of my_func (GNU-syntax)

    #if (__ARM_ARCH == 6) && (__ARM_ARCH_PROFILE == 'M')

    ...
    .global my_func

my_func:
    ...
    <assembly source implementation>
    ...

    #endif
```

You must then instruct the tiarmclang compiler to pre-process the GNU-syntax Arm assembly source file in one of two ways:

- **Use .S file extension** - Using a *.S* file extension for your GNU-syntax Arm assembly source file will instruct the tiarmclang compiler to run its preprocessor over the assembly source before passing the preprocessed assembly source on to the integrated GNU-syntax Arm assembler.

```
%> tiarmclang -mcpu=cortex-m0 -c my_func.S
```

- **Use -x assembler-with-cpp option** - If the *-x assembler-with-cpp* option is specified on the tiarmclang command-line, then tiarmclang will interpret input files that follow the *-x assembler-with-cpp* option as GNU-syntax assembly source files that will be preprocessed before being assembled.

```
%> tiarmclang -mcpu=cortex-m0 -c -x assembler-with-cpp my_func.
↪xyz
```

The remainder of this section will list the predefined symbolic constants that are supported in the TI-syntax Arm assembler and, if a given symbolic constant has a functional equivalent predefined macro symbol expression that is supported via tiarmclang's C/C++ preprocessor, then a description of how the TI-syntax symbolic constant reference can be converted into a preprocessing directive will be provided.

Most of the preprocessing directive examples below make use of predefined macro symbols that

---

are part of the Arm C Language Extensions (ACLE) support included in the tiarmclang compiler. You can find more information about ACLE in the *Arm C Language Extensions - ACLE* section of this migration guide.

- **.TI_ARM**

  The *.TI_ARM* predefined symbol is always set to 1 when the TI-syntax Arm assembler is invoked. The following reference to *.TI_ARM*:

  ```
  ;* TI-syntax reference to .TI_ARM

    .if .TI_ARM
    .int      0x1234
    .endif
  ```

  is functionally equivalent to the following GNU-syntax Arm assembly:

  ```
  // GNU-syntax conversion of .TI_ARM reference

    #if defined(__ti_version__) && (__ARM_ARCH != 0)
    .int      0x1234
    #endif
  ```

  where *__ti_version__* is a predefined macro symbol unique to tiarmclang and *__ARM_ARCH* is an ACLE representation of the Arm architecture version.

- **.TI_ARM_16BIS**

  The *.TI_ARM_16BIS* predefined symbol is set to 1 if the compiler has been instructed to generate THUMB mode instructions for an Arm processor. The following reference to *.TI_ARM_16BIS*:

  ```
  ;* TI-syntax reference to .TI_ARM_16BIS

    .if .TI_ARM_16BIS
    .int      0x1234
    .endif
  ```

  is functionally equivalent to the following GNU-syntax Arm assembly:

  ```
  // GNU-syntax conversion of .TI_ARM_16BIS reference

    #if defined(__thumb__)
    .int      0x1234
    #endif
  ```

  where *__thumb__* is a tiarmclang predefined macro symbol that indicates that the compiler has been instructed to generate THUMB mode instructions.

- **.TI_ARM_32BIS**

  The *.TI_ARM_32BIS* predefined symbol is set to 1 if the compiler has been instructed to generate 32-bit ARM mode instructions for an Arm processor. The following reference to *.TI_ARM_32BIS*:

  ```
  ;* TI-syntax reference to .TI_ARM_32BIS

    .if .TI_ARM_32BIS
    .int     0x1234
    .endif
  ```

  is functionally equivalent to the following GNU-syntax Arm assembly:

  ```
  // GNU-syntax conversion of .TI_ARM_32BIS reference

    #if !defined(__thumb__)
    .int     0x1234
    #endif
  ```

  where *__thumb__* is a tiarmclang predefined macro symbol that indicates that the compiler has been instructed to generate THUMB mode instructions. The expression "!defined(__thumb__) in this case indicates that the compiler will generate ARM mode instructions.

- **.TI_ARM_T2IS**

  The *.TI_ARM_T2IS* predefined symbol is set to 1 if an Arm version 7 architecture processor is selected and targeted to generate THUMB mode instructions (this is also referred to as the THUMB2 or T32 instruction set). The following reference to *.TI_ARM_T2IS*:

  ```
  ;* TI-syntax reference to .TI_ARM_T2IS

    .if .TI_ARM_T2IS
    .int     0x1234
    .endif
  ```

  is functionally equivalent to the following GNU-syntax Arm assembly:

  ```
  // GNU-syntax conversion of .TI_ARM_T2IS reference

    #if defined(__thumb__) && \
        (__ARM_ARCH == 7 && \
         (__ARM_ARCH_PROFILE == 'R' || \
          (__ARM_ARCH_PROFILE == 'M' && defined(__ARM_FEATURE_
  ↪SIMD32))))
  ```

  <div align="right">(continues on next page)</div>

```
    .int        0x1234
    #endif
```

where __*thumb*__ is a tiarmclang predefined macro symbol that indicates that the compiler has been instructed to generate THUMB mode instructions. The *__ARM_ARCH*, *__ARM_ARCH_PROFILE*, and *__ARM_FEATURE_SIMD32* are ACLE predefined macro symbols, the combination of which is used to identify the Cortex-M4 or a Cortex-R series Arm processor that can be selected via tiarmclang's *-mcpu=cortex-m4*, *-mcpu=cortex-r4*, or *-mcpu=cortex-r5* option.

- **.TI_ARM_LITTLE**

  The *.TI_ARM_LITTLE* predefined symbol is set to 1 if little-endian mode is selected via armcl's *-me* or *--endian=little* option. The following reference to *.TI_ARM_LITTLE*:

  ```
  ;* TI-syntax reference to .TI_ARM_LITTLE

    .if .TI_ARM_LITTLE
    .int        0x1234
    .endif
  ```

  is functionally equivalent to the following GNU-syntax Arm assembly:

  ```
  // GNU-syntax conversion of .TI_ARM_LITTLE reference

    #if defined(__ARM_LITTLE_ENDIAN)
    .int        0x1234
    #endif
  ```

  where *__ARM_LITTLE_ENDIAN* is an ACLE predefined macro symbol. *__ARM_BIG_ENDIAN* indicates that the compiler will generate little-endian object code as dictated by the tiarmclang *-mlittle-endian* option.

- **.TI_ARM_BIG**

  The *.TI_ARM_BIG* predefined symbol is set to 1 if big-endian mode is selected via armcl's *--endian=big* option (which is also the default behavior for armcl). The following reference to *.TI_ARM_BIG*:

  ```
  ;* TI-syntax reference to .TI_ARM_BIG

    .if .TI_ARM_BIG
    .int        0x1234
    .endif
  ```

  is functionally equivalent to the following GNU-syntax Arm assembly:

```
// GNU-syntax conversion of .TI_ARM_BIG reference

  #if defined(__ARM_BIG_ENDIAN)
  .int      0x1234
  #endif
```

where *__ARM_BIG_ENDIAN* is an ACLE predefined macro symbol.
__ARM_BIG_ENDIAN indicates that the compiler will generate big-endian object
code as dictated by the tiarmclang *-mbig-endian* option.

- **__TI_ARM_V6M0__**

  The *__TI_ARM_V6M0__* predefined symbol is set to 1 if the Cortex-M0 processor is targeted
  via armcl's *-mv6m0* option. The following reference to *__TI_ARM_V6M0__*:

```
;* TI-syntax reference to __TI_ARM_V6M0__

  .if __TI_ARM_V6M0__
  .int      0x1234
  .endif
```

  is functionally equivalent to the following GNU-syntax Arm assembly:

```
// GNU-syntax conversion of __TI_ARM_V6M0__ reference

  #if (__ARM_ARCH == 6 && __ARM_ARCH_PROFILE == 'M')
  .int      0x1234
  #endif
```

  where *__ARM_ARCH* and *__ARM_ARCH_PROFILE* are ACLE predefined macro symbols,
  the combination of which is used to identify the Cortex-M0 processor that can be selected
  via tiarmclang's *-mcpu=cortex-m0* option.

- **__TI_ARM_V7__**

  The *__TI_ARM_V7__* predefined symbol is set to 1 if any processor based on version
  7 of the Arm architecture is selected via an armcl option. The following reference to
  __TI_ARM_V7__:

```
;* TI-syntax reference to __TI_ARM_V7__

  .if __TI_ARM_V7__
  .int      0x1234
  .endif
```

  is functionally equivalent to the following GNU-syntax Arm assembly:

```
// GNU-syntax conversion of __TI_ARM_V7__ reference

  #if (__ARM_ARCH == 7)
  .int      0x1234
  #endif
```

where *__ARM_ARCH* is a ACLE predefined macro symbol used to identify a processor that is based on version 7 of the Arm architecture.

- **__TI_ARM_V7M3__**

The *__TI_ARM_V7M3__* predefined symbol is set to 1 if the Cortex-M3 Arm processor is targeted via armcl's *-mv7m3* option. The follow when the TI-syntax Arm assembler is invoked. The following reference to *.TI_ARM*:

```
;* TI-syntax reference to __TI_ARM_V7M3__

  .if __TI_ARM_V7M3__
  .int      0x1234
  .endif
```

is functionally equivalent to the following GNU-syntax Arm assembly:

```
// GNU-syntax conversion of __TI_ARM_V7M3__ reference

  #if (__ARM_ARCH == 7 && __ARM_ARCH_PROFILE == 'M' && \
       !defined(__ARM_FEATURE_SIMD32))
  .int      0x1234
  #endif
```

where *__ARM_ARCH*, *__ARM_ARCH_PROFILE*, and *__ARM_FEATURE_SIMD32* are ACLE predefined macro symbols, the combination of which is used to identify the Cortex-M3 processor that can be selected via tiarmclang's *-mcpu=cortex-m3* option.

- **__TI_ARM_V7M4__**

The *__TI_ARM_V7M$__* predefined symbol is set to 1 if the Cortex-M4 Arm processor is targeted via armcl's *-mv7m4* option. The following reference to *__TI_ARM_V7M4__*:

```
;* TI-syntax reference to __TI_ARM_V7M4__

  .if __TI_ARM_V7M4__
  .int      0x1234
  .endif
```

is functionally equivalent to the following GNU-syntax Arm assembly:

2.5. Migrating Assembly Language Source Code   **173**

```
// GNU-syntax conversion of __TI_ARM_V7M4__ reference

#if (__ARM_ARCH == 7 && __ARM_ARCH_PROFILE == 'M' && \
     defined(__ARM_FEATURE_SIMD32))
.int      0x1234
#endif
```

where *__ARM_ARCH*, *__ARM_ARCH_PROFILE*, and *__ARM_FEATURE_SIMD32* are ACLE predefined macro symbols, the combination of which is used to identify the Cortex-M4 processor that can be selected via tiarmclang's *-mcpu=cortex-m4* option.

- **__TI_ARM_V7R4__**

  The *__TI_ARM_V7R4__* predefined symbol is set to 1 if the Cortex-R4 Arm processor is targeted via armcl's *-mv7r4* option. The following reference to *__TI_ARM_V7R4__*:

```
;* TI-syntax reference to __TI_ARM_V7R4__

.if __TI_ARM_V7R4__
.int      0x1234
.endif
```

  is functionally equivalent to the following GNU-syntax Arm assembly:

```
// GNU-syntax conversion of __TI_ARM_V7R4__ reference

#if (__ARM_ARCH == 7 && __ARM_ARCH_PROFILE == 'R')
.int      0x1234
#endif
```

  where *__ARM_ARCH* and *__ARM_ARCH_PROFILE* are ACLE predefined macro symbols, the combination of which is used to identify the Cortex-R4 processor that can be selected via tiarmclang's *-mcpu=cortex-r4* option.

- **__TI_VFP_SUPPORT__**

  The *__TI_VFP_SUPPORT__* predefined symbol is set to 1 if floating-point coprocessor support is enabled via the armcl's *--float_support* option usage. The following reference to *__TI_VFP_SUPPORT__*:

```
;* TI-syntax reference to __TI_VFP_SUPPORT__

.if __TI_VFP_SUPPORT__
.int      0x1234
.endif
```

  is functionally equivalent to the following GNU-syntax Arm assembly:

```
// GNU-syntax conversion of __TI_VFP_SUPPORT__  reference

  #if defined(__ARM_FP)
  .int      0x1234
  #endif
```

where *__ARM_FP* is an ACLE predefined macro symbol. *__ARM_FP* indicates that use of the floating-point coprocessor is enabled via the *-mfloat-abi=hard* option.

- **__TI_VFPV3D16_SUPPORT__**

  The *__TI_VFPV3D16_SUPPORT__* predefined symbol is set to 1 if floating-point coprocessor support is enabled via the armcl's *--float_support=vfpv3d16* option. The following reference to *__TI_VFPV3D16_SUPPORT__*:

```
;* TI-syntax reference to __TI_VFPV3D16_SUPPORT__

  .if __TI_VFPV3D16_SUPPORT__
  .int      0x1234
  .endif
```

is functionally equivalent to the following GNU-syntax Arm assembly:

```
// GNU-syntax conversion of __TI_VFPV3D16_SUPPORT__  reference

  #if (defined(__ARM_FP) && __ARM_ARCH == 7 && __ARM_ARCH_
→PROFILE == 'R')
  .int      0x1234
  #endif
```

where *__ARM_FP*, *__ARM_ARCH*, and *__ARM_ARCH_PROFILE* are ACLE predefined macro symbols. *__ARM_FP* indicates that use of the floating-point coprocessor is enabled via the *-mfloat-abi=hard* option. The combination of *__ARM_ARCH* and *__ARM_ARCH_PROFILE* is used to identify the Cortex-R4 processor that is paired with the floating point coprocessor selected via the *-mfpu=vfpv3-d16* option.

- **__TI_FPV4SPD16_SUPPORT__**

  The *__TI_FPV4SPD16_SUPPORT__* predefined symbol is set to 1 if floating-point coprocessor support is enabled via the armcl's *--float_support=fpv4spd16* option. The following reference to *__TI_FPV4SPD16_SUPPORT__*:

```
;* TI-syntax reference to __TI_FPV4SPD16_SUPPORT__

  .if __TI_FPV4SPD16_SUPPORT__
  .int      0x1234
```

2.5. Migrating Assembly Language Source Code   **175**

```
   .endif
```

is functionally equivalent to the following GNU-syntax Arm assembly:

```
// GNU-syntax conversion of .TI_ARM reference

  #if (defined(__ARM_FP) && __ARM_ARCH == 7 && __ARM_ARCH_
→PROFILE == 'M')
   .int        0x1234
  #endif
```

where *__ARM_FP*, *__ARM_ARCH*, and *__ARM_ARCH_PROFILE* are ACLE prede-
fined macro symbols. *__ARM_FP* indicates that use of the floating-point coprocessor
is enabled via the *-mfloat-abi=hard* option. The combination of *__ARM_ARCH* and
__ARM_ARCH_PROFILE is used to identify the Cortex-M4 processor that is paired with
the floating point coprocessor selected via the *-mfpu=fpv4-sp-d16* option.

Other TI-Syntax Symbolic Constants

The following list of TI-syntax predefined symbolic constants do not serve any real purpose in the
context of an application that is to be built with the tiarmclang compiler tools since tiarmclang
only generates code that is compatible with the EABI and only supports a subset of existing Arm
processor variants (Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M33, Cortex-R4,
and Cortex-R5).

- **__TI_EABI_ASSEMBLER**

  The *__TI_EABI_ASSEMBLER* predefined symbol is set to 1 when the armcl compiler is
  invoked to generate EABI compatible object code via armcl's *--abi=eabi* option. The latest
  available version of the armcl compiler tools only support generation of EABI compatible
  object code, so the use of *__TI_EABI_ASSEMBLER* has become superfluous.

  Likewise, the tiarmclang compiler tools will only generate EABI compatible object code, so
  there is no need to migrate the *__TI_EABI_ASSEMBLER* predefined symbolic constant.

- **__TI_ARM7ABI_ASSEMBLER**

- **__TI_ARM9ABI_ASSEMBLER**

  Both of these ABIs have been deprecated and are no longer supported in the latest version
  of the armcl compiler tools.

- **__TI_NEON_SUPPORT__**

  The tiarmclang compiler does not currently support the Cortex-A8 Arm processor and there-
  fore does not support the neon extensions that are available with a Cortex-A8 Arm processor.

- **__TI_ARM_V4__**

- **__TI_ARM_V5E__**

- **__TI_ARM_V6__**

- **__TI_ARM_V7A8__**

  These TI-syntax Arm assembler predefined symbolic constants serve no purpose in the context of an application intended to be built with the tiarmclang compiler tools. The tiarmclang compiler only supports the following Arm Cortex-M and Cortex-R series processor variants:

    – Cortex-M0

    – Cortex-M0+

    – Cortex-M3

    – Cortex-M4

    – Cortex-M33

    – Cortex-R4

    – Cortex-R5

- **__TI_VFPV3_SUPPORT__**

  The tiarmclang does not currently support the Cortex-A8 Arm processor and therefore does not support use of the VFPV3 floating point coprocessor that is available with a Cortex-A8 Arm processor.

2.5.2 Converting Legacy TI Arm asm() Statements Embedded in C/C++ Source

Embedded in the C/C++ source code for your TI Arm application, you may be making use of asm() statements. In general, asm() statements are used to insert literal assembly language code into the compiler generated code for a given compilation unit. The armcl compiler supports a no-frills implementation of asm() statements; what you specify in the string argument to the asm() statement is exactly what will be inserted into the compiler generated code. The armcl's implementation of asm() statements does not support the notion of C expression operands, for example.

The tiarmclang compiler supports the GCC-style asm() statements that allows for specifying C expression operands. For example, the following definition of add() contains an example of an embedded GCC-style asm() statement:

```
int add(int i, int j) {
  int res;
  asm("\tADD %0, %1, %2\n"
      : "=r" (res)
      : "r" (i), "r" (j));
  return res;
}
```

where (res) is an output operand and (i) and (j) are input operands. If compiled with optimization (-O1 option), the tiarmclang compiler will generate the following instructions for the above function:

```
add:
        add     r0, r1
        bx      lr
```

For more information about using GCC-style asm() statements, please refer to How to Use Inline Assembly Language in C Code in the C Extensions part of Using the GNU Compiler Collection (GCC) online documentation.

Note: Use Caution When Defining Symbols Inside an asm() Statement

Inlining a function that contains an asm() statement that contains a symbol definition when compiling with the tiarmclang compiler can cause a "symbol multiply defined" error.

Please see *Inlining Functions that Contain asm() Statements* for more details.

2.5.3 Invoking the TI-Syntax ARM Assembler from tiarmclang

The tiarmclang compiler can be instructed to process input files with its TI-syntax ARM assembler. The assembly language source files, written using legacy TI-syntax ARM assembly code, can be indicated to the tiarmclang compiler using the "-x *language*" option, where *language* is "ti-asm". This option applies to all subsequent input files, so be sure to reset the input file type if there are non-TI-syntax ARM assembly source files on the command line following the TI-syntax ARM assembly source files.

For example, in the following command:

```
%> tiarmclang c-source1.c -x ti-asm ti-asm-source1.asm -x none c-
↪source2.c
```

the "-x ti-asm" option indicates that the ti-asm-source1.asm file is to be processed by the TI-syntax ARM assembler, and the subsequent "-x none" option resets the input file type to a default state so that the tiarmclang compiler knows to process the c-source2.c input file as a C file.

While many important options that need to be passed to the TI-syntax ARM assembler, such as silicon version and FPU version, can be inferred from the compiler's normal command line options, other options that you might need may be missing or need to be overridden. To support this, two new options, "-Wti-a," and "-Xti-assembler" are supported. These behave similarly to tiarmclang's other "-W" and "-X" options. Please see -Xlinker flag and -Wl flag for examples of the "-X" and "-W" options that apply to the linker.

Some helpful options available for use are:

- **--define/--undefine** - pre-define (or undefine) an identifier to a value, similar to tiarmclang's -D and -U compiler options.

- **--include_file** - effectively process an ".include *file*" directive before reading the assembly source file itself.

- **--include_path** - similar to the tiarmclang's -I compiler option, adding a specified directory to the search path used to find the subject of a ".include *file*" directive.

- **--no_warnings** - suppresses all warnings that are emitted by the TI-syntax ARM assembler.

Remember that when passing one of the above options to the TI-syntax ARM assembler from the tiarmclang command line, to precede the option with the tiarmclang's "-Xti-assembler" or "-Wti-a," option. In this example,

```
%> tiarmclang ... -x ti-asm tia.asm -Xti-assembler --define=MY_
↪PREDEF_SYM=10 ...
```

the "--define=MY_PREDEF_SYM=10" option is passed to the TI-syntax ARM assembler when processing the TI-syntax ARM assembly source file, "tia.asm".

Note: The .cdecls directive is not supported

The .cdecls directive, which for armcl projects allows programmers in mixed assembly and C/C++ environments to share C headers containing declarations and prototypes between the C and assembly code, is not supported by the TI-Syntax ARM Assembler included with tiarmclang.

2.6 Migrating Linker Command Files for Use With tiarmclang

To a large extent, linker command files for Arm applications that manage the placement of code and data generated by the armcl compiler will also work with object files that are generated by the tiarmclang compiler. However, a few adjustments may be needed to make your linker command file tiarmclang-friendly.

2.6.1 Explicit Specification of Compiler-Generated Object Files and Libraries

If your linker command file refers to specific object files, you may need to adjust how those files are referenced. There are two significant differences that you are likely to run into:

- When compiling and linking in a single step, the tiarmclang compiler creates temporary names for compiler-generated object files. For example, given a C source file named xyz.c, the tiarmclang compiler generates a temporary object file called xyz-<auto generated number sequence>.o that you might want to reference from your linker command file. You should reference such an object file using a wild-card, 'xyz*.o'.

- The tiarmclang compiler generates object files with a '.o' file extension, whereas the default file extension for an armcl-generated object file is '.obj'. You will need to update references to specific object files in your linker command file to use the '.o' file extension instead of the '.obj' file extension.

2.6.2 Compiler-Generated Section Names

Both the tiarmclang and armcl compilers generate code and data into object file sections. However, there are some differences to be aware of:

Compiler-Generated Section Names:

Section Description	armcl Generated Section	tiarmclang Generated Section
function definitions / code	.text	.text
const data	.const	.const, .rodata
initialization tables	.cinit	.cinit
initialized data	.data	.data
uninitialized data	.bss	.bss

As you will notice, the difference is that the tiarmclang compiler-generated string constants and some other constants into the .rodata section. In the linker command file, you may need to account for the placement of .rodata sections.

Other sections that you typically find in an Arm application are typically defined in the C/C++ runtime libraries and underlying run-time operating system layer.

RTS or RTOS Defined Section Names:

Section Description	armcl Generated Section	tiarmclang Generated Section
arguments (argc/argv)	.args	.args
stack space	.stack	.stack
heap space	.sysmem	.sysmem

2.6.3 Linker Options

The --rom_model (-c) linker option, which is the default for armcl, is not set by default by the tiarm-clang compiler when running the linker. Therefore, either the -rom_model (-c) or --ram_model (-cr) option must be passed to the linker using either -Xlinker or -Wl on the tiarmclang command line or must be specified in the linker command file.

THREE

TIARMCLANG COMPILER USER MANUAL

The TI Arm Clang Compiler Tools, commonly referred to in this user guide as tiarmclang, support the development of software applications intended to run on an Arm processor.

The tiarmclang compiler toolchain currently supports the development of applications that run on Arm Cortex-M and Cortex-R series processors, including: Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M33, Cortex-R4, and Cortex-R5.

This section of the documentation provides a detailed description of each of the parts of the tiarmclang compiler toolchain. It provides guidance on how these tools can be used to develop Arm applications.

Contents:

3.1 Using the C/C++ Compiler

This section of the *tiarmclang Compiler User Manual* describes the tiarmclang compiler, the compiler options that can be specified on the **tiarmclang** command-line, and how the compiler works with the linker to produce static executables that can be loaded and run on an Arm processor.

Contents:

3.1.1 About the Compiler

The TI Arm Clang Compiler (tiarmclang) lets you compile, optimize, assemble, and link an application in one step. The compiler performs the following steps on one or more source modules:

- The tiarmclang compiler compiles or assembles (as appropriate) one or more of the following types of input files:

  - **C** source files (with *.c* file extension)

  - **C++** source files (with *.C* and/or *.cpp* file extensions)

  - **GNU-Syntax Arm Assembly** source files

* By default, the integrated GNU-syntax assembler is used to process assembly source files with a *.s* file extension

* By default, an assembly source file with a *.S* extension will be pre-processed by the compiler before being passed to the integrated GNU-syntax assembler.

– **TI-Syntax Arm Assembly** source files (with *.asm* file extension)

Note: The **-x** option can be used on the **tiarmclang** command line to instruct the compiler how to interpret input files if the default file extension interpretation is not appropriate for your application. For more information about the **-x** option, see *Using -x Option to Control Input File Interpretation*.

---

* By default, the tiarmclang compiler then invokes the **linker** to create a static executable file from the object modules that were generated in the compile/assemble step. (The linker is also available using the **tiarmlnk** command line.)

Note: The link step can be disabled using the **-c** option on the **tiarmclang** command line. See *Stop Compiler After Object File Output (Omit Linking)*

3.1.2 Invoking the Compiler

Usage

To invoke the tiarmclang compiler, enter:

> **tiarmclang** [*options*] [*filenames*]

* **tiarmclang** - Command that runs the compiler and other tools (the linker, for example).

* *options* - Options that affect the way that the compiler tools process input files. These may include:

  – tiarmclang options - affect the behavior of the C/C++ compiler or the integrated GNU-syntax Arm assembler. These are described in more detail in the *Compiler Options* section. Additional information about the integrated GNU-syntax Arm assembler can be found in the *Integrated GNU-Syntax Arm Assembler* section.

  – Legacy TI-syntax Arm assembler options - are prefixed with either the *-Wti-a,* or *-Xti-assembler* option indicating that the option that follows should be passed directly to the legacy TI-syntax Arm assembler. Legacy TI-syntax Arm assembler options are described in more detail in the *TI-Syntax Arm Assembler* section. See *Passing TI-Syntax Arm Assembler Options: -Wti-a, and -Xti-assembler* for more about passing options to the assembler.

- **–** Linker options - are prefixed with either the *-Wl,* or *-Xlinker* option indicating that the option that follows should be passed directly to the linker. Linker options are described in more detail in *Linker Options*. See *Passing Linker Options: -Wl, and -Xlinker* for more about passing options to the linker.

Note: The linker is invoked by default from a **tiarmclang** command line, but you can disable the link step by specifying the *-c* option on the **tiarmclang** command line. You can invoke the linker by itself using the **tiarmlnk** command line.

- *filenames* - One or more input files. These may include:

  - **– C source files** - by default, an input file with a *.c* file extension is interpreted as a C source file. You may also use the *-x c* option to instruct **tiarmclang** to interpret subsequent input files as C source files.

  - **– C++ source files** - by default, an input file with a *.C* or *.cpp* file extension is interpreted as a C++ source file. You may also use the *-x c++* option to instruct **tiarmclang** to interpret subsequent input files as C++ source files.

  - **– ELF object files** - by default, an input file with a *.o* file extension is interpreted as an ELF object file.

- **GNU-syntax Arm assembly source files** - by default, an input file with either a *.s* or *.S* file extension is interpreted as a GNU-syntax Arm assembly source file. In the case of an input file with a *.S* extension, the input file is run through the C pre-processor and the output of the C pre-processor is passed to the integrated GNU-syntax Arm assembler. You may use the *-x assembler* option to instruct **tiarmclang** to interpret an input file as a GNU-syntax Arm assembly source file. Similarly, you may use the *-x assembler-with-cpp* option to instruct **tiarmclang** to run the input file through the C pre-processor before passing the output of the C pre-processor to the GNU-syntax Arm assembler.

- **TI-syntax Arm assembly source files** - unlike other **tiarmclang** input file types, a *-x ti-asm* option must be used to instruct **tiarmclang** to interpret a subsequent input file as a TI-syntax Arm assembly source file.

Note: For more information about the **tiarmclang** *-x* option and controlling how **tiarmclang** interprets input files, see the *Using -x Option to Control Input File Interpretation* section.

Example

The following simple example shows how the **tiarmclang** command can be used to build an ELF format static executable file that can be loaded and run on a Cortex-M0 Arm processor.

Source Files There are two input files to specify on the **tiarmclang** command line.

The C file *print_global.c* references a global variable that is defined in a GNU-syntax Arm assembly source file *def_global.S* and prints out the value of that global variable.

Contents of *print_global.c*:

```c
#include <stdio.h>

extern int a_global;

int main() {
  printf("a_global: %d\n", a_global);
  return 0;
}
```

Contents of *def_global.S*:

```
#if defined(__ti_version__)
            .global  a_global
            .data
a_global:   .int      12345
#else
#error "a_global is not defined"
#endif
```

In addition, the linker command file *lnkme.cmd* is stored in the current working directory. This linker command file provides a specification of the available memory and how to place compiler/linker generated output sections in that memory.

Contents of *lnkme.cmd*:

```
/
↪*****************************************************************
↪
/* Example Linker Command File                                   ␣
↪         */
/
↪*****************************************************************
↪
-c                                          /* LINK USING C␣
↪CONVENTIONS     */
```

```
-stack  0x8000                                      /* SOFTWARE STACK␣
↪SIZE           */
-heap   0x2000                                      /* HEAP AREA SIZE   ␣
↪           */
--args  0x1000


/* SPECIFY THE SYSTEM MEMORY MAP */


MEMORY
{
    P_MEM     : org = 0x00000020   len = 0x20000000   /* PROGRAM␣
↪MEMORY (ROM) */
    D_MEM     : org = 0x20000020   len = 0x20000000   /* DATA␣
↪MEMORY    (RAM) */
}


/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */


SECTIONS
{
    .intvecs    : {} > 0x0                  /* INTERRUPT VECTORS    ␣
↪           */
    .bss        : {} > D_MEM                /* GLOBAL & STATIC VARS  ␣
↪           */
    .data       : {} > D_MEM
    .sysmem     : {} > D_MEM                /* DYNAMIC MEMORY␣
↪ALLOCATION AREA    */
    .stack      : {} > D_MEM                /* SOFTWARE SYSTEM STACK ␣
↪           */

    .text       : {} > P_MEM                /* CODE                  ␣
↪           */
    .cinit      : {} > P_MEM                /* INITIALIZATION TABLES ␣
↪           */
    .const      : {} > P_MEM                /* CONSTANT DATA         ␣
↪           */
    .rodata     : {} > P_MEM
    .ARM.exidx  : {} > P_MEM
    .init_array : {} > P_MEM                /* C++ CONSTRUCTOR TABLES␣
↪           */
}
```

Compile and Link Steps Explained The following **tiarmclang** command compiles and links the

input files to create a static executable file called *a.out*:

```
%> tiarmclang -mcpu=cortex-m0 print_global.c def_global.S -o a.
↪out -Xlinker lnkme.cmd
```

The above **tiarmclang** command performs the following actions during the process of building the static executable file *a.out*:

- **tiarmclang** compiles and generates a first input object file from *print_global.c*. This input object file is subsequently fed into the link step.

- **tiarmclang** runs the C pre-processor over *def_global.c*, passing the result of the C pre-processor to the integrated GNU-syntax assembler that assembles the source file and produces a second input object file to be fed into the link step.

- **tiarmclang** runs the linker:

   - the previously generated input object files are linked with the Cortex-M0 version of the runtime libraries that are provided with the **tiarmclang** compiler tools installation,

   - the linker resolves the call to *printf* with a definition of the *printf* function from the C runtime library,

   - the linker reads the linker command file *lnkme.cmd* to determine how code and data sections are to be placed in Cortex-M0 target memory, and

   - the linker writes out the resulting ELF executable file a.out.

Compile and Link (Default Operation)

The default behavior of the tiarmclang compiler is to compile the specified source files into temporary object files, and then pass those object files along with any explicitly specified object files and any specified linker options to the linker.

In the following example, assume that the C code in **file1.c** references a data object that is defined in an object file named **file2.o**. The specified **tiarmclang** command compiles **file1.c** into a temporary object file. That object file, along with **file2.o** and a linker command file, **link_test.cmd**, is input to the linker and linked with applicable object files from the tiarmclang runtime libraries to create an executable output file named **test.out**:

```
tiarmclang -mcpu=cortex-m0 file1.c file2.o -o test.out -Wl,link_
↪test.cmd
```

More About Invoking the Linker With tiarmclang

Note that there is no mention of the tiarmclang runtime libraries on the **tiarmclang** command line or inside of the **link_test.cmd** linker command file. When the linker is invoked from the **tiarmclang** command line, the tiarmclang compiler implicitly tells the linker where to find applicable runtime libraries like the C runtime library (libc.a).

More specifically, the following options are implicitly passed from **tiarmclang** directly to the linker:

```
-I<install directory>/lib
--start-group -llibc++.a -llibc++abi.a -llibc.a -llibsys.a \
             -llibsysbm.a -llibclang_rt.builtins.a \
             -llibclang_rt.profile.a --end-group
```

- **-I <install directory>/lib** - tells the linker where to find the tiarmclang runtime libraries

- **--start_group/--end_group** - specifies exactly which runtime libraries are incorporated into the link

In the above **tiarmclang** command line, the **-Wl,** prefix in front of the specification of the **link_test.cmd** file name indicates to the compiler that the **link_test.cmd** file should be input directly into the linker (you can also use the **-Xlinker** prefix for this purpose).

Run Preprocesser Only

The **-E** option causes the compiler to halt after running the C preprocessor and emitting the preprocessed output to *stdout*. (The preprocessor output can also be piped to a file.)

> **tiarmclang -E** [*options*] [*filenames*]

The **-E** option is often combined with other preprocessor options like **-dD**, **-dI**, or **-dM** to further regulate the behavior of the C preprocessor. In the following example, the **-E** option is combined with **-dD** to print macro definitions in addition to normal preprocessor output to *stdout*:

```
tiarmclang -mcpu=cortex-m0 -E -dD file1.c
```

For more information about preprocessor options, see the *Preprocessor Options* section.

Run Preprocesser and Syntax-Checking Only

The **-fsyntax-only** option instructs **tiarmclang** to run the C preprocessor, parse the C/C++ input file to check for syntax errors, and perform type checking before halting compilation.

> **tiarmclang -fsyntax-only** [*options*] [*filenames*]

The **-fsyntax-only** option can be useful for finding simple syntax and type usage errors in the C/C++ source without incurring additional compile time early on in the development of newly written code.

Stop Compiler After Assembly Output

Using the **-S** option causes the compiler to generate assembly files from C or C++ source files that are specified on the command line. When **-S** is specified on the command line, compilation stops after the assembly files are emitted, preventing the compiler from generating object files or invoking the linker.

> **tiarmclang -S** [*options*] [*filenames*]

The following example generates assembly files, **file1.s** and **file2.s**, each containing compiler-generated GNU-syntax Arm assembly language directives and instructions:

```
tiarmclang -S -mcpu=cortex-m0 file1.c file2.c
```

Stop Compiler After Object File Output (Omit Linking)

You can avoid invoking the linker by specifying the **-c** option on the **tiarmclang** command line.

> **tiarmclang -c** [*options*] [*filenames*]

The following example generates object files **file1.o** and **file2.o** from the C files **file1.c** and **file2.c**, respectively:

```
tiarmclang -c -mcpu=cortex-m0 file1.c file2.c
```

3.1.3 Compiler Options

This section of the *tiarmclang Compiler User Manual* serves as a reference guide for the available command-line options that affect the behavior of the **tiarmclang** executable.

Contents:

Commonly Used Options

The commonly used options listed in the subsections below are available on the **tiarmclang** compiler command line.

- *Processor Options*
- *Include Options*
- *Predefined Symbol Options*
- *Optimization Options*
- *Debug Options*
- *Control Options*
- *Compiler Output Option*

Processor Options

Select a Target Arm Processor

−mcpu=<processor>
    Select the target *<processor>* version.

The tiarmclang compiler supports the following Arm Cortex-M processor variants which support 16-bit and T32 THUMB instructions (as indicated):

- **-mcpu=cortex-m0** - 16-bit THUMB
- **-mcpu=cortex-m0plus** - 16-bit THUMB
- **-mcpu=cortex-m3** - T32 THUMB
- **-mcpu=cortex-m4** - T32 THUMB
- **-mcpu=cortex-m33** - T32 THUMB

The tiarmclang compiler also supports Arm Cortex-R type processor variants, Cortex-R4 and Cortex-R5, which can execute ARM and T32 THUMB instructions. By default, the tiarmclang compiler assumes ARM instruction mode, but you can instruct the compiler to assume the use of T32 THUMB instructions for a given compilation by specifying the **-mthumb** option.

- **-mcpu=cortex-r4** - ARM (default) and/or T32 THUMB
- **-mcpu=cortex-r5** - ARM (default) and/or T32 THUMB

Instruction Mode Options

The Cortex-M type Arm processors only support execution of THUMB mode instructions, so **-mthumb** is selected by default when using the **-mcpu=cortex-m0**, **-mcpu=cortex-m0plus**, **-mcpu=cortex-m3**, **-mcpu=cortex-m4**, or **-mcpu=cortex-m33** processor options.

-mthumb

> Select THUMB mode instructions (16-bit THUMB or T32 THUMB depending on which processor variant is selected) for current compilation; default for Cortex-M type architectures

> Cortex-R type Arm processors can execute both ARM and T32 THUMB mode instructions. When either the **-mcpu=cortex-r4** or **-mcpu=cortex-r5** processor option is specified on the command line, the tiarmclang compiler assumes ARM instruction mode by default. This can be overridden for a given compilation by specifying the **-mthumb** option in combination with **-mcpu=cortex-r4** or **-mcpu=cortex-r5** to cause the compiler to generate T32 THUMB instructions.

-marm

> Select ARM mode instructions for current compilation; default for Cortex-R type processors

Endianness

All of the Arm Cortex-M type processor variants supported by the tiarmclang compiler are little-endian. The Arm Cortex-R type processor variants supported by the tiarmclang compiler may be big-endian or little-endian. The following options can be used to select the endian-ness to be assumed for a given compilation:

-mlittle-endian

> Select little-endian; default

-mbig-endian

> Select big-endian; only applicable for Cortex-R

Floating-Point Support Options

Some Arm processor variants can be used in conjunction with floating-point hardware processors to perform floating-point operations more efficiently. To enable this support, you must specify the **-mfloat-abi=hard** option in combination with the appropriate **-mfpu** option depending on what Arm processor variant is in use.

-mfloat-abi=<arg>

> Select floating point ABI indicated by *<arg>*.

> - **hard** - floating-point hardware is available; select appropriate hardware with **-mfpu** option

---

3.1. Using the C/C++ Compiler 191

- **soft** - unless **-mfloat-abi=hard** is selected, floating-point operations are emulated in software

-mfpu=<arg>

Select floating-point hardware indicated by *<arg>*

Only certain Arm processor variants can be used in conjunction with floating-point hardware processors. The **cortex-m0**, **cortex-m0plus**, and **cortex-m3** Arm processor variants are not compatible with any floating-point hardware processor.

- **vfpv3-d16** - available in combination with **-mcpu=cortex-r4** or **-mcpu=cortex-r5**

- **fpv4-sp-d16** - available in combination with **-mcpu=cortex-m4**

- **fpv5-sp-d16** - available in combination with **-mcpu=cortex-m33**

Include Options

The tiarmclang compiler utilizes the include file directory search path to locate a header file that is included by a C/C++ source file via an **#include** preprocessor directive. The tiarmclang compiler implicitly defines an initial include file directory search path to contain directories relative to the tools installation area where C/C++ standard header files can be found. These C/C++ standard header files are considered part of the tiarmclang compiler package and should be used in combination with linker and the runtime libraries that are included in the tiarmclang compiler tools installation.

-I<dir>

The **-I** option lets you add your own directories to the include file directory path, allowing user-created header files to be easily accessible during compilation.

Predefined Symbol Options

In addition to the pre-defined macro symbols that the tiarmclang compiler defines depending on which processor options are selected, you can also manage your own symbols at compile-time using the **-D** and **-U** options. These options are useful when the source code is configured to behave differently based on whether a compile-time symbol is defined and/or what value it has.

-D<name>[=<value>]

A user-created pre-defined compile symbol can be defined and given a value using the **-D** option. In the following example, **MySym** will be defined and given a value 123 at compile-time. **MySym** will then be available for use during the compilation of the **test.c** source file.

```
tiarmclang -mcpu=cortex-m0 -DMySym=123 -c test.c
```

–U<name>

> The **-U** option can be used to cancel a previous definition of a specified **<name>** whether it was pre-defined implicitly by the compiler or with a prior **-D** option.

Optimization Options

To enable optimization passes in the tiarmclang compiler, select a level of optimization from among the following **-O[0|1|2|3|fast|g|s|z]** options. In general, the options below represent various levels of optimization with some options designed to favor smaller compiler generated code size over performance, while others favor performance at the cost of increased compiler generated code size.

Among the options listed below, **-Oz** is recommended as the optimization option to use if small compiler generated code size is a priority for an application. Using **-Oz** retains performance gains from many of the **-O2** level optimizations that are performed.

–O0

> No optimization. This setting is not recommended, because it can make debugging difficult.

–O1 or **–O**

> Restricted optimizations, providing a good trade-off between code size and debug-ability.

–O2

> Most optimizations enabled; some optimizations that require significantly additional compile time are disabled.

–O3

> All optimizations available at **-O2** plus others that require additional compile time to perform.

–Ofast

> All optimizations available at **-O3** plus additional aggressive optimizations with potential for additional performance gains, but also not guaranteed to be in strict compliance with language standards.

–Og

> Restricted optimizations while preserving debuggability. All optimizations available at **-O1** are performed with the addition of some optimizations from **-O2**.

–Os

> All optimizations available at **-O2** plus additional optimizations that are designed to reduce code size while mitigating negative impacts on performance.

–Oz

> All optimizations available at **-O2** plus additional optimizations to further reduce code size with the risk of sacrificing performance.

Note: Optimization Option Recommendations:

- The **-O1** option is recommended for maximum debuggability.

- The **-Oz** option is recommended for optimizing code size.

- The **-O3** option is recommended for optimizing performance, but it is likely to increase compiler generated code size.

Debug Options

The tiarmclang compiler generates DWARF debug information if the **-g** or **-gdwarf-3** option is selected.

-g or -gdwarf-3

Emit DWARF version 3 debug information

Control Options

The default behavior of the tiarmclang compiler is to compile the specified source files into temporary object files, then pass those object files along with any explicitly specified object files and any specified linker options to the linker.

Several tiarmclang compiler options can be used to change this behavior and halt compilation at different stages:

-c

Stop compilation after emitting compiler-generated object files; do not call linker.

-E

Stop compilation after the pre-processing phase of the compiler; this option can be used in conjunction with several other options that provide further control over the pre-processor output:

- **-dD** - Print macro definitions in addition to normal preprocessor output.

- **-dI** - Print include directives in addition to normal preprocessor output.

- **-dM** - Print macro symbol definitions *instead of* normal preprocessor output.

-S

Stop compilation after emitting compiler-generated assembly files; do not call assembler or linker.

See *Invoking the Compiler* for examples.

Compiler Output Option

-o<file>

The **-o** option names the output file that results from a **tiarmclang** command. If **tiarmclang** is used to compile and link an executable output file, then the **-o** option's *<file>* argument names that output file. If no **-o** option is specified in a compile and link invocation of **tiarmclang**, then the linker produces an executable output file named **a.out**.

If the compiler is used to process a single source file, then the **-o** option names the output of the compilation. This is sometimes useful in case there is a need to name the output file from the compiler something other than what the compiler produces by default.

In the following example, the output object file from the compilation of C source file *task_42.c* is named *task.o* by the **-o** option, replacing the *task_42.o* file that would normally be generated by the compiler:

```
tiarmclang -mcpu=cortex-m0 -c task_42.c -o task.o
```

Processor Options

- *Select Processor*
- *Select Floating-Point Code Generated*
- *Select Endianness*

Select Processor

-mcpu=<arg>

Instruct the compiler to generate code for the Arm processor variant indicated by *<arg>*, where *<arg>* can be:

- **cortex-m0**
- **cortex-m0plus**
- **cortex-m3**
- **cortex-m4**
- **cortex-m33**
- **cortex-r4**
- **cortex-r5**

Select Architecture

–march=<arg>

Instruct the compiler to generate code for the Arm architecture variant indicated by *<arg>*, where
<arg> can be:

- **thumbv6m** - appropriate for *-mcpu=cortex-m0* or *-mcpu=cortex-m0plus*

- **thumbv7m** - appropriate for *-mcpu=cortex-m3*

- **thumbv7em** - appropriate for *-mcpu=cortex-m4*

- **thumbv8m.main** - appropriate for *-mcpu=cortex-m33*

- **thumbv8.1-m.main+cdecp0** - can be used to enable CDE intrinsics on a Cortex-m33 device
  that is equipped with a coprocessor that can execute CDE instructions. See *Custom Datapath
  Extension (CDE) Intrinsics* for more information.

- **armv7r** - appropriate for *-mcpu=cortex-r4* or *-mcpu=cortex-r5* with *-marm* and *-mlittle-
  endian*

- **armebv7r** - appropriate for *-mcpu=cortex-r4* or *-mcpu=cortex-r5* with *-marm* and *-mbig-
  endian*

- **thumbv7r** - appropriate for *-mcpu=cortex-r4* or *-mcpu=cortex-r5* with *-mthumb* and -
  mlittle-endian

- **thumbebv7r** - appropriate for *-mcpu=cortex-r4* or *-mcpu=cortex-r5* with *-mthumb* and -
  mbig-endian

Select Instruction Set

–mthumb

Instruct the compiler to generate THUMB mode instructions (16-bit THUMB or T32 THUMB
depending on which processor variant is selected) for current compilation; default for Cortex-M
type architectures

–marm

Instruct the compiler to generate ARM mode instructions for current compilation; default for
Cortex-R series processors

Select Floating-Point Code Generated

−mfloat−abi=<arg>

Instruct the compiler to generate code in accordance with the floating-point ABI indicated by *<arg>*, where *<arg>* can be:

- **hard** - assume floating=point hardware (FPU) is available; pass floating-point type arguments and return values in FPU registers; generate FPU instructions for floating-point type operations

- **soft** - assume no floating-point hardware is available; pass arguments and return values in general-purpose registers (GPRs); perform floating-point operations in software with CPU instructions or in calls to runtime support library functions.

−mfpu=<arg>

Instruct compiler to use floating-point hardware (FPU) registers and instructions to perform floating-point operations; select the FPU specified by *<arg>*, where *<arg>* can be:

- **vfpv3-d16** - available in combination with **-mcpu=cortex-r4** or **-mcpu=cortex-r5**

- **fpv4-sp-d16** - available in combination with **-mcpu=cortex-m4**

- **fpv5-sp-d16** - available in combination with **-mcpu=cortex-m33**

Select Endianness

−mlittle−endian, −EL

Instruct the compiler to generate little-endian object code; default.

−mbig−endian, −EB

Instruct the compiler to generate big-endian object code; only applicable for Cortex-R series processors.

C/C++ Language Options

The **tiarmclang** compiler's **-std** option allows you to specify which C or C++ language standard the compiler should adhere to when processing C or C++ source files.

The supported C and C++ language variants are described below.

Note: **Default C/C++ Language Standard**

If no *-std* option is specified on the **tiarmclang** command line, then *-std=gnu17* is assumed for C source files and *-std=gnu++17* is assumed for C++ source files.

C Language Variants (-std)

For C *<language-variants>* of the form *cNN*, the compiler pre-defines the *__STRICT_ANSI__* macro symbol to 1.

-std=c89, **-std**=c90
> C as defined in the ISO C 1990 standard

-std=c99, **-std**=c9x
> C as defined in the ISO C 1999 standard

-std=c11, **-std**=c1x
> C as defined in the ISO C 2011 standard

-std=c17, **-std**=c18
> C as defined in the ISO C 2017 standard, which addressed C11 defects without adding any new features

For C *<language-variants>* of the form *gnuNN*, GNU C language extensions are supported and the compiler does not define the *__STRICT_ANSI__* macro symbol.

-std=gnu89, **-std**=gnu90
> C as defined in the ISO C 1990 standard with GNU extensions

-std=gnu99, **-std**=gnu9x
> C as defined in the ISO C 1999 standard with GNU extensions

-std=gnu11, **-std**=gnu1x
> C as defined in the ISO C 2011 standard with GNU extensions

-std=gnu17, **-std**=gnu18
> C as defined in the ISO C 2017 standard with GNU extensions. This is the default for C files if no *-std* option is defined.

C++ Language Variants (-std)

For C++ *<language-variants>* of the form *c++NN*, the compiler pre-defines the *__STRICT_ANSI__* macro symbol to 1.

-std=c++98, **-std**=c++03
> C++ as defined in the ISO C++ 1998 standard with amendments

–std=c++11

C++ as defined in the ISO C++ 2011 standard with amendments

–std=c++14

C++ as defined in the ISO C++ 2014 standard with amendments

–std=c++17

C++ as defined in the ISO C++ 2017 standard with amendments

For C++ *<language-variants>* of the form *gnuNN*, GNU C language extensions are supported and the compiler does not define the *__STRICT_ANSI__* macro symbol.

–std=gnu++98, **–std**=gnu++03

C++ as defined in the ISO C++ 1998 standard with amendments and GNU extensions

–std=gnu++11

C++ as defined in the ISO C++ 2011 standard with amendments and GNU extensions

–std=gnu++14

C++ as defined in the ISO C++ 2014 standard with amendments and GNU extensions

–std=gnu++17

C++ as defined in the ISO C++ 2017 standard with amendments and GNU extensions. This is the default for C++ files if no *-std* option is defined.

See *Characteristics and Implementation of Arm C++* for details about C++ 2017 support.

C/C++ Run-Time Standard Header and Library Options

–nostdlib, --no-standard-libraries

Avoid linking in the C/C++ standard libraries. This is useful when partially linking an application, or when you want to link against your own standards-compliant libraries.

–nostdinc, --no-standard-includes

Do not incorporate the C/C++ runtime header file directory, the compiler builtin include directory, or the standard system include directory in the default definition of the include file directory search path.

–nostdlibinc

Do not incorporate the C/C++ runtime header file directory or the standard system include directory into the include file directory search path, but do incorporate the compiler's builtin include directory.

Run-Time Type Information (RTTI) Options

–frtti, –fno–rtti

> The **tiarmclang** compiler allows you to support Run-Time Type Information (RTTI) features, such as the dynamic_cast operator, the typeid operator, and the type_info class.
>
> By default RTTI support is disabled, which is equivalent to using the -fno-rtti option. When RTTI support is disabled, use of the `typeid` operator causes an error. Use of `dynamic_case` causes an error only in certain situations.
>
> To explicitly enable RTTI support, use the -frtti option.

C++ Exception Handling

–fexceptions, –fno–exceptions

> By default, the **tiarmclang** compiler provides no exception handling support. To enable exception handling, use the compiler's -fexceptions option. See *C++ Exception Handling* for details.

Runtime Model Options

–fcommon, –fno–common

> The *-fcommon* option is enabled by default.
>
> For C source code, this option causes uninitialized global variable definitions to be treated as *tentative* definitions. If the *-fcommon* option is enabled, uninitialized global variables are placed in a *common* block. The linker then resolves all *tentative* definitions of the same global variable in different compilation units to a single data object definition.
>
> This behavior can be disabled with the *-fno-common* option.

–fdata–sections, –fno–data–sections

> The *-fdata-sections* option is enabled by default and instructs the tiarmclang compiler to generate code for the definition of a data object into its own section. This default behavior can be overridden by specifying the *fno-datasections* on the **tiarmclang** command line. You can also dictate what section the data object will be defined in by attaching a *section* attribute to the data object in the C/C++ source code.

–fexceptions, –fno–exceptions

> By default, support for C++ exceptions is disabled. To enable support for C++ exceptions, use the *-fexceptions* compiler option on the **tiarmclang** command line. If the *-fexceptions* option is specified on the **tiarmclang** command line, then the linker links the application with runtime libraries that have been built with C++ exceptions enabled.

–ffunction–sections, –fno–function–sections

The *-ffunction-sections* option is enabled by default and instructs the tiarmclang compiler to generate code for a function definition into its own section. This default behavior can be overridden by specifying the *fno-function-sections* on the **tiarmclang** command line. You can also dictate what section the compiler will generate code for a function definition into by attaching a *section* attribute to the function in the C/C++ source code.

–fshort–enums, –fno–short–enums

The *-fshort-enums* option instructs the compiler to only allocate as much space for an enum type data object as is needed to represent the declared range of possible values. This is the default behavior assumed by the tiarmclang behavior. You can override this default behavior using the *-fno-short-enums* option that allocates 4 bytes for an enum type data object even if the range of values for a given enum type data object can be represented with fewer bytes.

–fshort–wchar, –fno–short–wchar

The default size for the *wchar_t* type is 32-bits, which is analogous to the *fno-short-wchar* option. The runtime libraries provided with the tiarmclang toolchain installation are all built assuming a *wchar_t* type size of 32-bits. If you compile a C/C++ source file with the *-fshort-wchar* option to indicate that the *wchar_t* type size should be assumed to be 16-bits, you will encounter a link-time warning indicating that the newly compiled object file is not compatible with object files in the runtime libraries.

–funsigned–char, –fsigned–char or –fno–unsigned–char

A plain *char* type is treated as *unsigned char* by default in the **tiarmclang** compiler. This matches the semantics for the *-funsigned-char* option. This behavior can be overridden with the use of the *-fsigned-char* or *-fno-unsigned-char* option, which indicates that a plain *char* type is to be interpreted as *signed char*.

–mexecute–only, –mno–execute only

The tiarmclang compiler tools support the generation of execute-only code for TI Arm Cortex-M0/M0+ processor variants. The execute-only feature is not enabled by default. Generation of execute-only code can be enabled by specifying the -mexecute-only option on the **tiarmclang** command line.

When an application's source files are compiled with *-mexecute-only*, the compiler prevents constant data from being embedded in the code section that contains the definition of a given function. Also, the linker is instructed to link with a version of the runtime support libraries that were built with execute-only enabled.

Consider a simple example with a function that contains a switch statement:

```c
#include <stdio.h>

void mySwitch(int n) {

  switch (n)
```

```
  {
    case 1:
      printf("Input value is 1\n");
      break;
    case 2:
      printf("Input value is 2\n");
      break;
    case 3:
      printf("Input value is 3\n");
      break;
    default:
      printf("Invalid input\n");
      break;
  }
}
```

When the above source file is compiled as follows:

```
%> tiarmclang -mcpu=cortex-m0plus -mexecute-only -S
↪ex_switch.c
```

The compiler generates an assembly language file that contains the following:

```
%> cat ex_switch.s
...
        .section        .text.mySwitch,"axy",
↪%progbits,unique,0
        .globl  mySwitch
        .p2align        1
        .code   16                                    @
↪@mySwitch
        .thumb_func
mySwitch:
...
.LBB0_5:                                        @ %sw.bb3
        movs    r0, :upper8_15:.L.str.2
        lsls    r0, r0, #8
        adds    r0, :upper0_7:.L.str.2
        lsls    r0, r0, #8
        adds    r0, :lower8_15:.L.str.2
        lsls    r0, r0, #8
        adds    r0, :lower0_7:.L.str.2
        bl      printf
```

```
...
        .section        .rodata.str1.1,"aMS",
↪%progbits,1
.L.str:
        .asciz  "Input value is 1\n"
        .size   .L.str, 18
...
.L.str.2:
        .asciz  "Input value is 3\n"
        .size   .L.str.2, 18
...
```

Without the execute-only feature enabled, the code generated to load the address of a string constant consists of a single PC-relative load instruction.

When execute-only is enabled, the compiler generates four 8-bit loads to get the address of a string contant into an argument register as we can see in the above assembly language excerpt.

While the execute-only compiler generated code is larger and less efficient, it is able to reside in special execute-only memory. This is useful when code security is a concern.

–munaligned–access, –mno–unaligned–access

The *-munaligned-access* option can be used to enable the use of unaligned memory accesses when generating code for an Arm processor variant that supports unaligned memory accesses. This is the default tiarmclang compiler behavior for Arm processors like the Cortex-M4 that have support for unaligned memory accesses. This default behavior can be disabled with the *-mno-unaligned-accesses* option.

For Arm processor variants like the Cortex-M0 that do not have support for unaligned memory accesses, the default tiarmclang behavior is *-mno-unaligned-accesses*. An attempt to use the *-munaligned-accesses* option with an Arm processor that does not support unaligned memory accesses will result in a warning diagnostic and the option will be ignored.

In general, C library memory access functions (e.g. memcpy, memset) assume normal memory, which for most Arm subtargets allows unaligned memory access. For special device/peripheral memory, unaligned access could cause an Alignment fault. When memory access functions are used for device memory, the pointers must be to an aligned address, or else customized functions should be used.

The compiler defines the *__ARM_FEATURE_UNALIGNED* macro when *-munaligned-access* is enabled. See *TI Arm-Specific Pre-Defined Macro Symbols* for more information. The macro can be used to select customized memory access functions with special implementations at compile-time:

```
#ifdef __ARM_FEATURE_UNALIGNED
#define __MEMCPY memcpy
#else
extern void memcpy_aligned(void *dst, void *src, size_t
 ↪bytes);
#define __MEMCPY memcpy_aligned
#endif

void copy(void *dst, void *src, size_t bytes) {
  __MEMCPY(dst, src, bytes);
}
```

-mlong-calls, -mno-long-calls

> The *-mlong-calls* option can be used to instruct the compiler to implement function calls using branch indirect instructions (i.e. load address of destination into a register and branch via the register). The tiarmclang compiler default behavior is to use direct branches to implement function calls.

> For functions that are out-of-range from their caller, the linker generates a *trampoline* to help an out-of-range caller reach its destination.

Symbol Management Options

Define/Undefine Symbols

You can define or undefine a symbol on the **tiarmclang** command line with the *-D* and *-U* options. These can be useful for selecting a particular configuration of your source code from the command line.

-D<symbol>[=<value>]

> Define a *<symbol>* with the specified *<value>*. If no *<value>* argument is provided, then the *<symbol>*'s value will be set to 1. A symbol defined on the command line via the *-D* option is equivalent to a pre-defined macro symbol.

-U<symbol>

> Undefine an existing pre-defined macro *<symbol>*.

Symbol Visibility

An important part of creating shared objects is managing which symbols defined within a shared object are available to be linked against from outside the shared object.

−fvisibility=<visibility_kind>

> Set the default ELF image symbol visibility to the specified *<visibility_kind>*. All symbols are marked with the specified *<visibility_kind>* unless explicitly overridden within the C/C++ source code.

> Symbols that are declared extern are not affected by the use of the *-fvisibility* option.

> The available *<visibility kind>* settings are:

> - *default* - Indicates that symbols have *public* visibility by default and can be linked against from outside a shared object. Global and weak symbols with *public* visibility can be preempted by definitions of a symbol with the same name from an object outside of the shared object.

> - *hidden* - Indicates that symbols are not available to be linked against from outside a shared object by default.

> - *protected* - Indicates that symbols defined in a shared object are visible outside of the shared object, but cannot be preempted. A reference to a protected symbol from within the shared object in which it is defined must be resolved by the definition in that shared object.

Preprocessor Options

With the **-E** option, you can instruct the tiarmclang compiler to stop after the preprocessor phase of compilation:

−E, −−preprocess

> Halt compilation after running the C preprocessor.

Preprocessor Options

tiarmclang options that control the behavior of the C preprocessor:

−C, −−comments

> Include comments in preprocessed output.

−CC, −−comments−in−macros

> Include comments from within macros in preprocessed output.

-D\<macro\>=\<value\>

> Define *\<macro\>* symbol to *\<value\>* (or 1 if *\<value\>* omitted).

-H, --trace-includes

> Show header includes and nesting depth.

-P, --no-line-commands

> Disable linemarker output in -E mode.

-U\<macro\>

> Undefine *\<macro\>* symbol.

-Wp,\<arg1\>,\<arg2\>...

> Pass the comma separated arguments in *\<argN\>* to the preprocessor.

-Xpreprocessor \<option\>

> Pass *\<option\>* to the preprocessor.

Dependency File Generation

tiarmclang options that control generation of a dependency file for make-like build systems.

-M, --dependencies

> Like -MD, but also implies -E and writes to stdout by default.

-MD, --write-dependencies

> Write a dependency file containing user and system headers.

-MF\<file\>

> Write dependency file output from -MMD, -MD, -MM, or -M to specified *\<file\>*.

-MG, --print-missing-file-dependencies

> Add missing headers to dependency file.

-MJ\<arg\>

> Write a compilation database entry per input.

-MM, --user-dependencies

> Like -MMD, but also implies -E and writes to stdout by default.

-MMD, --write-user-dependencies

> Write a dependency file containing user headers.

-MP

> Create phony target for each dependency (other than main file).

-MQ<arg>

Specify name of main file output to quote in dependency file.

-MT<arg>

Specify name of main file output in dependency file

Dumping Preprocessor State

tiarmclang options that allow the state of the preprocessor to be dumped in various ways.

-dD

Print macro definitions in -E mode in addition to normal output.

-dI

Print include directives in -E mode in addition to normal output.

-dM

Print macro definitions in -E mode instead of normal output.

Optimization Options

To enable optimization passes in the tiarmclang compiler, select a level of optimization from among the following **-O[0|1|2|3|fast|g|s|z]** options. In general, the options below represent various levels of optimization with some options designed to favor smaller compiler generated code size over performance, while others favor performance at the cost of increased compiler generated code size.

For a more precise list of optimizations performed for each level, please see *Details of Optimizations Performed at Each Level*.

Optimization Level Options

Note: Optimization Option Recommendations

- The **-Oz** option is recommended if small compiler generated code size is a priority.

- The **-O3** option is recommended for optimizing performance, but it is likely to increase compiler-generated code size.

-O0

Performs no optimization.

-O1, -O

> Enables restricted optimizations, providing a good trade-off between code size and debuggability. This option is recommended for maximum debuggability.

-O2

> Enables most optimizations, but some optimizations that require significant additional compile time are disabled.

-O3

> Enables all optimizations available at **-O2** plus others that require additional compile time to perform. This option is recommended for optimizing performance, but it is likely to increase compiler-generated code size.

-Ofast

> Enables all optimizations available at **-O3** plus additional aggressive optimizations that have the potential for additional performance gains, but are not guaranteed to be in strict compliance with language standards.

-Og

> Enables restricted optimizations while preserving debuggability. All optimizations available at **-O1** are performed with the addition of some optimizations from **-O2**.

-Os

> Enables all optimizations available at **-O2** plus additional optimizations that are designed to reduce code size while mitigating negative impacts on performance.

-Oz

> Enables all optimizations available at **-O2** plus additional optimizations to further reduce code size with the risk of sacrificing performance. Using **-Oz** retains performance gains from many of the **-O2** level optimizations that are performed. This optimization setting is recommended if small code size is a priority.

Link-Time Optimization

-flto -O<1|2|3|fast|s|z>

> When link-time optimization is enabled in the tiarmclang toolchain via the **-flto** compiler/linker option, the linker is able to combine all input object files into a merged representation of the application and present that representation back to the compiler where inter-module optimizations can be performed.

> The types of inter-module optimizations that are performed depends on which of the **-O<1|2|3|fast|s|z>** options is specified in combination with **-flto**:

> - **-flto -O1** - perform basic inter-module optimizations.

- **-flto -O2** - perform all **-O1** level inter-module optimizations plus performance-oriented inter-module optimizations, such as function inlining and global duplicate constant merging.

- **-flto -O3** - perform all **-O2** level inter-module optimizations plus aggressive performance-oriented inter-module optimizations, like loop optimizations and call-site splitting. Performance improving inter-module optimizations are favored over code size reducing inter-module optimizations.

- **-flto -Ofast** - perform all **-O3** level inter-module optimizations plus more aggressive performance-oriented inter-module optimizations, including floating-point math optimizations that are potentially not value-safe.

- **-flto -Os** - perform all **-O2** level inter-module optimizations, but several of these optimizations will favor reducing code size over improving performance.

- **-flto -Oz** - perform all **-O2** level inter-module optimizations plus other code size reducing optimizations, like machine outlining. Also, several **-O2** optimizations are tuned to favor aggressive code size reduction even if performance is degraded.

More Specialized Optimization Options

Floating-Point Arithmetic

-ffast-math, -fno-fast-math

Enable or disable 'fast-math' mode during compilation. By default, the 'fast-math' mode is disabled. Enabling 'fast-math' mode allows the compiler to perform aggressive, not necessarily value-safe, assumptions about floating-point math, such as:

- Assume floating-point math is consistent with regular algebraic rules for real numbers (e.g. addition and multiplication are associative, $x/y == x * 1/y$, and $(a + b) * c == a * c + b * c$).

- Operands to floating-point operations are never *NaNs* or *Inf* values.

- *+0* and *-0* are interchangeable.

Enabling the -ffast-math option also causes the following options to be set:

- -ffp-contract=fast

- -fno-honor-nans

- -ffp-model=fast

- -fno-rounding-math

- -fno-signed-zeros

Use of the 'fast-math' mode also instructs the compiler to predefine the __FAST_MATH__ macro symbol.

-ffp-model=<precise|strict|fast>

-ffp-model is an umbrella option that is used to establish a model of floating-point semantics that the compiler will operate under. The available arguments to the **-ffp-model** option will imply settings for the other, single-purpose floating-point options, including **-ffast-math**, **-ffp-contract**, and **frounding-math** (described below).

The available arguments to the **-ffp-model** option are:

- **precise** - With the exception of floating-point contraction optimizations, all other optimizations that are not value-safe on floating-point data are disabled (*ffp-contract=on* and *-fno-fast-math*). The tiarmclang compiler assumes this floating-point model by default.

- **strict** - Disables floating-point contraction optimizations (*-ffp-contract=off*), honors dynamically-set floating-point rounding modes (*-frounding-math*), and disables all 'fast-math' floating-point optimizations (*-fno-fast-math*). Also sets *-ffp-exception-behavior=strict*.

- **fast** - Enables all 'fast-math' floating-point optimizations (*-ffast-math*) and enables floating-point contraction optimizations across C/C++ statements (*-ffp-contract=fast*).

-ffp-contract=<fast|on|off|fast-honor-pragmas>

Instruct the compiler whether and to what degree it is allowed to form fused floating-point operations, such as floating-point multiply and add (FMA) instructions. This optimization is also known as *floating-point contraction*. Fused floating-point operations are permitted to produce more precise results than would be otherwise computed if the operations were performed separately.

The available arguments to the **-ffp-contract** option are:

- **fast** - Allows fusing of floating-point operations across C/C++ statements, and ignores any *FP_CONTRACT* or *clang fp contract* pragmas that would otherwise affect the compiler's ability to apply floating-point contraction optimizations.

- **on** - Allows floating-point contraction within a given C/C++ statement. The floating-point contraction behavior can be affected by the use of *FP_CONTRACT* or *clang fp contract* pragmas.

- **off** - Disables all floating-point contraction optimizations.

- **fast-honor-pragma** - Same as the *fast* argument, but the user can alter the behavior via the use of the *FP_CONTRACT* and/or *clang fp contract* pragmas.

-ffp-exception-behavior=<ignore|strict|maytrap>

This option determines the behavior in coordination with potential floating-point hardware exceptions.

- **ignore** - This is the default setting. The compiler expects exception status flags to not be read and floating point exceptions to be masked.

- **maytrap** - This setting causes the compiler to avoid performing floating point transformations that may raise exceptions if the original code would not have raise them. The compiler may still perform constant folding.

- **strict** - This setting causes the compiler to strictly preserve the floating point exception semantics of the original code when performing any floating point transformations. Setting -ffp-model=strict causes -ffp-exception-behavior to be set to strict.

The maytrap and strict settings also prevent the following optimizations:

- Floating point instruction speculation

- Hoisting floating point instructions into code where they may execute despite guard conditions that would otherwise prevent their execution

Floating-Point Speculation Example

Consider a floating-point divide operation that is guarded by an if statement in a C function:

```
__attribute__((noinline)) void cp_fpu_trace(float32 *f_Info)
{
    float32 f_array[10] = {0};
    float32 float1 = *f_Info;

    if (float1 > CP_MIN)
    {
      f_array[1u] = (1.f / *f_Info);
    }
    else
    {
      f_array[1u] = CP_MAX;
    }

    *f_Info = f_array[1u];
}
```

In this code the if statement is intended to prevent the floating-point divide operation from performing a divide-by-zero that will trigger a hardware floating-point exception. However, if optimization is enabled to perform floating-point speculation, the compiler may generate code to perform the divide instruction prior to the compare instruction that carries out the intent of the if statement:

```
...
cp_fpu_trace:
    vldr        s0, [r0]
```

```
    vmov.f32    s2, #1.000000e+00
    vldr        s4, .LCPI1_0          ; <- load of CP_MIN
    vldr        s6, .LCPI1_1          ; <- load of CP_MAX
    vdiv.f32    s2, s2, s0            ; <- divide
    vcmp.f32    s0, s4               ; <- compare
    vmrs        APSR_nzcv, fpscr
    vmovgt.f32  s6, s2               ; <- conditional move:
                                     ;    if (float1 > CP_
↪MIN)
                                     ;      s6 = result of␣
↪divide
    vstr        s6, [r0]
    bx          lr
...
```

In this case, if the value loaded into s0 is zero, then the divide instruction will trigger a
floating-point hardware exception before the compare instruction has a chance to prevent that
from happening. As mentioned above, including the *-ffp-exception-behavior=maytrap* or
-ffp-exception-behavior=strict option on the compiler command-line will prevent the com-
piler from performing floating-point speculation optimizations and generating code that will
execute the divide instruction prior to the compare and conditional move instructions.

-fhonor-nans, -fno-honor-nans

Instructs the compiler to check for and properly handle floating-point NaN values. Use of
the **-fno-honor-nans** can improve code if the compiler can assume that it doesn't need to
check for and enforce the proper handling of floating-point NaN values.

-frounding-math, -fno-rounding-math

By default, the compiler assumes that the **-fno-rounding-mode** option is in effect. This
instructs the compiler to always round-to-nearest for floating-point operations.

The C standard runtime library provides functions such as *fesetround* and *fesetenv* that allow
you to dynamically alter the floating-point rounding mode. If the **-frounding-math** option
is specified, the compiler honors any dynamically-set floating-point rounding mode. This
can be used to prevent optimizations that may affect the result of a floating-point operation
if the current rounding mode has changed or is different from the default (round-to-nearest).
For example, floating-point constant folding may be inhibited if the result is not exactly
representable.

-fsigned-zeros, -fno-signed-zeros

Assumes the presence of signed floating-point zero values. Use of the **-fno-signed-zeros**
option can improve code if the compiler can assume that it doesn't need to account for the
presence of signed floating-point zero values.

Inlining and Outlining

-finline-functions, -fno-inline-functions

    Inline suitable functions. The *-fno-inline-functions* option disables this optimization.

-finline-hint-functions

    Inline functions that are explicitly or implicitly marked as inline.

-mllvm -arm-memset-max-stores=<n>

    When optimization is turned on during a compilation, the tiarmclang compiler inlines calls to the *memset* and *memclr* runtime support routines if the size of the data is below a certain threshold. For example, in the following source file:

```c
#include <string.h>

struct {
  int t1;
  int t2;
  int t3;
  int t4;
  short t5;
  long t6;
} my_struct_inline;

void func()
{
  memset(&my_struct_inline, 0, sizeof(my_struct_inline));
}
```

When compiled with *-O[1|2|3|fast]*, the call to *memset* is inlined if the clearing of the *my_struct_inline* data object can be done with <= 8 store instructions:

```
%> tiarmclang -mcpu=cortex-m0 -O3 -S struct_inline.c
%> cat struct_inline.s
...
func:
        ldr     r0, .LCPI0_0
        movs    r1, #0
        str     r1, [r0]
        str     r1, [r0, #4]
        str     r1, [r0, #8]
        str     r1, [r0, #12]
        str     r1, [r0, #16]
```

```
            str       r1, [r0, #20]
            bx        lr
            .p2align       2
.LCPI0_0:
            .long   my_struct_inline
...
```

However, when compiled with *-O[s|z]*, where the compiler is attempting to generate smaller code, the call to *memset* is inlined only if clearing the *my_struct_inline* data object can be done with <= 4 store instructions. When compiled in combination with the *-mcpu=cortex-m0* option, the call to *memset* is not inlined, but it will be implemented with a call to *__aeabi_memclr*:

```
%> tiamclang -mcpu=cortex-m0 -Oz -S struct_inline.c
%> cat struct_inline.s
...
func:
        push      {r7, lr}
        ldr       r0, .LCPI0_0
        movs      r1, #24
        bl        __aeabi_memclr4
        pop       {r7, pc}
        .p2align       2
.LCPI0_0:
        .long   my_struct_inline
```

The *-mllvm -arm-memset-max-stores=<n>* option allows you to control the criteria used by the compiler to decide whether or not to inline a call to the *memset* or *memclr* function. If the above example is re-compiled with *-mcpu=cortex-m0 -Oz -mllvm -arm-memset-max_stores=6*, then the call to *memset* will get inlined since the clearing of *my_struct_inline* can be accomplished on Cortex-M0 with 6 store instructions:

```
%> tiarmclang -mcpu=cortex-m0 -Oz -mllvm -arm-memset-max-
 →stores=6 -S struct_inline.c
%> cat struct_inline.s
...
func:
        ldr       r0, .LCPI0_0
        movs      r1, #0
        str       r1, [r0]
        str       r1, [r0, #4]
        str       r1, [r0, #8]
        str       r1, [r0, #12]
```

```
        str     r1, [r0, #16]
        str     r1, [r0, #20]
        bx      lr
        .p2align        2
.LCPI0_0:
        .long   my_struct_inline
...
```

The optimal value for the argument *<n>* to use with the *-mllvm -arm-memset-max-stores=<n>* option will vary depending on each particular use-case. Adjusting this value will only be beneficial if you are able to control the limit as needed.

Note: Use Caution When Defining Symbols Inside an asm() Statement

Inlining a function that contains an asm() statement that contains a symbol definition when compiling with the tiarmclang compiler can cause a "symbol multiply defined" error.

Please see *Inlining Functions that Contain asm() Statements* for more details.

–moutline

Function outlining (aka "machine outlining") is an optimization that saves code size by identifying recurring sequences of machine code and replacing each instance of the sequence with a call to a new function that the identified sequence of operations.

Function outlining is enabled when the *-Oz* option is specified on the **tiarmclang** command line. There are 3 settings for the function outlining optimization when using the *-Oz* option:

The *-moutline* option is the default setting; it performs machine outlining within functions. This is less aggressive than *-moutline-inter-function*, but it is guaranteed to be applied only when doing so will reduce the net code size.

–moutline–inter–function

The *-moutline-inter-function* option can be specified in combination with the *-Oz* option to enable inter-function outlining. While this is the more aggressive of the function outlining settings, it does not always guarantee an overall code size reduction if, for example, outlining occurs across multiple functions in a given compilation unit yet only one of those functions is included in the linked application, the application will include the outlined code as well as the additional instructions required to call that code. However, it is likely to be beneficial when all functions defined in a given compilation unit are included in the linked application.

–mno–outline

The *-mno-outline* option can be used to disable function outlining for a given compilation unit when using the *-Oz* option.

Loop Unrolling

–funroll–loops, –fno–unroll–loops
>   Enable optimizer to unroll loops. The *-fno-unroll-loops* option disables this optimization.

Details of Optimizations Performed at Each Level

This table lists examples of optimizations performed at each optimization level. (These optimizations are *not* listed in the order they are performed.)

Optimization Level	Optimizations Performed
−O0	None
−O1	• Control Flow Simplification • Merge contiguous icmps into a memcmp • memcpy/memset/memcmp inlining • Constant Hoisting • Partially inline calls to library functions • Inline for always_inline functions • Global Variable Merging • Merge disjoint stack slots • Loop Strength Reduction • Loop Invariant Code Motion • Common Subexpression Elimination • Dead Argument Elimination • Machine code sinking • Peephole optimization • Tail Predication • Tail Duplication • Load/store optimization • Simple Register Coalescing • Copy Propagation • Conditional Constant Propagation • Called Value Propagation • Control Flow optimization • If-conversion • Thumb2 instruction size reduction • Dead Code Elimination • Loop Vectorization • Printf function specialization • *Small* memcpy/memset function specialization • Conditionally eliminate dead library calls • Loop Rotation • Loop Unrolling
−O2	• Performs all (**-O1**) optimizations, plus: • Function Integration/Inlining • Instruction speculation • Value Propagation • Jump Threading (non-DFA) • Tail Call Elimination • Merged Load/Store Motion • Global Value Numbering • Memory Dependence Analysis • Dead Store Elimination

3.1. Using the C/C++ Compiler   **217**

Using -x Option to Control Input File Interpretation

The tiarmclang compiler interprets source files with a recognized file extension in a predictable manner. The recognized file extensions include:

- **.c** - C source file

- **.C** or **.cpp** - C++ source file

- **.s** - GNU-syntax Arm assembly source file

- **.S** - GNU-syntax Arm assembly source file to be preprocessed by the compiler

- **.o** - object file to be forwarded on to the linker

The tiarmclang compiler also supports a **-x** option that permits you to dictate how an input file is to be interpreted by the compiler. This can be used to override default file extension interpretations or to instruct the compiler how to interpret a file extension that is not automatically recognized by the compiler.

−x `<language>`

Interpret subsequent input files on the command line as *<language>* type files.

The following *<language>* types are available to the **-x** option:

- **-x none** - Reset compiler to default file extension interpretation.

- **-x c** - Interpret subsequent input files as C source files.

- **-x c++** - Interpret subsequent input files as C++ source files.

- **-x assembler** - Interpret subsequent files as GNU-syntax Arm assembly source files.

- **-x assembler-with-cpp** - Interpret subsequent files as GNU-syntax Arm assembly source files to be preprocessed by the compiler.

- **-x ti-asm** - Interpret subsequent source files as legacy TI-syntax assembly source files causing the compiler to invoke the legacy TI-syntax Arm assembler to process them. For more information about handling legacy TI-syntax Arm assembly source files, please see *Invoking the TI-Syntax ARM Assembler from tiarmclang*.

Note:

The **-x<language>** option is position-dependent. A given **-x** option on the **tiarmclang** command line is in effect until the end of the command line *or* until a subsequent **-x** option is encountered on the command line.

In the following example, the *-x ti-asm* option is used to force *a_ti_asm.asm* to be interpreted as a TI-syntax assembly source file. The subsequent *-x none* is used to instruct tiarmclang to use its default input file interpretations for the remaining input files on the command line:

```
tiarmclang -mcpu=cortex-m0 -c -x ti-asm a_ti_asm.asm -x none a_
↪gnu_asm.s a_c.c
```

The following example uses input files with missing or non-standard file extensions. In this case, the *-x* options serve to inform tiarmclang how each input file is to be interpreted:

```
tiarmclang -mcpu=cortex-m0 -c -x c file1 -x assembler
↪file2.xyz
```

Passing Options to Other Tools from tiarmclang

This section of the *tiarmclang Compiler User Manual* describes how the tiarmclang's **-W<x>,** and **-X<y>** options can be used to pass options from tiarmclang to other tools in the compiler toolchain.

Passing Linker Options: *-Wl,* and *-Xlinker*

While the *-Wl,* (W + lowercase L + comma) option allows you to pass multiple linker options from tiarmclang to the linker using a single instance of the *-Wl,* option, the *-Xlinker* alternative may be useful when you want to explicitly control each particular linker option in a **tiarmclang** command line.

Using the *-Wl,* Option

The tiarmclang *-Wl,* option can be used to identify one or more linker command line options to be forwarded from tiarmclang to the linker when the linker is invoked from the **tiarmclang** command line.

tiarmclang [*options*] [*filenames*] **-Wl,*<opt-list>***

- **-Wl,** - is the tiarmclang option that prefixes a list of linker options

- *<opt-list>* - is a comma-separated list of one or more linker options

In the following example, the *-Wl,* option passes both the *--rom_model* linker option and the *lnkme.cmd* linker command file directly to the linker:

```
tiarmclang -mcpu=cortex-m0 hello.c -o a.out -Wl,--rom_model,
↪lnkme.cmd
```

Using *-Xlinker* Options

Alternatively, you can use the tiarmclang *-Xlinker* option to identify a single linker command line option to be forwarded from tiarmclang to the linker when the linker is invoked from the **tiarmclang** command line.

tiarmclang [*options*] [*filenames*] **-Xlinker** *<option>*

- **-Xlinker** - is the tiarmclang option that prefixes a single linker option

- *<option>* - is the linker option to be passed to the linker

The example command for the **-Wl,** option could also be written using the *-Xlinker* option as follows:

```
tiarmclang -mcpu=cortex-m0 hello.c -o a.out -Xlinker --rom_model␣
↪-Xlinker lnkme.cmd
```

You can find more information about linker options in the *Linker Options* section.

Passing Preprocessor Options: *-Wp,* and *-Xpreprocessor*

See *Preprocessor Options* for a list of options that can be used to control the preprocessor.

-Wp,<arg1>,<arg2>...
    Pass the comma separated arguments in *<argN>* to the preprocessor.

-Xpreprocessor <option>
    Pass *<option>* to the preprocessor.

Passing TI-Syntax Arm Assembler Options: *-Wti-a,* and *-Xti-assembler*

Using the *-Wti-a,* Option

The **tiarmclang** *-Wti-a,* option can be used to identify one or more legacy TI-syntax Arm assembler command line options that are to be forwarded from **tiarmclang** to the TI-syntax Arm assembler when it is invoked from the **tiarmclang** command line.

    tiarmclang *-x ti-asm <ti_asm_file>* [*options*] [*filenames*] **-Wti-a,** *<opt-list>*

- **-Wti-a,** - is the **tiarmclang** option that prefixes a list of legacy TI-syntax Ar assembler options

- *<opt-list>* - is a comma-separated list of one or more TI-syntax Arm assembler options

Using *-Xti-assembler* Options

Alternatively, you can use the **tiarmclang** *-Xti-assembler* option to identify a legacy TI-syntax Arm assembler command line option that is to be forwarded from **tiarmclang** to the TI-syntax assembler when it is invoked from the **tiarmclang** command line.

> **tiarmclang** *-x ti-asm <ti)asm_file>* [*options*] [*filenames*] **-Xti-assembler** *<option>*

- **-Xti-assembler** - is the **tiarmclang** option that prefixes a single legacy TI-syntax Arm assembler option
- *<option>* - is the option to be passed to the TI-syntax Arm assembler

Examples

While the *-Wti-a,* option allows you to pass multiple options from **tiarmclang** to the legacy TI-syntax Arm assembler in a single instance, the *-Xti-assembler* alternative may be useful in some instances where you want to explicitly control each particular TI-syntax Arm assembler option in a **tiarmclang** command line.

In the following example, the *-Wti-a,* option is used to pass both a *-d* and a *--include_file* option directly to the legacy TI-syntax Arm assembler:

```
tiarmclang -mcpu=cortex-m0 -c -x ti-asm ti_src.asm -Wti-a,-d=__
↪ti__=1,--include_file=ti.inc
```

The above command could also be written using the *-Xti-assembler* option as follows:

```
tiarmclang -mcpu=cortex-m0 -c -x ti-asm ti_src.asm -Xti-
↪assembler -d=__ti__=1 -Xti-assembler --include_file=ti.inc
```

You can find more information about TI-syntax Arm assembler options in the *TI-Syntax Arm Assembler* section or in the Arm Assembly Language Tools User's Guide.

Diagnostic Options

Controlling Error, Warning, and Remark Diagnostics

The tiarmclang compiler provides the following options to assist with controlling what errors, warnings, and remarks are emitted during compilation:

-R<remark>
> Enable the specified remark category.

-Wall

> Enable most warning categories.

-Werror

> Treat detected warnings as errors.

-Werror=<warning-category>

> Treat detected warnings in the specified category as errors.

-W<warning-category>

> Enable the specified warning category.

-Wno-<warning-category>

> Disable the specified warning category.

Optimization Feedback Options

The following options can be used to instruct the tiarmclang compiler to emit information about the different optimizations and code transformations that are performed during compilation:

-Rpass-analysis=<arg>

> Report transformation analysis from optimization passes whose name matches the given POSIX regular expression.

-Rpass-missed=<arg>

> Report missed transformations by optimization passes whose name matches the given POSIX regular expression.

-Rpass=<arg>

> Report transformations performed by optimization passes whose name matches the given POSIX regular expression.

Debug Options

The tiarmclang compiler supports the following command-line option to facilitate generation of C/C++ source debug information:

-g, -gdwarf

> Generate source-level debug information with the default DWARF version (3)

-gdwarf-2

> Generate source-level debug information with DWARF version 2

-gdwarf-3

> Generate source-level debug information with DWARF version 3

3.1. Using the C/C++ Compiler

-gdwarf-4

> The tiarmclang compiler does not support generating DWARF version 4 debug information yet. If the *-gdwarf-4* option is specified, the compiler will emit a warning diagnostic and emit DWARF version 3 instead.

-gdwarf-5

> The tiarmclang compiler does not support generating DWARF version 5 debug information yet. If the *-gdwarf-5* option is specified, the compiler will emit a warning diagnostic and emit DWARF version 3 instead.

Instrumentation Options

Stack Smashing Detection Options

The compiler provides the capability to instrument protection for stack smashing attacks.

See *Stack Smashing Detection*.

Function Entry/Exit Hook Options

The compiler provides the capability to instrument functions with entry and exit hook function calls using the *-finstrument-functions* option:

-finstrument-functions

> For each function being compiled, instruct the compiler to generate a call to the entry hook function, *__cyg_profile_func_enter*, just after entry to a given function, and a call to exit hook function, *__cyg_profile_func_exit*, just prior to exit from a given function.
>
> The compiler also calls *__cyg_profile_func_enter* and *__cyg_profile_func_exit* on behalf of a function that is inlined into another function. This means that an addressable version of an inlined function must be available in the linked application to facilitate lookup of the inlined function symbol. If all uses of a function are inlined, the definition of the inlined function may incur some growth in code size for the linked application.

Enabling Use of Function Entry/Exit Hooks

To enable the use of function entry/exit hooks in your application, you need to provide definitions of:

- **__cyg_profile_func_entry**

  The signature of the *__cyg_profile_func_enter* function is as follows:

```
void __cyg_profile_func_entry(void *this_fcn, void *call_site);
```

An example definition of this function might look like this:

```cpp
#include "func_timer.h"

extern "C" {

// Entry Hook Function
__attribute__((no_instrument_function))
void __cyg_profile_func_enter(void *this_fcn, void *call_site) {
  // Non-NULL function address is required
  if (!this_fcn) return;

  // Find function address in function timer map;
  // If this is the first call to the specified function,
  // then create a timer record for it and insert record into map
  auto func_iter = func_timer_map.find((unsigned long)this_fcn);
  func_timer_record *func_timer;
  if (func_iter == func_timer_map.end()) {
    func_timer = new func_timer_record((unsigned long)this_fcn);
    func_timer_map[(unsigned long)this_fcn] = func_timer;
  }
  else {
    func_timer = func_iter->second;
  }

  // If function is not already on the call stack, start the
→clock
  if (func_timer->recur_level == 0) {
    func_timer->clock_start = clock();
  }
  else {
    func_timer->recur_level++;
  }
}

} /* extern "C" */
```

- **__cyg_profile_func_exit**

    The signature of the *__cyg_profile_func_exit* function is as follows:

```
void __cyg_profile_func_exit(void *this_fcn, void *call_site);
```

An example definition of this function might look like this:

```cpp
#include "func_timer.h"

extern "C" {

// Function Exit Hook
__attribute__((no_instrument_function))
void __cyg_profile_func_exit(void *this_fcn, void *call_site) {
  // Non-NULL function address is required
  if (!this_fcn) return;

  // Find function in function timer map; error if not found
  auto func_iter = func_timer_map.find((unsigned long)this_fcn);
  func_timer_record *func_timer;
  if (func_iter == func_timer_map.end()) {
    printf("ERROR: expected function in func_timer_map\n");
    return;
  }

  func_timer = func_iter->second;

  // If we're about to remove the function from the call stack,
  // add elapsed time to total accumulated time for this function
  if (func_timer->recur_level == 1) {
    func_timer->acc_func_time += (long)(clock() - func_timer->
clock_start);
  }

  func_timer->recur_level--;
}

} /* extern "C" */
```

For both of the above functions, the first argument, *this_fcn*, is the address of the start of the current function, which can be looked up in the symbol table, and the second argument, *call_site*, is the return address of the current function that can be used to determine where the current function was called from.

Note: Define __cyg_profile_func_enter and __cyg_profile_func_exit as "C" Symbols

---

When using the *-finstrument-functions* option with a C++ source file, the tiarmclang compiler instruments a given function with calls to *__cyg_profile_func_enter* and *__cyg_profile_func_exit* using the "C" names of those function symbols. Consequently, when you define the *__cyg_profile_func_enter* and *__cyg_profile_func_exit* functions for use in a C++ application, you must enclose the definitions of these functions in an *extern "C"* construct, as indicated in the examples above.

Disabling Instrumentation with *no_instrument_function* Attribute

While applying the *-finstument-functions* option to an application, there may be some functions that you may want to exclude from being instrumented, such as the definitions of *__cyg_profile_func_enter* and *__cyg_profile_func_exit* described above. In such cases, the *no_instrument_function* function attribute can be applied to prevent calls to the entry and exit hooks from being generated for a given function.

The above definition of *__cyg_profile_func_enter* contains an example of how to apply the *no_instrument_function* attribute to a function:

```
__attribute__((no_instrument_function))
void __cyg_profile_func_enter(void *this_fcn, void *call_site) {
    ...
}
```

Function Entry/Exit Hooks Example

One useful application of function entry and exit hook functions is to gather profile data for the functions in an application. The above definitions of *__cyg_profile_func_enter* and *__cyg_profile_func_exit* collect the accumulated time spent in each instrumented function in an application.

The profile data is collected and recorded in a map of *function_timer_record* objects as detailed in *func_timer.h*:

```
#include <stdio.h>
#include <time.h>
#include <map>


class func_timer_record {
  public:
    unsigned long func_address;
    unsigned int  recur_level;
    clock_t       clock_start;
```

(continues on next page)

```
    long            acc_func_time;

    func_timer_record(unsigned long func_addr) :
      func_address(func_addr),
      recur_level(0),
      clock_start(0),
      acc_func_time(0) { }
    ~func_timer_record() { }
} func_timer_record;

extern std::map<unsigned long, func_timer_record *> func_timer_
↪map;

__attribute__((no_instrument_function)) void report_function_
↪times(void);
```

In this simplistic example, it is anticipated that the application being profiled will call *report_function_times* that writes out a comma-separated list of the function addresses and their corresponding recorded execution times:

```
#include "func_timer.h"
#include <list>

std::map<unsigned long, func_timer_record *> func_timer_map;

__attribute__((no_instrument_function)) void report_function_
↪times(void) {
  // Print CSV output of function addresses and corresponding␣
↪times
  std::list<function_timer_record *> curr_func_list;
  for (auto it = func_timer_map.begin(); it != func_timer_map.
↪end(); ++it) {
    unsigned long curr_func_addr = it->first;
    unsigned long curr_func_time = (it->second)->acc_func_time;
    printf("func_address: 0x%08lx, cumulative time in function:␣
↪%ld\n",
          curr_func_addr, curr_func_time);
  }
}
```

The application to be profiled can then be compiled with the *-finstrument-functions* option:

```
%> tiarmclang -mcpu=cortex-m4 -finstrument-functions <app source␣
↪files> \
```

3.1. Using the C/C++ Compiler

```
func_timer.cpp func_enter.cpp func_exit.cpp -o app.out ...
```

While the functions defined in the application source files will be instrumented, the instrumentation functions themselves will not since they have been annotated with the *no_instrument_function* attribute.

When loaded and run, *app.out* produces the function time statistics that can then be analyzed and processed by a program that has access to the *app.out* file's symbol table.

Control Flow Integrity Options

The compiler includes an implementation of a number of control flow integrity (CFI) schemes, which are designed to abort the program upon detecting certain forms of undefined behavior that can potentially allow attackers to subvert the program's control flow.

See *Control Flow Integrity*.

3.2 C/C++ Language Implementation

Contents:

3.2.1 Data Types

Scalar Data Types

The table below lists the size and range of each scalar type as supported in the tiarmclang compiler. Many of the minimum and maximum values for each range are available as standard macros in the C standard header file *limits.h*.

The storage and alignment of data types is described in *Object Representation*.

Type	Size	Min Value	Max Value
signed char	8 bits	-128	127
char, unsigned char, bool	8 bits	0	255
short, signed short	16 bits	-32768	32767
unsigned short	16 bits	0	65535
int, signed int, long, signed long	32 bits	-2147483648	2147483647
enum	packed	-2147483648	2147483647
unsigned int, unsigned long, wchar_t	32 bits	0	4294967295
long long, signed long long	64 bits	-9223372036854775808	9223372036854775807
unsigned long long	64 bits	0	18446744073709551615
float	32 bits	1.175494e-38	3.40282346e+38
double, long double	64 bits	2.22507385e-308	1.79769313e+308
pointers, references, data member ptrs	32 bits	0	0xFFFFFFFF

Notes:

- The "plain" *char* type has the same representation as either *signed char* or *unsigned char*. The *-fsigned-char* and *-funsigned-char* options control whether "plain" *char* is signed or unsigned. The default is unsigned.

- The *wchar_t* type has the same representation as *unsigned int*. The tiarmclang runtime libraries do not support a 16-bit *wchar_t* type. Attempts to use the **tiarmclang** *-fshort-wchar* option may cause issues when linked with the tiarmclang runtime libraries.

- Further discussion about the size of *enum* types can be found below in the *Enum Type Storage*.

- Specified minimum values for floating-point types in the table above indicate the smallest precision value > 0.

- Negative values for signed types are represented using two's complement.

- 64-bit data types are aligned to 64-bit (8-byte) boundaries.

- Both 32-bit pointers and 64-bit pointers are aligned to 32-bit (4-byte) boundaries.

- Arm Cortex-M type processor variants supported by the tiarmclang compiler are little-endian. Arm Cortex-R type processor variants supported by the tiarmclang compiler may

be big-endian or little-endian depending on the use of command-line options.

Enum Type Storage

The type of the storage container for an enumerated type is the smallest integer type that contains all the enumerated values. The container types for enumerators are shown in the following table:

Lower Bound Range	Upper Bound Range	Enumerator Type
0 to 255	0 to 255	unsigned char
-128 to 1	-128 to 127	signed char
0 to 65535	256 to 65535	unsigned short
-128 to 1, -32768 to -129	128 to 32767, -32768 to 32767	signed short
0 to 4294967295	2147483648 to 4294967295	unsigned int
-32768 to -1, -2147483648 to -32769, 0 to 2147483647	32767 to 2147483647, -2147483648 to 2147483647, 65536 to 2147483647	signed int

The compiler determines the type based on the range of the lowest and highest elements of the enumerator.

For example, the following code results in an enumerator type of int:

```
enum COLORS {
  green  = -200,
  blue   = 1,
  yellow = 2,
  red    = 60000
};
```

The following code results in an enumerator type of short:

```
enum COLORS {
  green  = -200,
  blue   = 1,
  yellow = 2,
  red    = 30000
};
```

Enum Type Size

An enum type is represented by an underlying integer type. The size of the integer type and whether it is signed is based on the range of values of the enumerated constants.

By default, the tiarmclang uses the smallest possible byte size for the enumeration type. The underlying type is the first type in the following list in which all the enumerated constant values can be represented: *signed char*, *unsigned char*, *short*, *unsigned short*, *int*, *unsigned int*, *long*, *unsigned long*, *long long*, *unsigned long long*. This default behavior is equivalent to the effect of using the **tiarmclang** *-fshort-enums* option.

In strict c89/c99/c11 mode, the compiler will limit enumeration constants to those values that fit in *int* or *unsigned int*.

For C++ and gnuXX C dialects (relaxed c89/c99/c11), the compiler allows enumeration constants up to the largest integral type (64 bits).

You can alter the default compiler behavior using the *-fno-short-enums* option. When the *-fno-short-enums* option is used in strict c89/c99/c11 mode, the enumeration type used to represent an *enum* will be *int*, even if the values of the enumeration constants fit into a smaller integer type.

When the *fno-short-enums* option is used with C++ or gnuXX C dialects, the underlying enumeration type will be the first type in the following list in which all the enumerated constant values can be represented: *int*, *unsigned int*, *long*, *unsigned long*, *long long*, *unsigned long long*.

The following enum uses 8 bits instead of 32 bits by default (since *-fshort-enums* option behavior is in effect):

```
enum example_enum {
  first  = -128,
  second = 0,
  third  = 127
};
```

The following enum fits into 16 bits instead of 32 by default:

```
enum a_short_enum {
  bottom = -32768,
  middle = 0,
  top    = 32767
};
```

Note: Do not link object files compiled with the *-fno-short-enums* option with object files that were compiled without it. If you use the *-fno-short-enums* option, you must use it with all of your C/C++ files; otherwise, you will encounter errors that cannot be detected until run time.

3.2.2 Characteristics of Arm C

Please see *C Language Variants (-std)* for supported C language variants as well as the options that control the language standard used, including GNU language extensions.

C11 atomic operations are supported as follows:

- On Cortex-M3, Cortex-M4, Cortex-R4, and Cortex-R5, atomic operations are implemented using processor-supported exclusive access instructions.

- On Cortex-M0, atomic operations are not supported.

In addition, the compiler supports many of the features described in the Arm C Language Extensions for all Cortex-M and Cortex-R variants. See *C Language Extensions*.

The ANSI/ISO standard identifies some features of the C language that may be affected by characteristics of the target processor, run-time environment, or host environment. This set of features can differ among standard compilers. Please see *Arm C Implementation-Defined Behavior*.

The following C library features are *not* currently supported for TI Arm:

- The run-time library includes the header file <locale.h>, but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hardcoded to the behavior of the C locale, and attempting to install a different locale by way of a call to setlocale() will return NULL.

- Some run-time functions and features in the C99/C11 specifications are not supported. See *Library Functions (J.3.12)* for more details.

In addition to support for the C language standard, the compiler supports many extensions that are commonly supported by C language compilers. See *C Language Extensions*.

3.2.3 C Language Extensions

The tiarmclang compiler supports many of the features described in the Arm C Language Extensions (ACLE) specification.

A significant benefit of using ACLE features in your C/C++ source code is that ACLE features are supported by other Arm ACLE-compliant compilers (including the legacy TI Arm C/C++ Compiler). C/C++ code developed using ACLE extensions will be portable from one Arm ACLE-compliant compiler to another.

The ACLE features that are supported in tiarmclang are those that are applicable to the Arm Cortex-M and Cortex-R processor variants that the compiler supports.

Extensions supported in tiarmclang include pre-defined macro symbols, attributes, and intrinsics. Standard Clang language extensions, such as those described in the Clang documentation in the Clang Language Extensions section of the Clang documentation, are generally supported.

A description of selected extensions is provided in the following sections of this user guide:

| *Pre-Defined Macro Symbols* |
| *Attributes* |
| *Intrinsics* |

3.2.4 Arm C Implementation-Defined Behavior

The C standard requires that conforming implementations provide documentation on how the compiler handles instances of implementation-defined behavior.

The TI Arm Clang compiler officially supports a freestanding environment. The C standard does not require a freestanding environment to supply every C feature; in particular the library need not be complete. However, the TI compiler strives to provide most features of a hosted environment.

The section numbers in the lists that follow correspond to section numbers in Appendix J of the C99 standard and Appendix J of the C11 standard. The numbers in parentheses at the end of each item are sections in each standard that discuss the topic. Certain items listed in Appendix J of the C99 standard have been omitted from this list.

Translation (J.3.1)

- The compiler and related tools emit diagnostic messages with several distinct formats. The more common form is the following:

    source-file:line-number:char-number: description [diagnostic-flag]

  Where 'description' is a text description of the error, and 'diagnostic-flag' is an option flag of the form -Wflag for messages that can be suppressed. (1.3.6)

- Diagnostic messages are emitted to stderr; any text on stderr may be assumed to be a diagnostic. If any errors are present, the tool will exit with an exit status indicating failure (non-zero). (C99/C11 3.10, 5.1.1.3)

- Each whitespace sequence is collapsed to a single space. For aesthetic reasons, the first token on each non-directive line of output is preceded with sufficient spaces that it appears in the same column as it did in the original source file. (C99/C11 5.1.1.2)

Environment (J.3.2)

- The compiler interprets the physical source file multibyte characters as UTF-8.

  Wide character (wchar_t) types and operations are supported by the compiler. However, wide character strings may not contain characters beyond 7-bit ASCII. The encoding of wide characters is 7-bit ASCII, 0 extended to the width of the wchar_t type. (C99/C11 5.1.1.2)

- The name of the function called at program startup is "main." Its parameter list may be "(void)" or "(int argc, char *argv[])." (C99/C11 5.1.2.1)

- Program termination does not affect the environment; there is no way to return an exit code to the environment. By default, the program is known to have halted when execution reaches the special C$$EXIT label. (C99/C11 5.1.2.1)

- In relaxed ANSI mode, the compiler accepts "void main(void)" and "void main(int argc, char *argv[])" as alternate definitions of main. The alternate definitions are rejected in strict ANSI mode. (C99/C11 5.1.2.2.1)

- If space is provided for program arguments at link time with the --args option and the program is run under a system that can populate the .args section (such as CCS), argv[0] will contain the filename of the executable, argv[1] through argv[argc-1] will contain the command-line arguments to the program, and argv[argc] will be NULL. Otherwise, the value of argv and argc are undefined. (C99/C11 5.1.2.2.1)

- Interactive devices include stdin, stdout, and stderr (when attached to a system that honors CIO requests). Interactive devices are not limited to those output locations; the program may access hardware peripherals that interact with the external state. (C99/C11 5.1.2.3)

- Signals are not supported. The function signal is not supported. (C99/C11 7.14, 7.14.1.1)

- The library function getenv is implemented through the CIO interface. If the program is run under a system that supports CIO, the system performs getenv calls on the host system and passes the result back to the program. Otherwise the operation of getenv is undefined. No method of changing the environment from inside the target program is provided. (C99 7.20.4.5, C11 7.22.4.6)

- The system function is not supported. (C99 7.20.4.6, C11 7.22.4.8)

Identifiers (J.3.3)

- Multibyte characters are allowed in identifiers whose UTF-8 decoded value is within the allowed ranges specified in Appendix D of ISO/IEC 9899:2011. The '$' character is allowed in identifiers.

- The number of significant initial characters in an identifier is unlimited. (C99/C11 5.2.4.1, 6.4.2)

Characters (J.3.4)

- The number of bits in a byte (CHAR_BIT) is 8. (C99/C11 3.6)

- The execution character set is the same as the basic execution character set: plain ASCII. Characters in the ISO 8859 extended character set are also supported. (C99/C11 5.2.1)

- The values produced for the standard alphabetic escape sequences are as follows: (C99/C11 5.2.2)

Escape Sequence	ASCII Meaning	Integer Value
\a	BEL (bell)	7
\b	BS (backspace)	8
\f	FF (form feed)	12
\n	LF (line feed)	10
\r	CR (carriage return)	13
\t	HT (horizontal tab)	9
\v	VT (vertical tab)	11

- The value of a char object into which any character other than a member of the basic execution character set has been stored is the ASCII value of that character. (C99/C11 6.2.5)

- Plain char is identical to unsigned char, but can be changed to signed char with the -fsigned-char option. (C99/C11 6.2.5, 6.3.1.1)

- The source character set and execution character set are identical. (C99/C11 6.4.4.4, 5.1.1.2)

- The value of an integer character constant containing more than one character is the same as the last source character. The compiler will emit a warning when an integer character constant containing more than one character is used. There are no characters or escape sequences that do not map to a single-byte execution character. (C99/C11 6.4.4.4)

- The compiler does not support multibyte characters in wide character constants. There are no wide characters or escape sequences that do not map to a single wide execution character. (C99/C11 6.4.4.4)

- The compiler currently supports only one locale, "C". (C99/C11 6.4.4.4)

- The compiler currently supports only one locale, "C". (C99/C11 6.4.5)

- The compiler does not support multibyte characters in string literals. There are no escape sequences that do not map to a single execution character. (C99/C11 6.4.5)

- The wchar_t type is 32-bits wide and is equivalent to the uint32_t type (unsigned int).

Integers (J.3.5)

- No extended integer types are supported. (C99/C11 6.2.5)

- Negative values for signed integer types are represented as two's complement, and there are no trap representations. (C99/C11 6.2.6.2)

- No extended integer types are supported, so there is no change to the integer ranks. (C99/C11 6.3.1.1)

- When an integer is converted to a signed integer type which cannot represent the value, the value is truncated (without raising a signal) by discarding the bits which cannot be stored in the destination type; the lowest bits are not modified. (C99/C11 6.3.1.3)

- Right shift of a signed integer value performs an arithmetic (signed) shift. The bitwise operations other than right shift operate on the bits in exactly the same way as on an unsigned value. That is, after the usual arithmetic conversions, the bitwise operation is performed without regard to the format of the integer type, in particular the sign bit. (C99/C11 6.5)

Floating Point (J.3.6)

- The accuracy of floating-point operations (+ - * /) is bit-exact. The accuracy of library functions that return floating-point results is not specified. (C99/C11 5.2.4.2.2)

- The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in <stdio.h>, <stdlib.h>, and <wchar.h> is not specified. (C11 5.2.4.2.2)

- The compiler does not provide non-standard values for FLT_ROUNDS (C99/C11 5.2.4.2.2)

- The compiler does not provide non-standard negative values of FLT_EVAL_METHOD (C99/C11 5.2.4.2.2)

- The rounding direction when an integer is converted to a floating-point number is IEEE-754 "round to even". (C99/C11 6.3.1.4)

- The rounding direction when a floating-point number is converted to a narrower floating-point number is IEEE-754 "round to even". (C99/C11 6.3.1.5)

- For floating-point constants that are not exactly representable, the implementation uses the nearest representable value. (C99/C11 6.4.4.2)

- The compiler does not contract float expressions, except when -ffast-math is used. (C99/C11 6.5)

- The default state for the FENV_ACCESS pragma is off. (C99/C11 7.6.1)

- The compiler does not define any additional float exceptions (C99/C11 7.6, 7.12)

- The default state for the FP_CONTRACT pragma is off. (C99/C11 7.12.2)

- The "inexact" floating-point exception cannot be raised if the rounded result equals the mathematical result. (F.9)

- The "underflow" and "inexact" floating-point exceptions cannot be raised if the result is tiny but not inexact. (F.9)

Arrays and Pointers (J.3.7)

- When converting a pointer to an integer or vice versa, the pointer is considered an unsigned integer of the same size, and the normal integer conversion rules apply. If the bitwise representation of the destination can hold all of the bits in the bitwise representation of the source, the bits are copied exactly. (C99/C11 6.3.2.3)

- The size of the result of subtracting two pointers to elements of the same array is the size of ptrdiff_t, which is 4 bytes. (C99/C11 6.5.6)

Hints (J.3.8)

- When the optimizer is used, the register storage-class specifier is ignored. When the optimizer is not used, the compiler will preferentially place register storage class objects into registers to the extent possible. The compiler reserves the right to place any register storage class object somewhere other than a register. (C99/C11 6.7.1)

- The inline function specifier is ignored unless the optimizer is used. For other restrictions on inlining, as well as ways to control inlining behavior, see the compiler manual. (C99/C11 6.7.4)

Structures, unions, enumerations, and bit-fields (J.3.9)

- A "plain" int bit-field is treated as a signed int bit-field. (C99/C11 6.7.2, 6.7.2.1)

- In addition to _Bool, signed int, and unsigned int, the compiler allows char, signed char, unsigned char, signed short, unsigned short, signed long, unsigned long, signed long long, unsigned long long, and enum types as bit-field types. (C99/C11 6.7.2.1)

- Atomic types are not allowed as bit-field types

- Bit-fields may not straddle a storage-unit boundary. (C99/C11 6.7.2.1)

- Bit-fields are allocated in endianness order within a unit. See the compiler manual for details. (C99/C11 6.7.2.1)

- Non-bit-field members of structures are aligned as required by the type of the member. There are user controls to override this behavior; see the compiler manual for details. (C99/C11 6.7.2.1)

- The integer type underlying each enumerated type is described in the compiler manual. (C99/C11 6.7.2.2)

Qualifiers (J.3.10)

- The compiler does not shrink or grow volatile accesses. It is the user's responsibility to make sure the access size is appropriate for devices that only tolerate accesses of certain widths.

    - The compiler does not change the number of accesses to a volatile variable unless absolutely necessary. In some cases, the compiler will be forced to use two accesses, one for the read and one for the write, it is not guaranteed that the compiler will be able to map such expressions to an instruction with a single memory operand. It is not guaranteed that the memory system will lock that memory location for the duration of the instruction.

    - The compiler will not reorder two volatile accesses, but it may reorder a volatile and a non-volatile access, so volatile cannot be used to create a critical section. Use some sort of lock if you need to create a critical section. (C98/C11 6.7.3)

Preprocessing directives (J.3.11)

- The compiler does not support pragmas that refer to headers. (C11 6.4, 6.4.7)

- The sequences are mapped to external source file names in both forms of the #include directive (C11 6.4.7)

- The value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (both are plain ASCII). (C99/C11 6.10.1)

- Single-character constants in a constant expression that controls conditional inclusion have a non-negative value. (C11 6.10.1)

- Include directives may have one of two forms, < > or " ". For both forms, the compiler will look for a real file on-disk by that name using the "system" or "user" include file search path. See the compiler manual for details on how the system and user include file search path can be controlled with environment variables and command-line options. (C99/C11 6.4.7)

    - The compiler uses the "system" include file search path to search for an included < > delimited header file. See the compiler manual for details on how the system and user include file search path can be controlled with environment variables and command-line options. (C99/C11 6.10.2)

    - The compiler uses the "user" include file search path to search for an included " " delimited header file. See the compiler manual for details on how the system and user include file search path can be controlled with environment variables and command-line options. (C99/C11 6.10.2)

- As a result of macro replacement, the sequence of tokens should be either a single string literal or a sequence of preprocessing tokens, starting with < and ending with >. Sequences of whitespace characters are replaced by a single space. (C99/C11 6.10.2)

- There is no arbitrary nesting limit for #include processing. (C99/C11 6.10.2)

- The # operator inserts a \ character before the \ character that begins a universal character name. (C11 6.10.3.2)

- See the compiler manual for a description of the recognized non-standard pragmas. (C99/C11 6.10.6)

- The date and time of translation are always available from the host. (C99 6.10.8, C11 6.10.8.1)

Library Functions (J.3.12)

- Almost all of the library functions required for a hosted implementation are provided by the TI library. (C99/C11 5.1.2.1)

  - However, the following list of run-time functions and features are not implemented or fully supported:

    * fenv.h

      · Floating-point exception functions

    * inttypes.h

      · wcstoimax() / wcstoumax()

    * stdio.h

      · The %e specifier may produce "-0" when "0" is expected by the standard snprintf() does not properly pad with spaces when writing to a wide character array

    * stdlib.h

      · vfscanf() / vscanf() / vsscanf() return value on floating point matching failure is incorrect

    * wchar.h

      · fgetws() / fputws()

      · mbrlen()

      · mbsrtowcs()

      · wcscat()

      · wcschr()

> · wcscmp() / wcsncmp()
>
> · wcscpy() / wcsncpy()
>
> · wcsftime()
>
> · wcsrtombs()
>
> · wcsstr()
>
> · wcstok()
>
> · wcsxfrm()
>
> · Wide character print / scan functions
>
> · Wide character conversion functions
>
> ∗ signal.h
>
> > · signal()
> >
> > · raise()

- The format of the diagnostic printed by the assert macro is "Assertion failed, *(assertion macro argument)*, file *file*, line *line*". (C99/C11 7.2.1.1)

- The feraiseexcept function is not supported. (C11 7.6.2.3)

- No strings other than "C" and "" may be passed as the second argument to the setlocale function (C99/C11 7.11.1.1)

- The types defined for float_t and double_t when the value of the FLT_EVAL_METHOD macro is less than 0 or greater than 2 are float and double, respectively. (C99/C11 7.12)

- On underflow range errors, the mathematics functions return 0.0 and the errno is set to ERANGE. Floating-point exceptions raised using the feraiseexcept function are not supported. (C99/C11 7.12.1)

- The base-2 logarithm of the modulus used by the remquo functions in reducing the quotient is 31. The last 31bits of the quotient are returned (values up to $2^{\{31\}}$). (C99/C11 7.12.10.3)

- No signal handling is supported. (C99/C11 7.14.1.1)

- The +INF, -INF, +inf, -inf, NAN, and nan styles can be used to print an infinity or NaN. (C99 7.19.6.1, 7.24.2.1; C11 7.21.6.1, 7.29.2.1)

- The output for %p conversion in the fprintf or fwprintf function is the same as %x of the appropriate size. (C99 7.19.6.1, 7.24.2.1; C11 7.21.6.1, 7.29.2.1)

- Any n-char or n-wchar sequence in a string, representing a NaN, that is converted by the strtod, strtof, or strtold functions, is ignored. The wcstod, wcstof, and wcstold functions are not supported. (C99 7.20.1.3, 7.24.4.1.1; C11 7.22.1.3, 7.29.4.1.1)

- The strtod, strtof, or strtold functions set errno to ERANGE when underflow occurs. The wcstod, wcstof, and wcstold functions are not supported. (C99 7.20.1.3, 7.24.4.1.1; C11 7.22.1.3, 7.29.4.1.1)

- Open streams with unwritten buffered data are flushed, open streams are closed, and temporary files are removed when the _Exit function is called. The function abort does not close or flush open streams nor does it remove temporary files when it is called. (C99 7.20.4.1, 7.20.4.4, C11 7.22.4.1, 7.22.4.5)

- The termination status returned to the host environment by the abort, exit, _Exit, or quickexit function is not returned to the host environment. (C99 7.20.4.1, 7.20.4.3, 7.20.4.4, C11 7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7)

- The system function is not supported. (C99 7.20.4.6, C11 7.22.4.8)

Architecture (J.3.13)

- The values or expressions assigned to the macros specified in the headers float.h, limits.h, and stdint.h are described along with the sizes and format of integer types in the compiler manual. (C99 5.2.4.2, 7.18.2, 7.18.3; C11 5.2.4.2, 7.20.2, 7.20.3)

- Thread storage is not supported. (C11 6.2.4)

- The number, order, and encoding of bytes in any object are described in the compiler manual. (C99/C11 6.2.6.1)

- Valid alignments as well as extended alignments up to 2^{28} bytes are supported. (C11 6.2.8)

- The value of the result of the sizeof and _Alignof operators is the storage size for each type, in terms of bytes. See the compiler manual (C99/C11 6.5.3.4)

Locale-specific behavior (J.4)

- The behavior of these points is dependent on the implementation of the C library. The compiler currently supports only one locale, "C".

3.2.5 Characteristics and Implementation of Arm C++

The tiarmclang compiler supports C++ as defined in the ANSI/ISO/IEC 14882:2017 standard (C++17), including these features:

- Complete C++ standard library support, with exceptions noted below.

- C++ Templates

- Exception handling, which can be enabled with the *-fexceptions* compiler option (see *C++ Exception Handling*)

- Run-time type information (RTTI), which can be enabled with the *-frtti* compiler option.

The following features are *not* implemented or fully supported:

- Features related to threads and concurrency, such as:

  - std::thread

  - std::unique_lock

  - std::shared_mutex

  - std::execution

  - C++ atomic operations

  - thread-local storage

- Some features related to memory management, such as:

  - std::pmr::memory_resource

  - std::align_val_t

- The Filesystem library

- The C++17 Mathematical Special Functions library

Please see *C++ Language Variants (-std)* for supported C++ language variants as well as the options that control the language standard used.

C++ Exception Handling

The tiarmclang compiler now supports the C++ exception handling features defined by the ANSI/ISO 14882 C++ Standard. See *The C++ Programming Language, Third Edition* by Bjarne Stroustrup.

(The C library provides no built-in support for C exception handling.)

To enable C++ exception handling, use the compiler's *-fexceptions* option. By default, the compiler provides no exception handling support; this is the equivalent of using the *-fno-exceptions* option.

For exceptions to work correctly, the following must be true:

- All C++ files in the application must have been compiled with the *-fexceptions* option, regardless of whether exceptions occur in that file. Mixing exception-enabled and exception disabled object files and libraries can lead to undefined behavior.

- You must link an application with a run-time-support library that contains exception handling support. The run-time-support libraries provided with the tiarmclang compiler come in both exception-enabled and exception-disabled forms. When you use automatic library selection

(the default), the linker automatically selects the correct library. If you specify the library manually, you must use a version in a sub-directory called `except` of the usual directory for your target. See *Library Naming Conventions*.

- The *-fno-rtti* option must not be used to compile any C++ files in the application.

Using the *-fexceptions* option causes the compiler to insert exception handling code. This inserted code increases the size of the program, but not by a large amount. Compiling for exception support also slightly increases the data size to accommodate the exception handling tables. Compiling for exception support has a minimal execution time cost if exceptions are never thrown.

If C++ exceptions are enabled, the **.ARM.exidx** and **.ARM.extab** sections are generated and initialized.

To optimize the size of exception-handling tables, the *nothrow* attribute may be applied to a function to inform the compiler that the function cannot throw an exception. See *nothrow* for details.

For details about C++ exception handling and the Arm ABI, see the Application Binary Interface for the Arm Architecture documentation. For device-specific details, see the Arm Developer information; for example, exception handling documentation for ARMv8-M is available here.

See *C Exception Handler Calling Convention* for information about C language exception handling.

3.2.6 Pre-Defined Macro Symbols

The tiarmclang compiler supports the use of pre-defined macro symbols. These are compile-time symbols that are defined with a value based on how the compiler is invoked.

Note: Viewing the List of Pre-Defined Macro Symbols for a Given Compilation

To view the pre-defined macro symbols that are defined for a given **tiarmclang** option combination, you can compile using the *-E* and *-dM* options. For example,

```
%> tiarmclang -mcpu=cortex-m0 -E -dM test.c
```

emits to *stdout* the list of pre-defined macro symbols that are defined when compiling with the *-mcpu=cortex-m0* option.

The following sub-sections contain tables listing the various pre-defined macro symbols that are created by the tiarmclang compiler:

TI-Specific Pre-Defined Macro Symbols

The tiarmclang compiler pre-defines the following TI-specific macro symbols, which can be used when configuring source code to be compiled on the basis of the compiler vendor identification or on the basis of the tiarmclang compiler version being used:

TI-Specific Compiler Predefined Macro Symbols

Symbol	Value Kind	Value / Description
__ti__	<constant>	Defined to 1 if TI is the compiler vendor.
__ti_major__	<version>	Identifies major version number.
__ti_minor__	<version>	Identifies minor version number.
__ti_patchlevel__	<version>	Identifies patch version number.
__ti_version__	<encoding>	Encoding of major, minor, and patch version number values associated with the current release, where: `<encoding> = <major>` ↪ `* 10000` `<minor>` ↪ `* 100` `<patch>` For 1.3.2.LTS, for example, the value of <encoding> would be 10302.
__TI_EABI__	1	Indicates the output format is EABI.

TI Arm-Specific Pre-Defined Macro Symbols

TI Arm C Language Extensions (ACLE) Predefined Macro Symbols

The tiarmclang compiler supports ACLE pre-defined symbols that are relevant for Arm Cortex-M and Cortex-R series processors. The following table summarizes the ACLE pre-defined macros that are supported in the tiarmclang compiler.

ACLE Predefined Macro Symbols

Symbol	Value Kind	Value / Description
__ARM_ARCH	<version>	Identifies the Arm architecture version being compiled for. One of the following values: 6 - for Cortex-M0/M0+ 7 - for Cortex-M3/M4/R4/R5 8 - for Cortex-M33
__ARM_ACLE	<version>	Identifies the version of ACLE specification that the compiler adheres to, where <version> = - <major>*100 + <minor> The tiarmclang compiler adheres to version 2.00 of the ACLE specification (<version> = 200).
__ARM_BIG_ENDIAN	<constant>	Defined to 1 if the -mbig-endian option is specified on the tiarmclang command-line. By default, __ARM_BIG_ENDIAN is not defined.
__ARM_ARCH_ISA_ARM	<constant>	Defined to 1 to indicate Arm processor being compiled for supports ARM mode instructions (Cortex- R4/R5).
__ARM_ARCH_ISA_THUMB	<constant>	Identifies the set of THUMB mode instructions supported by the Arm processor being compiled for. Value is: 1: THUMB1(Cortex-M0/M0+) 2: THUMB2(Cortex-M3/M4/M33, - Cortex-R4/R5)
__ARM_32BIT_STATE	<constant>	Defined to 1 if the Arm architecture being compiled for is an AArch32 type.
__ARM_ARCH_PROFILE	<character>	Identifies the Arm architecture profile being compiled for: 'M' - for Cortex-M series 'R' - for Cortex-R series 'A' - for Cortex-A series - (not supported by - tiarmclang).
__ARM_FEATURE_UNALIGNED	<constant>	Defined if Arm processor being compiled for supports unaligned memory accesses (i.e. if -munaligned-access is on). Defined to 1 by default for Cortex-M3/M4/M33/R4/ R5, but can be overridden for the processors using the -mno-unaligned-access compiler option.
__ARM_FEATURE_LDREX	<bitmap>	Identifies which access- widths are supported for LDREX/STREX instructions on the Arm processor being compiled for. <bitmap> values/width: bit 0 - byte bit 1 - half-word bit 2 - word bit 3 - double-word <bitmap> values supported: 0x7 - Cortex-M3/M4 0xF - Cortex-M33/R4/R5
__ARM_FEATURE_CLZ	<constant>	Defined to 1 if the CLZ instruction is available on the Arm processor being compiled for (including Cortex-M3/M4/M33/R4/R5).
__ARM_FEATURE_QBIT	<constant>	Defined to 1 if the Q saturation flag exists and relevant Q saturation arithmetic intrinsics are available on the Arm processor being compiled for (Cortex-M3/M4/M33/R4/ R5).
__ARM_FEATURE_DSP	<constant>	Defined to 1 if DSP instructions are supported and relevant intrinsics are available on the Arm processor being compiled for (Cortex-M4/M33/R4/R5).
__ARN_FEATURE_SAT	<constant>	Defined to 1 if the SSAT and USAT instructions are supported and the relevant intrinsics are available on the Arm processor being compiled for (Cortex-M3/ M4/M33/R4/R5).
__ARM_FEATURE_SIMD32	<constant>	Defined to 1 if the 32-bit SIMD instructions are supported and the relevant intrinsics are available on the Arm processor being compiled for (Cortex-M4/M33/R4/R5).

Symbol	Value Kind	Value / Description
__ARM_FEATURE_IDIV	<constant>	Defined to 1 if hardware integer divide instructions are available on the Arm processor being compiled for (Cortex-M3/ M4/M33).
__ARM_FP	<bitmap>	Defined if floating-point hardware is available. The value assigned to

More details about ACLE pre-defined macro symbols can be found in the Arm C Language Extensions Documentation page.

Other Arm-Specific Predefined Macro Symbols

Arm-Specific Architecture/Processor/Instruction Set Macro Symbols

Symbol	Value Kind	Value / Description
__arm__	\<constant\>	Defined to 1 if target of compile is an Arm processor.
__arm	\<constant\>	Defined to 1 if target of compile is an Arm processor.
__thumb__	\<constant\>	Defined to 1 if compiling for an Arm processor in Thumb mode.
__thumb2__	\<constant\>	Defined to 1 if T32 (Thumb2) instructions are available when compiling for an Arm processor.
__ARM_ARCH_6M__	\<constant\>	Defined to 1 if compiling for an Arm processor based on the version 6M Arm architecture (Cortex-M0/M0+).
__ARM_ARCH_7M__	\<constant\>	Defined to 1 if compiling for an Arm processor based on the version 7M Arm architecture (Cortex-M3).
__ARM_ARCH_7EM__	\<constant\>	Defined to 1 if compiling for an Arm processor based on the version 7EM Arm architecture (Cortex-M4).
__ARM_ARCH_8M_MAIN__	\<constant\>	Defined to 1 if compiling for an Arm processor based on the version 8M- MAIN Arm architecture (Cortex-M33).
__ARM_ARCH_7R__	\<constant\>	Defined to 1 if compiling for an Arm processor based on the version 7R Arm architecture (Cortex-R4/R5).
__VFP_FP__	\<constant\>	Defined to 1 if floating-point hardware use is enabled for a tiarmclang compilation.
__ARM_VFPV2__	\<constant\>	Defined to 1 if use of the VFPV2 floating-point hardware is enabled for a tiarmclang compilation.
__ARM_VFPV3__	\<constant\>	Defined to 1 if use of the VFPV3 floating-point hardware is enabled for a tiarmclang compilation.
__ARM_VFPV4__	\<constant\>	Defined to 1 if use of the VFPV4 floating-point hardware is enabled for a tiarmclang compilation.
__ARM_FPV5__	\<constant\>	Defined to 1 if use of the FPV5 floating-point hardware is enabled for a tiarmclang compilation.
__ARMEB__	\<constant\>	Defined to 1 if compiler has been instructed to generate big-endian object code.
__ARMEL__	\<constant\>	Defined to 1 if compiler has been instructed to generate little-endian object code (default).
__THUMBEB__	\<constant\>	Defined to 1 if compiling for a Cortex-M series Arm processor in big-endian mode.
__THUMBEL__	\<constant\>	Defined to 1 if compiling for a Cortex-M series Arm processor in little-endian mode.

Arm-Specific Feature Test Predefined Macro Symbols

Symbol	Value Kind	Value / Description
__APCS_32__	<con- stant>	Defined to 1 unless a different procedure calling convention is in effect for a given compilation.
__ARM_ARCH_EXT_IDIV__	<con- stant>	Defined to 1 if hardware integer divide instructions are available on the Arm processor being compiled for (Cortex-M3/M4/M33/R5).
__ARM_FEATURE_CMSE	<con- stant>	Defined to 1 if Cortex-M Security Extensions (CMSE) are supported on the Arm processor being compiled for (Cortex-M33).
__ARM_FP_FAST	<con- stant>	Defined to 1 if the -ffast-math or -ffp-mode=fast option is enabled (- ffast-math is implied when the -Ofast optimization level is specified on the compiler command- line).

Generic Compiler Pre-Defined Macro Symbols

Version-Related Predefined Macro Symbols

Symbol	Value Kind	Value / Description
__clang__	<con- stant>	Defined to 1 if compiler uses Clang- based front-end.
__clang_major__	<ver- sion>	Identifies major version number of Clang front-end.
__clang_minor__	<ver- sion>	Identifies minor version number of Clang front-end.
__clang_patchlevel__	<ver- sion>	Identifies patch number of Clang front-end.
__clang_version__	<string>	String representation of Clang front-end version identi- fication.
__GNUC_MINOR__	<ver- sion>	2
__GNUC_PATCHLEVEL	<ver- sion>	1
__GNUC__	<ver- sion>	4
__GXX_ABI_VERSION	<ver- sion>	1002
__llvm__	<con- stant>	Defined to 1 if compiler uses LLVM back-end.
__VERSION__	<string>	Full string representation of Clang front-end version identification.

C Language-Related Predefined Macro Symbols

Symbol	Value Kind	Value / Description
__GNUC_GNU_INLINE__	\<constant\>	Defined to 1 if functions declared inline are defined and externally visible in compiler generated object files if such functions are not declared static or extern.
__GNUC_STDC_INLINE__	\<constant\>	Defined to 1 if functions declared inline are defined and externally visible in compiler generated object files only if such functions are declared extern.
__STDC__	\<constant\>	Defined to 1 if the compiler conforms to ISO Standard C.
__STDC_HOSTED__	\<constant\>	Defined to 1 if the target of the compiler is a hosted environment in which the compiler package supplies standard C runtime libraries.
__STDC_UTF16__	\<constant\>	Defined to 1 if char16_t type values are UTF-16 encoded.
__STDC_UTF32__	\<constant\>	Defined to 1 if char32_t type values are UTF-32 encoded.
__STDC_VERSION__	\<version\>	Defined to the C Standard being applied for a given compilation based on the -std=\<language\> option. By default, tiarmclang assumes "-std=gnu17" for C source files (\<version\>=201710L). *
__STRICT_ANSI__	\<constant\>	Defined to 1 if a strictly-conforming C language variant is specified as the argument to the -std option (c89/90/99/9x/11/1x/17/18).

- See Pre-defined Compiler Macros for a list of definitions of __STDC_VERSION__ that correspond to versions of the C language standards.

C++ Language Standard Predefined Macro Symbol (__cplusplus)

Symbol	Value Kind	Value / Description
__cplusplus	\<standard\>	Indicates the C++ \<standard\> that is in effect for a given compilation, where \<standard\> is one of the following values: ```value => C++ ↪Standard ------ --------- ↪--- 199711L C++98 199711L C++03 201103L C++11 201402L C++14 201703L C++17```

- See Pre-defined Compiler Macros for a list of definitions of __cplusplus that correspond to versions of the C++ language standards.

C++ Language Feature Test Predefined Macro Symbols

Symbol / Feature	Available	Value / Adoption
__cpp_aggregate_bases	C++17	201603L
__cpp_aggregate_nsdmi	C++14	201304L
__cpp_alias_templates	C++11	200704L
__cpp_aligned_new	C++17	201606L
__cpp_attributes	C++11	200809L
__cpp_binary_literals	C++14	201304L
__cpp_capture_star_this	C++17	201603L
__cpp_constexpr	C++11 C++14 C++17	200704L 201304L 201603L
__cpp_contexpr_in_decltype	C++11	201711L
__cpp_decltype	C++11	200707L
__cpp_decltype_auto	C++14	201304L
__cpp_deduction_guides	C++17	201703L
__cpp_delegating_constructors	C++11	200604L
__cpp_digit_separators	C++14	201309L
__cpp_enumerator_attributes	C++17	201411L
__cpp_fold_expressions	C++17	201603L
__cpp_generic_lambdas	C++14	201304L
__cpp_guaranteed_copy_elision	C++17	201606L
__cpp_hex_float	C++17	201603L
__cpp_if_constexpr	C++17	201606L
__cpp_impl_destroying_delete	C++98	201806L
__cpp_inheriting_constructors	C++11	201511L
__cpp_init_captures	C++14	201304L
__cpp_initializer_lists	C++11	200806L
__cpp_inline_variables	C++17	201606L
__cpp_lambdas	C++11	200907L
__cpp_namespace_attributes	C++17	201411L
__cpp_nested_namespace_definitions	C++17	201411L
__cpp_noexcept_function_type	C++17	201510L
__cpp_nontype_template_args	C++17	201411L
__cpp_nontype_template_parameter_auto	C++17	201606L
__cpp_nsdmi	C++11	200809L
__cpp_range_based_for	C++11 C++17	200907L 201603L
__cpp_raw_strings	C++11	200710L
__cpp_ref_qualifiers	C++11	200710L
__cpp_return_type_deduction	C++14	201304L
__cpp_rvalue_references	C++11	200610L

continues on next page

Table 3.1 – continued from previous page

Symbol / Feature	Available	Value / Adoption
__cpp_static_assert	C++11 C++17	200410L 201411L
__cpp_structured_bindings	C++17	201606L
__cpp_template_auto	C++17	201606L
__cpp_threadsafe_static_init	C++98	200806L
__cpp_unicode_characters	C++11	200704L
__cpp_unicode_literals	C++11	200710L
__cpp_user_defined_literals	C++11	200809L
__cpp_variable_templates	C++14	201304L
__cpp_variadic_templates	C++11	200704L
__cpp_variadic_using	C++17	201611L

Compiler Generated Object Format (__ELF__)

Symbol	Value Kind	Value / Description
__ELF__	<constant>	Defined to 1 if compiler generates object code that conforms to the ELF object file format (default).

Predefined Macro Symbols Related to Endian-ness

Symbol	Value Kind	Value / Description
__BIG_ENDIAN__	<constant>	Defined to 1 if compiler has been instructed to generate big-endian object code.
__LITTLE_ENDIAN__	<constant>	Defined to 1 if compiler has been instructed to generate little- endian object code (default).
__BYTE_ORDER__	<constant>	Matches the value of either the __ORDER_BIG_ENDIAN__ (4321) or __ORDER_LITTLE_ENDIAN__ (1234) pre-defined macro symbol depending on which endian-ness the compiler assumes for a given compilation.
__ORDER_BIG_ENDIAN__	<constant>	4321
__ORDER_LITTLE_ENDIAN__	<constant>	1234

Predefined Macro Symbols Related to Optimization

Symbol	Value Kind	Value / Description
__FAST_MATH__	<constant>	Defined to 1 if the -ffast-math or -ffp-mode=fast option is enabled (-ffast-math is implied when the -Ofast optimization level is specified on the compiler command- line).
_INLINE	<constant>	Defined to 1 if automatic inlining optimizations are enabled.
__NO_INLINE__	<constant>	Defined to 1 if automatic inlining optimizations are disabled.
__OPTIMIZE__	<constant>	Defined to 1 if optimization is in use (-O[123fastszg]).
__OPTIMIZE_SIZE__	<constant>	Defined to 1 if optimizations intended to reduce compiler- generated code size are in use (-Os or -Oz option).

Predefined Macro Symbols Related to Scalar Types

Symbol	Value Kind	Value / Description
__BIGGEST_ALIGNMENT__	<bytes>	Indicates the largest alignment in <bytes> ever used for any da
__CHAR16_TYPE__	<type>	unsigned short
__CHAR32_TYPE__	<type>	unsigned int
__CHAR_BIT__	<bits>	8
__CHAR_UNSIGNED__	<constant>	Defined to 1 if "plain" char types (not qualified with a "signed"
__INT8_FMTd__	<string>	"hhd"
__INT8_FMTi__	<string>	"hhi"
__INT8_MAX__	<constant>	127
__INT8_TYPE__	<type>	signed char
__UINT8_FMTX__	<string>	"hhX"
__UINT8_FMTo__	<string>	"hho"
__UINT8_FMTu__	<string>	"hhu"
__UINT8_FMTx__	<string>	"hhx"
__UINT8_MAX__	<constant>	255
__UINT8_TYPE__	<type>	unsigned char
__SCHAR_MAX__	<constant>	127
__INT_FAST8_FMTd__	<string>	"hhd"
__INT_FAST8_FMTi__	<string>	"hhi"
__INT_FAST8_MAX__	<constant>	127
__INT_FAST8_TYPE__	<type>	signed char
__UINT_FAST8_FMTX__	<string>	"hhX"
__UINT_FAST8_FMTo__	<string>	"hho"
__UINT_FAST8_FMTu__	<string>	"hhu"
__UINT_FAST8_FMTx__	<string>	"hhx"

continues on next page

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
__UINT_FAST8_MAX__	<constant>	255
__UINT_FAST8_TYPE__	<type>	unsigned char
__INT_LEAST8_FMTd__	<string>	"hhd"
__INT_LEAST8_FMTi__	<string>	"hhi"
__INT_LEAST8_MAX__	<constant>	127
__INT_LEAST8_TYPE__	<type>	signed char
__UINT_LEAST8_FMTX__	<string>	"hhX"
__UINT_LEAST8_FMTo__	<string>	"hho"
__UINT_LEAST8_FMTu__	<string>	"hhu"
__UINT_LEAST8_FMTx__	<string>	"hhx"
__UINT_LEAST8_MAX__	<constant>	255
__UINT_LEAST8_TYPE__	<type>	unsigned char
__INT16_FMTd__	<string>	"hd"
__INT16_FMTi__	<string>	"hi"
__INT16_MAX__	<constant>	32767
__INT16_TYPE__	<type>	short
__UINT16_FMTX__	<string>	"hX"
__UINT16_FMTo__	<string>	"ho"
__UINT16_FMTu__	<string>	"hu"
__UINT16_FMTx__	<string>	"hx"
__UINT16_MAX__	<constant>	65535
__UINT16_TYPE__	<type>	unsigned short
__SHRT_MAX__	<constant>	32767
__SIZEOF_SHORT__	<bytes>	4
__INT_FAST16_FMTd__	<string>	"hd"
__INT_FAST16_FMTi__	<string>	"hi"
__INT_FAST16_MAX__	<constant>	32767
__INT_FAST16_TYPE__	<type>	short
__UINT_FAST16_FMTX__	<string>	"hX"
__UINT_FAST16_FMTo__	<string>	"ho"
__UINT_FAST16_FMTu__	<string>	"hu"
__UINT_FAST16_FMTx__	<string>	"hx"
__UINT_FAST16_MAX__	<constant>	65535
__UINT_FAST16_TYPE__	<type>	unsigned short
__INT_LEAST16_FMTd__	<string>	"hd"
__INT_LEAST16_FMTi__	<string>	"hi"
__INT_LEAST16_MAX__	<constant>	32767
__INT_LEAST16_TYPE__	<type>	short
__UINT_LEAST16_FMTX__	<string>	"hX"
__UINT_LEAST16_FMTo__	<string>	"ho"

continues on next page

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
__UINT_LEAST16_FMTu__	<string>	"hu"
__UINT_LEAST16_FMTx__	<string>	"hx"
__UINT_LEAST16_MAX__	<constant>	65535
__UINT_LEAST16_TYPE__	<type>	unsigned short
__INT32_FMTd__	<string>	"d"
__INT32_FMTi__	<string>	"i"
__INT32_MAX__	<constant>	2147483647
__INT32_TYPE__	<type>	int
__UINT32_C_SUFFIX__	<text>	U
__UINT32_FMTX__	<string>	"X"
__UINT32_FMTo__	<string>	"o"
__UINT32_FMTu__	<string>	"u"
__UINT32_FMTx__	<string>	"x"
__UINT32_MAX__	<constant>	4294967295U
__UINT32_TYPE__	<type>	unsigned int
__INT_MAX__	<constant>	2147483647
__SIZEOF_INT__	<bytes>	4
__LONG_MAX__	<constant>	2147483647
__SIZEOF_LONG__	<bytes>	4
__INT_FAST32_FMTd__	<string>	"d"
__INT_FAST32_FMTi__	<string>	"i"
__INT_FAST32_MAX__	<constant>	2147483647
__INT_FAST32_TYPE__	<type>	int
__UINT_FAST32_FMTX__	<string>	"X"
__UINT_FAST32_FMTo__	<string>	"o"
__UINT_FAST32_FMTu__	<string>	"u"
__UINT_FAST32_FMTx__	<string>	"x"
__UINT_FAST32_MAX__	<constant>	4294967295U
__UINT32_TYPE__	<type>	unsigned int
__INT_LEAST32_FMTd__	<string>	"d"
__INT_LEAST32_FMTi__	<string>	"i"
__INT_LEAST32_MAX__	<constant>	2147483647
__INT_LEAST32_TYPE__	<type>	int
__UINT_LEAST32_FMTX__	<string>	"X"
__UINT_LEAST32_FMTo__	<string>	"o"
__UINT_LEAST32_FMTu__	<string>	"u"
__UINT_LEAST32_FMTx__	<string>	"x"
__UINT_LEAST32_MAX__	<constant>	4294967295U
__UINT_LEAST32_TYPE__	<type>	unsigned int
__INT64_C_SUFFIX__	<text>	LL

continues on next page

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
__INT64_FMTd__	\<string>	"lld"
__INT64_FMTi__	\<string>	"lli"
__INT64_MAX__	\<constant>	9223372036854775807LL
__INT64_TYPE__	\<type>	long long int
__UINT64_C_SUFFIX__	\<text>	ULL
__UINT64_FMTX__	\<string>	"llX"
__UINT64_FMTo__	\<string>	"llo"
__UINT64_FMTu__	\<string>	"llu"
__UINT64_FMTx__	\<string>	"llx"
__UINT64_MAX__	\<constant>	18446744073709551615ULL
__UINT64_TYPE__	\<type>	long long unsigned int
__LONG_LONG_MAX__	\<constant>	9223372036854775807LL
__SIZEOF_LONG_LONG__	\<bytes>	8
__INT_FAST64_FMTd__	\<string>	"lld"
__INT_FAST64_FMTi__	\<string>	"lli"
__INT_FAST64_MAX__	\<constant>	9223372036854775807LL
__INT_FAST64_TYPE__	\<type>	long long int
__UINT_FAST64_FMTX__	\<string>	"llX"
__UINT_FAST64_FMTo__	\<string>	"llo"
__UINT_FAST64_FMTu__	\<string>	"llu"
__UINT_FAST64_FMTx__	\<string>	"llx"
__UINT_FAST64_MAX__	\<constant>	18446744073709551615ULL
__UINT_FAST64_TYPE__	\<type>	long long unsigned int
__INT_LEAST64_FMTd__	\<string>	"lld"
__INT_LEAST64_FMTi__	\<string>	"lli"
__INT_LEAST64_MAX__	\<constant>	9223372036854775807LL
__INT_LEAST64_TYPE__	\<type>	long long int
__UINT_LEAST64_FMTX__	\<string>	"llX"
__UINT_LEAST64_FMTo__	\<string>	"llo"
__UINT_LEAST64_FMTu__	\<string>	"llu"
__UINT_LEAST64_FMTx__	\<string>	"llx"
__UINT_LEAST64_MAX__	\<constant>	18446744073709551615ULL
__UINT_LEAST64_TYPE__	\<type>	long long unsigned int
__INTMAX_C_SUFFIX__	\<text>	LL
__INTMAX_FMTd__	\<string>	"lld"
__INTMAX_FMTi__	\<string>	"lli"
__INTMAX_MAX__	\<constant>	9223372036854775807LL
__INTMAX_TYPE__	\<type>	long long int
__INTMAX_WIDTH__	\<bits>	64
__UINTMAX_C_SUFFIX__	\<text>	ULL

continues on next page

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
__UINTMAX_FMTX__	<string>	"llX"
__UINTMAX_FMTo__	<string>	"llo"
__UINTMAX_FMTu__	<string>	"llu"
__UINTMAX_FMTx__	<string>	"llx"
__UINTMAX_MAX__	<constant>	18446744073709551615ULL
__UINTMAX_TYPE__	<type>	long long unsigned int
__INTPTR_FMTd__	<string>	"d"
__INTPTR_FMTi__	<string>	"i"
__INTPTR_MAX__	<constant>	2147483647
__INTPTR_TYPE__	<type>	int
__INTPTR_WIDTH__	<bits>	32
__UINTPTR_FMTX__	<string>	"X"
__UINTPTR_FMTo__	<string>	"o"
__UINTPTR_FMTu__	<string>	"u"
__UINTPTR_FMTx__	<string>	"x"
__UINTPTR_MAX__	<constant>	4294967295U
__UINTPTR_TYPE__	<type>	unsigned int
__UINTPTR_WIDTH__	<constant>	32
__POINTER_WIDTH__	<constant>	32
_ILP32	<constant>	Defined to 1 if pointer type width and long type width is 32-bit
__ILP32__	<constant>	Defined to 1 if pointer type width and long type width is 32-bit
__SIZEOF_POINTER__	<bytes>	4
__PTRDIFF_FMTd__	<string>	"d"
__PTRDIFF_FMTi__	<string>	"i"
__PTRDIFF_MAX__	<constant>	2147483647
__PTRDIFF_TYPE__	<type>	int
__PTRDIFF_WIDTH__	<bits>	32
__SIZEOF_PTRDIFF_T__	<bytes>	4
__SIZE_FMTX__	<string>	"X"
__SIZE_FMTo__	<string>	"o"
__SIZE_FMTu__	<string>	"u"
__SIZE_FMTx__	<string>	"X"
__SIZE_MAX__	<constant>	4294967295U
__SIZE_TYPE__	<type>	unsigned int
__SIZE_WIDTH__	<bits>	32
__SIZEOF_SIZE_T__	<bytes>	4
__WCHAR_MAX__	<constant>	2147483647
__WCHAR_TYPE__	<constant>	int
__WCHAR_WIDTH__	<bits>	32
__SIZEOF_WCHAR_T__	<bytes>	4

continues on next page

3.2. C/C++ Language Implementation   **256**

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
__WINT_MAX__	\<constant\>	2147483647
__WINT_TYPE__	\<constant\>	int
__WINT_WIDTH__	\<bits\>	32
__SIZEOF_WINT_T__	\<bytes\>	4
__FLT_DECIMAL_DIG__	\<digits\>	9
__FLT_DENORM_MIN__	\<constant\>	1.40129846e-45F
__FLT_DIG__	\<digits\>	6
__FLT_EPSILON__	\<constant\>	1.19209290e-7F
__FLT_MANT_DIG__	\<bits\>	24
__FLT_MAX_10_EXP__	\<constant\>	38
__FLT_MAX_EXP__	\<constant\>	128
__FLT_MAX__	\<constant\>	3.40282347e+38F
__FLT_MIN_10_EXP__	\<constant\>	-37
__FLT_MIN_EXP__	\<constant\>	-125
__FLT_MIN__	\<constant\>	1.17549435e-38F
__FLT_RADIX__	\<constant\>	2
__SIZEOF_FLOAT__	\<bytes\>	4
__DBL_DECIMAL_DIG__	\<digits\>	17
__DBL_DENORM_MIN__	\<constant\>	4.9406564584124654e-324
__DBL_DIG__	\<digits\>	15
__DBL_EPSILON__	\<constant\>	2.2204460492503131e-16
__DBL_MANT_DIG__	\<bits\>	53
__DBL_MAX_10_EXP__	\<constant\>	308
__DBL_MAX_EXP__	\<constant\>	1024
__DBL_MAX__	\<constant\>	1.7976931348623157e+308
__DBL_MIN_10_EXP__	\<constant\>	-307
__DBL_MIN_EXP__	\<constant\>	-1021
__DBL_MIN__	\<constant\>	2.2250738585072014e-308
__DECIMAL_DIG__	\<constant\>	__LDBL_DECIMAL_DIG__
__LDBL_DECIMAL_DIG__	\<digits\>	17
__LDBL_DENORM_MIN__	\<constant\>	4.9406564584124654e-324
__LDBL_DIG__	\<digits\>	15
__LDBL_EPSILON__	\<constant\>	2.2204460492503131e-16
__LDBL_MANT_DIG__	\<bits\>	53
__LDBL_MAX_10_EXP__	\<constant\>	308
__LDBL_MAX_EXP__	\<constant\>	1024
__LDBL_MAX__	\<constant\>	1.7976931348623157e+308
__LDBL_MIN_10_EXP__	\<constant\>	-307
__LDBL_MIN_EXP__	\<constant\>	-1021
__LDBL_MIN__	\<constant\>	2.2250738585072014e-308

3.2.7 Attributes

Contents:

Attribute Syntax

There are three different kinds of attributes supported by the tiarmclang compiler:

- *Function Attributes*
- *Variable Attributes*
- *Type Attributes*

General Syntax

In general, an attribute can be applied to a function, variable, or type in the following ways:

> *attribute-specifier* <function, variable, or type>

> <function, variable, or type> *attribute=specifier*

where an *attribute-specifier* consists of the following parts:

> __attribute__((*attribute-list*))

An *attribute-list* consists of zero or more comma-separated *attributes* where each *attribute* can be:

- an attribute name that takes no arguments (like *noinit* or *persistent*),
- an attribute name that expects a list of arguments enclosed in parentheses (like *aligned* or *section*). Further details about argument requirements are provided in the descriptions of those attributes that take arguments, or
- empty, in which case the *attribute-specifier* is ignored.

Examples

- An *attribute-specifier* can precede a variable definition:

```
__attribute__((section("my_sect"))) int my_var;
```

- An *attribute-specifier* can be specified at the end of an uninitialized variable declaration:

```
int my_var __attribute__((section("my_sect")));
```

- An *attribute-specifier* can be applied to an initialized variable:

```
int my_var __attribute__((section("my_sect"))) = 5;
```

The *attribute-specifier* in this case must precede the initializer.

- An *attribute-specifier* can be applied to a structure member:

```
struct {
  char m1;
  int  m2 __attribute__((packed));
  int  m3;
} packed_struct = { 10, 20, 30 };
```

In this case, struct member *m2* is aligned on a 1-byte boundary relative to the beginning of the struct due to the *packed* attribute, but *m3* is aligned on a 4-byte boundary relative to the beginning of the struct.

- An *attribute-specifier* applied to a struct type can apply to all members of the struct:

```
struct __attribute__((packed)) {
  char m1;
  int  m2;
  int  m3;
} packed_struct = { 10, 20, 30 };
```

In this case, all members of the struct are aligned on a 1-byte boundary relative to the beginning of the struct due to the *packed* attribute.

- Multiple *attributes* can be applied in a single *attribute-specifier*:

```
__attribute__((noinit,location(0x100))) int noinit_location_
↪global;
```

- An *attribute-specifier* that precedes a list of function declarations applied to all of the declarations in the same statement:

```
_attribute__((noreturn)) void d0 (void),
   __attribute__((format(printf, 1, 2))) d1 (const char *, ..
↪.),
   d2 (void);
```

In this case, the *noreturn* attribute applies to all the declared functions, but the *format* attribute only applies to *d1*.

Function Attributes

The following function attributes are supported by the compiler:

- *alias*
- *aligned*
- *always_inline*
- *const*
- *constructor*
- *deprecated*
- *format*
- *format_arg*
- *fully_populate_jump_tables*
- *interrupt*
- *location*
- *malloc*
- *naked*
- *noinline*
- *nomerge*
- *nonnull*
- *noreturn*
- *optnone*
- *pure*
- *section*
- *used/retain*
- *visibility*
- *weak*
- *weakref*

The following additional function attributes are specific to Arm.

- *cmse_nonsecure_call*
- *cmse_nonsecure_entry*

- *interrupt_save_fp*

- *local*

- *nothrow*

- *offchip*

- *onchip*

Note: Function Attribute Syntax

A function attribute specification can appear at the beginning or end of a function declaration statement:

```
<function declaration> __attribute__((<attribute-list>));
```

or

```
__attribute__((<attribute-list>)) <function declaration>;
```

However, when a function attribute is specified with the function definition, if it appears between the function specification and the opening curly brace that indicates the start of the function body, the compiler will emit a warning diagnostic. For example,

```
// always_inline_function_attr.c
...
void emit_msg(void) __attribute__((always_inline)) {
  printf("this is call #%d\n", ++counter);
}
...
```

```
%> tiarmclang -mcpu=cortex-m0 -c always_inline_function_attr.c
always_inline_function_attr.c:8:36: warning: GCC does not allow
'always_inline' attribute in this position on a function
definition [-Wgcc-compat]
void emit_msg(void) __attribute__((always_inline)) {
                                   ^
1 warning generated.
```

The warning can be disabled using the -Wgcc-compat option or by moving the function attribute specification before the function specification.

---

alias

The *alias* function attribute can be applied to a function declaration to instruct the compiler to interpret the function symbol being declared as an alias for another function symbol that is defined in the same compilation unit.

Syntax

> <return type> *source symbol* (<arguments>) __attribute__((alias(*target symbol*)));

- *source symbol* - is the subject of the function declaration that will become an alias of the *target symbol*.

- *target symbol* - is a function symbol defined in the same compilation unit as the declaration of *source symbol*, to which references to *source symbol* will resolve to.

Example

In the following C code, both a *weak* and an *alias* attribute are applied to the declaration of *event_handler* so that calls to *event_handler* will be resolved by *default_handler* unless a strong definition of *event_handler* overrides the weak definition at link-time:

```c
// alias_func_attr.c
#include <stdio.h>

void default_handler() {
  printf("This is the default handler\n");
}

void event_handler(void) __attribute__((weak, alias("default_
 ↪handler")));

int main() {
  event_handler();
  return 0;
}
```

If the above code is compiled and linked and run, the output reveals that the reference to *event_handler* resolves to a call to *default_handler*:

```
%> tiarmclang -mcpu=cortex-m0 alias_func_attr.c -o a.out -Wl,-
 ↪llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
This is the default handler
```

If we add a strong definition of *event_handler* to the build, then the reference to *event_handler* will be resolved by the strong function definition of *event_handler*:

```c
#include <stdio.h>

void event_handler() {
  printf("This is the event handler implementation\n");
}
```

```
%> tiarmclang -mcpu=cortex-m0 alias_func_attr.c event_handler_
↪impl.c -o a.out -Wl,-llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
This is the event handler implementation
```

aligned

The *aligned* function attribute can be applied to a function in order to set a minimum byte-alignment constraint on the target memory address where the function symbol is defined.

Syntax

> <return type> *symbol* (<arguments>) __attribute__((aligned(*alignment*)));

> - *alignment* - is the minimum byte-alignment for the definition of the *symbol*. The *alignment* value must be a power of 2. The default alignment for a *symbol* is 4-bytes.

Example

The following example shows a program that calls a function with an *aligned* attribute applied to it:

```c
#include <stdio.h>

__attribute__((aligned(64))) void aligned_func() {
  printf("This functions address is: 0x%08lx\n",
         (unsigned long)&aligned_func);
}

int main() {
  aligned_func();
  return 0;
}
```

When compiled and linked and run, the output of the program shows that the effective address of *aligned_func* is on a 64-byte boundary (the 1 in the LS bit of the printed address indicates the code state for the function; in this case, 1 indicates that *aligned_func* is a THUMB function):

```
%> tiarmclang -mcpu=cortex-m0 aligned_func_attr.c -o a.out -Wl,-
 ↪llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
This function's address is: 0x00001981
```

always_inline

The *always_inline* function attribute can be applied to a function to instruct the compiler that the function is to be inlined at any call sites, even if no optimization is specified on the **tiarmclang** command line.

Syntax

        <return type> *symbol* (<arguments>) __attribute__((always_inline));

Example

The following use of the *always_inline* function attribute will cause the body of *emit_msg* to be inlined where *main* calls *emit_msg*:

```c
#include <stdio.h>

int counter = 0;

__attribute__((always_inline)) void emit_msg() {
  printf("this is call #%d\n", ++counter);
}

int main() {
  emit_msg();
  emit_msg();
  emit_msg();
}
```

The compiler-generated assembly source for the above C code shows that the body of *emit_msg* has been inlined even though no optimization was specified on the command line:

```
%> tiarmclang -mcpu=cortex-m0 -S always_inline_function_attr.c
%> cat always_inline_function_attr.s
...
        .section        .text.main,"ax",%progbits
        .hidden main                                 @ -- Begin␣
↪function main
        .globl  main
        .p2align        2
        .type   main,%function
        .code   16                                   @ @main
        .thumb_func
main:
        .fnstart
@ %bb.0:                                             @ %entry
        .save   {r7, lr}
        push    {r7, lr}
        .pad    #8
        sub     sp, #8
        ldr     r0, .LCPI1_0
        str     r0, [sp]                             @ 4-byte Spill
        ldr     r1, [r0]
        adds    r1, r1, #1
        str     r1, [r0]
        ldr     r0, .LCPI1_1
        str     r0, [sp, #4]                         @ 4-byte Spill
        bl      printf
...
```

The compiler-generated assembly code also contains the definition of the *emit_msg* function, but when the program is compiled and linked, the section containing the definition of *emit_msg* will not be included in the link since all of the references to it have been inlined.

const

The *const* function attribute applied to a function declaration informs the compiler that the function has no side effects except for the value it returns. The function will not examine any values outside its body with the exception of arguments that are passed into it. It does not access any global data.

Note that if the function has a pointer argument and it accesses memory via that pointer, then the *const* attribute should not be applied to its declaration. Also, the *const* attribute should not be applied to a function which calls a non-const function.

Syntax

<return type> *symbol* (<arguments>) __attribute__((const));

3.2. C/C++ Language Implementation   **265**

Example

The following C code is a simple example of the use of the *const* attribute:

```c
#include <stdio.h>

float square_flt(float a_flt) __attribute__((const));
float square_flt(float a_flt) {
  return (a_flt * a_flt);
}

float tot_flt = 0.0;

int main(void) {
  int i;
  for (i = 0; i < 10; i++)
  {
    tot_flt += square_flt(i);
    printf ("iter #%d, tot_flt is: %f\n", i, tot_flt);
  }
}
```

constructor

Applying a *constructor* attribute to a function causes the function to be called prior to executing the code in *main*. Use of this attribute provides a means of initializing data that is used implicitly during the execution of a program.

Syntax

> void *constructor function name* () __attribute__((constructor[(*priority*)]));
>
> - *priority* - is an optional integer argument used to indicate the order in which this constructor function is to be called relative to other functions that have been annotated with the *constructor* attribute. If no *priority* argument is specified, then constructor-annotated functions that do not have *priority* arguments will be called in the order in which they are encountered in the compilation unit. If a *priority* argument is specified, then a constructor-annotated function with a lower *priority* value will be called before a constructor-annotated function with a higher *priority* value or no *priority* argument.

Example

Consider the following C program containing 4 functions that have been annotated with a *constructor* attribute:

```
#include <stdio.h>

__attribute__((constructor)) void init2() {
  printf("run init2\n");
}

__attribute__((constructor)) void init1() {
  printf("run init1\n");
}

__attribute__((constructor(5))) void init3() {
  printf("run init3\n");
}

__attribute__((constructor(50))) void init4() {
  printf("run init4\n");
}

int main() {
  printf("run main\n");
  return 0;
}
```

When compiled and linked and run, the output of the program shows the order in which the constructor-annotated functions are called:

```
%> tiarmclang -mcpu=cortex-m0 constructor_function_attr.c -o a.
→out -Wl,-llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
run init3
run init4
run init2
run init1
run main
```

The *iniit3* constructor is run first since its *priority* value is lower than *init4*. The *init2* and *init1* constructors are run after *init4* because they don't have a *priority* argument. Finally, *init2* is run before *init1* since it is encountered before *init1* in the compilation unit.

deprecated

The *deprecated* function attribute can be applied to a function to mark it as deprecated so that the compiler will emit a warning when it sees a reference to the function in its compilation unit. This can be useful during program development when trying to remove references to a function whose definition will eventually be removed.

Syntax

> <return type> *symbol* (<arguments>) __attribute__((deprecated));

Example

In this example, the function *dep_func* has been marked with a *deprecated* attribute, but *main* contains a call to the function:

```c
#include <stdio.h>

__attribute__((deprecated)) void dep_func(void) {
  printf("this function has been deprecated\n");
}

int main() {
  dep_func();
  printf("run main\n");
  return 0;
}
```

When compiled, tiarmclang will emit a warning diagnostic to indicate a reference to the *deprecated* function *dep_func* has been encountered:

```
%> tiarmclang -mcpu=cortex-m4 deprecated_function_attr.c -o a.
↪out -Wl,-llnk.cmd,-ma.map
deprecated_function_attr.c:9:3: warning: 'dep_func' is
↪deprecated [-Wdeprecated-declarations]
  dep_func();
  ^
deprecated_function_attr.c:4:16: note: 'dep_func' has been
↪explicitly marked deprecated here
__attribute__((deprecated)) void dep_func(void) {
              ^
1 warning generated.
```

format

The *format* function attribute can be applied to a function to indicate that the function accepts a *printf* or *scanf*-like format string and corresponding arguments or a *va_list* that contains these arguments.

The tiarmclang compiler performs two kinds of checks with this attribute.

1. The compiler will check that the function is called with a format string that uses format specifiers that are allowed, and that arguments match the format string. If the compiler encounters an issue with this check, a warning diagnostic will be emitted at compile-time.

2. If the *format-nonliteral* warning category is enabled (off by default), then the compiler will emit a warning if the format string argument is not a literal string.

Syntax

<return type> *symbol* (<arguments>) __attribute__((format(*archtype*, *string-index*, *first-to-check*)));

- *archtype* - identifies the runtime library function that informs the compiler how to interpret the format string argument. The compiler will accept *printf*, *scanf*, and *strftime* as valid *archtype* argument values.

- *string-index* - is an integer value identifying which argument in the argument list is the format string argument. Index numbering starts with the integer 1.

- *first-to-check* - is an integer value identifying the first argument to check against the format string. Index numbering starts with the integer 1. For format functions where the arguments are not available to be checked, the *first-to-check* argument for the *format* attribute should be zero.

Example

Consider the following C code that uses a wrapper function for printf called *my_printf* that has been declared with a *format* attribute:

```c
#include <stdio.h>

__attribute__((format(printf, 2, 3)))
int my_printf(int n, const char* fmt, ...);

int main() {
  my_printf(10, "call with int: %d\n", 20);
  my_printf(30, "wrong number of args: %d\n", 40, 50);

  return 0;
}
```

When the above code is compiled, tiarmclang reports a warning diagnostic about the second call to *my_printf* since the call provides more arguments than can be handled by the specified format string.

```
%> tiarmclang -mcpu=cortex-m4 -c format_function_attr.c
format_function_attr.c:9:51: warning: data argument not used by␣
↪format string [-Wformat-extra-args]
  my_printf(30, "wrong number of args: %d\n", 40, 50);
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~     ^
1 warning generated.
```

format_arg

The *format_arg* function attribute can be applied to a function that takes a format string as an argument and returns a potentially updated version of the format string that is to be used as a format string argument for a printf-style function (like *printf*, *scanf*, *strftime*, etc.). The *format_arg* attribute enables the compiler to perform a type check between the format specifiers in the format string and the other arguments that are passed to the printf-style function.

Syntax

> <return type> *symbol* (<arguments>) __attribute__((format_arg(*string-index*)));

Example

In the following C code, the *my_format* function is declared with a *format_arg* attribute, identifying the index of the format string argument in the *my_format* function interface. The *main* function then contains a series of calls to *printf* where the format string to the *printf* call is the return value of the *my_format* function. The compiler will check the format string passed to *my_format* against the other arguments that are passed to *printf* in each case:

```c
#include <stdio.h>

char *my_format(int n, const char *fmt) __attribute__((format_
↪arg(2)));

extern int i1, i2;
extern float f1, f2;

int main() {
  printf(my_format(10, "one int: %d\n"), i1);
  printf(my_format(20, "too many ints: %d\n"), i1, i2);
  printf(my_format(20, "wrong type: %d %f\n"), f1, f2);
  return 0;
}
```

When the above code is compiled, tiarmclang reports warning diagnostics about the incorrect number of arguments in the second call to *printf* and the argument type mismatch in the third call to *printf*.

```
%> tiarmclang -mcpu=cortex-m0 -c format_arg_function_attr.c
%format_arg_function_attr.c:11:52: warning: data argument not
 ↪used by format string [-Wformat-extra-args]
  %printf(my_format(20, "too many ints: %d\n"), i1, i2);
                        %~~~~~~~~~~~~~~~~~~~~~          ^
%format_arg_function_attr.c:12:48: warning: format specifies
 ↪type 'int' but the argument has type 'float' [-Wformat]
  %printf(my_format(20, "wrong type: %d %f\n"), f1, f2);
                        %~~                   ^~
                        %%f
%2 warnings generated.
```

fully_populate_jump_tables

The *fully_populate_jump_tables* function attribute will allow a function's switch statements to use fully populated jump tables (if possible) with no minimum density up to a maximum range limit of 100 entries. This capability can eliminate non-deterministic control flow and is useful in embedded systems that require ISRs to execute with deterministic timing.

Note: **This attribute may negatively impact code size depending on the size of the jump table**

---

interrupt

The *interrupt* function attribute can be applied to a function definition to identify it as an interrupt function so that the compiler can generate additional code on function entry and exit to preserve the system state. The function can then be used to handle errors that led to function exits.

The tiarmclang compiler re-aligns the stack pointer to an 8-byte boundary on entry into a function marked with the *interrupt* attribute because the Arm Procedure Call Standard (AAPCS) requires this alignment for all public interfaces.

For Arm Cortex-M processor applications, the architecture includes mechanisms to preserve all general-purpose integer registers. When compiling for an Arm Cortex-R processor, the compiler will generate code to preserve general-purpose registers and a sequence of instructions to effect an appropriate exception return for the processor.

Note: **FPU Registers are Not Automatically Preserved by Interrupt Functions**

The compiler will not generate code to preserve FPU registers in a function marked with the *interrupt* attribute. If floating-point operations are required in the scope of an interrupt, then the *interrupt_save_fp* attribute should be used instead. Otherwise, FPU registers must be preserved manually.

Syntax

> __attribute__((interrupt)) <return type> *symbol* (<arguments>) { ... }
>
> __attribute__((interrupt[("*interrupt-type*")])) void *symbol* () { ... }
>
> - *interrupt-type* - is the optional string literal argument that indicates the type of interrupt being defined. The tiarmclang compiler supports the following *interrupt-type* identifiers:
>
>   - **IRQ**
>
>   - **FIQ**
>
>   - **SWI**
>
>   - **ABORT**
>
>   - **UNDEF**
>
> Interrupt types other than **FIQ** save all core registers except the *lr* and *sp* registers. **FIQ** type interrupts save registers r0-r7.
>
> Note that if an *interrupt-type* argument is specified with the *interrupt* attribute specification, then the function interface may not take any arguments and may not return a value.

Example

Here are some examples of the *interrupt* attribute applied to functions:

```
extern int  do_something1(int n) __attribute__((interrupt));
extern void do_something2(void) __attribute__((interrupt("IRQ
↪")));
extern void do_something3(void) __attribute__((interrupt("FIQ
↪")));
extern void do_something4(void) __attribute__((interrupt("SWI
↪")));
extern void do_something5(void) __attribute__((interrupt("ABORT
↪")));
extern void do_something6(void) __attribute__((interrupt("UNDEF
↪")));


__attribute__((interrupt)) void int1(int n) {
```

```
do_something1(n);
}
__attribute__((interrupt("IRQ"))) void int2() {
  do_something2();
}
__attribute__((interrupt("FIQ"))) void int3() {
  do_something3();
}
__attribute__((interrupt("SWI"))) void int4() {
  do_something4();
}
__attribute__((interrupt("ABORT"))) void int5() {
  do_something5();
}
__attribute__((interrupt("UNDEF"))) void int6() {
  do_something6();
}
```

When compiled with the *-mcpu=cortex-r5* option, the compiler-generated assembly snippet shows general-purpose registers being preserved during the function's entry and exit code, the re-alignment of the stack pointer via a forced adjustment to the stack and a "bfc sp, #0, #3" instruction. In the example shown, the stack is adjusted prior to the "bfc sp, #0, #3" instruction by pushing an extra two registers (r10 and r11) onto the stack. In other cases, the compiler may use a "sub sp, sp, #8" instruction to force an adjustment to the stack prior to the "bfc sp, #0, #3" instruction. The return from the interrupt function is effected with the addition of the "subs pc, lr, #4" instruction.

```
%> tiarmclang -mcpu=cortex-r5 -S interrupt_function_attr.c
%> cat interrupt_function_attr.s
...
int2:
        push    {r0, r1, r2, r3, r10, r11, r12, lr}
        add     r11, sp, #20
        bfc     sp, #0, #3
        bl      do_something2
        sub     sp, r11, #20
        pop     {r0, r1, r2, r3, r10, r11, r12, lr}
        subs    pc, lr, #4
...
```

location

The *location* function attribute can be used to specify a function's run-time address from within the C/C++ source. The tiarmclang compiler will embed linker instructions within a given compiler-generated object file that will dictate where in target memory the function definition will be placed at link-time.

Syntax

> <return type> *symbol* (<arguments>) __attribute__((location(*address*)));

> - *address* - is the run-time target memory address where the definition of the function *symbol* is to be placed at link-time.

When specifying the address for a THUMB mode function, a 1 must be set in the least significant bit of the specified *address* value since it is assumed that the specified value includes the ARM/THUMB mode bit.

Example

In the following example, the *location* attribute is applied to *main* using an *address* argument of 0x2001 to place the definition of *main* at target address 0x2000.

```c
#include <stdio.h>

__attribute__((location(0x2001))) int main() {
  printf("Good morning, Dave.\n");
  return 0;
}
```

```
%> tiarmclang -mcpu=cortex-m0 location_function_atr.c -o a.out -
↪Wl,-llnk.cmd,-ma.map
%> cat a.map
SECTION ALLOCATION MAP

 output                                           attributes/
section    page     origin      length       input sections
--------   ----   ----------  ----------    ----------------
...
.text.main
*          0     00002000     0000002c
                 00002000     0000001c     location_function_
↪attr-9108ac.o    (.text.main)
...
```

malloc

The *malloc* function attribute can be applied to a function to inform the compiler that the function performs dynamic memory allocation. This information will then be incorporated into associated optimizations that the compiler may perform. For example, the compiler can assume that the pointer returned by the function that is annotated with a *malloc* attribute cannot alias any other pointer that is valid at the time the function returns. The compiler can also assume that no other pointer to a valid object has access to any storage that was allocated by the malloc-like function.

Syntax

        <object type> * *symbol* (<arguments>) __attribute__((malloc));

Example

The following C code is an example of a function that is designated as malloc-like with the application of the *malloc* attribute:

```c
char *alloc_buffer(int sz) __attribute__((malloc));
```

naked

When a *naked* function attribute is applied to a function, it informs the compiler that the function is written entirely in GNU-syntax Arm assembly language via the use of *asm()* statements. The compiler will not generate function prologue or epilogue code for such functions.

Note that only simple *asm()* statements can be used to compose the assembly language content of a *naked* function, and the content must adhere to the applicable C/C++ calling conventions in terms of how arguments are passed into the function and how the return value is placed in the proper return register (for a function with a non-void return type).

Syntax

        <return type> *symbol* (<arguments>) __attribute__((naked));

Example

Here is a simple example of a *naked* function:

```c
__attribute__((naked)) int sub(int arg1, int arg2) {
  __asm(" SUB r0, r0, r1");
}
```

noinline

The *noinline* function attribute will suppress the inlining of the function that it applies to at any function call sites in the same compilation unit.

Syntax

    <return type> *symbol* (<arguments>) __attribute__((noinline));

Example

Consider the following program in which *incr_counter* is marked with a *noinline* attribute so that it cannot be inlined at the site of *main*'s call to the function, even with optimization enabled:

```c
#include <stdio.h>

int my_counter = 0;

__attribute__((noinline)) void incr_counter(void) { ++my_counter;
↪ }

int main() {
  int i;
  for (i = 0; i < 10; i++) {
    incr_counter();
  }

  printf("my counter is: %d\n", my_counter);
}
```

When compiled with *-O2* optimization, the compiler generated assembly for *main* shows that calls to *incr_counter* are not inlined, but the loop has been unrolled to eliminate the loop control flow overhead:

```
%> tiarmclang -mcpu=cortex-m0 -S -O2 noinline_function_attr.c
%> cat noinline_function_attr.s
...
        .section        .text.main,"ax",%progbits
        .hidden main                                    @ -- Begin␣
↪function main
        .globl  main
        .p2align        2
        .type   main,%function
        .code   16                                      @ @main
        .thumb_func
main:
```

```
        push    {r7, lr}
        bl      incr_counter
        bl      incr_counter
        bl      incr_counter
        bl      incr_counter
        bl      incr_counter
        bl      incr_counter
        bl      incr_counter
        bl      incr_counter
        bl      incr_counter
        bl      incr_counter
        ldr     r0, .LCPI1_0
        ldr     r1, [r0]
        ldr     r0, .LCPI1_1
        bl      printf
        movs    r0, #0
...
```

nomerge

When a *nomerge* attribute is applied to a function, the compiler will be prevented from merging calls to the function. The *nomerge* attribute will not affect indirect calls to the function.

Syntax

> <return type> *symbol* (<arguments>) __attribute__((nomerge));

Example

Consider the following C code containing an if block and an else block both of which call *callee_func* with slightly different arguments. If the *nomerge* attribute is not applied to the *callee_func* declaration, the compiler will tail-merge the calls into a single call when using optimization. However, the use of the *nomerge* attribute prevents the calls to *callee_func* from being merged:

```c
void callee_func(const char *str) __attribute__((nomerge));

int caller_func(int n) {

  if (n < 10) {
    callee_func("string for return 1");
    return 1;
  }
```

```
  else {
    callee_func("string for return 2");
    return 2;
  }

  return 0;
}
```

The compiler-generated assembly code for the definition of *caller_func* shows that the calls to *callee_func* are not merged:

```
%> tiarmclang -mcpu=cortex-m4 -Oz -S nomerge_function_attr.c
%> cat nomerge_function_attr.s
...
        .section        .text.caller_func,"ax",%progbits
        .hidden caller_func                      @ -- Begin␣
↪function    caller_func
        .globl  caller_func
        .p2align        2
        .type   caller_func,%function
        .code   16                               @ @caller_func
        .thumb_func
caller_func:
@ %bb.0:                                         @ %entry
        push    {r7, lr}
        cmp     r0, #9
        bgt     .LBB0_2
@ %bb.1:                                         @ %if.then
        ldr     r0, .LCPI0_1
        bl      callee_func
        movs    r0, #1
        pop     {r7, pc}
.LBB0_2:                                         @ %if.else
        ldr     r0, .LCPI0_0
        bl      callee_func
        movs    r0, #2
        pop     {r7, pc}
...
```

nonnull

The *nonnull* function attribute can be applied to a function to inform the compiler that pointer arguments to the function should not be null pointers. The compiler will emit a warning diagnostic if it detects an incoming null pointer argument.

Syntax

> <return type> *symbol* (<arguments>) __attribute__((nonnull));

Example

In the following C code, when the declaration of *callee_func* is annotated with a *nonnull* attribute, it enables the compiler to emit a warning diagnostic when it detects a NULL pointer being passed as an argument at one of the call sites for *callee_func*:

```c
void callee_func(int n, int *p) __attribute__((nonnull));

void caller_func(int n) {
  callee_func(n, &n);
  callee_func(n, (int *)0);
}
```

```
%> tiarmclang -mcpu=cortex-m0 -c nonnull_function_attr.c
nonnull_function_attr.c:6:26: warning: null passed to a callee␣
↪that requires a non-null argument [-Wnonnull]
  callee_func(n, (int *)0);
                 ~~~~~~~~^
1 warning generated.
```

noreturn

The *noreturn* function attribute can be used to identify a function that should not return to its caller. With *noreturn* applied to a function, the compiler will generate a warning diagnostic if a return from the function is detected.

The *noreturn* attribute cannot be applied to a function pointer.

Syntax

> void *symbol* (<arguments>) __attribute__((noreturn));

Example

The following example, when compiled, will emit a warning since *fake_return* contains a return statement:

```
__attribute__((noreturn)) void fake_noreturn() { return; }
```

```
%> tiarmclang -mcpu=cortex-m0 -c noreturn_warn.c
noreturn_warn.c:2:50: warning: function 'fake_noreturn' declared
↪'noreturn' should not return [-Winvalid-noreturn]
__attribute__((noreturn)) void fake_noreturn() { return; }
                                                 ^
1 warning generated.
```

optnone

The *optnone* function attribute can be applied to a function to instruct the compiler not to apply non-trivial optimizations in the generation of code for the function.

Syntax

> <return type> *symbol* (<arguments>) __attribute__((optnone));

Example

In the following C example, the *park_and_wait* function contains an empty while loop that the compiler would optimize away and remove references to the function if not for the application of the *optnone* attribute to *park_and_wait*:

```c
void check_peripheral();
__attribute__((optnone)) void park_and_wait();

void run_a_check_on_peripheral(void) {
  check_peripheral();
}

void check_peripheral() {
  if (*((unsigned long *)(268612608)) != 286529877) {
    park_and_wait();
  }
}

__attribute__((optnone)) void park_and_wait() {
  while(1) {
    ;
  }
}
```

The compiler-generated assembly for the above code shows that even though the *-O2* optimization option was specified, the reference to *park_and_wait* remains intact. If the *optnone* attribute had

---

3.2. C/C++ Language Implementation

not been applied to the *park_and_wait* function, the optimizer would have detected the empty while loop in *park_and_wait* and removed the reference to it in *check_peripheral*:

```
%> tiarmclang -mcpu=cortex-m4 -O2 -S optnone_function_attr.c
%> cat optnone_function_attr.s
...
        .section        .text.check_peripheral,"ax",%progbits
        .hidden check_peripheral            @ -- Begin function␣
↪check_peripheral
        .globl  check_peripheral
        .p2align        2
        .type   check_peripheral,%function
        .code   16                                  @ @check_
↪peripheral
        .thumb_func
check_peripheral:
        movw    r0, #46080
        movt    r0, #4098
        ldr     r0, [r0]
        movw    r1, #6485
        movt    r1, #4372
        cmp     r0, r1
        it      eq
        bxeq    lr
.LBB1_1:                                            @ %if.then
        bl      park_and_wait
...
```

pure

The *pure* function attribute can be applied to a function that is known to have no other observable effects on the state of a program other than to return a value. This information is useful to the compiler in performing optimizations such as common subexpression elimination.

Syntax

<return type> *symbol* (<arguments>) __attribute__((pure));

Example

Here is a simple example of applying the *pure* attribute to a function:

```
int decode(const char *str) __attribute__((pure));
```

The implication is that *decode* computes a value based on the string pointed to by *str* without modifying any other part of the program state.

---

section

The *section* function attribute can be used to instruct the compiler to place the definition of a function in a specific section. This is useful if you'd like to place specific functions separately from their default sections (e.g. .text).

Syntax

> <return type> *symbol* (<arguments>) __attribute__((section("*section name*")));

Example

In this program, *main* will get defined in a section called *main_section*:

```c
#include <stdio.h>

__attribute__((section("main_section"))) int main() {
  printf("hello\n");
  return 0;
}
```

When compiled and linked, the linker-generated map file shows that the symbol *main* is defined at address 0x22e1, which corresponds to the location of the *main_section* (with the THUMB function mode bit cleared to discern the actual target memory address of the section):

```
%> tiarmclang -mcpu=cortex-m0 section_function_attr.c -o a.out -
↪Wl,-llnk.cmd,-ma.map
%> cat a.map
...
SECTION ALLOCATION MAP

 output                                        attributes/
section    page    origin      length      input sections
--------   ----   ----------  ----------   ----------------
...
main_section
*          0     000022e0    0000001c
                 000022e0    0000001c     section_function_attr-
↪2eb101.o (main_section)
...
GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

address    name
-------    ----
...
000022e1   main
```

```
...
```

used/retain

The *used* or *retain* function attribute, when applied, will instruct the tiarmclang compiler to embed information in the compiler-generated code to instruct the linker to include the definition of the function in the link of a given application, even if it is not referenced elsewhere in the application.

Syntax

<return type> *symbol* (<arguments>) __attribute__((used));

<return type> *symbol* (<arguments>) __attribute__((retain));

Example

In the following program, the *retain* attribute is applied to *gb* so that its definition is kept in the link even though it is not called:

```c
#include <stdio.h>

void __attribute__((retain)) gb(void) {
  printf("goodbye\n");
}

int main() {
  printf("hello\n");
  return 0;
}
```

When compiled and linked, the linker-generated map file *a.map* shows that space has been allocated in target memory for the section where *gb* is defined:

```
%> tiarmclang -mcpu=cortex-m0 retain_global_func.c -o a.out -Wl,-
→llnk.cmd,-ma.map
%> cat a.map
...
SECTION ALLOCATION MAP

 output                                        attributes/
section    page     origin       length        input sections
--------   ----   ----------   ----------    ----------------
.text       0     00000020     000012b4
...
```

```
                    0000125c     00000010      retain_global_func-
↪242de9.o (.text.gb)
...
```

visibility

The *visibility* function attribute provides a way for you to dictate what visibility setting is to be associated with a function in the compiler-generated ELF symbol table. Visibility is particularly applicable for applications that make use of dynamic linking.

Syntax

>   <return type> *symbol* (<arguments>) __attribute__((visibility(”*visibility-kind*”)));

>   - *visibility-kind* indicates the visibility setting to be written into the symbol table entry for *symbol* in the compiler-generated ELF object file. The specified *visibility kind* will override the visibility setting that the compiler would otherwise assign to the *symbol*. The specified *visibility-kind* must be one of the following:

>   - *default* - external linkage; symbol will be included in the dynamic symbol table, if applicable, and can be accessed from other dynamic objects in the same application. This is the default visibility if no *visibility-kind* argument is specified with the *visibility* attribute.

>   - *hidden* - not included in the dynamic symbol table; symbol cannot be directly accessed from outside the current object, but may be accessed via an indirect pointer.

>   - *protected* - the symbol is included in the dynamic symbol table; references from within the same dynamic module will bind to the symbol and other dynamic modules cannot override the symbol.

Example

In the following C code, the *visibility* attribute is applied to *my_func* to mark it as *protected*:

```c
#include <stdio.h>

int my_func(int n) __attribute__((visibility("protected")));

void print_result(int n) {
  printf("my func returns: %d\n", my_func(n));
}
```

When compiled to an object file, the visibility setting for the *my_func* symbol is set to STV_PROTECTED in the symbol table:

```
%> tiarmclang -mcpu=cortex-m0 -c visibility_function_attr.c
%> tiarmofd -v visibility_function_attr.o
...
 Symbol Table ".symtab"
...
    <6> "my_func"
       Value:    0x00000000  Kind:        undefined
       Binding:  global       Type:        none
       Size:     0x0          Visibility:  STV_PROTECTED
...
```

weak

The *weak* function attribute causes the tiarmclang compiler to emit a weak symbol to the symbol table for the function symbol's declaration. At link-time, if a strong definition of a function symbol with the same name is included in the link, then the strong definition of the function will override the weak definition.

Syntax

        <return type> *symbol* (<arguments>) __attribute__((weak));

Example

Consider the following program with *weak_func_attr.c*:

```c
#include <stdio.h>

extern const char *my_func();

int main() {
  printf("my_func is: %s\n", my_func());
  return 0;
}
```

and *weak_func_def.c*:

```c
__attribute__((weak)) const char *my_func() {
  return "this is a weak definition of my_func\n";
}
```

and *strong_func_def.c*:

```
const char *my_func() {
  return "this is a strong definition of my_func\n";
}
```

If the program is compiled and linked without *strong_func_def.c*, then the weak definition of *my_func* will be chosen by the linker to resolve the reference to it in *weak_func_attr.c*:

```
%> tiarmclang -mcpu=cortex-m0 weak_func_attr.c weak_func_def.c -
↪o a.out -Wl,-llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
my_func is: this is a weak definition of my_func
```

If both *weak_func_def.c* and *strong_func_def.c* are included in the program build, then the linker will choose the strong definition of *my_func* to resolve the reference to it in *weak_func_attr.c*:

```
%> tiarmclang -mcpu=cortex-m0 weak_func_attr.c weak_func_def.c␣
↪strong_func_def.c -o a.out -Wl,-llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
my_func is: this is a strong definition of my_func
```

Note: Strong vs. Weak and Object Libraries

At link-time, if a weak definition of a symbol is available in the object files that are input to the linker and a strong definition of the symbol exists in an object library that is made available to the link, then the linker will not use the strong definition of the symbol since the reference to the symbol has already been resolved.

---

weakref

The *weakref* function attribute can be used to mark a declaration of a static function as a weak reference. The function *symbol* that the attribute applies to is interpreted as an alias of a *target symbol*, and also indicates that a definition of the *target symbol* is not required.

Syntax

> <return type> *symbol* (<arguments>) __attribute__((weakref("*target symbol*")));

> <return type> *symbol* (<arguments>) __attribute__((weakref, alias("*target symbol*")));

> > • *target symbol* - identifies the name of a function that the *symbol* being declared is to be treated as an alias for. If a *target symbol* argument is provided with

---

3.2. C/C++ Language Implementation

the *weakref* attribute, then *symbol* is interpreted as an alias of *target symbol*. Otherwise, an *alias* attribute must be combined with the *weakref* attribute to identify the *target symbol*.

Example

Consider the following program with *weakref_func_attr.c*:

```c
#include <stdio.h>

extern const char *my_func();
static const char *my_alias() __attribute__((weakref("my_func
 ↪")));

int main(void) {
  printf("my_alias returns %s", my_alias());
  return 0;
}
```

and *weak_func_def.c*:

```c
__attribute__((weak)) const char *my_func() {
  return "this is a weak definition of my_func\n";
}
```

and *strong_func_def.c*:

```c
const char *my_func() {
  return "this is a strong definition of my_func\n";
}
```

If the above program is compiled and linked without *strong_def.c*, the linker will choose the weak definition of *my_func* to resolve the call to *my_func* that goes through the *my_alias* weakref symbol:

```
%> tiarmclang -mcpu=cortex-m0 weakref_func_attr.c weak_func_def.
 ↪c -o a.out -Wl,-llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
my_alias returns this is a weak definition of my_func
```

If *strong_func_def.c* is included in the program build, the *my_alias* will resolve to the strong definition of *my_func*:

```
%> tiarmclang -mcpu=cortex-m0 weakref_func_attr.c weak_func_def.
 ↪c strong_func_def.c -o a.out -Wl,-llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
my_alias returns this is a strong definition of my_func
```

cmse_nonsecure_call

The *cmse _nonsecure_call* function attribute is a Cortex-M (ARMv8-M only) Security Extension (CMSE) and is only accepted by the tiarmclang compiler when the *-mcpu=cortex-m33* and *-mcmse* options are specified on the compiler command-line.

The *cmse_nonsecure_call* attribute is used to declare a non-secure function type to facilitate the use of a non-secure function pointer in secure memory code so that the secure code can execute a call to a function that is defined in non-secure memory. Secure code can only call non-secure code via a function pointer.

Syntax

> typedef void __attribute__((cmse_nonsecure_call)) *non-secure function type name* (<arguments>);

Example

The following example C code contains a definition of a secure function that is callable from non-secure code (via *cmse_nonsecure_entry* attribute) that is passed a pointer to a callback function. The secure function checks whether the callback function is defined in secure or non-secure memory and then calls the callback function.

```c
// cmse_nonsecure_call_ex.c
#include <arm_cmse.h>

// Declare non-secure function type
typedef void __attribute__((cmse_nonsecure_call)) nsfunc(void);

// Set up non-secure function pointer to call default handler
extern void secure_callback_handler(void);
void secure_defaul(void) { secure_callback_handler(); }
nsfunc *cbp = (nsfunc *) secure_default;

// Definition of non-secure callable secure function
void __attribute__((cmse_nonsecure_entry)) ns_callable(nsfunc
 *callback) {
  // Convert the callback function address so that it can be
 queried
  // as to whether it resides in secure or non-secure memory
  cbp = cmse_nsfptr_create(callback); // non-secure function
 pointer
```

(continues on next page)

```
}

// Callback function is called via non-secure function pointer,␣
↪cbp
void call_callback(void) {
  // If callback function is defined in non-secure memory, call␣
↪it
  // via a non-secure function pointer
  if (cmse_is_nsfptr(cbp)){ cbp(); }

  // Otherwise, the non-secure function pointer will be cast to a
  // normal function pointer before making the call
  else { ((void (*)(void)) cbp)(); }
}
```

Note: #include <arm_cmse.h>

The code that facilitates calls from secure code to non-secure code relies on the *cmse_nsfptr_create* and *cmse_is_nsfptr* intrinsics that are declared in *arm_cmse.h*. When writing secure code that needs to call non-secure code, be sure to include the *arm_cmse.h* header file in the compilation unit. The *arm_cmse.h* header file is provided with the **tiarmclang** compiler tools installation.

Related Reference

- *Cortex-M Security Extensions (CMSE)*

cmse_nonsecure_entry

The *cmse _nonsecure_entry* function attribute is a Cortex-M (ARMv8-M only) Security Extension (CMSE) and is only accepted by the **tiarmclang** compiler when the *-mcpu=cortex-m33* and *-mcmse* options are specified on the compiler command-line.

The *cmse_nonsecure_entry* attribute can be applied to a function defined in secure memory to indicate that the function is callable from non-secure memory. For secure functions that are marked with the *cmse_nonsecure_entry* attribute, to prevent information leakage from the secure state, the compiler takes the precaution of setting general-purpose registers and the N, Z, C, V, Q, and GE bits of the APSR register to a known value. If floating-point support is enabled when compiling a non-secure entry function, then the compiler also sets floating point registers and the FPSCR register to a known value.

Syntax

> void __attribute__((cmse_nonsecure_entry)) *symbol* (<arguments>);

extern "C" void __attribute__((cmse_nonsecure_entry)) *symbol* (<arguments>);

Example

The following example C code defines a secure function that is callable from non-secure memory:

```c
#include <arm_cmse.h>

extern void handle_service_request(int v);

void __attribute__((cmse_nonsecure_entry)) ns_service_
↪request(int val) {
  handle_service_request(val);
}
```

Related Reference

- *Cortex-M Security Extensions (CMSE)*

interrupt_save_fp

Like the *interrupt* attribute, the *interrupt_save_fp* function attribute, when applied, will identify a function as an interrupt function and instruct the compiler to generate code on entry and exit to preserve the system state. In addition, the *interrupt_save_fp* function attribute will instruct the compiler to preserve FPU registers, including arithmetic registers (d0-d16 on the fpv4-sp-d16 FPU, for example) and the FPEXC and FPSCR registers. The standard *interrupt* attribute does not save and restore FPU registers.

The tiarmclang compiler re-aligns the stack pointer to an 8-byte boundary on entry into an interrupt function because the Arm Procedure Call Standard (AAPCS) requires this alignment for all public interfaces.

For Arm Cortex-M processor applications, the architecture includes mechanisms to preserve all general-purpose integer registers. When compiling for an Arm Cortex-R processor, the compiler will generate code to preserve general-purpose registers and a sequence of instructions to effect an appropriate exception return for the processor.

Syntax

__attribute__((interrupt_save_fp)) <return type> *symbol* (<arguments>) { … }

__attribute__((interrupt_save_fp[("*interrupt-type*")])) void *symbol* () { … }

- *interrupt-type* - is the optional string literal argument that indicates the type of interrupt being defined. The tiarmclang compiler supports the following *interrupt-type* identifiers:

    - **IRQ**

- **FIQ**

- **SWI**

- **ABORT**

- **UNDEF**

Interrupt types other than **FIQ** save all core registers except the *lr* and *sp* registers. **FIQ** type interrupts save registers r0-r7.

Note that if an *interrupt-type* argument is specified with the *interrupt_save_fp* attribute specification, then the function interface may not take any arguments and may not return a value.

Example

Here are some examples of the *interrupt* attribute applied to functions:

```
extern int  do_something1(int n) __attribute__((interrupt_save_
↪fp));
extern void do_something2(void) __attribute__((interrupt_save_fp(
↪"IRQ")));
extern void do_something3(void) __attribute__((interrupt_save_fp(
↪"FIQ")));
extern void do_something4(void) __attribute__((interrupt_save_fp(
↪"SWI")));
extern void do_something5(void) __attribute__((interrupt_save_fp(
↪"ABORT")));
extern void do_something6(void) __attribute__((interrupt_save_fp(
↪"UNDEF")));

__attribute__((interrupt_save_fp)) void int1(int n) {
do_something1(n);
}
__attribute__((interrupt_save_fp("IRQ"))) void int2() {
  do_something2();
}
__attribute__((interrupt_save_fp("FIQ"))) void int3() {
  do_something3();
}
__attribute__((interrupt_save_fp("SWI"))) void int4() {
  do_something4();
}
__attribute__((interrupt_save_fp("ABORT"))) void int5() {
  do_something5();
}
__attribute__((interrupt_save_fp("UNDEF"))) void int6() {
```

(continues on next page)

```
    do_something6();
}
```

When compiled with the *-mcpu=cortex-r5* option, the compiler-generated assembly snippet below shows FPU as well as general-purpose registers being preserved during the function's entry and exit code. The preservation of the FPCSR and FPEXC registers must be done by first copying the FPCSR/FPEXC register into a general purpose register before pushing or popping the value onto/off the stack. As described in the *interrupt* attribute example, the re-alignment of the stack pointer is performed via a forced adjustment to the stack and a "bfc sp, #0, #3" instruction. The compiler will sometimes push and pop extra registers in order to effect a stack adjustment. In other cases, the compiler may use a "sub sp, sp, #8" instruction to force an adjustment to the stack prior to the "bfc sp, #0, #3" instruction. The return from the interrupt function is effected with the addition of the "subs pc, lr, #4" instruction.

```
%> tiarmclang -mcpu=cortex-r5 -S interrupt_save_fp_attr.c
%> cat interrupt_save_fp_attr.s
...
int2:
  push    {r0, r1, r2, r3, r4, r5, r10, r11, r12, lr}
  add     r11, sp, #28
  vmrs    r4, fpscr
  vmrs    r5, fpexc
  push    {r4, r5}
  vpush   {d0, d1, d2, d3, d4, d5, d6, d7}
  bfc     sp, #0, #3
  bl      do_something2
  sub     sp, r11, #100
  vpop    {d0, d1, d2, d3, d4, d5, d6, d7}
  pop     {r4, r5}
  vmsr    fpscr, r4
  vmsr    fpexc, r5
  pop     {r0, r1, r2, r3, r4, r5, r10, r11, r12, lr}
  subs    pc, lr, #4
...
```

local

When using *Smart Function and Data Placement*, the *local* function attribute can be used to more easily place a function in memory local to the processor (for example, in Tightly Coupled Memory (TCM)).

Syntax

> __attribute__((local(*priority*))) <return type> *symbol* (<arguments>) { … }

> - *priority* - is an optional integer argument indicating the placement order of the object relative to other objects aggregated in the same output section. This influences how the placements are sorted at link-time. If omitted, the priority is assumed to be the highest priority, which is '1'.

Example

```
__attribute__((local(3))) int main() {
  printf("Good morning, Dave.\n");
  return 0;
}
```

nothrow

When applied to a function, the *nothrow* attribute informs the compiler that the function cannot throw an exception. If the compiler knows that a function cannot throw an exception, it may be able to optimize the size of exception-handling tables for callers of the function.

Syntax

> <return type> *symbol* (<arguments>) __attribute__((nothrow));

See *C++ Exception Handling* for more about exception handling.

offchip

When using *Smart Function and Data Placement*, the *offchip* function attribute can be used to more easily place a function in offchip FLASH.

Syntax

> __attribute__((offchip(*priority*))) <return type> *symbol* (<arguments>) { … }

> - *priority* - is an optional integer argument indicating the placement order of the object relative to other objects aggregated in the same output section. This influences how the placements are sorted at link-time. If omitted, the priority is assumed to be the highest priority, which is '1'.

Example

```
__attribute__((offchip(2))) int main() {
  printf("Good morning, Dave.\n");
  return 0;
}
```

onchip

When using *Smart Function and Data Placement*, the *onchip* function attribute can be used to more easily place a function in onchip RAM.

Syntax

> __attribute__((onchip(*priority*))) <return type> *symbol* (<arguments>) { … }

>   • *priority* - is an optional integer argument indicating the placement order of the object relative to other objects aggregated in the same output section. This influences how the placements are sorted at link-time. If omitted, the priority is assumed to be the highest priority, which is '1'.

Example

```
__attribute__((onchip(5))) int main() {
  printf("Good morning, Dave.\n");
  return 0;
}
```

Variable Attributes

The following variable attributes are supported by the tiarmclang compiler:

   • *alias*

   • *aligned*

   • *deprecated*

   • *location*

   • *noinit*

   • *packed*

   • *persistent*

   • *section*

   • *unused*

- *used/retain*

- *visibility*

- *weak*

- *weakref*

- *local*

- *offchip*

- *onchip*

alias

The *alias* variable attribute allows you to create multiple symbol aliases that effectively refer to the same definition of a data object. However, an alias must be declared in the same compilation unit as the definition of the data object that it is an alias for. Furthermore, you cannot declare aliases to local variables. The tiarmclang compiler interprets an alias declared in a local block as a local variable, ignoring the alias attribute in such cases.

Syntax

> <type> *new symbol* __attribute((alias("*old symbol*")));

> - *new symbol* - is the name of the alias.

> - *old symbol* - is the name of the variable to be aliased.

Example

Consider the following C code:

```c
#include <stdio.h>

int red_fish = 10;
extern int blue_fish __attribute__((alias("red_fish")));

int main() {
  printf("blue_fish: %d\n", blue_fish);
  return 0;
}
```

The compiler generated code for the above code contains the following assembly:

```
...
        .hidden red_fish                           @ @red_fish
        .type   red_fish,%object
```

3.2. C/C++ Language Implementation   **295**

```
        .section        .data.red_fish,"aw",%progbits
        .globl  red_fish
        .p2align        2
red_fish:
        .long   10                                      @ 0xa
        .size   red_fish, 4
...
        .globl  blue_fish
        .hidden blue_fish
        .set    blue_fish, red_fish
```

In this case, there is a definition of the global variable *red_fish* and the subsequent *.set* directive
creates a symbolic link from *blue_fish* to *red_fish* so that any references to *blue_fish* are resolved
by the definition of *red_fish*.

aligned

The *aligned* attribute can be used to specify a minimum alignment for a given data object, where
the alignment boundary is specified in bytes.

Syntax

> <type> *symbol* __attribute((aligned(*alignment*)));

> - *symbol* - is the variable/data object that is subject to the specified minimum alignment.

> - *alignment* - is the minimum alignment (in bytes) relative to the section that *symbol* is defined in. If an *alignment* value is not specified, then the compiler assumes default alignment based on the type of the data object.

Example

Consider the C source code below (align_var_attr.c):

```c
int var1 __attribute__((aligned(8))) = 5;

unsigned char var2[10] __attribute__((aligned(16))) = { 15, 25,
↪35 };

struct {
  int m1;
  short m2;
  char m3 __attribute__((aligned(4)));
  short m4;
```

```
} var3 = { 10, 20, 30, 40 };

short var4[3] __attribute__((aligned)) = { 100, 200, 300 };
```

The compiler generated code for the above code contains the following assembly:

```
...
        .hidden var1                                @ @var1
        .type   var1,%object
        .section        .data.var1,"aw",%progbits
        .globl  var1
        .p2align        3
var1:
        .long   5                                   @ 0x5
        .size   var1, 4

        .hidden var2                                @ @var2
        .type   var2,%object
        .section        .data.var2,"aw",%progbits
        .globl  var2
        .p2align        4
var2:
        .asciz  "\017\031#\000\000\000\000\000\000"
        .size   var2, 10

        .hidden var3                                @ @var3
        .type   var3,%object
        .section        .data.var3,"aw",%progbits
        .globl  var3
        .p2align        2
var3:
        .long   10                                  @ 0xa
        .short  20                                  @ 0x14
        .zero   2
        .byte   30                                  @ 0x1e
        .zero   1
        .short  40                                  @ 0x28
        .size   var3, 12

        .hidden var4                                @ @var4
        .type   var4,%object
        .section        .data.var4,"aw",%progbits
        .globl  var4
```

```
        .p2align        3
var4:
        .short  100                                     @ 0x64
        .short  200                                     @ 0xc8
        .short  300                                     @ 0x12c
        .size   var4, 6
```

Some things to notice about the compiler-generated assembly:

- *var1* is aligned to an 8-byte boundary via the *.p2align 3* directive (the *.p2align* operand is interpreted as an exponent in the expression "2 ^^ *exp*").

- *var2* is aligned to a 16-byte boundary via the *.p2align 4* directive.

- The alignment of *var3*'s char type member, *m3*, is effected via the *.zero 2* directive that pads the start of *m3* to a 4-byte boundary relative to the start of the structure.

- The *aligned* attribute associated with *var4* does not take an *alignment* argument, so the compiler assumes default alignment for an array of short, 8-bytes.

deprecated

The *deprecated* variable attribute can be used to mark a symbol as deprecated to enable the compiler to detect and report warnings on uses of a symbol whose definition is known to be deprecated.

Syntax

<type> *symbol* __attribute__((deprecated));

- *symbol* - identifies the name of the variable being marked as deprecated.

Example

Consider the following C code (deprecated_var_attr.c):

```
extern int dep_var __attribute__((deprecated));
void foo() {
  dep_var = 5;
}
```

When compiled, the compiler emits the following diagnostic information:

```
%> tiarmclang -mcpu=cortex-m0 -c deprecated_var_attr.c
deprecated_var_attr.c:4:3: warning: 'dep_var' is deprecated [-
↪Wdeprecated-declarations]
  dep_var = 5;
  ^
```

```
deprecated_var_attr.c:2:35: note: 'dep_var' has been explicitly␣
↪marked deprecated here
extern int dep_var __attribute__((deprecated));
                                  ^
1 warning generated.
```

The *deprecated* attribute can be particularly useful in a large C/C++ source file when trying to find all the references to a deprecated symbol that need to be modified.

location

The *location* variable attribute can be used to specify a variable's run-time address from within the C/C++ source. The tiarmclang compiler embeds linker instructions within a given compiler-generated object file that dictates where in target memory the variable definition are placed at link-time.

Syntax

> <type> *symbol* __attribute__((location(*address*)));

> - *address* - is the run-time target memory address where the definition of *symbol* is to be placed at link-time.

Example

Consider the following C source where a *location* attribute applied to a global variable (location_var_attr.c):

```c
#include <stdio.h>

int xyz __attribute__((location(0x30000000))) = 10;

int main()
{
  printf("address of xyz is 0x%lx\n", (unsigned long)&xyz);
  return 0;
}
```

The compiler defines *xyz* in a special *.TI.bound:xyz* section. It also emits symbol metadata information to instruct the linker to place *xyz* at target memory address 0x30000000 (805306368 in decimal) at link-time.

```
...
        .hidden xyz                                   @ @xyz
        .type   xyz,%object
```

```
        .section        ".TI.bound:xyz","aw",%progbits
        .globl  xyz
        .p2align        2
xyz:
        .long   10                              @ 0xa
        .size   xyz, 4
        .sym_meta_info  xyz, "location", 805306368
```

If the above program is compiled and linked, the linker-generated map file, *a.map*, reveals that the
run-time address of *xyz* is indeed 0x30000000:

```
%> tiarmclang -mcpu=cortex-m0 location_var_attr.c -o a.out -Wl,-
↪llnk.cmd,-ma.map
%> cat a.map
*****************************************************************************
TI ARM Clang Linker Unix v1.2.0
*****************************************************************************
>> Linked Tue Jan 19 18:49:47 2021


OUTPUT FILE NAME:   <a.out>
ENTRY POINT SYMBOL: "_c_int00"  address: 0000187d


...
SECTION ALLOCATION MAP

 output                                      attributes/
section    page    origin      length      input sections
--------   ----    ----------  ----------  ----------------
...
.TI.bound:xyz
*          0     30000000    00000004    UNINITIALIZED
                 30000000    00000004    location_var_attr-
↪baf510.o (.TI.bound:xyz)
...
```

noinit

The *noinit* variable attribute is especially useful in applications where non-volatile memory is in use. The *noinit* attribute identifies a global or static variable that should not be initialized at startup or reset (typically, global and static variables that aren't explicitly initialized in the source code are zero-initialized at startup and reset).

The *noinit* attribute can be used in conjunction with the *location* attribute to specify the placement of variables at special target memory locations, like memory-mapped registers, without generating unwanted writes.

The *noinit* attribute may only be used with uninitialized variables.

Syntax

<type> *symbol* __attribute__((noinit));

Example

Consider the following C source (noinit_var_attr.c):

```c
#include <stdio.h>

extern void usei(int *x);

__attribute__((noinit)) int noinit_global;
__attribute__((noinit,location(0x100))) int noinit_location_
↪global;

int main() {
  usei(&noinit_global);
  usei(&noinit_location_global);
  return 0;
}
```

The compiler generated code for the above code contains the following assembly:

```
...
        .hidden noinit_global                       @ @noinit_global
        .type    noinit_global,%object
        .section         .TI.noinit,"aw",%nobits
        .globl   noinit_global
        .p2align         2
noinit_global:
        .long    0                                  @ 0x0
        .size    noinit_global, 4
        .sym_meta_info   noinit_global, "noinit", 1
```

3.2. C/C++ Language Implementation   **301**

```
        .hidden noinit_location_global              @ @noinit_
↪location_global
        .type    noinit_location_global,%object
        .section         ".TI.bound:noinit_location_global","aw",
↪%nobits
        .globl  noinit_location_global
        .p2align        2
noinit_location_global:
        .long   0                                    @ 0x0
        .size   noinit_location_global, 4
        .sym_meta_info  noinit_location_global, "location", 256
        .sym_meta_info  noinit_location_global, "noinit", 1
...
```

Some things to notice about the above compiler-generated code for the definitions of *noinit_global* and *noinit_location_global*:

- *noinit_global* is defined in a special section, *.TI.noinit*, to keep such data objects apart from other variable definitions that will be initialized

- Symbol metadata information is emitted along with the definition of *noinit_global* to indicate that the section where *noinit_global* is defined is not to be initialized

- *noinit_location_global* is also defined in a special section, *.TI.bound:noinit_location_global*, that the linker will consider for placement early in the link-step

- There are two pieces of symbol metadata information emitted by the compiler with the definition of *noinit_location_global*. The first instructs the linker to place the section where *noninit_location_global* is defined at target memory address 0x100 (256 decimal), and the second indicates to the linker that the section where *noinit_location_global* is defined is not to be initialized

packed

If the program in question is being built for an Arm processor variant that has support for unaligned memory accesses, then the *packed* variable attribute can be used to compress data layout.

The *packed* attribute specifies that a variable or structure field should have the smallest possible alignment - one byte for a variable, and one bit for a bit field - unless a larger alignment requirement is indicated with an *aligned* attribute.

Syntax

<type> *symbol* __attribute__((packed));

Example

Consider the following C code in which a *packed* attribute is applied to a struct member:

```
struct _stag {
  char m1;
  int  m2 __attribute__((packed));
} my_struct = { 10, 20 };
```

In this case, the *m2* member of *my_struct* is aligned to a 1 byte boundary. Support for unaligned memory accesses need to be in effect for code to access the content of *m2*.

Note that when accessing a packed member of a struct, the member should be accessed via a reference through the base of the structure itself (e.g. "my_struct.m2") or via an offset from a pointer that has been set to the base of the struct (e.g. "struct _stag *ps = &my_struct; ps->m2 = 30;").

persistent

The *persistent* variable attribute is especially useful in applications where non-volatile memory is in use. The *persistent* attribute identifies a global or static variable that is to be initialized at load-time, but should not be re-initialized at reset.

The *persistent* attribute can be used in conjunction with the *location* attribute to specify the placement of variables at special target memory locations, like memory-mapped registers, without generating unwanted writes.

The *persistent* attribute may only be used with statically initialized variables.

Syntax

> <type> *symbol* __attribute__((persistent);

Example

If you are using non-volatile RAM, you can define a *persistent* variable with an initial value of zero loaded into RAM. The program can increment that variable over time as a counter, and that count does not disappear if the device loses power and restarts, because the memory is non-volatile and the boot routines do not initialize it back to zero.

For example, compiling the following C code:

```
extern void run_init(void);
extern void run_actions(int n);
extern void delay(unsigned int cycles);

__attribute__((persistent, location(0xC200))) int x = 0;
```

3.2. C/C++ Language Implementation

```
void main() {
  run_init();
  while (1) {
    run_actions(x);
    delay(1000000);
    x++;
  }
}
```

generates the following definition of *x*:

```
...
        .hidden x                                    @ @x
        .type    x,%object
        .section         ".TI.bound:x","aw",%progbits
        .globl  x
        .p2align        2
x:
        .long   0                                    @ 0x0
        .size   x, 4
        .sym_meta_info  x, "location", 49664
        .sym_meta_info  x, "noinit", 1
...
```

The variable *x* is directly initialized by the *.long* directive in the initialized section where *x* is defined. Symbol metadata for *x* is embedded in the compiler-generated code to instruct the linker to place the section where *x* is defined at address 0xC200 (49664 decimal), and to not initialize the definition of *x* on reset.

section

The *section* variable attribute can be used to instruct the compiler to place the definition of a data object in a specific section. This is useful if you'd like to place specific data objects separately from their default sections (e.g. .bss, .rodata, .data).

Syntax

> <type> *symbol* __attribute__((section("*section_name*")));

> - *section name* - is the name of the section where *symbol* will be defined. It must be specified as a string argument in the *section* attribute specification. used to instruct the compiler to place the definition of a data object in a spec

Example

3.2. C/C++ Language Implementation

The following C code uses the *section* attribute to generate the definition of *bufferB* into a different section from *bufferA*:

```
char bufferA[512];
__attribute__((section("my_sect"))) char bufferB[512];
```

The compiler-generated assembly snippet for the above C code shows that *bufferA* gets defined in a common block by the *.comm* directive, whereas *bufferB* gets defined in the *my_sect* section.

```
...
        .hidden bufferA                                 @ @bufferA
        .type   bufferA,%object
        .comm   bufferA,512,1
        .hidden bufferB                                 @ @bufferB
        .type   bufferB,%object
        .section        my_sect,"aw",%nobits
        .globl  bufferB
bufferB:
        .zero   512
        .size   bufferB, 512
```

unused

When the *unused-variable* category of warning diagnostics is enabled, the tiarmclang compiler generates a warning if a variable is declared in a compilation unit, but never referenced in the same compilation unit. The *unused* variable attribute can be applied to a variable declaration to disable the *unused-variable* warning with respect to that variable.

Syntax

> <type> *symbol* __attribute__((unused));

Example

In the following C code, *a_var* is marked as *unused*:

```
void foo()
{
  static int my_stat = 0;
  int a_var __attribute__((unused));
  int b_var;
  my_stat;
}
```

When compiled with *unused-variable* warnings enabled (via -*Wall* option in this case), tiarmclang emits a warning about unused variable *b_var*, but not *a_var*.

```
%> tiarmclang -mcpu=cortex-m0 -Wall -c unused_var_attr.c
unused_var_attr.c:6:3: warning: expression result unused [-
↪Wunused-value]
  my_stat;
  ^~~~~~~
unused_var_attr.c:5:7: warning: unused variable 'b_var' [-
↪Wunused-variable]
int b_var;
    ^
2 warnings generated
```

used/retain

The *used* or *retain* variable attribute, when applied, instructs the tiarmclang compiler to embed information in the compiler-generated code to instruct the linker to include the definition of the variable in the link of a given application, even if it is not referenced elsewhere in the application.

Syntax

> <type> *symbol* __attribute__((used));

> <type> *symbol* __attribute__((retain));

Example

In the following C code example, the tiarmclang compiler generates a definition of the file static data object *keep_this* even though it is not referenced elsewhere in the compilation unit:

```
static int lose_this = 1;
static int keep_this __attribute__((used)) = 2; // retained in
↪object file
```

The compiler-generated code also includes a *.no_dead_strip* directive that instructs the linker to include the definition of the *keep_this* in a link that includes the compiler generated object file from the above code:

```
...
       .type    keep_this,%object              @ @keep_this
       .section         .data.keep_this,"aw",%progbits
       .p2align        2
keep_this:
       .long    2                              @ 0x2
       .size    keep_this, 4
```

```
        .no_dead_strip  keep_this
...
```

The compiler does not produce a definition of *lose_this*.

The example below shows a variable *used_varX* that is annotated with a *retain* attribute:

```c
#include <stdio.h>

int used_varX __attribute__((retain)) = 10;

int main()
{
  printf("hello\n");
  return 0;
}
```

After compiling and linking with the following command:

```
%> tiarmclang -mcpu=cortex-m0 retain_init_global_var.c -o a.out -
→Wl,-llnk.cmd,-ma.map
```

The contents of *a.map* reveals that *used_varX* was retained in the linked program:

```
%> cat a.map
...
SECTION ALLOCATION MAP

 output                                          attributes/
section     page     origin       length       input sections
--------    ----    ----------   ----------    ----------------
.text        0      00000020     000012a0
...
.data        0      2000a020     000001d1     UNINITIALIZED
                    2000a020     000000f0      libc.a : defs.c.obj (.
→data._ftable)
                    ...
                    2000a1ec     00000004      retain_init_global_
→var-d1e7b9.o (.data.used_varX)
                    ...
```

visibility

The *visibility* variable attribute provides a way for you to dictate what visibility setting is to be associated with a variable in the compiler-generated ELF symbol table. Visibility is particularly applicable for applications that make use of dynamic linking.

Syntax

> <type> *symbol* __attribute__((visibility("*visibility-kind*)));

> - *visibility-kind* indicates the visibility setting to be written into the symbol table entry for *symbol* in the compiler-generated ELF object file. The specified *visibility kind* overrides the visibility setting that the compiler would otherwise assign to the *symbol*. The specified *visibility-kind* must be one of the following:
>
>   - *default* - external linkage; symbol ise included in the dynamic symbol table, if applicable, and can be accessed from other dynamic objects in the same application. This is the default visibility if no *visibility-kind* argument is specified with the *visibility* attribute.
>
>   - *hidden* - not included in the dynamic symbol table; symbol cannot be directly accessed from outside the current object, but may be accessed via an indirect pointer.
>
>   - *protected* - the symbol is included in the dynamic symbol table; references from within the same dynamic module bind to the symbol and other dynamic modules cannot override the symbol.

Example

The following use of the *visibility* attribute sets the visibility of *my_var* to *protected*:

```
int my_var __attribute__((visibility("protected"))) = 1;
```

When compiled to an object file, the symbol table entry for *my_var* reflects that it has a *protected* visibility kind:

```
%> tiarmclang -mcpu=cortex-m0 -c visibility_var_attr.c
%> tiarmofd -v visibility_var_attr.o
...
 Symbol Table ".symtab"

    <0> ""
       Value:    0x00000000  Kind:        undefined
       Binding:  local        Type:        none
       Size:     0x0          Visibility:  STV_DEFAULT

    <1> "visibility_var_attr.c"
```

```
        Value:     0x00000000  Kind:        absolute
        Binding:   local        Type:        file
        Size:      0x0          Visibility:  STV_DEFAULT


    <2> "my_var" (defined in section ".data.my_var" (3))
        Value:     0x00000000  Kind:        defined
        Binding:   global       Type:        object
        Size:      0x4          Visibility:  STV_PROTECTED
 ...
```

weak

The *weak* variable attribute causes the tiarmclang compiler to emit a weak symbol to the symbol table for the symbol's declaration. At link-time, if a strong definition of a symbol with the same name is included in the link, then the strong definition of the symbol overrides the weak definition.

Syntax

<type> *symbol* __attribute__((weak));

Example

Consider the following program with *weak_var_attr.c*:

```c
#include <stdio.h>

extern int my_var;

int main() {
  printf("my_var is: %d\n", my_var);
  return 0;
}
```

weak_def.c:

```c
int my_var __attribute__((weak)) = 5;
```

and *strong_def.c*:

```c
int my_var = 10;
```

If the program is compiled without *strong_def.c*, then the weak definition of *my_var* is chosen by the linker to resolve the reference to it in *weak_var_attr.c*:

```
%> tiarmclang -mcpu=cortex-m0 weak_var_attr.c weak_def.c -o a.
↪out -Wl,-llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
my_var is: 5
```

If both *weak_def.c* and *strong_def.c* are included in the program build, then the linker chooses the strong definition of *my_var* to resolve the reference to it in *weak_var_attr.c*:

```
%> tiarmclang -mcpu=cortex-m0 weak_var_attr.c weak_def.c strong_
↪def.c -o a.out -Wl,-llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
my_var is: 10
```

Note: Strong vs. Weak and Object Libraries

At link-time, if a weak definition of a symbol is available in the object files that are input to the linker and a strong definition of the symbol exists in an object library that is made available to the link, then the linker does not use the strong definition of the symbol since the reference to the symbol has already been resolved.

---

weakref

The *weakref* variable attribute can be used to mark a declaration of a static variable as a weak reference. The *symbol* that the attribute applies to is interpreted as an alias of a *target symbol*, and also indicates that a definition of the *target symbol* is not required.

Syntax

<type> *symbol* __attribute__((weakref(”*target symbol*)));

<type> *symbol* __attribute__((weakref, alias(”*target symbol*”)));

- *target symbol* - identifies the name of a variable that the *symbol* being declared is to be treated as an alias for. If a *target symbol* argument is provided with the *weakref* attribute, then *symbol* is interpreted as an alias of *target symbol*. Otherwise, an *alias* attribute must be combined with the *weakref* attribute to identify the *target symbol*.

Example

Consider the following program with *weakref_var_attr.c*:

```
#include <stdio.h>

extern int my_var;
static int a_sym __attribute__((weakref("my_var")));

int main(void) {
  int my_loc = a_sym;

  printf("my_loc is %d\n", my_loc);
  return 0;
}
```

and *strong_def.c*:

```
int my_var = 10;
```

If the above program is compiled and linked without *strong_def.c*, the build succeeds, but the run-time behavior will be unpredictable as there will be a reference to an undefined symbol:

```
%> tiarmclang -mcpu=cortex-m0 weakref_var_attr.c -o a.out -Wl,-
↪llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
Data fetch: 00000000 is outside configured memory
```

However, if we include *strong_def.c* in the link, then the reference to *a_sym* resolves to the definition of *my_var* via the *weakref* attribute:

```
%> tiarmclang -mcpu=cortex-m0 weakref_var_attr.c strong_def.c -o␣
↪a.out   -Wl,-llnk.cmd,-ma.map
```

The output when the program is loaded and run is as follows:

```
my_loc is 10
```

If we were to replace the *weakref* attribute with an *alias* attribute in *weakref_var_attr.c*:

```
#include <stdio.h>

extern int my_var;
// static int a_sym __attribute__((weakref("my_var")));
static int a_sym __attribute__((alias("my_var")));

int main(void) {
```

```
  int my_loc = a_sym;

  printf("my_loc is %d\n", my_loc);
  return 0;
}
```

and *strong_def.c* was not included in the build, then tiarmclang reports an unresolved symbol
reference:

```
%> tiarmclang -mcpu=cortex-m0 weakref_var_attr.c -o a.out -Wl,-
↪llnk.cmd,-ma.map

weakref_var_attr.c:6:33: error: alias must point to a defined
↪variable or function
static int a_sym __attribute__((alias("my_var")));
                                ^
1 error generated.
```

local

When using *Smart Function and Data Placement*, the *local* variable attribute can be used to more
easily place data in memory local to the processor (for example, in Tightly Coupled Memory
(TCM)).

Syntax

> <type> *symbol* __attribute__((local(*priority*)));

> - *priority* - is an optional integer argument indicating the placement order of the
>   object relative to other objects aggregated in the same output section. This in-
>   fluences how the placements are sorted at link-time. If omitted, the priority is
>   assumed to be the highest priority, which is '1'.

Example

```
int abc __attribute__((local(3))) = 10;
```

offchip

When using *Smart Function and Data Placement*, the *offchip* variable attribute can be used to more easily place data in offchip FLASH.

Syntax

> <type> *symbol* __attribute__((offchip(*priority*)));

> - *priority* - is an optional integer argument indicating the placement order of the object relative to other objects aggregated in the same output section. This influences how the placements are sorted at link-time. If omitted, the priority is assumed to be the highest priority, which is '1'.

Example

```
int abc __attribute__((offchip(2))) = 10;
```

onchip

When using *Smart Function and Data Placement*, the *onchip* variable attribute can be used to more easily place data in onchip RAM.

Syntax

> <type> *symbol* __attribute__((onchip(*priority*)));

> - *priority* - is an optional integer argument indicating the placement order of the object relative to other objects aggregated in the same output section. This influences how the placements are sorted at link-time. If omitted, the priority is assumed to be the highest priority, which is '1'.

Example

```
int abc __attribute__((onchip(5))) = 10;
```

Type Attributes

The tiarmclang compiler supports the application of type attributes to *enum*, *struct*, or *union* declarations or definitions. Type attributes can also be applied to a type that is defined via *typedef* declarations.

The following type attributes are supported by the tiarmclang compiler:

- *aligned*
- *packed*

aligned

The *aligned* type attribute indicates a minimum byte boundary alignment for variables of the specified type. This attribute is especially useful for overriding the default compiler-imposed constraint on a particular data object, especially when a more restrictive alignment requirement is warranted.

Syntax

> <type specification> __attribute__((aligned(*alignment*)));

> - *alignment* - the minimum alignment for the indicated type, specified in bytes. The *alignment* value must be an integer power of two. The compiler imposes the maximum of the default alignment for the type and the specified *alignment* on data objects of the type.

Examples

- An *aligned* attribute applied to a *typedef*:

```c
typedef short a_short_type __attribute__((aligned(4)));
```

Use of the *a_short_type* in C source code forces the definition of any data objects of that type to be placed on a 4-byte boundary.

- An *aligned* attribute applied to a struct type:

```c
struct myS {
  char  m1;
  int   m2
  int   m3
  char  m4;
  short m5;
} __attribute__((aligned(8)));
```

In this case, the *myS* struct is aligned to an 8-byte boundary instead of what the compiler would impose by default (4-byte boundary).

- An *aligned* attribute applied to a union within a struct:

```c
#include <stdio.h>

typedef struct {
  char m1;
  union {
    short m2_u_m1;
    int   m2_u_m2;
    char  m2_u_m3;
  } m2_u __attribute__((aligned(16)));
```

3.2. C/C++ Language Implementation   **314**

```
} myS;

myS myS_obj;

int main() {
  printf("address of myS_obj: 0x%08lx\n", (unsigned long)&
→myS_obj);
  printf("address of m2_u_m1: 0x%08lx\n",
         (unsigned long)&myS_obj.m2_u.m2_u_m1);
  return 0;
}
```

When compiled and linked and run, the output reveals that the target memory location where the first member of the *m2_u* union resides in memory is on a 16-byte boundary relative to the start of the struct *myS_obj*:

```
%> tiarmclang -mcpu=cortex-m0 aligned_type_attr.c -o a.out -
→Wl,-llnk.cmd,-ma.map
```

The output is:

```
address of myS_obj: 0x2000a1e0
address of m2_u_m1: 0x2000a1f0
```

Note that the *myS* type alignment is also 16-bytes since the alignment of the struct is determined by the struct member with the most restrictive alignment constraint (*m2_u* in this case).

packed

The *packed* type attribute can be applied to struct or union types to indicate that each member of a given struct or union is placed on a 1-byte boundary.

Syntax

> <type specification> __attribute__((packed));

Examples

Consider the following C code in which a *packed* attribute is applied to a struct type:

```
struct __attribute__((packed)) {
  char m1;
  int  m2;
} packed_struct = { 10, 20 };
```

In this case, the size of *packed_struct* is 5 bytes, whereas without the *packed* attribute the int type member *m2* would have been aligned to a 4-byte boundary causing the size of the struct to be 8 bytes.

3.2.8 Pragmas

The following pragmas are supported by the tiarmclang compiler:

- *clang section text*
- *clang section data*
- *clang section bss*
- *clang section rodata*

clang section text

The *clang section text* pragma places enclosed functions within a named section, which can then be placed with the linker using a linker command file.

Syntax

> #pragma clang section text="scn_name"

The setting is reset to the default section name using

> #pragma clang section text=""

Example

The following use of the *clang section text* pragma causes the enclosed function to be included in a section called *.text.functions*

```
#include <stdio.h>

#pragma clang section text=".text.functions"

int main() {
  emit_msg();
  emit_msg();
  emit_msg();
}

#pragma clang section text=""
```

clang section data

The *clang section data* pragma places enclosed variables within a named section, which can then be placed with the linker using a linker command file.

Syntax

> #pragma clang section data="scn_name"

The setting is reset to the default section name using

> #pragma clang section data=""

Example

The following use of the *clang section data* pragma causes the enclosed variables to be included in a section called *.data.variables*

```c
#include <stdio.h>

#pragma clang section data=".data.variables"

int var1 = 39;
char *myString = "this is a test";

#pragma clang section data=""

extern void func(int, char*);

int main() {
  func(var1, myString);
}
```

Note: Variables that are not initialized with a constant expression are not defined in .data

For example, in the above example, if either of the *var1* or *myString* definitions were uninitialized, then they would not be defined in *.data.variables*. They would instead be defined in *.bss.var1* and/or *.bss.myString"*.

---

clang section bss

The *clang section bss* pragma places enclosed variables within a named section, which can then be placed with the linker using a linker command file.

Syntax

#pragma clang section bss="scn_name"

The setting is reset to the default section name using

#pragma clang section bss=""

Example

The following use of the *clang section bss* pragma causes the definition of *myString* to be included in a section called *.bss.variables*

```c
#include <stdio.h>

#pragma clang section bss=".bss.variables"

int var1 = 39;
char *myString;

#pragma clang section bss=""

extern void init_myString(const char*);
extern void func(int, char*);

int main() {
  init_myString("hello world");
  func(var1, myString);
}
```

Note: Variables that are initialized with a constant expression are not defined in .bss

For example, in the above example, *myString* is defined in *.bss.variables*, but *var1* is defined in *.data.var1* since it is initialized with a constant expression.

clang section rodata

The *clang section rodata* pragma places enclosed variables within a named section, which can then be placed with the linker using a linker command file.

Syntax

#pragma clang section rodata="scn_name"

The setting is reset to the default section name using

#pragma clang section rodata=""

Example

The following use of the *clang section rodata* pragma causes the enclosed const qualified data object definitions to be included in a section called *MyRodata*

```c
#include <stdio.h>

#pragma clang section rodata="MyRodata"

const int var1 = 39;
const char *myString = "this is a test";

#pragma clang section rodata=""

extern void func(const int, const char*);

int main() {
  func(var1, myString);
}
```

Note: A General Note About *clang section* Pragmas

In general, only variable definitions that match the type of the preceding *#pragma clang section <type>="scn_name"* are affected by that *clang section* pragma.

You can specify more than one section type in a *clang section* pragma. For example,

```c
#pragma clang section bss="myBSS" data="myData" rodata="myRodata"
int x2 = 5;                      // Goes in myData section.
int y2;                          // Goes in myBss section.
const int z2 = 42;               // Goes in myRodata section.
```

If you were to turn off the *clang section rodata* between definitions of const qualified data objects:

3.2. C/C++ Language Implementation   **319**

```
#pragma clang section bss="myBSS" data="myData" rodata="myRodata"
int x2 = 5;                        // Goes in myData section.
int y2;                            // Goes in myBss section.
const int z2 = 42;                 // Goes in myRodata section.


#pragma clang section rodata="" // Use default name for rodata␣
 ↪section.
int x3 = 5;                        // Goes in myData section.
int y3;                            // Goes in myBss section.
const int z3 = 42;                 // Goes in .rodata section
```

Note that *z3* is not defined in *myBSS* or *myData* because it does not match the *bss* or *data* type specified in the first *clang section* pragma.

3.2.9 Intrinsics

The ACLE compiler intrinsics are fully documented in the Arm C Language Extensions - Release ACLE Q3 2020. A summary of each of the ACLE compiler intrinsics that are supported by the **tiarmclang** compiler is provided below. Note that many of the intrinsics documented in the ACLE specification are only available on Arm processors that are not currently supported by **tiarmclang**.

Note: #include <arm_acle.h>

The *<arm_acle.h>* header file should be included in your compilation unit before using any of the ACLE intrinsics.

Data-Processing Intrinsics

Rotate and Bit-Manipulation Intrinsics

__ror, __rorl, __rorll

Signatures

```
uint32_t        __ror(uint32_t x, uint32_t y);
unsigned long __rorl(unsigned long x, unsigned long y);
uint64_t        __rorll(uint64_t x, uint64_t y);
```

Description

Rotate *x* by *y* bits; return the result.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__clz, __clzl, __clzll

Signatures

```
unsigned int __clz(uint32_t x);
unsigned int __clzl(unsigned long x);
unsigned int __clzll(uint64_t x);
```

Description

Return the number of leading zero bits in *x*. If *x* == 0, then the return value is (width of *x*).

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__cls, __clsl, __clsll

Signatures

```
unsigned int __cls(uint32_t x);
unsigned int __clsl(unsigned long x);
unsigned int __clsll(uint64_t x);
```

Description

Return the number of leading sign bits in *x*. If *x* == 0, then the return value is ((width of *x*) - 1).

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__rev, __revl, __revll

Signatures

```
uint32_t        __rev(uint32_t x);
unsigned long __revl(unsigned long x);
uint64_t        __revll(uint64_t x);
```

Description

Reverse the byte order within the argument *x*; return the result.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__rev16, __rev16l, __rev16ll

Signatures

```
uint32_t        __rev16(uint32_t x);
unsigned long __rev16l(unsigned long x);
uint64_t        __rev16ll(uint64_t x);
```

Description

Reverse the byte order within each 16-bit half-word of the input argument *x*; return the result.
For example,

```
__rev16(0x12345678) -> 0x34127856
__rev16ll(0x1122334455667788) -> 0x2211443366558877
```

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__revsh

Signatures

```
uint16_t __revsh(uint16_t x);
```

Description

Reverse the byte order within the 16-bit input argument *x*; return the signed result.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__rbit, __rbitl, __rbitll

Signatures

```
uint32_t        __rbit(uint32_t x);
unsigned long __rbitl(unsigned long x);
uint64_t        __rbitll(uint64_t x);
```

Description

Reverse the bits within the input argument *x*; return the result.

Availability

Cortex-M0/M0+/R4/R5

16-Bit Multiplication Intrinsics

__smulbb

Signatures

```
int32_t __smulbb(int32_t x, int32_t y);
```

Description

```
result = (x & 0xffff) * (y & 0xffff)
```

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smulbt

Signatures

```
int32_t __smulbt(int32_t x, int32_t y);
```

Description

```
result = (x & 0xffff) * (y >> 16)
```

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smultb

Signatures

```
int32_t __smultb(int32_t x, int32_t y);
```

Description

```
result = (x >> 16) * (y & 0xffff)
```

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smultt

Signatures

```
int32_t __smultt(int32_t x, int32_t y);
```

Description

```
result = (x >> 16) * (y >> 16)
```

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smulwb

Signatures

```
int32_t __smulwb(int32_t x, int32_t y);
```

Description

```
result = (x * (int16_t)(y & 0xffff)) >> 16
```

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smulwt

Signatures

```
int32_t __smulwt(int32_t x, int32_t y);
```

Description

```
result = (x * (int16_t)(y >> 16)) >> 16
```

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

Saturating Intrinsics

__ssat

Signatures

```
int32_t __smlawb(int32_t x, int32_t y, int32_t z);
```

Description Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__usat

Signatures

```
int32_t __smlawb(int32_t x, int32_t y, int32_t z);
```

Description Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__qadd

Signatures

```
int32_t __qadd(int32_t x, int32_t y);
```

Description

Add *x* and *y* with saturation; return result and set Q bit if the addition saturates.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__qsub

Signatures

```
int32_t __qsub(int32_t x, int32_t y);
```

Description

Subtract *y* from *x* with saturation; return result and set Q bit if the subtraction saturates.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__qdbl

Signatures

```
int32_t __qdbl(int32_t x);
```

Description

Add *x* to itself with saturation; return result if the addition saturates.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smlabb

Signatures

```
int32_t __smlabb(int32_t x, int32_t y, int32_t z);
```

Description

```
result = ((x & 0xffff) * (y & 0xffff)) + z
```

Set Q bit if addition overflows.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smlabt

Signatures

```
int32_t __smlabt(int32_t x, int32_t y, int32_t z);
```

Description

```
result = ((x & 0xffff) * (y >> 16)) + z
```

Set Q bit if addition overflows.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smlatb

Signatures

```
int32_t __smlatb(int32_t x, int32_t y, int32_t z);
```

Description

```
result = ((x >> 16) * (y & 0xffff)) + z
```

Set Q bit if addition overflows.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smlatt

Signatures

```
int32_t __smlatt(int32_t x, int32_t y, int32_t z);
```

Description

```
result = ((x >> 16) * (y >> 16)) + z
```

Set Q bit if addition overflows.

Availability

3.2. C/C++ Language Implementation   **326**

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smlawb

Signatures

```
int32_t __smlawb(int32_t x, int32_t y, int32_t z);
```

Description

```
result = (x * (int16_t)(y & 0xffff)) + z
```

Set Q bit if addition overflows.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

__smlawt

Signatures

```
int32_t __smlawt(int32_t x, int32_t y, int32_t z);
```

Description

```
result = (x * (int16_t)(y >> 16)) + z
```

Set Q bit if addition overflows.

Availability

Cortex-M0/M0+/M3/M4/M33/R4/R5

32-Bit SIMD Intrinsics

Note: Data Types for 32-Bit SIMD Intrinsics

The *<arm_acle.h>* header file defines the following types:

```
typedef int32_t int8x4_t;
typedef int32_t int16x2_t;
typedef uint32_t uint8x4_t;
typedef uint32_t uint16x2_t;
```

These are used in the descriptions below to indicate that a 32-bit term in an arithmetic operation is being broken into 2 16-bit half-words or 4 8-bit bytes. In the intrinsic descriptions below, terms will refer explicitly to the bits from the 32-bit input argument or return value. For example,

---

```
result[16..31] = x[16..31] + (int16_t)y[16..23]
result[0..15] = x[0..15] + (int16_t)y[0..7]
```

indicates that the top 16-bits of *x* are added to the *y[2]* byte and the result is placed in the top 16-bits of the return value, while the bottom 16-bits of *x* are added to the *y[0]* byte and the result is placed in the bottom 16-bits of the return value (see the description of the *__sctab16* intrinsic below).

__ssat16

Signatures

```
int16x2_t __ssat16(int16x2_t x, unsigned int y);
```

Description

Saturate *x[16..23]* and *x[0..15]* to a bit width, *y*, in the range [1..16]. The Q bit is set if either operation saturates.

Availability

Cortex-M4/M33/R4/R5

__usat16

Signatures

```
int16x2_t __usat16(int16x2_t x, unsigned int y);
```

Description

Saturate *x[16..23]* and *x[0..15]* to a bit width, *y*, in the range [0..15]. The input values are signed and the output values are non-negative, with all negative inputs going to zero. The Q bit is set if either operation saturates.

Availability

Cortex-M4/M33/R4/R5

__sxtab16

Signatures

```
int16x2_t __sxtab16(int16x2_t x, int8x4_t y);
```

Description

```
result[16..31] = x[16..31] + (int16_t)y[16..23]
result[0..15] = x[0..15] + (int16_t)x[0..7]
```

Availability

Cortex-M4/M33/R4/R5

__sxtb16

Signatures

```
int16x2_t __sxtb16(int8x4_t x);
```

Description

```
result[16..31] = (int16_t)x[16..23]
result[0..15] = (int16_t)x[0..7]
```

Availability

Cortex-M4/M33/R4/R5

__uxtab16

Signatures

```
uint16x2_t __uxtab16(uint16x2_t x, uint8x4_t y);
```

Description

```
result[16..31] = x[16..31] + (uint16_t)y[16..23]
result[0..15] = x[0..15] + (uint16_t)y[0..7]
```

Availability

Cortex-M4/M33/R4/R5

__uxtb16

Signatures

```
uint16x2_t __uxtb16(uint8x4_t x);
```

Description

```
result[16..31] = (uint16_t)x[16..23]
result[0..15] = (uint16_t)x[0..7]
```

Availability

Cortex-M4/M33/R4/R5

__sel

Signatures

```
uint8x4_t  __sel(uint8x4_t x, uint8x4_t y);
uint16x2_t __sel(uint16x2_t x, uint16x2_t y);
```

Description

Selects each byte/half-word of the result from either *x* or *y* based on the values of the GE
bits. For each byte, if the corresponding GE bit is set, then the byte from the first operand
(*x*) is selected. If the corresponding GE bit is clear, then the byte from the second operand
(*y*) is selected.

In the case of half-word operations, two GE bits are set/cleared per half-word operation, so
the *__sel** intrinsic can be used for either uint8x4_t or uint16x2_t type data.

Availability

Cortex-M4/M33/R4/R5

__qadd8

Signatures

```
int8x4_t __qadd8(int8x4_t x, int8x4_t y);
```

Description

```
result[24..31] = x[24..31] + y[24..31]
result[16..23] = x[16..23] + y[16..23]
result[8..15] = x[8..15] + y[8..15]
result[0..7] = x[0..7] + y[0..7]
```

Each signed addition result saturates to the range [-128..127].

Availability

Cortex-M4/M33/R4/R5

__qsub8

Signatures

```
int8x4_t __qsub8(int8x4_t x, int8x4_t y);
```

Description

```
result[24..31] = x[24..31] - y[24..31]
result[16..23] = x[16..23] - y[16..23]
result[8..15] = x[8..15] - y[8..15]
result[0..7] = x[0..7] - y[0..7]
```

Each signed subtraction result saturates to the range [-128..127].

Availability

Cortex-M4/M33/R4/R5

__sadd8

Signatures

```
int8x4_t __sadd8(int8x4_t x, int8x4_t y);
```

Description

```
result[24..31] = x[24..31] + y[24..31]
result[16..23] = x[16..23] + y[16..23]
result[8..15] = x[8..15] + y[8..15]
result[0..7] = x[0..7] + y[0..7]
```

Set GE bits according to the results.

Availability

Cortex-M4/M33/R4/R5

__shadd8

Signatures

```
int8x4_t __shadd8(int8x4_t x, int8x4_t y);
```

Description

```
result[24..31] = (x[24..31] + y[24..31]) / 2
result[16..23] = (x[16..23] + y[16..23]) / 2
result[8..15] = (x[8..15] + y[8..15]) / 2
result[0..7] = (x[0..7] + y[0..7]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__shsub8

Signatures

```
int8x4_t __shsub8(int8x4_t x, int8x4_t y);
```

Description

```
result[24..31] = (x[24..31] - y[24..31]) / 2
result[16..23] = (x[16..23] - y[16..23]) / 2
result[8..15] = (x[8..15] - y[8..15]) / 2
result[0..7] = (x[0..7] - y[0..7]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__ssub8

Signatures

```
int8x4_t __ssub8(int8x4_t x, int8x4_t y);
```

Description

```
result[24..31] = x[24..31] - y[24..31]
result[16..23] = x[16..23] - y[16..23]
result[8..15] = x[8..15] - y[8..15]
result[0..7] = x[0..7] - y[0..7]
```

Set GE bits according to the results.

Availability

Cortex-M4/M33/R4/R5

__uadd8

Signatures

```
uint8x4_t __uadd8(uint8x4_t x, uint8x4_t y);
```

Description

```
result[24..31] = x[24..31] + y[24..31]
result[16..23] = x[16..23] + y[16..23]
result[8..15] = x[8..15] + y[8..15]
result[0..7] = x[0..7] + y[0..7]
```

Set GE bits according to the results.

Availability

Cortex-M4/M33/R4/R5

__uhadd8

Signatures

```
uint8x4_t __uhadd8(uint8x4_t x, uint8x4_t y);
```

Description

```
result[24..31] = (x[24..31] + y[24..31]) / 2
result[16..23] = (x[16..23] + y[16..23]) / 2
```

```
result[8..15] = (x[8..15] + y[8..15]) / 2
result[0..7] = (x[0..7] + y[0..7]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__uhsub8

Signatures

```
uint8x4_t __uhsub8(uint8x4_t x, uint8x4_t y);
```

Description

```
result[24..31] = (x[24..31] - y[24..31]) / 2
result[16..23] = (x[16..23] - y[16..23]) / 2
result[8..15] = (x[8..15] - y[8..15]) / 2
result[0..7] = (x[0..7] - y[0..7]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__uqadd8

Signatures

```
uint8x4_t __uqadd8(uint8x4_t x, uint8x4_t y);
```

Description

```
result[24..31] = x[24..31] + y[24..31]
result[16..23] = x[16..23] + y[16..23]
result[8..15] = x[8..15] + y[8..15]
result[0..7] = x[0..7] + y[0..7]
```

Result of each unsigned addition saturates to the range of [0..0xff].

Availability

Cortex-M4/M33/R4/R5

__uqsub8

Signatures

```
uint8x4_t __uqsub8(uint8x4_t x, uint8x4_t y);
```

Description

```
result[24..31] = x[24..31] - y[24..31]
result[16..23] = x[16..23] - y[16..23]
result[8..15] = x[8..15] - y[8..15]
result[0..7] = x[0..7] - y[0..7]
```

Result of each unsigned subtraction saturates to the range of [0..0xff].

Availability

Cortex-M4/M33/R4/R5

__usub8

Signatures

```
uint8x4_t __usub8(uint8x4_t x, uint8x4_t y);
```

Description

```
result[24..31] = x[24..31] - y[24..31]
result[16..23] = x[16..23] - y[16..23]
result[8..15] = x[8..15] - y[8..15]
result[0..7] = x[0..7] - y[0..7]
```

Set GE bits according to the results.

Availability

Cortex-M4/M33/R4/R5

__usad8

Signatures

```
uint32_t __usad8(uint8x4_t x, uint8x4_t y);
```

Description

```
result = abs(x[24..31] - y[24..31]) +
         abs(x[16..23] - y[16..23]) +
         abs(x[8..15] - y[8..15]) +
         abs(x[0..7] - y[0..7])
```

Availability

Cortex-M4/M33/R4/R5

__usada8

Signatures

```
uint32_t __usada8(uint8x4_t x, uint8x4_t y, uint32_t z);
```

Description

```
result = abs(x[24..31] - y[24..31]) +
         abs(x[16..23] - y[16..23]) +
         abs(x[8..15] - y[8..15]) +
         abs(x[0..7] - y[0..7]) + z
```

Availability

Cortex-M4/M33/R4/R5

__qadd16

Signatures

```
int16x2_t __qadd16(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = x[16..31] + y[16..31]
result[0..15] = x[0..15] + y[0..15]
```

Result of each addition saturates to the range [0..0xffff}.

Availability

Cortex-M4/M33/R4/R5

__qasx

Signatures

```
int16x2_t __qasx(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = x[16..31] + y[0..15]
result[0..15] = x[0..15] - y[16..31]
```

Result of each operation saturates to the range [0..0xffff}.

Availability

Cortex-M4/M33/R4/R5

__qsax

Signatures

```
int16x2_t __qsax(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = x[16..31] - y[0..15]
result[0..15] = x[0..15] + y[16..31]
```

Result of each operation saturates to the range [0..0xffff}.

Availability

Cortex-M4/M33/R4/R5

__qsub16

Signatures

```
int16x2_t __qsub16(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = x[16..31] - y[16..31]
result[0..15] = x[0..15] - y[0..15]
```

Result of each subtraction saturates to the range [0..0xffff}.

Availability

Cortex-M4/M33/R4/R5

__sadd16

Signatures

```
int16x2_t __sadd16(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = x[16..31] + y[16..31]
result[0..15] = x[0..15] + y[0..15]
```

Set GE bits according to the results.

Availability

Cortex-M4/M33/R4/R5

__sasx

Signatures

```
int16x2_t __sasx(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = x[16..31] + y[0..15]
result[0..15] = x[0..15] - y[16..31]
```

Set GE bits according to the results.

Availability

Cortex-M4/M33/R4/R5

__shadd16

Signatures

```
int16x2_t __shadd16(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = (x[16..31] + y[16..31]) / 2
result[0..15] = (x[0..15] + y[0..15]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__shasx

Signatures

```
int16x2_t __shasx(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = (x[16..31] + y[0..15]) / 2
result[0..15] = (x[0..15] - y[16..31]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__shsax

Signatures

```
int16x2_t __shsax(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = (x[16..31] - y[0..15]) / 2
result[0..15] = (x[0..15] + y[16..31]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__shsub16

Signatures

```
int16x2_t __shsub16(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = (x[16..31] - y[16..31]) / 2
result[0..15] = (x[0..15] - y[0..15]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__ssax

Signatures

```
int16x2_t __ssax(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = x[16..31] - y[0..15]
result[0..15] = x[0..15] + y[16..31]
```

The GE bits are set according to the results.

Availability

Cortex-M4/M33/R4/R5

__ssub16

Signatures

```
int16x2_t __ssub16(int16x2_t x, int16x2_t y);
```

Description

```
result[16..31] = x[16..31] - y[16..31]
result[0..15] = x[0..15] - y[0..15]
```

The GE bits are set according to the results.

3.2. C/C++ Language Implementation   **338**

Availability

Cortex-M4/M33/R4/R5

__uadd16

Signatures

```
uint16x2_t __uadd16(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = x[16..31] + y[16..31]
result[0..15] = x[0..15] + y[0..15]
```

The GE bits are set according to the results.

Availability

Cortex-M4/M33/R4/R5

__uasx

Signatures

```
uint16x2_t __uasx(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = x[16..31] + y[0..15]
result[0..15] = x[0..15] - y[16..31]
```

Set GE bits according to the result of the unsigned addition.

Availability

Cortex-M4/M33/R4/R5

__uhadd16

Signatures

```
uint16x2_t __uhadd16(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = (x[16..31] + y[16..31]) / 2
result[0..15] = (x[0..15] + y[0..15]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__uhasx

Signatures

```
uint16x2_t __uhasx(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = (x[16..31] + y[0..15]) / 2
result[0..15] = (x[0..15] - y[16..31]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__uhsax

Signatures

```
uint16x2_t __uhsax(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = (x[16..31] - y[0..15]) / 2
result[0..15] = (x[0..15] + y[16..31]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__uhsub16

Signatures

```
uint16x2_t __uhsub16(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = (x[16..31] - y[16..31]) / 2
result[0..15] = (x[0..15] - y[0..15]) / 2
```

Availability

Cortex-M4/M33/R4/R5

__uqadd16

Signatures

```
uint16x2_t __uqadd16(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = x[16..31] + y[16..31]
result[0..15] = x[0..15] + y[0..15]
```

Result of each addition saturates to the range [0..0xffff}.

Availability

Cortex-M4/M33/R4/R5

__uqasx

Signatures

```
uint16x2_t __uqasx(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = x[16..31] + y[0..15]
result[0..15] = x[0..15] - y[16..31]
```

Result of each operation saturates to the range [0..0xffff}.

Availability

Cortex-M4/M33/R4/R5

__uqsax

Signatures

```
uint16x2_t __uqsax(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = x[16..31] - y[0..15]
result[0..15] = x[0..15] + y[16..31]
```

Result of each operation saturates to the range [0..0xffff}.

Availability

Cortex-M4/M33/R4/R5

__uqsub16

Signatures

```
uint16x2_t __uqsub16(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = x[16..31] - y[16..31]
result[0..15] = x[0..15] - y[0..15]
```

Result of each subtraction saturates to the range [0..0xffff}.

Availability

Cortex-M4/M33/R4/R5

__usax

Signatures

```
uint16x2_t __usax(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = x[16..31] - y[0..15]
result[0..15] = x[0..15] + y[16..31]
```

Set GE bits according to the result of unsigned addition.

Availability

Cortex-M4/M33/R4/R5

__usub16

Signatures

```
uint16x2_t __usub16(uint16x2_t x, uint16x2_t y);
```

Description

```
result[16..31] = x[16..31] - y[16..31]
result[0..15] = x[0..15] - y[0..15]
```

Set GE bits according to the results.

Availability

Cortex-M4/M33/R4/R5

__smlad

Signatures

```
int32_t __smlad(int16x2_t x, int16x2_t y, int32_t z);
```

Description

```
result = (x[0..15] * y[0..15]) + (x[16..31] * y[16..31]) + z
```

Set Q bit if addition overflows.

Availability

Cortex-M4/M33/R4/R5

__smladx

Signatures

```
int32_t __smladx(int16x2_t x, int16x2_t y, int32_t z);
```

Description

```
result = (x[0..15] * y[16..31]) + (x[16..31] * y[0..15]) + z
```

Set Q bit if addition overflows.

Availability

Cortex-M4/M33/R4/R5

__smlald

Signatures

```
int64_t __smlald(int16x2_t x, int16x2_t y, int64_t z);
```

Description

```
result = (x[0..15] * y[0..15]) + (x[16..31] * y[16..31]) + z
```

Availability

Cortex-M4/M33/R4/R5

__smlaldx

Signatures

```
int64_t __smlaldx(int16x2_t x, int16x2_t y, int64_t z);
```

Description

```
result = (x[0..15] * y[16..31]) + (x[16..31] * y[0..15]) + z
```

Availability

Cortex-M4/M33/R4/R5

__smlsd

Signatures

```
int32_t __smlsd(int16x2_t x, int16x2_t y, int32_t z);
```

Description

```
result = (x[0..15] * y[0..15]) - (x[16..31] * y[16..31]) + z
```

Set Q bit if the addition overflows.

Availability

Cortex-M4/M33/R4/R5

__smlsdx

Signatures

```
int32_t __smlsdx(int16x2_t x, int16x2_t y, int32_t z);
```

Description

```
result = (x[0..15] * y[16..31]) - (x[16..31] * y[0..15]) + z
```

Set Q bit if the addition overflows.

Availability

Cortex-M4/M33/R4/R5

__smlsld

Signatures

```
int64_t __smlsld(int16x2_t x, int16x2_t y, int64_t z);
```

Description

```
result = (x[0..15] * y[0..15]) - (x[16..31] * y[16..31]) + z
```

Availability

Cortex-M4/M33/R4/R5

__smlsldx

Signatures

```
int64_t __smlsldx(int16x2_t x, int16x2_t y, int64_t z);
```

Description

3.2. C/C++ Language Implementation

```
result = (x[0..15] * y[16..31]) - (x[16..31] * y[0..15]) + z
```

Availability

Cortex-M4/M33/R4/R5

__smuad

Signatures

```
int32_t __smuad(int16x2_t x, int16x2_t y);
```

Description

```
result = (x[0..15] * y[0..15]) + (x[16..31] * y[16..31])
```

Set Q bit if the addition overflows.

Availability

Cortex-M4/M33/R4/R5

__smuadx

Signatures

```
int32_t __smuadx(int16x2_t x, int16x2_t y);
```

Description

```
result = (x[0..15] * y[16..31]) + (x[16..31] * y[0..15])
```

Set Q bit if the addition overflows.

Availability

Cortex-M4/M33/R4/R5

__smusd

Signatures

```
int32_t __smusd(int16x2_t x, int16x2_t y);
```

Description

```
result = (x[0..15] * y[0..15]) - (x[16..31] * y[16..31])
```

Availability

Cortex-M4/M33/R4/R5

`__smusdx`

Signatures

```
int32_t __smusdx(int16x2_t x, int16x2_t y);
```

Description

```
result = (x[0..15] * y[16..31]) - (x[16..31] * y[0..15])
```

Availability

Cortex-M4/M33/R4/R5

Floating-Point Data-Processing Intrinsics

`__sqrt, __sqrtf`

Signatures

```
double __sqrt(double x);
float  __sqrtf(float x);
```

Description

```
result = sqrt(x)
```

If $x < 0$, result will be a default NaN value (with possible floating-point exception thrown).

Availability

- Cortex-M4 w/ FPv4 (-mfpu=fpv4-sp-d16)

- Cortex-M33 w/ FPv5 (-mfpu=fpv5-sp-d16)

- Cortex-R4/R5 w/ VFPv3 (-mfpu=vfpv3-d16)

`__fma, __fmaf`

Signatures

```
double __fma(double x, double y, double z);
float  __fmaf(float x, float y, float z);
```

Description

```
result = (x * y) + z
```

Availability

- Cortex-M4 w/ FPv4 (-mfpu=fpv4-sp-d16)

- Cortex-M33 w/ FPv5 (-mfpu=fpv5-sp-d16)

- Cortex-R4/R5 w/ VFPv3 (-mfpu=vfpv3-d16)

__rintnf, __rintn

Signatures

```
double __rintn (double);
float  __rintnf (float);
```

Description

Convert double/float type *x* to integer. The return type of these intrinsics are double/float and should be cast to integer when assigning to an integer typedata object. For example,

```
int my_int = (int)__rintn(1.25);
```

Availability

- Cortex-M4 w/ FPv4 (-mfpu=fpv4-sp-d16)

- Cortex-M33 w/ FPv5 (-mfpu=fpv5-sp-d16)

- Cortex-R4/R5 w/ VFPv3 (-mfpu=vfpv3-d16)

Custom Datapath Extension (CDE) Intrinsics

Some TI Arm Cortex-M33 devices may be equipped with a coprocessor that can execute custom datapath extension (CDE) instructions via the use of CDE intrinsics.

To enable the use of CDE intrinsics in your C/C++ source file,

1. You should include the *arm_cde.h* header file in your compilation unit prior to any references to a CDE intrinsic. For example,

   ```
   #include <arm_cde.h>

   void foo(void) {
     ...
     uint32_t = __arm_cx2a(10, 20, 30, 40);
     ...
   }
   ```

2. You must specify one of the following *-march* compiler options on the tiarmclang command-line:

   ```
   -march=thumbv8.1-m.main+cdecp0
   ```

   or

```
-march=armv8.1-m.main+cdecp0
```

Note: Selecting a Coprocessor for CDE

The tiarmclang compiler will parse and encode CDE intrinsic instructions for any coprocessor numbered 0 through 7. However, there are currently no TI Arm Cortex-M33 devices that support CDE intrinsic instructions on coprocessors other than coprocessor 0.

This is why "cdecp0" is used in the *-march* options specified above.

The available CDE intrinsics include the following:

- uint32_t __arm_cxl(int, uint32_t);
- uint32_t __arm_cx1a(int, uint32_t, uint32_t);
- uint64_t __arm_cx1d(int, uint32_t);
- uint64_t __arm_cx1da(int, uint64_t, uint32_t);
- uint32_t __arm_cx2(int, uint32_t, uint32_t);
- uint32_t __arm_cx2a(int, uint32_t, uint32_t, uint32_t);
- uint64_t __arm_cx2d(int, uint32_t, uint32_t);
- uint64_t __arm_cx2da(int, uint64_t, uint32_t, uint32_t);
- uint32_t __arm_cx3(int, uint32_t, uint32_t, uint32_t);
- uint32_t __arm_cx3a(int, uint32_t, uint32_t, uint32_t, uint32_t);
- uint64_t __arm_cx3d(int, uint32_t, uint32_t, uint32_t);
- uint64_t __arm_cx3da(int, uint64_t, uint32_t, uint32_t, uint32_t);
- uint32_t __arm_vcx1_u32(int, uint32_t);
- uint32_t __arm_vcx1a_u32(int, uint32_t, uint32_t);
- uint64_t __arm_vcx1d_u64(int, uint32_t);
- uint64_t __arm_vcx1da_u64(int, uint64_t, uint32_t);
- uint32_t __arm_vcx2_u32(int, uint32_t, uint32_t);
- uint32_t __arm_vcx2a_u32(int, uint32_t, uint32_t, uint32_t);
- uint64_t __arm_vcx2d_u64(int, uint64_t, uint32_t);
- uint64_t __arm_vcx2da_u64(int, uint64_t, uint64_t, uint32_t);
- uint32_t __arm_vcx3_u32(int, uint32_t, uint32_t, uint32_t);

3.2. C/C++ Language Implementation   **348**

- uint32_t __arm_vcx3a_u32(int, uint32_t, uint32_t, uint32_t, uint32_t);

- uint64_t __arm_vcx3d_u64(int, uint64_t, uint64_t, uint32_t);

- uint64_t __arm_vcx3da_u64(int, uint64_t, uint64_t, uint64_t, uint32_t);

Each of the CDE intrinsics is defined in *arm_cde.h* as a static inline function and implemented via a compiler runtime built-in function that is defined in the relevant version of the libclang_rt.builtins.a runtime library, which is included in the tiarmclang 3.0.0.STS compiler tools installation.

3.2.10 Keywords

The tiarmclang compiler supports C and C++ language keywords defined in the relevant language standards. You can find information about these keywords in an up-to-date version of the C and C++ language standards. The C++ reference web site is also a very useful resource for information about elements of the C and C++ programming languages.

const Keyword

The `const` keyword is part of the C standard. The tiarmclang compiler supports this keyword in all language modes (see *C/C++ Language Options*),

This keyword gives you greater optimization and control over allocation for certain data objects. You can apply the `const` qualifier to the definition of any variable or array to ensure that its value is not altered.

Global objects qualified as const are placed in the .rodata section. The linker allocates the .rodata section from ROM or FLASH, which are typically more plentiful than RAM. The const data storage allocation rule has the following exceptions:

- If the object has automatic storage (function scope).

- If the object is a C++ object with a "mutable" member.

- If the object is initialized with a value that is not known at compile time (such as the value of another variable).

In these cases, the storage for the object is the same as if the const keyword were not used.

The placement of the const keyword is important. For example, the first statement below defines a constant pointer p to a modifiable int. The second statement defines a modifiable pointer q to a constant int:

```
int * const p = &x;
const int * q = &x;
```

Using the const keyword, you can define large constant tables and allocate them into system ROM. For example, to allocate a ROM table, you could use the following definition:

```
const int digits[] = {0,1,2,3,4,5,6,7,8,9};
```

inline Keyword

The `inline` keyword is part of the C standard beginning with C99. The tiarmclang compiler supports inlining on a per-function basis in all language modes (see *C/C++ Language Options*). However, if you are using -std=c89, which requires the compiler to follow the C89 standard strictly, use the `__inline` keyword instead.

The compiler inlines a function only if it is legal to do so. Functions are never inlined if the compiler is invoked with the `-O0` option or the `-fno-inline-functions` option.

A function may be inlined even if the function is not declared with the inline keyword. The tiarmclang compiler inlines functions with the `inline` keyword and some library functions when the `-O1` optimization option is used. It performs additional inlining when the `-O2` option is used and aggressive inlining when the `-O3` option is used. A function mayn be inlined even if the compiler is not invoked with any `-O` command-line option. The `-Og` and `-Os` options reduce inlining.

restrict Keyword

The `restrict` keyword is part of the C standard beginning with C99. The tiarmclang compiler supports specifying restricted access to pointers, references, and arrays in all language modes (see *C/C++ Language Options*). However, if you are using -std=c89, which requires the compiler to follow the C89 standard strictly, use the `__restrict` keyword instead.

To help the compiler determine memory dependencies, you can qualify a pointer, reference, or array with the `restrict` keyword. The restrict keyword is a type qualifier that can be applied to pointers, references, and arrays. Its use represents a guarantee by you, the programmer, that within the scope of the pointer declaration the object pointed to can be accessed only by that pointer. Any violation of this guarantee renders the program undefined. This practice helps the compiler optimize certain sections of code because aliasing information can be more easily determined.

The following example uses the restrict keyword to tell the compiler that the function `func1` is never called with the pointers `a` and `b` pointing to objects that overlap in memory. You are promising that accesses through `a` and `b` will never conflict; therefore, a write through one pointer cannot affect a read from any other pointers. The precise semantics of the restrict keyword are described in the 1999 version of the ANSI/ISO C Standard.

```
void func1(int * restrict a, int * restrict b)
{
    /* func1's code here */
}
```

The following example uses the restrict keyword when passing arrays to a function. Here, the arrays c and d must not overlap, nor may c and d point to the same array.

```
void func2(int c[restrict], int d[restrict])
{
    int i;
    for(i = 0; i < 64; i++)
    {
        c[i] += d[i];
        d[i] += 1;
    }
}
```

volatile Keyword

The `volatile` keyword is part of the C standard. The tiarmclang compiler supports this keyword in all language modes (see *C/C++ Language Options*).

The `volatile` keyword indicates to the compiler that there is something about how the variable is accessed that requires that the compiler not use overly-clever optimization on expressions involving that variable. For example, the variable may also be accessed by an external program, an interrupt, another thread, or a peripheral device.

The compiler eliminates redundant memory accesses whenever possible, using data flow analysis to figure out when it is legal. However, some memory accesses may be special in some way that the compiler cannot see, and in such cases you should use the volatile keyword to prevent the compiler from optimizing away something important. The compiler does not optimize out any accesses to variables declared volatile. The number of volatile reads and writes will be exactly as they appear in the C/C++ code, no more and no less and in the same order.

Any variable that might be modified by something external to the obvious control flow of the program (such as an interrupt service routine) must be declared volatile. This tells the compiler that an interrupt function might modify the value at any time, so the compiler should not perform optimizations which will change the number or order of accesses of that variable. This is the primary purpose of the volatile keyword. In the following example, the loop intends to wait for a location to be read as 0xFF:

```
unsigned int *ctrl;
while (*ctrl !=0xFF);
```

However, in this example, `*ctrl` is a loop-invariant expression, so the loop is optimized down to a single-memory read. To get the desired result, define ctrl as:

```
volatile unsigned int *ctrl;
```

Here the `*ctrl` pointer is intended to reference a hardware location, such as an interrupt flag.

3.2. C/C++ Language Implementation   **351**

The volatile keyword must also be used when accessing memory locations that represent memory-mapped peripheral devices. Such memory locations might change value in ways that the compiler cannot predict. These locations might change if accessed, or when some other memory location is accessed, or when some signal occurs.

Volatile must also be used for local variables in a function that calls setjmp, if the value of the local variables needs to remain valid if a longjmp occurs.

```
#include <stdlib.h>
jmp_buf context;

void function()
{
    volatile int x = 3;
    switch(setjmp(context))
    {
        case 0: setup(); break;
        default:
        {
            /* We only reach here if longjmp occurs. Because x's␣
 ↪lifetime begins before setjmp
            and lasts through longjmp, the C standard requires x␣
 ↪be declared "volatile". */
            printf("x == %d\n", x);
            break;
        }
    }
}
```

3.3 Run-Time Environment

This chapter describes the C/C++ run-time environment. To ensure successful execution of C/C++ programs, it is critical that all run-time code maintain this environment. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface with C/C++ code.

3.3.1 Memory Model

The compiler treats memory as a single linear block that is partitioned into subblocks of code and data. Each subblock of code or data generated by a C program is placed in its own continuous memory space. The compiler assumes that a full 32-bit address space is available in target memory.

Note: The Linker Defines the Memory Map

The linker, not the compiler, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces. For example, you can use the linker to allocate global variables into on-chip RAM or to allocate executable code into external ROM. You can allocate each block of code or data individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although you can access physical memory locations with C/C++ pointer types).

Sections

The compiler produces relocatable blocks of code and data called *sections*. The sections are allocated into memory in a variety of ways to conform to a variety of system configurations. For more information about sections and allocating them, see *Introduction to Object Modules*.

There are two basic types of sections:

- **Initialized sections** contain data or executable code. Initialized sections are usually, but not always, read-only. The C/C++ compiler creates the following initialized sections:

  - The **.binit section** contains linker-generated boot time copy tables. This is a read-only section. For details on BINIT, see *Boot-Time Copy Tables*.

  - The **.cinit section** contains auto-initialization records for global variables. See *Automatic Initialization of Variables* for more information. The *.cinit* section is read-only.

  - The **.pinit section** contains a table of pointers to global constructor functions to be run at system boot time. See *Automatic Initialization of Variables* for more information. The *.pinit* section is read-only.

  - The **.init_array section** contains a table of pointers to global constructor functions for a dynamic shared object. This section is a read-only section.

  - The **.fini_array section** contains a table of pointers to global destructor functions for a dynamic shared object. This section is read-only.

  - The **.ovly section** contains linker-generated copy tables for unions in which different sections have the same run address. See *Linker-Generated Copy Table Sections and*

Symbols for an example of copy tables used in conjunction with a `UNION` in a linker command file. The *.ovly* section is read-only.

- The **.data section** contains initialized non-const global and static variables. This *.data* section is read-write.

- The **.rodata section** contains read-only data, typically string constants and static-scoped objects defined with the C/C++ qualifier `const`. Note that not all static-scoped objects marked with `const` are placed in the .rodata section (see *const Keyword*).

- The **.text section** contains all the executable code. It also contains string literals, switch tables, and compiler-generated constants. This section is usually read-only. Note that some string literals may instead be placed in `.rodata.str` sections.

- The **.TI.crctab section** contains CRC checking tables. This is a read-only section.

• **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at run time to create and store variables. The compiler creates the following uninitialized sections:

- The **.bss section** reserves space for uninitialized global and static variables. Uninitialized variables that are also unused are usually created as common symbols (unless you specify --common=off) instead of being placed in .bss so that they can be excluded from the resulting application.

- The **.stack section** reserves memory for the C/C++ software stack.

- The **.sysmem section** reserves space for dynamic memory allocation. This space is used by dynamic memory allocation routines, such as malloc(), calloc(), realloc(), or new().

The **.ARM.exidx** and **.ARM.extab** sections are generated if C++ exceptions are enabled. These are initialized sections.

The assembler creates the default *.text* section, even if it is empty. You can instruct the compiler to create additional sections by using the section *function* and *variable* attributes.

The linker takes the individual sections from different object files and combines sections that have the same name. The resulting output sections and the appropriate placement in memory for each section are listed in the following table. You can place these output sections anywhere in the address space as needed to meet system requirements.

Summary of Sections and Memory Placement

Section	Type of Memory	Section	Type of Memory
.bss	RAM	.fini_array	ROM or RAM
.cinit	ROM or RAM	.pinit	ROM or RAM
.rodata	ROM or RAM	.stack	RAM
.data	RAM	.sysmem	RAM
.init_array	ROM or RAM	.text	ROM or RAM

You can use the `SECTIONS` directive in the linker command file to customize the section-allocation process. For more information about allocating sections into memory, see *Linker Description*.

C/C++ System Stack

The C/C++ compiler uses a stack to:

- Allocate local variables

- Pass arguments to functions

- Save register contents

The run-time stack grows from the high addresses to the low addresses.

The compiler uses the R13 register to manage this stack. R13 is the *stack pointer* (SP), which points to the next unused location on the stack.

The linker sets the stack size, creates a global symbol, `__TI_STACK_SIZE`, and assigns it a value equal to the stack size in bytes. The default stack size is 2048 bytes. You can change the stack size at link time by using the *--stack_size* option with the **tiarmlnk** command (use *-Wl,* or *-Xlinker* prefix for the linker option if invoking the linker from **tiarmclang**, e.g. *-Wl,--stack_size=256*). For more information on the *--stack_size* option, see *Linker Description*.

At system initialization, `SP` is set to a designated address for the top of the stack. This address is the first location past the end of the `.stack` section. Since the position of the stack depends on where the `.stack` section is allocated, the actual address of the stack is determined at link time.

The C/C++ environment automatically decrements `SP` at the entry to a function to reserve all the space necessary for the execution of that function. The stack pointer is incremented at the exit of the function to restore the stack to the state before the function was entered. If you interface assembly language routines to C/C++ programs, be sure to restore the stack pointer to the same state it was in before the function was entered.

For more information about using the stack pointer, see *Register Conventions*; for more information about the stack, see *Function Structure and Calling Conventions*.

To debug issues related to the stack size, we recommend using the CCS Stack Usage view to see the static stack usage of each function in the application. See Stack Usage View in CCS for more information. Using the Stack Usage View requires that source code be built with *debug enabled*. This feature relies on the *--call_graph* capability provided by the *tiarmofd - Object File Display Utility*.

Note: Stack Overflow and Stack Smashing Detection

A stack overflow disrupts the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow. You can use the *-finstrument-functions* option to add code to

---

the beginning of each function to check for stack overflow. See *Function Entry/Exit Hook Options* for more information.

Stack smashing occurs when a given function writes past the stack space that has been allocated for it. You can use the *fstack-protector* option to enable stack smashing detection for your application. See *Stack Smashing Detection Options* for more information.

Dynamic Memory Allocation

The run-time-support library supplied with the compiler contains several functions (such as malloc, calloc, and realloc) that allow you to allocate memory dynamically for variables at run time.

Memory is allocated from a global pool, or heap, that is defined in the .sysmem section. You can set the size of the .sysmem section by using the *--heap_size=<n>* option with the **tiarmlnk** command ((use *-Wl,* or *-Xlinker* prefix for the linker option if invoking the linker from **tiarmclang**, e.g. *-Wl,--heap_size=1024*). The linker also creates a global symbol, __TI_SYSMEM_SIZE, and assigns it a value equal to the size of the heap in bytes. The default size is 2048 bytes. For more information on the *--heap_size* option, see *Linker Description*.

If you use any C I/O function (e.g. printf), the RTS library allocates an I/O buffer for each file you access. This buffer will be a bit larger than BUFSIZ, which is defined in stdio.h and defaults to 256. Make sure you allocate a heap large enough for these buffers or use *setvbuf()* to change the buffer to a statically-allocated buffer.

Dynamically allocated objects are not addressed directly (they are always accessed with pointers) and the memory pool is in a separate section (.sysmem); therefore, the dynamic memory pool can have a size limited only by the amount of available memory in your system. To conserve space in the .bss section, you can allocate large arrays from the heap instead of defining them as global or static. For example, instead of a definition such as:

```
struct big table[100];
```

Use a pointer and call the malloc function:

```
struct big *table;
table = (struct big *)malloc(100*sizeof(struct big));
```

> **Warning:** When allocating from a heap, make sure the size of the heap is large enough for the allocation. This is particularly important when allocating variable-length arrays.

3.3.2 Object Representation

For general information about data types, see *Data Types*. This section explains how various data objects are sized, aligned, and accessed.

Data Type Storage

The following table lists register and memory storage for various data types:

Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char, signed char	Bits 0-7 of register (Note 1 below)	8 bits aligned to 8-bit boundary
unsigned char, bool	Bits 0-7 of register	8 bits aligned to 8-bit boundary
short, signed short	Bits 0-15 of register (Note 1 below)	16 bits aligned to 16-bit (halfword) boundary
unsigned short, wchar_t	Bits 0-15 of register	16 bits aligned to 16-bit (halfword) boundary
int, signed int	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
unsigned int	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
long, signed long	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
unsigned long	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
long long	Even/odd register pair	64 bits aligned to 64-bit boundary (Note 2 below)
unsigned long long	Even/odd register pair	64 bits aligned to 64-bit boundary (Note 2 below)
float	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
double	Register pair	64 bits aligned to 64-bit boundary (Note 2 below)
long double	Register pair	64 bits aligned to 64-bit boundary (Note 2 below)
struct	Members are stored as their individual types require.	Members are stored as their individual types require; aligned according to the member with the most restrictive alignment requirement.
array	Members are stored as their individual types require.	Members are stored as their individual types require; aligned to 32-bit (word) boundary. All arrays inside a structure are

Note 1) Negative values are sign-extended to bit 31.

Note 2) 64-bit data is aligned on a 64-bit boundary. 64-bit pointers are aligned to a 32-bit boundaries.

For details about the size of an enum type, see *Enum Type Storage*.

char and short Data Types (signed and unsigned)

The char and unsigned char data types are stored in memory as a single byte and are loaded to and stored from bits 0-7 of a register (see the following figure). Objects defined as short or unsigned short are stored in memory as two bytes at a halfword (2 byte) aligned address and they are loaded to and stored from bits 0-15 of a register.

Figure 3.1: Char and Short Data Storage Format

float, int, and long Data Types (signed and unsigned)

The int, unsigned int, float, long and unsigned long data types are stored in memory as 32-bit objects at word (4 byte) aligned addresses. Objects of these types are loaded to and stored from bits 0-31 of a register, as shown in the following figure.

- In big-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 24-31 of the register, moving the second byte of memory to bits 16-23, moving the third byte to bits 8-15, and moving the fourth byte to bits 0-7.

- In little-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0-7 of the register, moving the second byte to bits 8-15,

moving the third byte to bits 16-23, and moving the fourth byte to bits 24-31.

Single-precision floating char

S	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M

31   23   0

MS / LS

Signed 32-bit integer or long char

S	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I

31   0

MS / LS

Unsigned 32-bit integer or long

U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U

31   0

MS / LS

LEGEND: S = sign, M = Mantissa, U = unsigned integer, E = exponent, I = signed integer, MS = most significant, LS = least significant
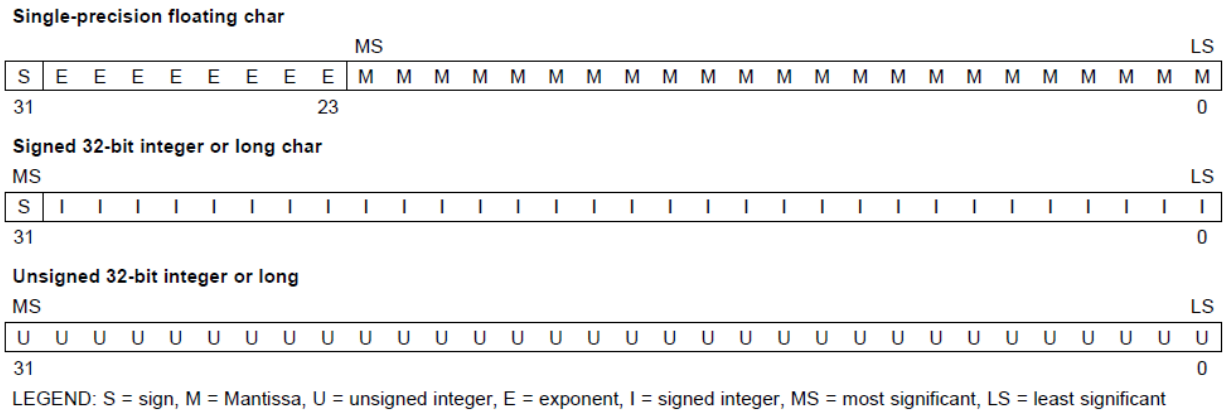
Figure 3.2: 32-Bit Data Storage Format

double, long double, and long long Data Types (signed and unsigned)

Double, long double, long long and unsigned long long data types are stored in memory in a pair of registers and are always referenced as a pair. These types are stored as 64-bit objects at 64-bit aligned addresses. For FPA mode, the word at the lowest address contains the sign bit, the exponent, and the most significant part of the mantissa. The word at the higher address contains the least significant part of the mantissa. This is true regardless of the endianness of the target. For VFP mode, the words are ordered based upon the endianness of the target.

Objects of this type are loaded into and stored in register pairs, as shown in the following figure. The most significant memory word contains the sign bit, exponent, and the most significant part of the mantissa. The least significant memory word contains the least significant part of the mantissa.

Address *x*

S	E	E	E	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M

31   20   0

MS

Address *x*+ 4

M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M

31   0

LS

LEGEND: S = sign, M = mantissa, E = exponent, MS = most significant, LS = least significant
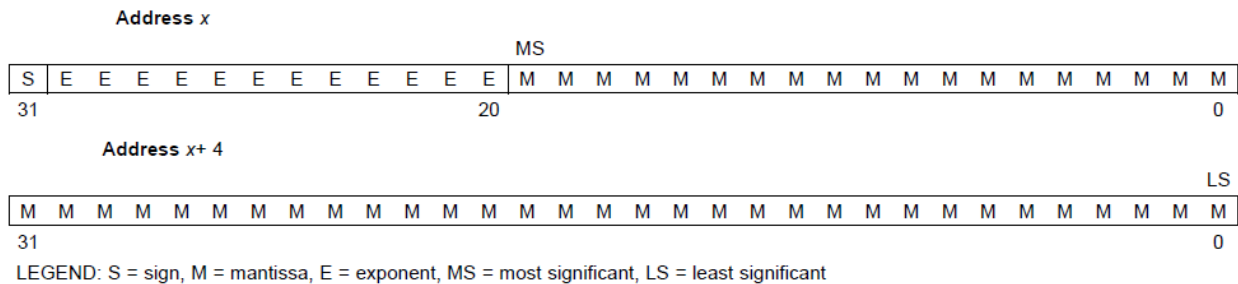
Figure 3.3: Double-Precision Floating-Point Data Storage Format

Pointer to Data Member Types

Pointer to data member objects are stored in memory like an unsigned int (32 bit) integral type. Its value is the byte offset to the data member in the class, plus 1. The zero value is reserved to represent the NULL pointer to the data member.

Pointer to Member Function Types

Pointer to member function objects are stored as a structure with three members, and the layout is equivalent to:

```
struct {
    short int d;
    short int i;
    union {
        void (f) ();
        long 0; }
};
```

The parameter d is the offset to be added to the beginning of the class object for this pointer. The parameter I is the index into the virtual function table, offset by 1. The index enables the NULL pointer to be represented. Its value is -1 if the function is non-virtual. The parameter f is the pointer to the member function if it is non-virtual, when I is 0. The 0 is the offset to the virtual function pointer within the class object.

Structure and Array Alignment

Structures are aligned according to the member with the most restrictive alignment requirement. Structures are padded so that the size of the structure is a multiple of its alignment. Arrays are always word aligned. Elements of arrays are stored in the same manner as if they were individual objects.

Bit Fields

Bit fields are the only objects that are packed within a byte. That is, two bit fields can be stored in the same byte. Bit fields can range in size from 1 to 32 bits, but they never span a 4-byte boundary.

For big-endian mode, bit fields are packed into registers from most significant bit (MSB) to least significant bit (LSB) in the order in which they are defined. Bit fields are packed in memory from most significant byte (MSbyte) to least significant byte (LSbyte). For little-endian mode, bit fields are packed into registers from the LSB to the MSB in the order in which they are defined, and packed in memory from LSbyte to MSbyte.

Here are some details about how bit fields are handled:

- Plain int bit fields are unsigned. Consider the following C code, where bar() is never called, since bit field 'a' is unsigned. Use signed int if you need a signed bit field.

```
struct st
{
    int a:5;
} S;

foo()
{
    if (S.a < 0)
    bar();
}
```

- Bit fields of type long long are supported.

- Bit fields are treated as the declared type.

- The size and alignment of the struct containing the bit field depends on the declared type of the bit field. For example, consider the struct, which uses up 4 bytes and is aligned at 4 bytes:

```
struct st {int a:4};
```

- Unnamed bit fields affect the alignment of the struct or union. For example, consider the struct, which uses 4 bytes and is aligned at a 4-byte boundary:

```
struct st{char a:4; int :22;};
```

- Bit fields declared volatile are accessed according to the bit field's declared type. A volatile bit field reference generates exactly one reference to its storage; multiple volatile bit field accesses are not merged.

The following figure illustrates bit-field packing, using the following bit field definitions:

```
struct{
    int A:7
    int B:10
    int C:3
    int D:2
    int E:9
}x;
```

A0 represents the least significant bit of the field A; A1 represents the next least significant bit, etc. Again, storage of bit fields in memory is done with a byte-by-byte, rather than bit-by-bit, transfer.
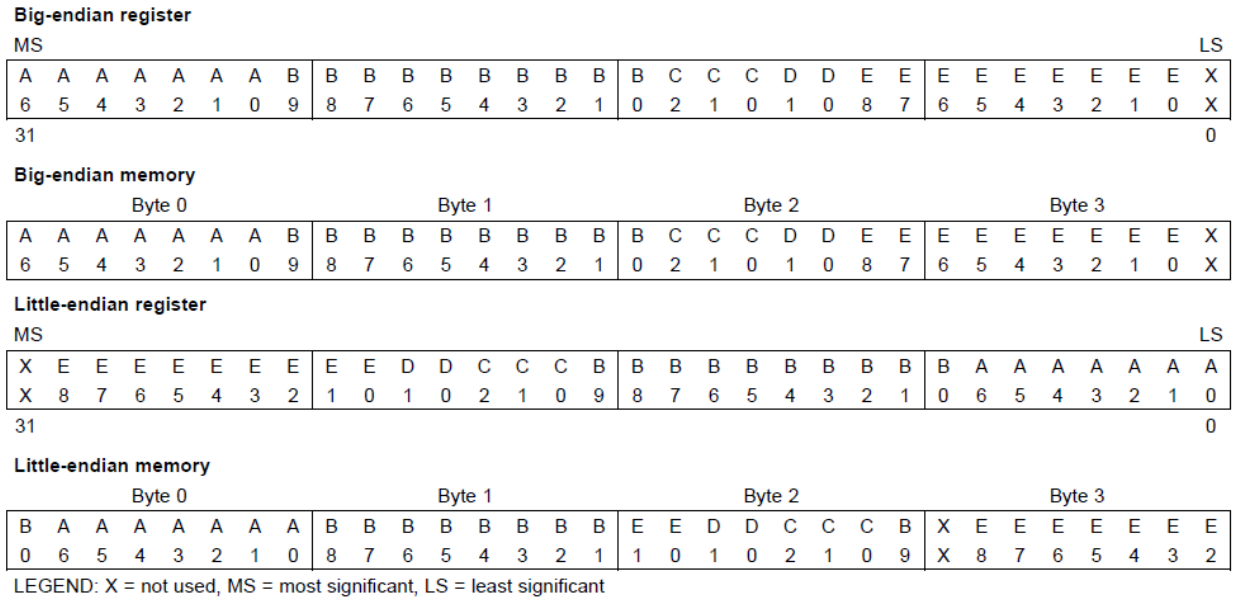
Figure 3.4: Bit-Field Packing in Big-Endian and Little-Endian Formats

Character String Constants

In C, a character string constant is used in one of the following ways:

To initialize an array of characters. For example:

```
char s[] = "abc";
```

When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about initialization, see *System Initialization*.

In an expression. For example:

```
strcpy (s, "abc");
```

When a string is used in an expression, the string itself is defined in the .const section with the .string assembler directive, along with a unique label that points to the string; the terminating 0 byte is included. For example, the following lines define the string abc, and the terminating 0 byte (the label SL5 points to the string):

```
        .sect ".const"
SL5:  .string "abc",0
```

String labels have the form SLn, where n is a number assigned by the compiler to make the label unique. The number begins at 0 and is increased by 1 for each string defined. All strings used in a source module are defined at the end of the compiled assembly language module.

The label SLn represents the address of the string constant. The compiler uses this label to reference the string expression.

Because strings are stored in the .const section (possibly in ROM) and shared, it is bad practice for a program to modify a string constant. The following code is an example of incorrect string use:

```
const char *a = "abc"
a[1] = 'x';              /* Incorrect! undefined behavior */
```

3.3.3 Register Conventions

Strict conventions associate specific registers with specific operations in the C/C++ environment. If you plan to interface an assembly language routine to a C/C++ program, you must understand and follow these register conventions.

The register conventions dictate how the compiler uses registers and how values are preserved across function calls. Table 1 below shows the types of registers affected by these conventions. Table 2 summarizes how the compiler uses registers and whether their values are preserved across calls. Table 3 and 4 provide register usage information for VFP and Neon registers, respectively. For information about how values are preserved across calls, see *Function Structure and Calling Conventions*.

Table 1: How Register Types Are Affected by the Conventions

Register Type	Description
Argument register	Passes arguments during a function call
Return register	Holds the return value from a function call
Expression register	Holds a value
Argument pointer	Used as a base value from which a function's parameters (incoming arguments) are accessed
Stack pointer	Holds the address of the top of the software stack
Link register	Contains the return address of a function call
Program counter	Contains the current address of code being executed

Table 2: Register Usage

Reg- ister	Alias	Usage	Preserved by Function (Note 1 below)
R0	A1	Argument register, return register, expression register	Parent
R1	A2	Argument register, return register, expression register	Parent
R2	A3	Argument register, expression register	Parent
R3	A4	Argument register, expression register	Parent
R4	V1	Expression register	Child
R5	V2	Expression register	Child
R6	V3	Expression register	Child
R7	V4, AP	Expression register, argument pointer	Child
R8	V5	Expression register	Child
R9	V6	Expression register	Child
R10	V7	Expression register	Child
R11	V8	Expression register	Child
R12	V9, 1P	Expression register, instruction pointer	Parent
R13	SP	Stack pointer	Child (Note 2 below)
R14	LR	Link register, expression register	Child
R15	PC	Program counter	N/A
CPSR		Current program status register	Child
SPSR		Saved program status register	Child

Note 1) The parent function refers to the function making the function call. The child function refers to the function being called.

Note 2) The SP is preserved by the convention that everything pushed on the stack is popped off before returning.

Table 3: VFP Register Usage

32-Bit Register	64-Bit Register	Usage	Preserved by Function (Note 1 below)
FPSCR		Status register	N/A
S0	D0	Floating-point expression, return values, pass arguments	N/A
S1			
S2	D1	Floating-point expression, return values, pass arguments	N/A

Table 3.3 – continued from previous page

32-Bit Register	64-Bit Register	Usage	Preserved by Function (Note 1 below)
S3			
S4 S5	D2	Floating-point expression, return values, pass arguments	N/A
S6 S7	D3	Floating-point expression, return values, pass arguments	N/A
S8 S9	D4	Floating-point expression, pass arguments	N/A
S10 S11	D5	Floating-point expression, pass arguments	N/A
S12 S13	D6	Floating-point expression, pass arguments	N/A
S14 S15	D7	Floating-point expression, pass arguments	N/A
S16 S17	D8	Floating-point expression	Child
S18 S19	D9	Floating-point expression	Child
S20 S21	D10	Floating-point expression	Child
S22 S23	D11	Floating-point expression	Child
S24 S25	D12	Floating-point expression	Child
S26 S27	D13	Floating-point expression	Child
S28 S29	D14	Floating-point expression	Child
S30 S31	D15	Floating-point expression	Child
	D16-D31	Floating-point expression	

Note 1) The child function refers to the function being called.

Table 4: Neon Register Usage

64-Bit Register	Quad Register	Usage	Preserved by Function (Note 1 below)
D0	Q0	SIMD register	N/A
D1			
D2	Q1	SIMD register	N/A
D3			
D4	Q2	SIMD register	N/A
D5			
D6	Q3	SIMD register	N/A
D7			
D8	Q4	SIMD register	Child
D9			
D10	Q5	SIMD register	Child
D11			
D12	Q6	SIMD register	Child
D13			
D14	Q7	SIMD register	Child
D15			
D16	Q8	SIMD register	N/A
D17			
D18	Q9	SIMD register	N/A
D19			
D20	Q10	SIMD register	N/A
D21			
D22	Q11	SIMD register	N/A
D23			
D24	Q12	SIMD register	N/A
D25			
D26	Q13	SIMD register	N/A
D27			
D28	Q14	SIMD register	N/A
D29			
D30	Q15	SIMD register	N/A
D31			
FPSCR		Status register	N/A

Note 1) The child function refers to the function being called.

3.3.4 Function Structure and Calling Conventions

The C/C++ compiler imposes a strict set of rules on function calls. Except for special run-time support functions, any function that calls or is called by a C/C++ function must follow these rules. Failure to adhere to these rules can disrupt the C/C++ environment and cause a program to fail.

The following sections use this terminology to describe the function-calling conventions of the C/C++ compiler:

- **Argument block**. The part of the local frame used to pass arguments to other functions. Arguments are passed to a function by moving them into the argument block rather than pushing them on the stack. The local frame and argument block are allocated at the same time.

- **Register save area**. The part of the local frame that is used to save the registers when the program calls the function and restore them when the program exits the function.

- **Save-on-call registers**. Registers R0-R3 and R12 (alternate names are A1-A4 and V9). The called function does not preserve the values in these registers; therefore, the calling function must save them if their values need to be preserved.

- **Save-on-entry registers**. Registers R4-R11 and R14 (alternate names are V1 to V8 and LR). It is the called function's responsibility to preserve the values in these registers. If the called function modifies these registers, it saves them when it gains control and preserves them when it returns control to the calling function.

The following figure illustrates a typical function call. In this example, arguments are passed to the function, and the function uses local variables and calls another function. The first four arguments are passed to registers R0-R3. This example also shows allocation of a local frame and argument block for the called function. Functions that have no local variables and do not require an argument block do not allocate a local frame.
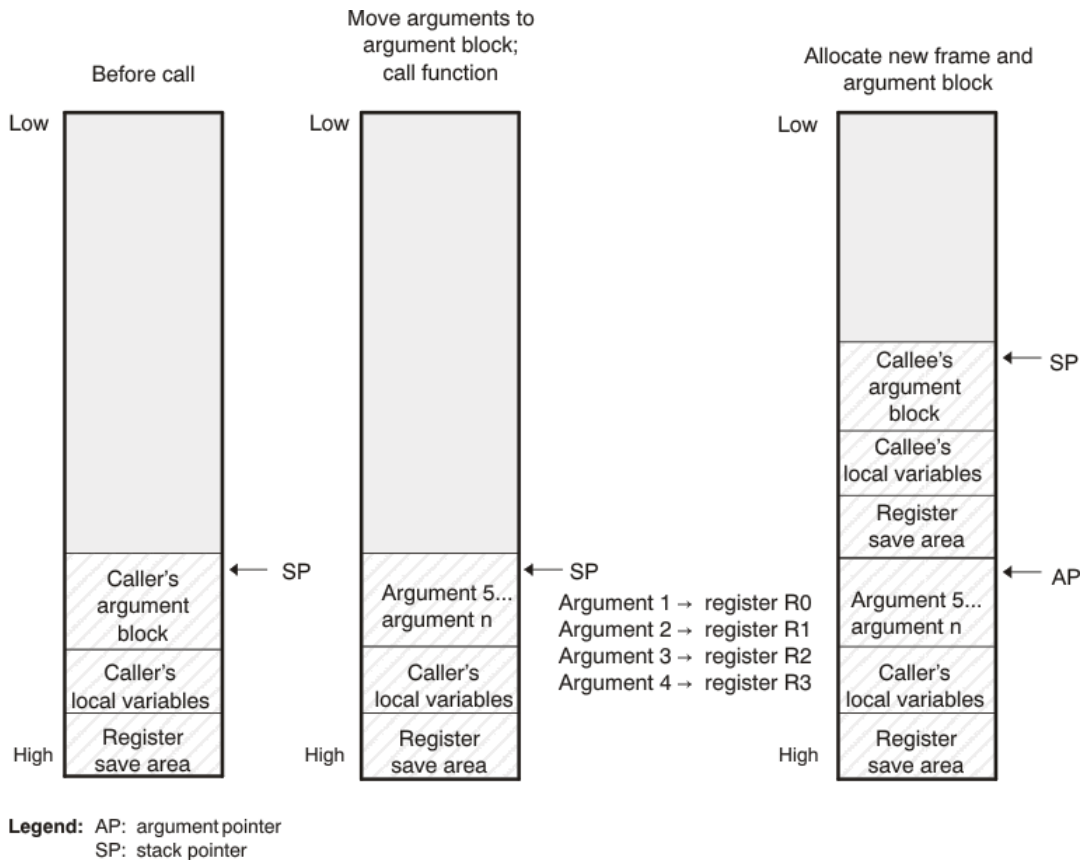
Figure 3.5: Use of the Stack During a Function Call

How a Function Makes a Call

A function (parent function) performs the following tasks when it calls another function (child function).

1. The calling function (parent) is responsible for preserving any save-on-call registers across the call that are live across the call. (The save-on-call registers are R0-R3 and R12 (alternate names are A1-A4 and V9).)

2. If the called function (child) returns a structure, the caller allocates space for the structure and passes the address of that space to the called function as the first argument.

3. The caller places the first arguments in registers R0-R3, in that order. The caller moves the remaining arguments to the argument block in reverse order, placing the leftmost remaining argument at the lowest address. Thus, the leftmost remaining argument is placed at the top of the stack.

4. If arguments were stored onto the argument block in step 3, the caller reserves a word in the argument block for dual-state support.

How a Called Function Responds

A called function (child function) must perform the following tasks:

1. If the function is declared with an ellipsis, it can be called with a variable number of arguments. The called function pushes these arguments on the stack if they meet both of these criteria:

   - The argument includes or follows the last explicitly declared argument.

   - The argument is passed in a register.

2. The called function pushes register values of all the registers that are modified by the function and that must be preserved upon exit of the function onto the stack. Normally, these registers are the save-on-entry registers (R4-R11 and R14 (alternate names are V1 to V8 and LR)) and the link register (R14) if the function contains calls. If the function is an interrupt, additional registers may need to be preserved. For more information, see *Interrupt Handling*.

3. The called function allocates memory for the local variables and argument block by subtracting a constant from the SP. This constant is computed with the following formula:

   ```
   size of all local variables + max = constant
   ```

   The max argument specifies the size of all parameters placed in the argument block for each call.

4. The called function executes the code for the function.

5. If the called function returns a value, it places the value in R0 (or R0 and R1 values).

6. If the called function returns a structure, it copies the structure to the memory block that the first argument, R0, points to. If the caller does not use the return value, R0 is set to 0. This directs the called function not to copy the return structure.

   Structures and unions are always passed by reference.

   In this way, the caller can be smart about telling the called function where to return the structure. For example, in the statement s = f(x), where s is a structure and f is a function that returns a structure, the caller can simply pass the address of s as the first argument and call f. The function f then copies the return structure directly into s, performing the assignment automatically.

   You must be careful to properly declare functions that return structures, both at the point where they are called (so the caller properly sets up the first argument) and at the point where they are declared (so the function knows to copy the result).

7. The called function deallocates the frame and argument block by adding the constant computed in Step 3.

8. The called function restores all registers that were saved in Step 2.

9. The called function (_f) loads the program counter (PC) with the return address.

The following example is typical of how a called function responds to a call:

```
                                  ; called function entry point
STMFD SP!, {V1, V2, V3, LR} ; save V1, V2, V3, and LR
SUB SP, SP, #16             ; allocate frame
...                         ; body of the function
ADD SP, SP, #16             ; deallocate frame
LDMFD SP!, {V1, V2, V3, PC} ; restore V1, V2, V3, and store LR
                            ; in the PC, causing a return
```

C Exception Handler Calling Convention

The C library provides no built-in support for C exception handling.

If a C exception handler calls other functions, the following must take place:

- The handler must set its own stack pointer.
- The handler saves all of the registers not preserved by the call: R0-R3, R-12, LR (R8-R12 saved by hardware for FIQ)
- Re-entrant exception handlers must save SPSR and LR.

See *C++ Exception Handling* for information about C++ exception handling.

Accessing Arguments and Local Variables

A function accesses its local nonregister variables indirectly through the stack pointer (SP or R13) and its stack arguments indirectly through the argument pointer (AP). If all stack arguments can be referenced with the SP, they are, and the AP is not reserved. The SP always points to the top of the stack (the most recently pushed value) and the AP points to the leftmost stack argument (the one closest to the top of the stack). For example:

```
LDR  A2 [SP, #4]                  ; load local var from stack
LDR  A1 [AP, #0]                  ; load argument from stack
```

Since the stack grows toward smaller addresses, the local and argument data on the stack for the C/C++ function is accessed with a positive offset from the SP or the AP register.

3.3.5 Accessing Linker Symbols in C and C++

See *Using Linker Symbols in C/C++ Applications* for information about referring to linker symbols in C/C++ code.

3.3.6 Interfacing C and C++ With Assembly Language

The following are ways to use assembly language with C/C++ code:

- Use separate modules of assembled code and link them with compiled C/C++ modules (see *Using Assembly Language Modules With C/C++ Code*).

- Use assembly language variables and constants in C/C++ source (see *Accessing Assembly Language Variables From C/C++*).

- Use inline assembly language embedded directly in the C/C++ source (see *Using Inline Assembly Language*).

- Modify the assembly language code that the compiler produces (see *Modifying Compiler Output*).

Using Assembly Language Modules With C/C++ Code

Interfacing C/C++ with assembly language functions is straightforward if you follow the calling conventions defined in *Function Structure and Calling Conventions*, and the register conventions defined in *Register Conventions*.

C/C++ code can access variables and call functions defined in assembly language, and assembly code can access C/C++ variables and call C/C++ functions.

Follow these guidelines to interface assembly language and C:

- You must preserve any dedicated registers modified by a function. Dedicated registers include:

    - Save-on-entry registers (R4-R11 (alternate names are V1 to V8 and LR))

    - Stack pointer (SP or R13)

    If the SP is used normally, it does not need to be explicitly preserved. In other words, the assembly function is free to use the stack as long as anything that is pushed onto the stack is popped back off before the function returns (thus preserving SP).

    Any register that is not dedicated can be used freely without first being saved.

- Interrupt routines must save *all* the registers they use. For more information, see *Interrupt Handling*.

- When you call a C/C++ function from assembly language, load the designated registers with arguments and push the remaining arguments onto the stack as described in *How a Function Makes a Call*.

  Remember that a function can alter any register not designated as being preserved without having to restore it. If the contents of any of these registers must be preserved across the call, you must explicitly save them.

- Functions must return values correctly according to their C/C++ declarations. Double values are returned in R0 and R1, and structures are returned as described in Step 2 of *How a Function Makes a Call*. Any other values are returned in R0.

- No assembly module should use the .cinit section for any purpose other than autoinitialization of global variables. The C/C++ startup routine assumes that the .cinit section consists *entirely* of initialization tables. Disrupting the tables by putting other information in .cinit can cause unpredictable results.

- The compiler assigns linknames to all external objects. Thus, when you write assembly language code, you must use the same linknames as those assigned by the compiler. See *Disable Name Demangling (--no_demangle)* for details.

- Any object or function declared in assembly language that is accessed or called from C/C++ must be declared with the .def or .global directive in the assembly language modifier. This declares the symbol as external and allows the linker to resolve references to it.

Accessing Assembly Language Functions From C/C++

Functions defined in C++ that will be called from assembly should be defined as extern "C" in the C++ file. Functions defined in assembly that will be called from C++ must be prototyped as extern "C" in C++.

Example 1 below illustrates a C++ function called main(), which calls an assembly language function called asmfunc, which is shown in Example 2. The asmfunc function takes its single argument, adds it to the C++ global variable called gvar, and returns the result.

Example 1: Calling an Assembly Language Function From a C/C++ Program

```
extern "C" {
extern int asmfunc(int a); /* declare external asm function */
int gvar = 0; /* define global variable */
}

void main()
{
    int I = 5;
```

(continues on next page)

```
      I = asmfunc(I); /* call function normally */
}
```

Example 2: Assembly Language Program Called by Example 1

```
        .global asmfunc
        .global gvar
asmfunc:
        LDR r1, gvar_a
        LDR r2, [r1, #0]
        ADD r0, r0, r2
        STR r0, [r1, #0]
        MOV pc, lr
gvar_a .field gvar, 32
```

In the C++ program in Example 1, the extern "C" declaration tells the compiler to use C naming conventions (that is, no name mangling). When the linker resolves the .global _asmfunc reference, the corresponding definition in the assembly file will match.

The parameter i is passed in R0, and the result is returned in R0. R1 holds the address of the global gvar. R2 holds the value of gvar before adding the value i to it.

Accessing Assembly Language Variables From C/C++

It is sometimes useful for a C/C++ program to access variables or constants defined in assembly language. There are several methods that you can use to accomplish this, depending on where and how the item is defined: a variable defined in the .bss section, a variable not defined in the .bss section, or a linker symbol.

Accessing Assembly Language Global Variables

Accessing variables from the .bss section or a section named with .usect is straightforward:

1. Use the .bss or .usect directive to define the variable.

2. Use the .def or .global directive to make the definition external.

3. Use the appropriate linkname in assembly language.

4. In C/C++, declare the variable as *extern* and access it normally.

Example 3 and Example 4 show how you can access a variable defined in .bss.

Example 3: Assembly Language Variable Program

```
.bss      var,4,4   ; Define the variable
.global   var       ; Declare the variable as external
```

Example 4: C Program to Access Assembly Language From Example 3

```
extern int var;        /* External variable */
var = 1;               /* Use the variable  */
```

Accessing Assembly Language Constants

You can define global constants in assembly language by using the .set directive in combination with either the .def or .global directive, or you can define them in a linker command file using a linker assignment statement. These constants are accessible from C/C++ only with the use of special operators.

For **variables** defined in C/C++ or assembly language, the symbol table contains the *address of the value* contained by the variable. When you access an assembly variable by name from C/C++, the compiler gets the value using the address in the symbol table.

For **assembly constants**, however, the symbol table contains the actual *value* of the constant. The compiler cannot tell which items in the symbol table are addresses and which are values. If you access an assembly (or linker) constant by name, the compiler tries to use the value in the symbol table as an address to fetch a value. To prevent this behavior, you must use the & (address of) operator to get the value. In other words, if x is an assembly language constant, its value in C/C++ is &x. See *Using Linker Symbols in C/C++ Applications* for more examples.

For more about symbols and the symbol table, refer to *Symbols*.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 5 and Example 6.

Example 5: Accessing an Assembly Language Constant From C

```
extern int table_size; /*external ref */
#define TABLE_SIZE ((int) (&table_size))
    . /* use cast to hide address-of */
    .
    .
for (I=0; i<TABLE_SIZE; ++I) /* use like normal symbol */
```

Example 6: Assembly Language Program for Example 5

```
_table_size   .set10000              ; define the constant
              .global _table_size ; make it global
```

Because you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 5, int is used. You can reference linker-defined symbols in a similar manner.

Sharing C/C++ Header Files With Assembly Source

Sharing C/C++ header files with assembly source is not supported.

Using Inline Assembly Language

Within a C/C++ program, you can use the asm statement to insert a single line of assembly language into the assembly language file created by the compiler. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code. For more information, see *naked*.

The asm statement is useful for inserting comments in the compiler output. Simply start the assembly code string with a semicolon (;) as shown below:

```
asm(";*** this is an assembly language comment");
```

Note: Using the asm Statement Keep the following in mind when using the asm statement:

- Be extremely careful not to disrupt the C/C++ environment. The compiler does not check or analyze the inserted instructions.

- Avoid inserting jumps or labels into C/C++ code because they can produce unpredictable results by confusing the register-tracking algorithms that the code generator uses.

- Do not change the value of a C/C++ variable when using an asm statement. This is because the compiler does not verify such statements. They are inserted as is into the assembly code, and potentially can cause problems if you are not sure of their effect.

- Do not use the asm statement to insert assembler directives that change the assembly environment.

- Avoid creating assembly macros in C code and compiling with the --symdebug:dwarf (or -g) option. The C environment's debug information and the assembly macro expansion are not compatible.

Modifying Compiler Output

You can inspect and change the compiler's assembly language output by compiling the source and then editing the assembly output file before assembling it. Specify the *-S* option on the compiler command line to capture the compiler generated assembly.

3.3.7 Interrupt Handling

As long as you follow the guidelines in this section, you can interrupt and return to C/C++ code without disrupting the C/C++ environment. When the C/C++ environment is initialized, the startup routine does not enable or disable interrupts. If the system is initialized by way of a hardware reset, interrupts are disabled. If your system uses interrupts, you must handle any required enabling or masking of interrupts. Such operations have no effect on the C/C++ environment and are easily incorporated with asm statements or calling an assembly language function.

Saving Registers During Interrupts

When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any functions called by the routine. With the exception of banked registers, register preservation must be explicitly handled by the interrupt routine.

All banked registers are automatically preserved by the hardware (except for interrupts that are reentrant. If you write interrupt routines that are reentrant, you must add code that preserves the interrupt's banked registers.) Each interrupt type has a set of banked registers. For information about interrupt types, see *interrupt*.

Using C/C++ Interrupt Routines

When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any functions called by the routine. Register preservation must be explicitly handled by the interrupt routine.

```
__interrupt void example (void)
{
    ...
}
```

If a C/C++ interrupt routine does not call any other functions, only those registers that the interrupt handler uses are saved and restored. However, if a C/C++ interrupt routine does call other functions, these functions can modify unknown registers that the interrupt handler does not use. For this reason, the routine saves all the save-on-call registers if any other functions are called. (This excludes banked registers.) Do not call interrupt handling functions directly.

Interrupts can be handled directly with C/C++ functions by using the __interrupt keyword. For information, see *interrupt*.

Using Assembly Language Interrupt Routines

You can handle interrupts with assembly language code as long as you follow the same register conventions the compiler does. Like all assembly functions, interrupt routines can use the stack, access global C/C++ variables, and call C/C++ functions normally. When calling C/C++ functions, be sure that any save-on-call registers are preserved before the call because the C/C++ function can modify any of these registers. You do not need to save save-on-entry registers because they are preserved by the called C/C++ function.

How to Map Interrupt Routines to Interrupt Vectors

Note: This section does not apply to Cortex-M devices.

To map Cortex-A interrupt routines to interrupt vectors you need to include a intvecs.asm file. This file will contain assembly language directives that can be used to set up Arm interrupt vectors with branches to your interrupt routines. Follow these steps to use this file:

1. Using Example 7 as a guide, create intvecs.asm and include your interrupt routines. For each routine:

    a. At the beginning of the file, add a .global directive that names the routine.

    b. Modify the appropriate .word directive to create a branch to the name of your routine.

2. Assemble and link intvecs.asm with your applications code and with the compiler's linker control file (lnk16.cmd or lnk32.cmd). The control file contains a SECTIONS directive that maps the .intvecs section into the memory locations 0x00-0x1F.

For example, on an Arm v4 target, if you have written a C interrupt routine for the IRQ interrupt called c_intIRQ and an assembly language routine for the FIQ interrupt called tim1_int, you should create intvecs.asm as in Example 7.

Example 7: Sample intvecs.asm File

```
    .if __TI_EABI_ASSEMBLER
    .asg c_intIRQ, C_INTIRQ
    .else
    .asg _c_intIRQ, C_INTIRQ
    .endif

.global _c_int00
```

(continues on next page)

```
.global C_INTIRQ
.global tim1_int

.sect ".intvecs"

B _c_int00              ; reset interrupt
.word 0                 ; undefined instruction interrupt
.word 0                 ; software interrupt
.word 0                 ; abort (prefetch) interrupt
.word 0                 ; abort (data) interrupt
.word 0                 ; reserved
B C_INTIRQ              ; IRQ interrupt
B tim1_int              ; FIQ interrupt
```

Using Software Interrupts

A software interrupt (SWI) is a synchronous exception generated by the execution of a particular instruction. Applications use software interrupts to request services from a protected system, such as an operating system, which can perform the services only while in a supervisor mode.

Some Arm documentation uses the term Supervisor Calls (SVC) instead of "software interrupt".

Since a call to the software interrupt function represents an invocation of the software interrupt, passing and returning data to and from a software interrupt is specified as normal function parameter passing with the following restriction:

All arguments passed to a software interrupt must reside in the four argument registers (R0-R3). No arguments can be passed by way of a software stack. Thus, only four arguments can be passed unless:

- Floating-point doubles are passed, in which case each double occupies two registers.

- Structures are returned, in which case the address of the returned structure occupies the first argument register.

For Cortex-M architectures, C SWI handlers cannot return values. Values may be returned by SWI handlers on other architectures.

The C/C++ compiler also treats the register usage of a called software interrupt the same as a called function. It assumes that all save-on-entry registers () are preserved by the software interrupt and that save-on-call registers (the remainder of the registers) can be altered by the software interrupt.

Other Interrupt Information

An interrupt routine can perform any task performed by any other function, including accessing global variables, allocating local variables, and calling other functions.

When you write interrupt routines, keep the following points in mind:

- It is your responsibility to handle any special masking of interrupts.

- A C/C++ interrupt routine cannot be called directly from C/C++ code.

- In a system reset interrupt, such as c_int00, you cannot assume that the run-time environment is set up; therefore, you *cannot allocate local variables*, and you *cannot save any information on the run-time stack*.

- In assembly language, remember to precede the name of a C/C++ interrupt with the appropriate linkname. For example, refer to c_int00 as _c_int00.

- When an interrupt occurs, the state of the processor (ARM or Thumb mode) is dependent on the device you are using. The compiler allows interrupt handlers to be defined in ARM or Thumb-2 mode. You should ensure the interrupt handler uses the proper mode for the device.

- The FIQ, supervisor, abort, IRQ, and undefined modes have separate stacks that are not automatically set up by the C/C++ run-time environment. If you have interrupt routines in one of these modes, you must set up the software stack for that mode. However, Arm Cortex-M processors have two stacks, and the main stack (MSP), which is used by IRQ (the only interrupt type for Cortex-M), is automatically handled by the compiler.

- Interrupt routines are not reentrant. If an interrupt routine enables interrupts of its type, it must save a copy of the return address and SPSR (the saved program status register) before doing so.

- Because a software interrupt is synchronous, the register saving conventions discussed in *Saving Registers During Interrupts* can be less restrictive as long as the system is designed for this. A software interrupt routine generated by the compiler, however, follows the conventions in *Saving Registers During Interrupts*.

3.3.8 Intrinsic Run-Time-Support Arithmetic and Conversion Routines

The intrinsic run-time-support library contains a number of assembly language routines that provide arithmetic and conversion capability for C/C++ operations that the 32-bit and 16-bit instruction sets do not provide. These routines include integer division, integer modulus, and floating-point operations.

There are two versions of each of the routines:

- A 16-bit version to be called only from the 16-BIS (bit instruction set) state

- A 32-bit version only to be called from the 32-BIS state

These routines do not follow the standard C/C++ calling conventions in that the naming and register conventions are not upheld.

See *Intrinsics* for a list of available intrinsics.

3.3.9 Built-In Functions

Built-in functions are predefined by the compiler. They can be called like a regular function, but they do not require a prototype or definition. The compiler supplies the proper prototype and definition.

The tiarmclang compiler supports the following built-in functions:

- The *__curpc* function, which returns the value of the program counter where it is called. The syntax of the function is:

```
void *__curpc(void);
```

- The *__run_address_check* function, which returns TRUE if the code performing the call is located at its run-time address, as assigned by the linker. Otherwise, FALSE is returned. The syntax of the function is:


```
int __run_address_check(void);
```

3.3.10 System Initialization

Before you can run a C/C++ program, you must create the C/C++ run-time environment. The C/C++ boot routine performs this task using a function called c_int00 (or _c_int00). The run-time-support source library, rts.src, contains the source to this routine in a module named boot.c (or boot.asm).

To begin running the system, the c_int00 function can be called by reset hardware. You must link the c_int00 function with the other object files. This occurs automatically when you use the --rom_model or --ram_model link option and include a standard run-time-support library as one of the linker input files.

When C/C++ programs are linked, the linker sets the entry point value in the executable output file to the symbol c_int00.

The c_int00 function performs the following tasks to initialize the environment:

1. Switches to the appropriate mode, reserves space for the run-time stack, and sets up the initial value of the stack pointer (SP). The stack is aligned on a 64-bit boundary.

2. Calls the function _ _TI_auto_init to perform the C/C++ autoinitialization.

    The _ _TI_auto_init function does the following tasks:

    - Processes the binit copy table, if present.

    - Performs C autoinitialization of global/static variables. For more information, see *Automatic Initialization of Variables*.

    - Calls C++ initialization routines for file scope construction from the global constructor table. For more information, see *Global Constructors*.

3. Calls the main() function to run the C/C++ program.

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the operations listed above to correctly initialize the C/C++ environment.

Boot Hook Functions for System Pre-Initialization

Boot hooks are points at which you may insert application functions into the C/C++ boot process. Default boot hook functions are provided with the run-time support (RTS) library. However, you can implement customized versions of these boot hook functions, which override the default boot hook functions in the RTS library if they are linked before the run-time library. Such functions can perform any application-specific initialization before continuing with the C/C++ environment setup.

If customized boot hook functions are defined in a user library, then in addition to linking the library *before* the run-time library, you may also need to use the *--priority* link option to ensure that unresolved symbol references to the boot functions are resolved by the first library that contains a symbol definition. This will prevent references to the boot functions from being resolved by the default implementations defined by the compiler run-time library. See *Exhaustively Read and Search Libraries (--reread_libs and --priority Options)*.

Note that RTOS kernels may use custom versions of the boot hook functions for system setup, so you should be careful about overriding these functions if you are using an RTOS.

The following boot hook functions are available:

__mpu_init(): This function provides an interface for initializing the MPU, if MPU support is included. The __mpu_init() function is called after the stack pointer is initialized but before any C/C++ environment setup is performed. This function should not return a value.

_system_pre_init(): This function provides a place to perform application-specific initialization. It is invoked after the stack pointer is initialized but before any C/C++ environment setup is performed. For targets that include MPU support, this function is called after __mpu_init().By default, _system_pre_init() should return a non-zero value. The default C/C++ environment setup is bypassed if _system_pre_init() returns 0.

_system_post_cinit(): This function is invoked during C/C++ environment setup, after C/C++ global data is initialized but before any C++ constructors are called. This function should not

return a value.

The _c_int00( ) initialization routine also provides a mechanism for an application to perform the setup (set I/O registers, enable/disable timers, etc.) before the C/C++ environment is initialized.

Run-Time Stack

The run-time stack is allocated in a single continuous block of memory and grows down from high addresses to lower addresses. The SP points to the top of the stack.

The code does not check to see if the run-time stack overflows. Stack overflow occurs when the stack grows beyond the limits of the memory space that was allocated for it. Be sure to allocate adequate memory for the stack.

The stack size can be changed at link time by using the --stack_size link option on the linker command line and specifying the stack size as a constant directly after the option.

The C/C++ boot routine shipped with the compiler sets up the user/thread mode run-time stack. If your program uses a run-time stack when it is in other operating modes, you must also allocate space and set up the run-time stack corresponding to those modes.

EABI requires that 64-bit data (type long long and long double) be aligned at 64-bits. This requires that the stack be aligned at a 64-bit boundary at function entry so that local 64-bit variables are allocated in the stack with correct alignment. The boot routine aligns the stack at a 64-bit boundary.

Automatic Initialization of Variables

Any global variables declared as preinitialized must have initial values assigned to them before a C/C++ program starts running. The process of retrieving these variables' data and initializing the variables with the data is called autoinitialization. Internally, the compiler and linker coordinate to produce compressed initialization tables. Your code should not access the initialization table.

Zero Initializing Variables

In ANSI C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler supports preinitialization of uninitialized variables by default. This can be turned off by specifying the linker option --zero_init=off.

Zero initialization takes place only if the --rom_model linker option, which causes autoinitialization to occur, is used. If you use the --ram_model option for linking, the linker does not generate initialization records, and the loader must handle both data and zero initialization.

Direct Initialization

The compiler uses direct initialization to initialize global variables. For example, consider the following C code:

```
int i    = 23;
int a[5] = { 1, 2, 3, 4, 5 };
```

The compiler allocates the variables 'i' and 'a[]' to .data section and the initial values are placed directly.

```
        .global i
        .data
     .align   4
i:
        .field          23,32       ; i @ 0
        .global a
        .data
        .align   4
a:
        .field          1,32        ; a[0] @ 0
        .field          2,32        ; a[1] @ 32
        .field          3,32        ; a[2] @ 64
        .field          4,32        ; a[3] @ 96
        .field          5,32        ; a[4] @ 128
```

Each compiled module that defines static or global variables contains these .data sections. The linker treats the .data section like any other initialized section and creates an output section. In the load-time initialization model, the sections are loaded into memory and used by the program. See *Initialization of Variables at Load Time*.

In the run-time initialization model, the linker uses the data in these sections to create initialization data and an additional compressed initialization table. The boot routine processes the initialization table to copy data from load addresses to run addresses. See *Autoinitialization of Variables at Run Time*.

Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the most common method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

Using this method, the linker creates a compressed initialization table and initialization data from the direct initialized sections in the compiled module. The table and data are used by the C/C++ boot routine to initialize variables in RAM using the table and data in ROM.

The following figure illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.
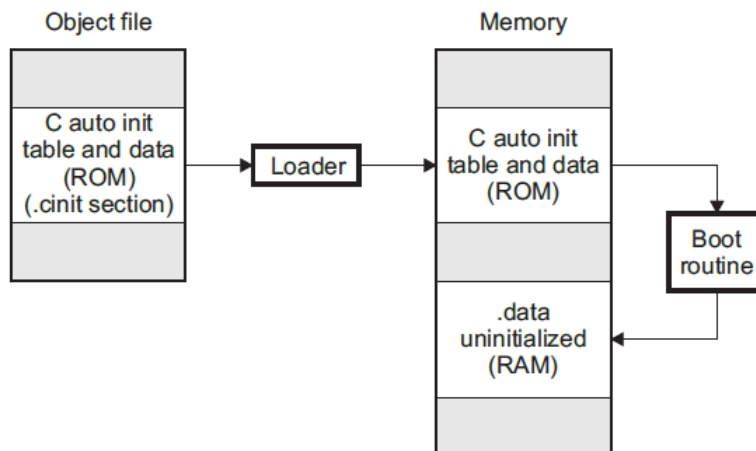
Figure 3.6: Autoinitialization at Run Time

Autoinitialization Tables

The compiled object files do not have initialization tables. The variables are initialized directly. The linker, when the --rom_model option is specified, creates C auto initialization table and the initialization data. The linker creates both the table and the initialization data in an output section named .cinit.

The autoinitialization table has the following format:
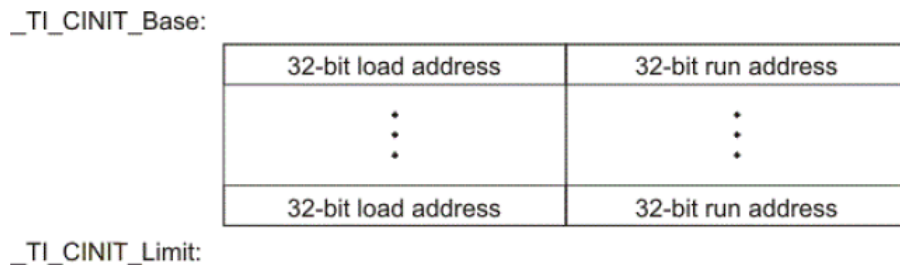
Figure 3.7: Autoinitialization Table Format

The linker defined symbols __TI_CINIT_Base and __TI_CINIT_Limit point to the start and end of the table, respectively. Each entry in this table corresponds to one output section that needs to be initialized. The initialization data for each output section could be encoded using different encoding.

The load address in the C auto initialization record points to initialization data with the following format:

8-bit index	Encoded data

Figure 3.8: Load Address Format

The first 8-bits of the initialization data is the handler index. It indexes into a handler table to get the address of a handler function that knows how to decode the following data.

The handler table is a list of 32-bit function pointers.

_TI_Handler_Table_Base:

32-bit handler 1 address
⋮
32-bit handler n address

_TI_Handler_Table_Limit:
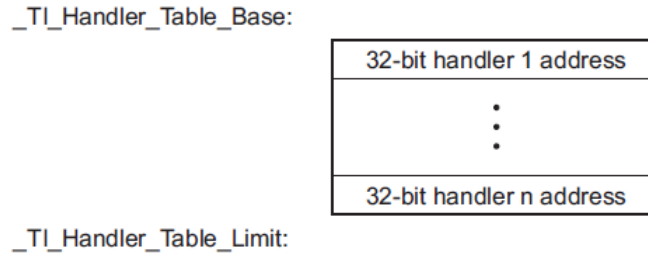
Figure 3.9: Handler Table Format

The *encoded data* that follows the 8-bit index can be in one of the following format types. For clarity the 8-bit index is also depicted for each format.

Length Followed by Data Format

8-bit index	24-bit padding	32-bit length (N)	N byte initialization data (not compressed)
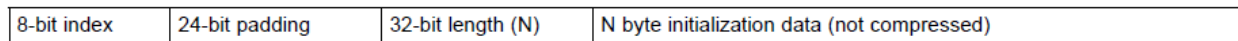
Figure 3.10: Encoded Data in Length Followed by Data Format

The compiler uses 24-bit padding to align the length field to a 32-bit boundary. The 32-bit length field encodes the length of the initialization data in bytes (N). N byte initialization data is not compressed and is copied to the run address as is.

The run-time support library has a function __TI_zero_init() to process this type of initialization data. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

Zero Initialization Format

8-bit index	24-bit padding	32-bit length (N)

Figure 3.11: Encoded Data in Zero Initialization Format

The compiler uses 24-bit padding to align the length field to a 32-bit boundary. The 32-bit length field encodes the number of bytes to be zero initialized.

The run-time support library has a function __TI_zero_init() to process the zero initialization. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

Run Length Encoded (RLE) Format

8-bit index	Initialization data compressed using run length encoding
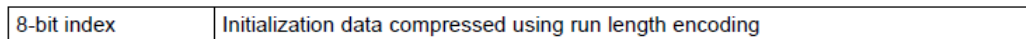
Figure 3.12: Encoded Data in RLE Format

The data following the 8-bit index is compressed using Run Length Encoded (RLE) format. uses a simple run length encoding that can be decompressed using the following algorithm:

1. Read the first byte, Delimiter (D).

2. Read the next byte (B).

3. If B != D, copy B to the output buffer and go to step 2.

4. Read the next byte (L).

   a. If L == 0, then length is either a 16-bit, a 24-bit value, or we've reached the end of the data, read next byte (L).

      1. If L == 0, length is a 24-bit value or the end of the data is reached, read next byte (L).

         a. If L == 0, the end of the data is reached, go to step 7.

         b. Else L <<= 16, read next two bytes into lower 16 bits of L to complete 24-bit value for L.

      2. Else L <<= 8, read next byte into lower 8 bits of L to complete 16-bit value for L.

   b. Else if L > 0 and L < 4, copy D to the output buffer L times. Go to step 2.

   c. Else, length is 8-bit value (L).

---

5. Read the next byte (C); C is the repeat character.

6. Write C to the output buffer L times; go to step 2.

7. End of processing.

The run-time support library has a routine __TI_decompress_rle24() to decompress data compressed using RLE. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

Note: **RLE Decompression Routine** The previous decompression routine, __TI_decompress_rle(), is included in the run-time-support library for decompressing RLE encodings generated by older versions of the linker.

Lempel-Ziv-Storer-Szymanski Compression (LZSS) Format

8-bit index	Initialization data compressed using LZSS
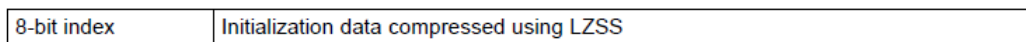
Figure 3.13: Encoded Data in LZSS Format

The data following the 8-bit index is compressed using LZSS compression. The run-time support library has the routine __TI_decompress_lzss() to decompress the data compressed using LZSS. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

Sample C Code to Process the C Autoinitialization Table

The run-time support boot routine has code to process the C autoinitialization table. The following C code illustrates how the autoinitialization table can be processed on the target.

Example: Processing the C Autoinitialization Table

```
typedef void (*handler_fptr)(const unsigned char *in,
unsigned char *out);

#define HANDLER_TABLE   __TI_Handler_Table_Base
#pragma WEAK(HANDLER_TABLE)
extern  unsigned int    HANDLER_TABLE;
extern  unsigned char *__TI_CINIT_Base;
extern  unsigned char *__TI_CINIT_Limit;
```

(continues on next page)

```
void auto_initialize()
{
    unsigned char **table_ptr;
    unsigned char **table_limit;

    /*------------------------------------------------------------
↪---*/
    /* Check if Handler table has entries.                       ␣
↪   */
    /*------------------------------------------------------------
↪---*/
    if (&__TI_Handler_Table_Base >= &__TI_Handler_Table_Limit)
        return;

    /*------------------------------------------------------------
↪----*/
    /* Get the Start and End of the CINIT Table.                 ␣
↪    */
    /*------------------------------------------------------------
↪----*/
    table_ptr   = (unsigned char **)&__TI_CINIT_Base;
    table_limit = (unsigned char **)&__TI_CINIT_Limit;
    while (table_ptr < table_limit)
    {

        /*--------------------------------------------------------
↪------*/
        /* 1. Get the Load and Run address.                      ␣
↪      */
        /* 2. Read the 8-bit index from the load address.        ␣
↪      */
        /* 3. Get the handler function pointer using the index␣
↪from   */
        /*    handler table.                                     ␣
↪      */
        /*--------------------------------------------------------
↪------*/
        unsigned char *load_addr   = *table_ptr++;
        unsigned char *run_addr    = *table_ptr++;
        unsigned char  handler_idx = *load_addr++;
        handler_fptr   handler     =
                        (handler_fptr)(&HANDLER_
↪TABLE)[handler_idx];
```

```
        /*----------------------------------------------------------
↪------*/
        /* 4. Call the handler and pass the pointer to the load␣
↪data    */
        /*    after index and the run address.                   ␣
↪        */
        /*----------------------------------------------------------
↪------*/
        (*handler)((const unsigned char *)load_addr, run_addr);
    }
}
```

Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the --ram_model option.

When you use the --ram_model link option, the linker does not generate C autoinitialization tables and data. The direct initialized sections (.data) in the compiled object files are combined according to the linker command file to generate initialized output sections. The loader loads the initialized output sections into memory. After the load, the variables are assigned their initial values.

Since the linker does not generate the C autoinitialization tables, no boot time initialization is performed.

The following figure illustrates the initialization of variables at load time.
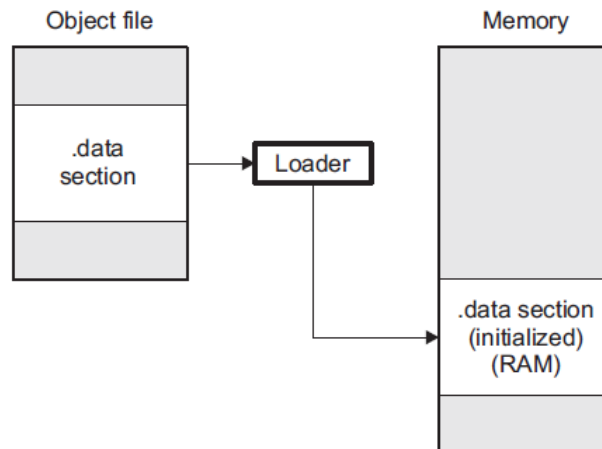
Figure 3.14: Initialization of Variables at Load Time

3.3. Run-Time Environment   **390**

Global Constructors

All global C++ variables that have constructors must have their constructor called before main(). The compiler builds a table of global constructor addresses that must be called, in order, before main() in a section called .init_array. The linker combines the .init_array section form each input file to form a single table in the .init_array section. The boot routine uses this table to execute the constructors. The linker defines two symbols to identify the combined .init_array table as shown below. This table is not null terminated by the linker.

Figure 3.15: Global Constructor Address Table

3.4 Using Run-Time-Support Functions and Building Libraries

Some of the features of C/C++ (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are provided as an ANSI/ISO C/C++ standard library, rather than as part of the compiler itself. The TI implementation of this library is the run-time-support library (RTS). The C/C++ compiler implements the ISO standard library except for those facilities that handle exception conditions, signal and locale issues (properties that depend on local language, nationality, or culture). Using the ANSI/ISO standard library ensures a consistent set of functions that provide for greater portability.

In addition to the ANSI/ISO-specified functions, the run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests. These are detailed in *C and C++ Run-Time Support Libraries* and *The C I/O Functions*.

3.4.1 C and C++ Run-Time Support Libraries

The TI Arm compiler installation includes pre-built run-time support (RTS) libraries that provide all the standard capabilities. Separate libraries are provided for each mode, big and little endian support, each ABI (compiler version 4.1.0 and later), various architectures, and C++ exception support. See *Library Naming Conventions* for information on the library file names and paths.

The run-time-support library contains the following:

- ANSI/ISO C/C++ standard library
- C I/O library
- Low-level support functions that provide I/O to the host operating system
- Fundamental arithmetic routines
- System startup routine, _c_int00
- Time routines
- Compiler helper functions (to support language features that are not directly efficiently expressible in C/C++)

The run-time-support libraries do not contain functions involving signals and locale issues.

The C++ library supports wide chars, in that template functions and classes that are defined for char are also available for wide char. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described in *C/C++ Language Options*.

TI does not provide documentation that covers the functionality of the C++ library. TI suggests referring to one of the following sources:

- *The Standard C++ Library: A Tutorial and Reference*, Nicolai M. Josuttis, Addison-Wesley, ISBN 0-201-37926-0
- *The C++ Programming Language* (Third or Special Editions), Bjarne Stroustrup, Addison-Wesley, ISBN 0-201-88954-4 or 0-201-70073-5

Note: You can avoid linking with the C and C++ Run-Time Support Libraries by using the **-nostdlib** compiler option. See *C/C++ Run-Time Standard Header and Library Options* for more information.

Linking Code With the Object Library

When you link your program, you must specify the object library as one of the linker input files so that references to the I/O and run-time-support functions can be resolved. You can either specify the library or allow the compiler to select one for you. See *Automatic Library Selection (-- disable_auto_rts Option)* for further information.

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see *Linker Description*.

C, C++, and mixed C and C++ programs can use the same run-time-support library. Run-time-support functions and variables that can be called and referenced from both C and C++ will have the same linkage.

Enabling AEABI Portability

If you want to link object files created with the TI CodeGen tools with object files generated by other compiler tool chains, the Arm standard specifies that you should define the _AEABI_PORTABILITY_LEVEL preprocessor symbol as follows before #including any standard header files, such as <stdlib.h> or <time.h>.

```
#define _AEABI_PORTABILITY_LEVEL 1
```

This definition enables full portability. Defining the symbol to 0 specifies that the "C standard" portability level will be used.

Header Files

You must use the header files provided with the compiler run-time support when using functions from C/C++ standard library.

The following header files provide TI extensions to the C standard:

- cpy_tbl.h – Declares the copy_in() RTS function, which is used to move code or data from a load location to a separate run location at run-time. This function helps manage overlays.

- file.h – Declares functions used by low-level I/O functions in the RTS library.

- _lock.h – Used when declaring system-wide mutex locks. This header file is deprecated; use _reg_mutex_api.h and _mutex.h instead.

- memory.h – Provides the memalign() function, which is not required by the C standard.

- _mutex.h – Declares functions used by the RTS library to help facilitate mutexes for specific resources that are owned by the RTS. For example, these functions are used for heap or file table allocation.

- _pthread.h – Declares low-level mutex infrastructure functions and provides support for recursive mutexes.

- _reg_mutex_api.h – Declares a function that can be used by an RTOS to register an underlying lock mechanism and/or thread ID mechanism that is implemented in the RTOS but is called indirectly by the RTS' _mutex.h functions.

- _reg_synch_api.h – Declares a function that can be used by an RTOS to register an underlying cache synchronization mechanism that is implemented in the RTOS but is called indirectly by the RTS' _data_synch.h functions.

- strings.h – Provides additional string functions, including bcmp(), bcopy(), bzero(), ffs(), index(), rindex(), strcasecmp(), and strncasecmp().

Support for String and Character Handling

The library includes the header files <string.h>, <strings.h>, and <wchar.h>, which provide the following functions for string handling beyond those required.

- string.h

  - memcpy(), which copies memory from one location to another

  - memcmp(), which compares sections of memory

  - strcmp() and strncmp(), which perform case-sensitive string comparisons

  - strdup(), which duplicates a string by dynamically allocating memory and copying the string to this allocated memory

  - strlen_s(), which is the same as the strlen() function, except that it provides checking for null pointers or strings that are not null-terminated. This function is available only if you are using C11/C++11 or later *and* if you have used #define to set __STDC_WANT_LIB_EXT1__ to the integer constant 1 prior to the #include statement for string.h.

- strings.h

  - bcmp(), which is equivalent to memcmp()

  - bcopy(), which is equivalent to memmove()

  - bzero(), which is equivalent to memset(.., 0, . . .);

  - ffs(), which finds the first bit set and returns the index of that bit

  - index(), which is equivalent to strchr()

  - rindex(), which is equivalent to strrchr()

  - strcasecmp() and strncasecmp(), which perform case-insensitive string comparisons

- wchar.h

- wcsnlen_s(), which is the same as the wcsnlen() function, except that it provides checking for null pointers or strings that are not null-terminated. This function is available only if you are using C11/C++11 or later *and* if you have used #define to set __STDC_WANT_LIB_EXT1__ to the integer constant 1 prior to the #include statement for wchar.h.

Support for time.h and time_t

The library includes the header file <time.h>, which provides the following functions for time handling beyond those required:

- asctime(), which returns a pointer to a string representing the day and time
- clock(), which returns the processor clock time
- ctime(), which returns a string representing the localtime
- difftime(), which returns the difference in seconds between two times
- gmtime(), which expresses the time value expressed in Greenwich Mean Time (GMT)
- localtime(), which returns the time value expressed in the local time zone
- mktime(), which converts the given time structure into a time_t value
- strftime(), which formats the time according to given format rules
- time(), which calculates the current calendar time and returns it as time_t

The library defines time_t by default as a *POSIX-compatible signed 64-bit value using the POSIX epoch of January 1, 1970*. This is different from the TI proprietary ARM compiler (armcl), which defines time_t as an unsigned 32-bit value using a legacy TI-defined epoch of January 1, 1900. You do not need to take any special steps or define any macros to use the new implementation, just be sure to include time.h and use the standard C time functions, which automatically map to 64-bit implementations.

Enabling AEABI Portability

Enabling AEABI portability mode *as described above* forcea the compiler to use an unsigned 32-bit representation for time_t with POSIX epoch. All calls to the standard C time functions refer to their 32-bit variants. In addition, the compiler also ensures that the _AEABI_PORTABLE macro is defined, as required by the standard.

Note: If your program attempts to use or manipulate any time_t with value larger than INT_MAX (which corresponds to a date in 2038), the time functions may not work properly.

You may also interlink third party object code with the tiarmclang linker and runtime library, including calling the standard C time functions, requiring no special steps. You are responsible for ensuring that AEABI portability is used consistently throughout your application.

Leveraging the Unsigned 32-bit Representation of time_t

You may also activate the unsigned 32-bit representation of time_t without activating AEABI portability mode. Instead of setting the _AEABI_PORTABILITY_LEVEL macro, simply set the macro __TI_TIME_USES_64=0.

As long as variables of type time_t aren't used globally, you may freely link object files built using __TI_TIME_USES_64=0 with those that do not since the actual time functions in the RTS are not changed.

Note: The epoch used for time_t functions is an implementation-defined property. For this reason, you should be cautious when attempting to link using time functions defined within a runtime library built using the TI proprietary ARM compiler (armcl) since that compiler does not define values for time_t using the POSIX epoch. You are responsible for ensuring that time_t is used consistently through your application.

Minimal Support for Internationalization

The library includes the header files <locale.h>, <wchar.h>, and <wctype.h>, which provide APIs to support non-ASCII character sets and conventions. Our implementation of these APIs is limited in the following ways:

- The library has minimal support for wide and multibyte characters. The type wchar_t is implemented as int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h> but does not include all the functions specified in the standard. See *Generic Compiler Pre-Defined Macro Symbols* for more information about extended character sets.

- The C library includes the header file <locale.h> but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale via a call to setlocale() returns NULL.

Allowable Number of Open Files

In the <stdio.h> header file, the value for the macro FOPEN_MAX has the value of the macro _NFILE, which is set to 10. The impact is that you can only have 10 files simultaneously open at one time (including the pre-defined streams stdin, stdout, and stderr).

The C standard requires that the minimum value for the FOPEN_MAX macro is 8. The macro determines the maximum number of files that can be opened at one time. The macro is defined in the stdio.h header file and can be modified by changing the value of the _NFILE macro and recompiling the library.

Nonstandard Header Files in the Source Tree

The source code in the lib/src subdirectory of the compiler installation contains these non-ANSI include files that are used to build the library:

- The *file.h* file includes macros and definitions used for low-level I/O functions.

- The *format.h* file includes structures and macros used in printf and scanf.

- The *trgcio.h* file includes low-level, target-specific C I/O macro definitions. If necessary, you can customize trgcio.h.

Library Naming Conventions

By default, which run-time-support library (see *Invoking the Compiler*) to link with is determined based on the command-line options used to compile your application, and you do not need to specify the RTS library to the linker.

If you select the library manually, you must select the matching library, using the following naming scheme:

arm*<sub><endian>*-ti-none-eabi*<float>*

Where the following must be indicated:

- *<sub>*: "v6m", "v7r", "v7m", "v7em", or "v8m.main"

- *<endian>*: "eb" for big-endian or *blank* for little-endian

- *<float>*: "hf" for VFP support or *blank* for no VFP support

Each directory variant contains the following files and subdirectories:

```
libc++.a
libc++abi.a
c/libc.a
c/libsysbm.a
```

(continues on next page)

```
except/libc++.a
except/libc++abi.a
```

The `except` directory provided for each variant contains C++ run-time-support libraries compiled with exception handling support. These library variants are used automatically if you use the -exceptions compiler option.

For example, on a Windows installation, the directory containing run-time-support libraries for Arm v7r using little-endian code, VFP support, and no exception handling support might be:

```
C:\mycompilers\ti-cgt-armllvm_3.2.0.LTS\lib\armv7r-ti-none-eabihf
```

The corresponding libraries *with* exception handling support would then be:

```
C:\mycompilers\ti-cgt-armllvm_3.2.0.LTS\lib\armv7r-ti-none-
 ↪eabihf\except
```

3.4.2 The C I/O Functions

The C I/O functions make it possible to access the host's operating system to perform I/O. The capability to perform I/O on the host gives you more options when debugging and testing code.

The I/O functions are logically divided into layers: high level, low level, and device-driver level.

With properly written device drivers, the C-standard high-level I/O functions can be used to perform I/O on custom user-defined devices. This provides an easy way to use the sophisticated buffering of the high-level I/O functions on an arbitrary device.

The formatting rules for long long data types require ll (lowercase LL) in the format string. For example:

```
printf("%lld", 0x0011223344556677);
printf("llx", 0x0011223344556677);
```

Note: Debugger Required for Default HOST For the default HOST device to work, there must be a debugger to handle the C I/O requests; the default HOST device cannot work by itself in an embedded system. To work in an embedded system, you need to provide an appropriate driver for your system.

Note: C I/O Mysteriously Fails If there is not enough space on the heap for a C I/O buffer, operations on the file will silently fail. If a call to printf() mysteriously fails, this may be the reason. The heap needs to be at least large enough to allocate a block of size BUFSIZ (defined in

---

stdio.h) for every file on which I/O is performed, including stdout, stdin, and stderr, plus allocations performed by the user's code, plus allocation bookkeeping overhead. Alternately, declare a char array of size BUFSIZ and pass it to setvbuf to avoid dynamic allocation. To set the heap size, use the --heap_size option when linking (refer to *Linker Description*).

Note: Open Mysteriously Fails The run-time support limits the total number of open files to a small number relative to general-purpose processors. If you attempt to open more files than the maximum, you may find that the open will mysteriously fail. You can increase the number of open files by extracting the source code from rts.src and editing the constants controlling the size of some of the C I/O data structures. The macro _NFILE controls how many FILE (fopen) objects can be open at one time (stdin, stdout, and stderr count against this total). (See also FOPEN_MAX.) The macro _NSTREAM controls how many low-level file descriptors can be open at one time (the low-level files underlying stdin, stdout, and stderr count against this total). The macro _NDEVICE controls how many device drivers are installed at one time (the HOST device counts against this total).

High-Level I/O Functions

The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, and so on). These functions call one or more low-level I/O functions to carry out the high-level I/O request. The high-level I/O routines operate on FILE pointers, also called *streams*.

Portable applications should use only the high-level I/O functions.

To use the high-level I/O functions:

- Include the header file stdio.h for each module that references a function.

- Allow for 320 bytes of heap space for each I/O stream used in your program. A stream is a source or destination of data that is associated with a peripheral, such as a terminal or keyboard. Streams are buffered using dynamically allocated memory that is taken from the heap. More heap space may be required to support programs that use additional amounts of dynamically allocated memory (calls to malloc()). To set the heap size, use the --heap_size option when linking; see *Define Heap Size (--heap_size Option)*.

For example, given the following C program in a file named main.c:

```c
#include <stdio.h>

void main()
{
    FILE *fid;
```

<div align="right">(continues on next page)</div>

```
    fid = fopen("myfile","w");
    fprintf(fid,"Hello, world\n");
    fclose(fid);

    printf("Hello again, world\n");
}
```

Issuing the following compiler command compiles, links, and creates the file main.out using the appropriate version of the run-time-support library:

```
tiarmclang main.c -Xlinker --heap_size=400 -Xlinker --output_
↪file=main.out
```

Executing main.out results in:

```
Hello, world
```

being output to a file and

```
Hello again, world
```

being output to your host's stdout window.

Formatting and the Format Conversion Buffer

The internal routine behind the C I/O functions—such as printf(), vsnprintf(), and snprintf()—reserves stack space for a format conversion buffer. The buffer size is set by the macro FORMAT_CONVERSION_BUFSIZE, which is defined in format.h. Consider the following issues before reducing the size of this buffer:

- The default buffer size is 510 bytes. If MINIMAL is defined, the size is set to 32, which allows integer values without width specifiers to be printed.

- Each conversion specified with %xxxx (except %s) must fit in FORMAT_CONVERSION_BUFSIZE. This means any individual formatted float or integer value, accounting for width and precision specifiers, needs to fit in the buffer. Since the actual value of any representable number should easily fit, the main concern is ensuring the width and/or precision size meets the constraints.

- The length of converted strings using %s are unaffected by any change in FORMAT_CONVERSION_BUFSIZE. For example, you can specify `printf("%s value is %d",some_really_long_string,intval)` without a problem.

- The constraint is for each individual item being converted. For example, a format string of `%d item1 %f item2 %e item3` does not need to fit in the buffer. Instead, each

converted item specified with a % format must fit.

- There is no buffer overrun check.

Overview of Low-Level I/O Implementation

The low-level functions are comprised of seven basic I/O functions: open, read, write, close, lseek, rename, and unlink. These low-level routines provide the interface between the high-level functions and the device-level drivers that actually perform the I/O command on the specified device.

The low-level functions are designed to be appropriate for all I/O methods, even those which are not actually disk files. Abstractly, all I/O channels can be treated as files, although some operations (such as lseek) may not be appropriate. See *Device-Driver Level I/O Functions* for more details.

The low-level functions are inspired by, but not identical to, the POSIX functions of the same names.

The low-level functions operate on file descriptors. A file descriptor is an integer returned by open, representing an opened file. Multiple file descriptors may be associated with a file; each has its own independent file position indicator.

open – Open File for I/O

Syntax

```
#include <file.h>

int open (const char * path,
          unsigned     flags,
          int          file_descriptor );
```

Description The open function opens the file specified by *path* and prepares it for I/O.

- The *path* is the filename of the file to be opened, including an optional directory path and an optional device specifier (see *The device Prefix*).

- The *flags* are attributes that specify how the file is manipulated. Low-level I/O routines allow or disallow some operations depending on the flags used when the file was opened. Some flags may not be meaningful for some devices, depending on how the device implements files. The flags are specified using the following symbols:

```
O_RDONLY (0x0000) /* open for reading */
O_WRONLY (0x0001) /* open for writing */
O_RDWR (0x0002) /* open for read & write */
O_APPEND (0x0008) /* append on each write */
O_CREAT (0x0200) /* open with file create */
```

(continues on next page)

- The *count* is the number of characters to read from the file.

Return Value The function returns one of the following values:

- 0 if EOF was encountered before any characters were read
- *positive value* to indicate number of characters read (may be less than *count*)
- -1 on failure

write – Write Characters to a File

Syntax

```
#include <file.h>

int write ( int           file_descriptor,
            const char * buffer,
            unsigned      count );
```

Description The write function writes the number of characters specified by *count* from the *buffer* to the file associated with *file_descriptor*.

- The *file_descriptor* is the number assigned by open to an opened file.
- The *buffer* is where the characters to be written are located.
- The *count* is the number of characters to write to the file.

Return Value The function returns one of the following values:

- *positive value* to indicate number of characters written if successful (may be less than *count*)
- -1 on failure

lseek – Set File Position Indicator

Syntax for C

```
#include <file.h>

off_t lseek ( int   file_descriptor,
              off_t offset,
              int   origin );
```

Description The lseek function sets the file position indicator for the given file to a location relative to the specified origin. The file position indicator measures the position in characters from the beginning of the file.

- The *file_descriptor* is the number assigned by open to an opened file.

- The *offset* indicates the relative offset from the *origin* in characters.

- The *origin* is used to indicate which of the base locations the *offset* is measured from. The *origin* must be one of the following macros:

    - SEEK_SET (0x0000) Beginning of file

    - SEEK_CUR (0x0001) Current value of the file position indicator

    - SEEK_END (0x0002) End of file

Return Value The return value is one of the following:

- *positive value* to indicate new value of the file position indicator if successful

- (off_t)-1 on failure

unlink – Delete File

Syntax

```
#include <file.h>

int unlink ( const char * path );
```

Description The unlink function deletes the file specified by *path*. Depending on the device, a deleted file may still remain until all file descriptors which have been opened for that file have been closed. See *Device-Driver Level I/O Functions*.

The *path* is the filename of the file, including path information and optional device prefix. (See *The device Prefix*.)

Return Value The function returns one of the following values:

- 0 if successful

- -1 on failure

rename – Rename File

Syntax for C

```
#include {<stdio.h> \| <file.h>}

int rename ( const char * old_name,
             const char * new_name );
```

Syntax for C++

```
#include {<cstdio> \| <file.h>}

int std::rename ( const char * old_name,
                  const char * new_name );
```

Description The rename function changes the name of a file.

- The *old_name* is the current name of the file.

- The *new_name* is the new name for the file.

Note: The optional device specified in the new name must match the device of the old name. If they do not match, a file copy would be required to perform the rename, and rename is not capable of this action.

Return Value The function returns one of the following values:

- 0 if successful

- -1 on failure

Note: Although rename is a low-level function, it is defined by the C standard and can be used by portable applications.

Device-Driver Level I/O Functions

At the next level are the device-level drivers. They map directly to the low-level I/O functions. The default device driver is the HOST device driver, which uses the debugger to perform file operations. The HOST device driver is automatically used for the default C streams stdin, stdout, and stderr.

The HOST device driver shares a special protocol with the debugger running on a host system so that the host can perform the C I/O requested by the program. Instructions for C I/O operations that the program wants to perform are encoded in a special buffer named _CIOBUF_ in the .cio section. The debugger halts the program at a special breakpoint (C$$IO$$), reads and decodes the target memory, and performs the requested operation. The result is encoded into _CIOBUF_, the program is resumed, and the target decodes the result.

The HOST device is implemented with seven functions, HOSTopen, HOSTclose, HOSTread, HOSTwrite, HOSTlseek, HOSTunlink, and HOSTrename, which perform the encoding. Each function is called from the low-level I/O function with a similar name.

A device driver is composed of seven required functions. Not all function need to be meaningful for all devices, but all seven must be defined. Here we show the names of all seven functions as

---

starting with DEV, but you may choose any name except for HOST.

DEV_open – Open File for I/O

Syntax

```
int DEV_open ( const char * path,
               unsigned     flags ,
               int          llv_fd );
```

Description This function finds a file matching *path* and opens it for I/O as requested by *flags*.

- The *path* is the filename of the file to be opened. If the name of a file passed to open has a device prefix, the device prefix will be stripped by open, so DEV_open will not see it. (See *The device Prefix* for details on the device prefix.)

- The *flags* are attributes that specify how the file is manipulated. See POSIX for further explanation of the flags. The flags are specified using the following symbols:

```
O_RDONLY (0x0000) /* open for reading */
O_WRONLY (0x0001) /* open for writing */
O_RDWR   (0x0002) /* open for read & write */
O_APPEND (0x0008) /* append on each write */
O_CREAT  (0x0200) /* open with file create */
O_TRUNC  (0x0400) /* open with truncation */
O_BINARY (0x8000) /* open in binary mode */
```

- The *llv_fd* is treated as a suggested low-level file descriptor. This is a historical artifact; newly-defined device drivers should ignore this argument. This differs from the low-level I/O open function.

This function must arrange for information to be saved for each file descriptor, typically including a file position indicator and any significant flags. For the HOST version, all the bookkeeping is handled by the debugger running on the host machine. If the device uses an internal buffer, the buffer can be created when a file is opened, or the buffer can be created during a read or write.

Return Value This function must return -1 to indicate an error if for some reason the file could not be opened; such as the file does not exist, could not be created, or there are too many files open. The value of errno may optionally be set to indicate the exact error (the HOST device does not set errno). Some devices might have special failure conditions; for instance, if a device is read-only, a file cannot be opened O_WRONLY.

On success, this function must return a non-negative file descriptor unique among all open files handled by the specific device. The file descriptor need not be unique across devices. The device file descriptor is used only by low-level functions when calling the device-driver-level functions. The low-level function open allocates its own unique file descriptor for the high-level functions

to call the low-level functions. Code that uses only high-level I/O functions need not be aware of these file descriptors.

DEV_close – Close File for I/O

Syntax

```
int DEV_close ( int dev_fd );
```

Description This function closes a valid open file descriptor.

On some devices, DEV_close may need to be responsible for checking if this is the last file descriptor pointing to a file that was unlinked. If so, it is responsible for ensuring that the file is actually removed from the device and the resources reclaimed, if appropriate.

Return Value This function should return -1 to indicate an error if the file descriptor is invalid in some way, such as being out of range or already closed, but this is not required. The user should not call close() with an invalid file descriptor.

DEV_read – Read Characters from a File

Syntax

```
int DEV_read ( int      dev_fd ,
               char *   buf ,
               unsigned count );
```

Description The read function reads *count* bytes from the input file associated with *dev_fd*.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buf* is where the read characters are placed.
- The *count* is the number of characters to read from the file.

Return Value This function must return -1 to indicate an error if for some reason no bytes could be read from the file. This could be because of an attempt to read from a O_WRONLY file, or for device-specific reasons.

If count is 0, no bytes are read and this function returns 0.

This function returns the number of bytes read, from 0 to count. 0 indicates that EOF was reached before any bytes were read. It is not an error to read less than count bytes; this is common if there are not enough bytes left in the file or the request was larger than an internal device buffer size.

DEV_write – Write Characters to a File

Syntax

```
int DEV_write ( int          dev_fd ,
                const char * buf ,
                unsigned     count );
```

Description This function writes *count* bytes to the output file.

- The *dev_fd* is the number assigned by open to an opened file.

- The *buffer* is where the write characters are placed.

- The *count* is the number of characters to write to the file.

Return Value This function must return -1 to indicate an error if for some reason no bytes could be written to the file. This could be because of an attempt to read from a O_RDONLY file, or for device-specific reasons.

DEV_lseek – Set File Position Indicator

Syntax

```
off_t DEV_lseek ( int   dev_fd,
                  off_t offset,
                  int   origin );
```

Description This function sets the file's position indicator for this file descriptor as *lseek – Set File Position Indicator*.

If lseek is supported, it should not allow a seek to before the beginning of the file, but it should support seeking past the end of the file. Such seeks do not change the size of the file, but if it is followed by a write, the file size increases.

Return Value If successful, this function returns the new value of the file position indicator.

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. For many devices, the lseek operation is nonsensical (e.g. a computer monitor).

DEV_unlink – Delete File

Syntax

```
int DEV_unlink (const char * path );
```

Description Remove the association of the pathname with the file. This means that the file may no longer be opened using this name, but the file may not actually be immediately removed.

Depending on the device, the file may be immediately removed, but for a device that allows open file descriptors to point to unlinked files, the file is not actually deleted until the last file descriptor is closed. See *Device-Driver Level I/O Functions*.

Return Value This function must return -1 to indicate an error if for some reason the file could not be unlinked (delayed removal does not count as a failure to unlink.)

If successful, this function returns 0.

DEV_rename – Rename File

Syntax

```
int DEV_rename ( const char * old_name,
                 const char * new_name );
```

Description This function changes the name associated with the file.

- The *old_name* is the current name of the file.

- The *new_name* is the new name for the file.

Return Value This function must return -1 to indicate an error if for some reason the file could not be renamed, such as the file doesn't exist, or the new name already exists.

Note: It is inadvisable to allow renaming a file so that it is on a different device. In general this would require a whole file copy, which may be more expensive than you expect.

If successful, this function returns 0.

Adding a User-Defined Device Driver for C I/O

The function add_device allows you to add and use a device. When a device is registered with add_device, the high-level I/O routines can be used for I/O on that device.

You can use a different protocol to communicate with any desired device and install that protocol using add_device; however, the HOST functions should not be modified. The default streams stdin, stdout, and stderr can be remapped to a file on a user-defined device instead of HOST by using freopen() as in the following example. If the default streams are reopened in this way, the buffering mode changes to _IOFBF (fully buffered). To restore the default buffering behavior, call setvbuf on each reopened file with the appropriate value (_IOLBF for stdin and stdout, _IONBF for stderr).

The default streams stdin, stdout, and stderr can be mapped to a file on a user-defined device instead of HOST by using freopen() as shown in the following example. Each function must set up and maintain its own data structures as needed. Some function definitions perform no action and should just return.

```c
#include <stdio.h>
#include <file.h>
#include "mydevice.h"

void main()
{
    add_device("mydevice", _MSA,
               MYDEVICE_open, MYDEVICE_close,
               MYDEVICE_read, MYDEVICE_write,
               MYDEVICE_lseek, MYDEVICE_unlink, MYDEVICE_
 ↪rename);


    /*------------------------------------------------------
 ↪------------*/
    /* Re-open stderr as a MYDEVICE file */
    /*------------------------------------------------------
 ↪------------*/
if (!freopen("mydevice:stderrfile", "w", stderr))
{
    puts("Failed to freopen stderr");
    exit(EXIT_FAILURE);
}


/*------------------------------------------------------
 ↪--------*/
/* stderr should not be fully buffered; we want errors to be␣
 ↪seen as */
```

```
/* soon as possible. Normally stderr is line-buffered, but this␣
↪example */
/* doesn't buffer stderr at all. This means that there will be␣
↪one call */
/* to write() for each character in the message. */
/*--------------------------------------------------------------
↪--------*/
if (setvbuf(stderr, NULL, _IONBF, 0))
{
    puts("Failed to setvbuf stderr");
    exit(EXIT_FAILURE);
}


/*--------------------------------------------------------------
↪--------*/
/* Try it out! */
/*--------------------------------------------------------------
↪--------*/
printf("This goes to stdout\n");
fprintf(stderr, "This goes to stderr\n"); }
```

Note: Use Unique Function Names The function names open, read, write, close, lseek, rename, and unlink are used by the low-level routines. Use other names for the device-level functions that you write.

Use the low-level function add_device() to add your device to the device_table. The device table is a statically defined array that supports *n* devices, where *n* is defined by the macro _NDEVICE found in stdio.h/cstdio.

The first entry in the device table is predefined to be the host device on which the debugger is running. The low-level routine add_device() finds the first empty position in the device table and initializes the device fields with the passed-in arguments. For a complete description, see *add_device – Add Device to Device Table*.

The device Prefix

A file can be opened to a user-defined device driver by using a device prefix in the pathname. The device prefix is the device name used in the call to add_device followed by a colon. For example:

```
FILE *fptr = fopen("mydevice:file1", "r");
int fd = open("mydevice:file2, O_RDONLY, 0);
```

If no device prefix is used, the HOST device is used to open the file.

add_device – Add Device to Device Table

Syntax for C

```
#include <file.h>

int add_device(
      char * name,
      unsigned flags,
      int    (* dopen )(const char *path, unsigned flags, int
→llv_fd),
      int    (* dclose )(int dev_fd),
      int    (* dread )(int dev_fd, char *buf, unsigned count),
      int    (* dwrite )(int dev_fd, const char *buf, unsigned
→count),
      off_t (* dlseek )(int dev_fd, off_t ioffset, int origin),
      int    (* dunlink )(const char * path),
      int    (* drename )(const char *old_name, const char *new_
→name) );
```

Defined in lowlev.c (in the lib/src subdirectory of the compiler installation)

Description The add_device function adds a device record to the device table allowing that device to be used for I/O from C. The first entry in the device table is predefined to be the HOST device on which the debugger is running. The function add_device() finds the first empty position in the device table and initializes the fields of the structure that represent a device.

To open a stream on a newly added device use fopen( ) with a string of the format *device-name:filename* as the first argument.

- The *name* is a character string denoting the device name. The name is limited to 8 characters.

- The *flags* are device characteristics. The defined flags are as follows, and more flags can be added by defining them in file.h:

  - **_SSA** Denotes that the device supports only one open stream at a time

– **_MSA** Denotes that the device supports multiple open streams

- The *dopen*, *dclose*, *dread*, *dwrite*, *dlseek*, *dunlink*, and *drename* specifiers are function pointers to the functions in the device driver that are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in *Overview of Low-Level I/O Implementation*. The device driver for the HOST that the Arm debugger is run on are included in the C I/O library.

Return Value The function returns one of the following values:

- 0 if successful

- -1 on failure

Example The following example illustrates adding and using a device for C I/O. It does the following:

- Adds the device *mydevice* to the device table

- Opens a file named *test* on that device and associates it with the FILE pointer *fid*

- Writes the string *Hello, world* into the file

- Closes the file

```c
#include <file.h>
#include <stdio.h>
/
 ↪********************************************************************************
 ↪
/* Declarations of the user-defined device drivers */
/
 ↪********************************************************************************
 ↪
extern int MYDEVICE_open(const char *path, unsigned flags, int
 ↪fno);
extern int MYDEVICE_close(int fno);
extern int MYDEVICE_read(int fno, char *buffer, unsigned count);
extern int MYDEVICE_write(int fno, const char *buffer, unsigned
 ↪count);
extern off_t MYDEVICE_lseek(int fno, off_t offset, int origin);
extern int MYDEVICE_unlink(const char *path);
extern int MYDEVICE_rename(const char *old_name, char *new_name);

main()
{
    FILE *fid;
    add_device("mydevice", _MSA, MYDEVICE_open, MYDEVICE_close,
 ↪MYDEVICE_read,
```

(continues on next page)

```
                MYDEVICE_write, MYDEVICE_lseek, MYDEVICE_unlink,␣
↪MYDEVICE_rename);
    fid = fopen("mydevice:test","w");
    fprintf(fid,"Hello, world\n");
    fclose(fid);
}
```

3.4.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions)

The C standard assumes only one thread of execution, with the only exception being extremely narrow support for signal handlers. The issue of reentrancy is avoided by not allowing you to do much of anything in a signal handler. However, multi-threaded systems, such as systems that integrate an RTOS, may have multiple threads that need to modify the same global program state, such as the CIO buffer, so reentrancy is a concern.

Part of the problem of reentrancy remains your responsibility, but the run-time-support environment does provide rudimentary support for multi-threaded reentrancy by providing support for critical sections. This implementation does not protect you from reentrancy issues such as calling run-time-support functions from inside interrupts; this remains your responsibility.

The run-time-support environment provides hooks to install critical section primitives. By default, a single-threaded model is assumed, and the critical section primitives are not employed. In a multi-threaded system, the kernel arranges to install semaphore lock primitive functions in these hooks, which are then called when the run-time-support enters code that needs to be protected by a critical section.

Throughout the run-time-support environment where a global state is accessed, and thus needs to be protected with a critical section, there are calls to the function _lock(). This calls the provided primitive, if installed, and acquires the semaphore before proceeding. Once the critical section is finished, _unlock() is called to release the semaphore.

Usually a kernel is responsible for creating and installing the primitives, so you do not need to take any action. However, this mechanism can be used in multi-threaded applications that use other locking mechanisms.

You should not define the functions _lock() and _unlock() functions directly; instead, the installation functions are called to instruct the run-time-support environment to use these new primitives:

```
void _register_lock  (void (  *lock)());

void _register_unlock(void (*unlock)());
```

The arguments to _register_lock() and _register_unlock() should be functions which take no arguments and return no values, and which implement some sort of global semaphore locking:

```
extern volatile sig_atomic_t *sema = SHARED_SEMAPHORE_LOCATION;
static int sema_depth = 0;
static void my_lock(void)
{
    while (ATOMIC_TEST_AND_SET(sema, MY_UNIQUE_ID) != MY_UNIQUE_
 ↪ID);
    sema_depth++;
}
static void my_unlock(void)
{
    if (!--sema_depth) ATOMIC_CLEAR(sema);
}
```

The run-time-support nests calls to _lock(), so the primitives must keep track of the nesting level.

3.5 Processing Assembly Source Code

Contents:

3.5.1 About the tiarmclang Arm Assemblers

The tiarmclang compiler tools provide two different Arm assemblers:

- The **tiarmclang** integrated GNU-syntax assembler can be used to assemble assembly source code written in GNU-syntax. Input files specified on the **tiarmclang** command line with a *.s* or *.S* file extension are assumed to be GNU-syntax Arm assembly source files by default. See *Integrated GNU-Syntax Arm Assembler* and *GNU-Syntax Arm Assembly Language Reference Guide* for more about the integrated GNU-syntax assembler.

- The **tiarmasm** standalone TI-syntax assembler can be used to assemble assembly source code written in TI-syntax. See *TI-Syntax Arm Assembler* for more about the standalone TI-syntax assembler. The **tiarmasm** assembler can also be invoked from the **tiarmclang** command-line using the *-x ti-asm* option. See *Invoking the TI-Syntax Arm Assembler from tiarmclang* for details.

Note: Assembly source code that is embedded in C/C++ source files via *asm()* statements will be processed inline by the **tiarmclang**'s integrated GNU-syntax assembler.

3.5.2 Integrated GNU-Syntax Arm Assembler

The **tiarmclang** compiler tools installation includes a GNU-syntax Arm assembler that is integrated into the **tiarmclang** compiler tool. The **tiarmclang** command can be used to assemble a source file containing GNU-syntax Arm assembly language.

Invoking the Integrated GNU-Syntax Arm Assembler

tiarmclang [*options*] <*gnu asm file*>.S [*filenames*]
tiarmclang [*options*] <*gnu asm file*>.s [*filenames*]

tiarmclang is the command to invoke the integrated GNU-syntax Arm assembler when presented with a GNU-syntax Arm assembly source input file.

options may be a set of command-line options to influence the behavior of the **tiarmclang** compiler and integrated assembler. These are described in the *Compiler Options* section.

`.S` files are GNU-syntax assembly source files that are run through the C preprocessor before presenting the output of the preprocessor to the GNU-syntax Arm assembler. Preprocessing directives in the assembly source file are useful for configuring assembly source content based on predefined macro symbols that are available to the C preprocessor.

`.s` files are GNU-syntax assembly source files that are passed directly to the GNU-syntax Arm assembler for processing.

Files with other file extensions are processed by the **tiarmclang** compiler as usual for C source files.

Additional documentation for the GNU-syntax assembler is provided in this documentation and elsewhere:

- For an overview of GNU Arm assembly syntax, see *GNU-Syntax Arm Assembly Source Anatomy*.

- For links to documentation of assembly instructions for your Arm device, see *GNU-Syntax Arm Assembly Instructions*.

- For documentation of the assembly directives supported with **tiarmclang**, see *GNU-Syntax Arm Assembly Directives*.

- For information about migrating legacy TI-syntax Arm assembly language instructions and directives to GNU-syntax Arm assembly, see *Migrating Assembly Language Source Code*.

- Other documentation of GNU-syntax Arm assembly language is available in GNU and Arm documentation online. The majority of the documented syntax is recognized and processed by **tiarmclang**'s integrated GNU-syntax Arm assembler. Here are some suggested sources:

    – Arm Developer's Instruction Set Architecture

- GNU Assembler - Using as

- GNU Assembler (as) - ARM Dependent Features

3.5.3 TI-Syntax Arm Assembler

tiarmasm is the name of the TI-syntax Arm assembler that is include in the **tiarmclang** compiler tools installation. It is essentially the same TI-syntax Arm assembler that is provided with the legacy TI Arm Code Generation Tools, **armasm**.

For more details about the legacy TI Arm Code Generation Tools support for processing TI-syntax Arm assembly source, please refer to either of the following documents:

- Arm Optimizing C/C++ Compiler User's Guide

- Arm Assembly Language Tools User's Guide

The remainder of this section will provide basic useful information about how to use **tiarmasm**, the standalone TI-syntax Arm assembler, in the context of the TI Arm Clang Compiler Tools.

Invoking the TI-Syntax Arm Assembler from tiarmclang

The tiarmclang compiler can be instructed to process input files with its TI-syntax Arm assembler. The assembly language source files, written using legacy TI-syntax Arm assembly code, can be indicated to the tiarmclang compiler using the "-x *language*" option, where *language* is "ti-asm". This option applies to all subsequent input files, so be sure to reset the input file type if there are non-TI-syntax Arm assembly source files on the command line following the TI-syntax Arm assembly source files.

> **tiarmclang -x ti-asm** <*ti-syntax asm file*> [*options*] [*filenames*]

For example, in the following command:

```
%> tiarmclang c-source1.c -x ti-asm ti-asm-source1.asm -x none c-
↪source2.c
```

the "-x ti-asm" option indicates that the ti-asm-source1.asm file is to be processed by the TI-syntax Arm assembler, and the subsequent "-x none" option resets the input file type to a default state so that the tiarmclang compiler knows to process the c-source2.c input file as a C file.

While many important options that need to be passed to the TI-syntax Arm assembler, such as silicon version and FPU version, can be inferred from the compiler's normal command line options, other options that you might need may be missing or need to be overridden. To support this, two **tiarmclang** options, **-Wti-a,** and **-Xti-assembler**, are supported. These behave similarly to **tiarmclang**'s other *-W* and *-X* options. Please see *Passing Options to Other Tools from tiarmclang* for a more detailed discussion and examples of how to use **tiarmclang**'s *-X* and *-W* options.

Invoking TI-Syntax Arm Assembler Directly

To invoke the TI-syntax Arm assembler, **tiarmasm**, from the command-line, enter the following:

> **tiarmasm** [*options*] [*filenames*]

- **tiarmasm** - Command that invokes the standalone TI-syntax Arm assembler.

- *options* - TI-syntax Arm assembler options. When **tiarmasm** is invoked directly, the assembler options need not be preceded by **tiarmclang**'s *-Xti-assembler* or *-Wti-a,* options.

- *filenames* - TI-syntax Arm assembly source files.

TI-Syntax Arm Assembler Options

As mentioned above, when the TI-syntax Arm assembler is invoked from the **tiarmclang** command-line, options that are intended for the TI-syntax Arm assembler must be preceded by **tiarmclang**'s *-Xti-assembler* or *-Wti-a,* option. In this example:

```
%> tiarmclang ... -x ti-asm tia.asm -Xti-assembler --define=MY_
 ↪PREDEF_SYM=10 ...
```

the *--define=MY_PREDEF_SYM=10* option is passed to the TI-syntax Arm assembler when processing the TI-syntax Arm assembly source file, *tia.asm*.

In the list of TI-syntax Arm assembler options described below, the syntax for each option is presented as if the option were specified when the TI-syntax Arm assembler is invoked directly from the command line.

-d=*<name>*[=*<value>*], **--define**=*<name>*[=*<value>*]

> Define assembly constant symbol <*name*> with <*value*>, if the <*value*> argument is specified. If the <*value*> argument is omitted, <*name*> is given a value of 1.

--depend, --asm_dependency

> Perform preprocessing for assembly files, but instead of writing preprocessed output, write a list of dependency lines suitable for input to a standard make utility. The list is written to a file with the same name as the source file but with a *.ppa* extension.

--includes, --asm_includes

> Perform preprocessing for assembly files, but instead of writing preprocessed output, write a list of files included with the .include directive. The list is written to a file with the same name as the source file but with a *.ppa* extension.

-hc=*<file>*, **--copy_file**=*<file>*

> Include the specified <*file*> for the assembly module. The <*file*> is included before source file statements. The included <*file*> will appear in assembly listing files that are generated by the assembler.

-hi=*<file>*, **--include_file**=*<file>*

> Include the specified *<file>* for the assembly module. The *<file>* is included before source file statements. The included *<file>* will not appear in any assembly listing files that are generated by the assembler.

-l, --asm_listing

> Produce a listing file with the same name as the input file with a *.lst* extension

-u=*<name>*, **--undefine**=*<name>*

> Undefine the predefined symbol constant *<name>*, which overrides any *--define* options for the specified *<name>*.

--thumb_state, -mt

> The *--thumb* option is a synonym for the *--code_state=16* option. Instruct the assembler to begin assembling instructions as 16-bit instructions. This option is off by default.

--code_state=[16|32]

> The *--code_state=16* option instructs the assembler to begin assembling instructions as 16-bit instructions. By default, the assembler begins assembling 32-bit instructions.

--endian=[little|big]

> Produce object code in little-endian or big-endian format as indicated by the option argument. The **tiarmclang** compiler tools only support big-endian object files for Cortex-R series Arm processors.

-i=<*path>*, **--include_path**=*<path>*

> Specify a directory *<path>* where the assembler can find files named by the .copy, .include, or .mlib directives. There is no limit to the number of directories you can specify in this manner; each pathname must be preceded by the *--include_path* option.

-q, --quiet

> Suppress the banner and progress information (assembler runs in quiet mode).

-v=*<cpu>*, **--silicon_version**=*<cpu>*

> Select target Arm processor variant to assemble for. The supported values for the specified *<cpu>* argument include:
>
> - **6M0** - Cortex-M0 processor
>
> - **7M3** - Cortex-M3 processor
>
> - **7M4** - Cortex-M4 processor
>
> - **7R4** - Cortex-R4 processor
>
> - **7R5** - Cortex-R5 processor

Note: The standalone TI-syntax Arm assembler does not currently support the Arm Cortex-m33 processor. Use of Cortex-m33 instructions must be compiled or assembled using the **tiarmclang**

---

compiler and its integrated GNU-syntax Arm assembler.

-@=*<file>*, **--cmd_file**=*<file>*

> Append the contents of a *<file>* to the command line. You can use this option to avoid limitations on command line length imposed by the host operating system.

--float_support=*<fpu>*

> Enable use of floating-point hardware registers and instructions. The supported values for the specified *<fpu>* argument include:
>
> > - **FPv4SPD16** - available in combination with Arm Cortex-M4 processor
> >
> > - **VFPv3D16** - available in combination with Arm Cortex-M4 or Cortex-R5 processor

--unaligned_access=[on|off]

> Enable (on) or disable (off) use of unaligned memory accesses for load and store instructions. Unaligned memory accesses are *not* available on the Arm Cortex-M0 processor.

--no_warnings

> Suppress assembler warnings.

-pdew, **--emit_warnings_as_errors**

> Treat warnings as errors.

-c, **--syms_ignore_case**

> Symbol names are parsed and converted to upper case.

-f, **--suppress_asm_extension**

> Don't assume source file has a *.asm* file extension.

-g, **--symdebug:dwarf**

> Encode symbolic debug information in generated object code.

-h, **--help**, **--usage**, **-?**

> Display help output.

3.6 Cortex-M Security Extensions (CMSE)

The **tiarmclang** compiler tools support the Cortex-M Security Extensions (CMSE) for the Armv8-M architecture, specifically the Arm Cortex-M33 processor.

- Documentation on how to incorporate the use of the CMSE security extensions into an application is provided in the TrustZone(R) technology for Armv8-M Architecture.

- A specification of the security mechanisms supported in the compiler tools can be found in the Armv8_M Security Extensions: Requirements on Development Tools - Engineering Specification.

This section of the user guide provides a tour through the CMSE security extension mechanisms that are supported in the compiler tools with particular attention to how the compiler tools are used to facilitate:

- Accessing code in secure memory from code in non-secure memory, and

- Calling code in non-secure memory from code in secure memory

3.6.1 Memory Partitions

As discussed in Section 3 of the TrustZone(R) technology for Armv8-M Architecture document, in an application that makes use of the security extensions, the memory space accessible to the Cortex-M33 processor is divided into *Secure* and *Non-secure* memory regions.
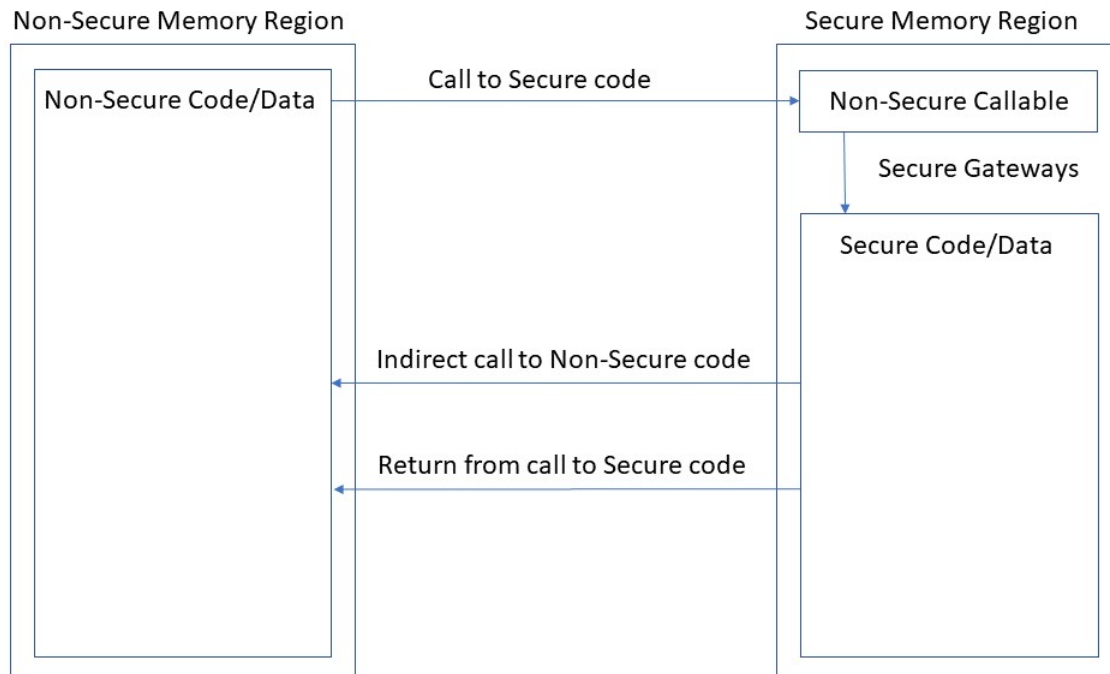
Secure Memory Region

The *Secure Memory Region* contains two partitions:

Secure

Memory in the *Secure* partition of the secure memory region contains code, data, and peripherals that are only accessible from secure software or secure masters.

Non-secure Callable

The *Non-secure Callable* partition of the secure memory region is the only area of target memory that may contain Secure Gateway (SG) instructions that facilitate transitions from Non-secure code

state to Secure code state. The table of linker-generated secure gateway veneers that is allocated to the *Non-secure Callable* partition constitutes the only means whereby a function defined in the *Non-secure Memory Region* can gain entry to a function defined in the *Secure* partition.

Each secure gateway veneer in the table begins with an SG instruction that changes the processor state from Non-secure to Secure, followed by a branch to the definition of the secure function being called. Thus, the actual address is only known to code that resides in the *Secure Memory Region*.

Non-secure Memory Region

Memory in the *Non-secure Memory Region* contains code, data, and peripherals that are accessible from all software running on the Arm processor. Code that resides in non-secure memory may only access code and data that is defined in non-secure memory. Code defined in non-secure memory may only call a function defined in secure memory via the secure gateway veneer that is associated with that secure function.

3.6.2 Enabling CMSE Support

The Cortex-M Security Extensions (CMSE) are only available when building an application to be run on the Arm Cortex-M33 processor. Code that makes use of CMSE support mechanisms must #include the *<arm_cmse.h>* header file prior to the use of any CMSE intrinsics in a compilation unit.

To enable CMSE support attributes and pre-defined macro symbols in the **tiarmclang** compiler, the following options should be used:

```
-mcpu=cortex-m33 -mcmse
```

3.6.3 Example Application

To explore how different aspects of the CMSE support mechanisms work in the **tiarmclang** tool chain, consider a simple example application where:

- Secure code on the Arm processor provides 3 functions:

    - **set_event_handler()** - registers the address of an event handler function that is to be called when an event occurs.

    - **wait_on_event()** - polls a secure peripheral for the occurrence of an event, and when the event occurs, will call the event handler function that is currently registered.

    - **default_event_handler()** - in the case where a custom event handler function has not been registered, the *default_event_handler()* function will get called when an event occurs

- The application running in non-secure memory relies on secure code to monitor a secure peripheral that is used to trigger an action to be performed by the non-secure application.

Defining Non-secure Entry Functions

An *non-secure entry* function in this context refers to a function defined in secure memory that can be called from non-secure code via a secure gateway veneer. Such a function must be annotated with a *cmse_nonsecure_entry* attribute.

For example, the definition of *set_event_handler()* could be written as follows:

```c
#include <arm_cmse.h>

typedef void __attribute__((cmse_nonsecure_call)) nsfunc(void);
extern int receive_signal(void);
extern void default_event_handler(void);


nsfunc *callback_fcn = (nsfunc *)default_event_handler;


__attribute__((cmse_nonsecure_entry)) void set_event_
→handler(nsfunc *fp)
{
  if (fp)
    callback_fcn = cmse_nsfptr_create(fp);
  else
    callback_fcn = (nsfunc *)default_event_handler;
}


__attribute__((cmse_nonsecure_entry)) void wait_on_event()
{
  while (!receive_signal()) ;

  if (cmse_is_nsfptr(callback_fcn))
    callback_fcn();
  else
    ((void (*)(void))callback_fcn)();
}
```

Compiling the source file with **tiarmclang** as follows:

```
%> tiarmclang -mcpu=cortex-m33 -mcmse -S secure_code.c
```

Yields the following compiler-generated assembly definition of the *set_event_handler()* function:

```
@ Definition of set_event_handler function
        .section        .text.set_event_handler,"ax",%progbits
        .hidden set_event_handler                  @ -- Begin␣
↪function
        .globl  set_event_handler
        .p2align        1
        .type   set_event_handler,%function
        .code   16                                  @ @set_event_
↪handler
        .thumb_func
        .globl  __acle_se_set_event_handler
        .type   __acle_se_set_event_handler,%function
__acle_se_set_event_handler:
set_event_handler:
        .pad    #4
        sub     sp, #4
        str     r0, [sp]
        ldr     r0, [sp]
        cbz     r0, .LBB0_2
        b       .LBB0_1
.LBB0_1:                                            @ %if.then
        ldr     r0, [sp]
        movw    r1, :lower16:callback_fcn
        movt    r1, :upper16:callback_fcn
        str     r0, [r1]
        b       .LBB0_3
.LBB0_2:
        movw    r1, :lower16:callback_fcn
        movt    r1, :upper16:callback_fcn
        movw    r0, :lower16:default_event_handler
        movt    r0, :upper16:default_event_handler
        str     r0, [r1]
        b       .LBB0_3
.LBB0_3:                                            @ %if.end
        add     sp, #4
        mrs     r12, control
        tst.w   r12, #8
        beq     .LBB0_5

        vmov    d0, lr, lr
        vmov    d1, lr, lr
        vmov    d2, lr, lr
        vmov    d3, lr, lr
```

(continues on next page)

```
        vmov    d4, lr, lr
        vmov    d5, lr, lr
        vmov    d6, lr, lr
        vmov    d7, lr, lr
        vmrs    r12, fpscr
        bic     r12, r12, #159
        bic     r12, r12, #4026531840
        vmsr    fpscr, r12
.LBB0_5:                                        @ %if.end
        mov     r0, lr
        mov     r1, lr
        mov     r2, lr
        mov     r3, lr
        mov     r12, lr
        msr     apsr_nzcvqg, lr
        bxns    lr
```

The application of the *cmse_nonsecure_entry* attribute to the definition of *set_default_handler()* has a few notable impacts on the above compiler-generated code:

- In addition to the normal *set_event_handler* label to mark the start of the function, the compiler also defines *__acle_se_set_event_handler* at the same address. When the program is linked, the label *set_even_handler* will get redefined to the address of the first instruction in the secure gateway veneer that is generated by the linker for *set_event_handler()*. The definition of the *__acle_se_event_handler* label will remain at the location of the actual function's first instruction. See *Building a Secure Image and Creating an Import Library* for more details about what happens at link-time.

- The compiler generates code to clear the contents of caller-saved registers (including FPU registers) except those that hold the result value and the return address of the entry function.

- The compiler generates code to clear registers and flags that have undefined values at the return of a procedure, according to the Arm procedure call standard.

- The compiler generates code to save and restore callee-saved registers as dictated by the Arm procedure call standard.

- The compiler generates a *BXNS* instruction to return to its caller. If the function was called from a non-secure function, then the *BXNS* instruction will change the state of the processor from Secure to Non-secure before returning to the non-secure caller function.

Calling Non-secure Functions

A function defined in non-secure memory may only be called from code in secure memory via a function pointer that has been annotated with a *cmse_nonsecure_call* attribute. Such a function pointer is referred to as a *non-secure function pointer*.

In the imagined CMSE example application under consideration, the non-secure application code relies on the *wait_on_event()* function that is defined in secure memory to monitor a secure peripheral for a signal and then call an event handler function. The definition of *wait_on_event()* must account for the possibility that a non-secure function has been registered as an event handler that must be called via a *non-secure function pointer*.

Here is a possible implementation of the *wait_on_event()* function:

```c
#include <arm_cmse.h>

typedef void __attribute__((cmse_nonsecure_call)) nsfunc(void);
extern int receive_signal(void);
extern void default_event_handler(void);

nsfunc *callback_fcn = (nsfunc *)default_event_handler;

__attribute__((cmse_nonsecure_entry)) void set_event_
↪handler(nsfunc *fp)
{
  if (fp)
    callback_fcn = cmse_nsfptr_create(fp);
  else
    callback_fcn = (nsfunc *)default_event_handler;
}

__attribute__((cmse_nonsecure_entry)) void wait_on_event()
{
  while (!receive_signal()) ;

  if (cmse_is_nsfptr(callback_fcn))
    callback_fcn();
  else
    ((void (*)(void))callback_fcn)();
}
```

Things to note about the above C code:

- The *set_event_handler()* function uses the *cmse_nsfptr_create* intrinsic that is defined in *<arm_cmse.h>* to convert the address of the non-secure function pointer argument to a value

that can be recognized as a non-secure function address by clearing the least significant bit of *fp* as its address is assigned to *callback_fcn*. Note that all functions in the application are Thumb functions which have their least significant bit set to indicate that they are thumb mode functions. The CMSE ABI exploits this fact and repurposes the least significant bit of a function address to enable secure code to recognize it as a non-secure function.

- The *wait_on_event()* function is also a non-secure entry function as indicated via application of the *cmse_nonsecure_entry* attribute.

- *wait_on_event()* uses the *cmse_is_nsfptr* intrinsic to recognize a function as being defined in non-secure memory. The intrinsic will recognize a function pointer as a non-secure function if the least significant bit of the function pointer value is zero.

The following code is generated by the compiler for the definition of *wait_on_event()*:

```
@ Generated code for wait_for_event function

        .section        .text.wait_on_event,"ax",%progbits
        .hidden wait_on_event                        @ -- Begin
→function wait_on_event
        .globl  wait_on_event
        .p2align        1
        .type   wait_on_event,%function
        .code   16                                   @ @wait_on_event
        .thumb_func
        .globl  __acle_se_wait_on_event
        .type   __acle_se_wait_on_event,%function
__acle_se_wait_on_event:
wait_on_event:
        push    {r7, lr}
        b       .LBB1_1
.LBB1_1:                                             @ %while.cond
                                                     @ =>This Inner Loop
→Header: Depth=1
        bl      receive_signal
        cbnz    r0, .LBB1_3
        b       .LBB1_2
.LBB1_2:                                             @ %while.body
                                                     @   in Loop: Header=BB1_
→1 Depth=1
        b       .LBB1_1
.LBB1_3:                                             @ %while.end
        movw    r0, :lower16:callback_fcn
        movt    r0, :upper16:callback_fcn
        ldrb    r0, [r0]
        lsls    r0, r0, #31
```

(continues on next page)

```
        cbnz     r0, .LBB1_5
        b        .LBB1_4
.LBB1_4:                                    @ %if.then
        movw     r0, :lower16:callback_fcn
        movt     r0, :upper16:callback_fcn
        ldr      r0, [r0]
        push.w   {r4, r5, r6, r7, r8, r9, r10, r11}
        bic      r0, r0, #1
        sub      sp, #136
        vlstm    sp
        mov      r1, r0
        mov      r2, r0
        mov      r3, r0
        mov      r4, r0
        mov      r5, r0
        mov      r6, r0
        mov      r7, r0
        mov      r8, r0
        mov      r9, r0
        mov      r10, r0
        mov      r11, r0
        mov      r12, r0
        msr      apsr_nzcvqg, r0
        blxns    r0
        vlldm    sp
        add      sp, #136
        pop.w    {r4, r5, r6, r7, r8, r9, r10, r11}
        b        .LBB1_6
.LBB1_5:                                    @ %if.else
        movw     r0, :lower16:callback_fcn
        movt     r0, :upper16:callback_fcn
        ldr      r0, [r0]
        blx      r0
        b        .LBB1_6
.LBB1_6:                                    @ %if.end
        pop.w    {r7, lr}
        mrs      r12, control
        tst.w    r12, #8
        beq      .LBB1_8

        vmov     d0, lr, lr
        vmov     d1, lr, lr
```

```
        vmov    d2, lr, lr
        vmov    d3, lr, lr
        vmov    d4, lr, lr
        vmov    d5, lr, lr
        vmov    d6, lr, lr
        vmov    d7, lr, lr
        vmrs    r12, fpscr
        bic     r12, r12, #159
        bic     r12, r12, #4026531840
        vmsr    fpscr, r12
.LBB1_8:                                @ %if.end
        mov     r0, lr
        mov     r1, lr
        mov     r2, lr
        mov     r3, lr
        mov     r12, lr
        msr     apsr_nzcvqg, lr
        bxns    lr
```

Some things to note about the compiler-generated code:

- Since *wait_on_event()* is also a non-secure entry function, it is impacted by the application of the *cmse_nonsecure_entry* attribute in the same way that *set_event_handler()* is. Namely,

    - An *__acle_se_wait_on_event* label is defined at the function entry point in addition to the normal *wait_on_event* label.

    - The compiler generates code to clear the contents of caller-saved registers.

    - The compiler generates code to clear registers and flags that have undefined values at the return of a procedure, according to the Arm procedure call standard.

    - The compiler generates code to save and restore callee-saved registers as dictated by the Arm procedure call standard.

    - The compiler generates a *BXNS* instruction to return to its caller.

- The effect of calling a function via a *non-secure function pointer* is:

    - The compiler will generate code to save all callee- and live caller-saved registers in secure memory.

    - The compiler will generate code to clear all callee- and caller-saved registers except:

        * The link register (lr),

        * Registers that hold the arguments of the function being called, and

        * Registers that do not hold secret information.

- The compiler generates code to clear registers and flags that have undefined values at the entry to a procedure, according to the Arm procedure call standard.

- The compiler generates a *BLXNS* instruction to effect an indirect call to the non-secure callback function.

Building the Example Application

An application that utilizes CMSE mechanisms will be built in two parts: the non-secure executable application and the secure image that the non-secure application can call into via a secure gateway to access data and functionality that is strictly managed by code in secure memory.

Using the above described example application as a model, this section describes first how to create a secure image, generate a table of secure gateway veneers, and produce an import library. Then how to incorporate the import library into the build of the non-secure executable application.

Building a Secure Image and Creating an Import Library

Recall that some non-secure entry functions have been defined in *secure_code.c*:

```c
#include <arm_cmse.h>

typedef void __attribute__((cmse_nonsecure_call)) nsfunc(void);
extern int receive_signal(void);
extern void default_event_handler(void);


nsfunc *callback_fcn = (nsfunc *)default_event_handler;

__attribute__((cmse_nonsecure_entry)) void set_event_
→handler(nsfunc *fp)
{
  if (fp)
    callback_fcn = cmse_nsfptr_create(fp);
  else
    callback_fcn = (nsfunc *)default_event_handler;
}

__attribute__((cmse_nonsecure_entry)) void wait_on_event()
{
  while (!receive_signal()) ;

  if (cmse_is_nsfptr(callback_fcn))
```

(continues on next page)

```
    callback_fcn();
  else
    ((void (*)(void))callback_fcn)();
}
```

In addition, dummy definitions of *main*, *default_event_handler()* and *receive_signal()* are provided in *more_secure_code.c*:

```c
#include <stdio.h>

int main() {
  printf("dummy entry  point for secure image\n");
  return 0;
}


void default_event_handler(void) {
  printf("Default event handler invoked\n");
}


__attribute__((optnone)) int receive_signal() {
  int i = 20;
  while (i--) {
    asm(" ");
  }

  return 1;
}
```

The secure image can then be compiled and linked as follows:

```
%> tiarmclang -mcpu=cortex-m33 -mcmse secure_code.c more_secure_
↪code.c \
   -o secure.out -Xlinker -lsecure_lnk.cmd -Xlinker --import_
↪cmse_lib_out=implib.o
```

In the above command, *secure.out* contains definitions of the secure functions *set_event_handler()* and *wait_on_event()* and their addresses are indicated by the values of the *__acle_se_set_default_handler* and *__acle_se_wait_on_event* labels in the following disassembly output:

```
%> tiarmdis secure.out secure.dis
%> cat secure.dis
TEXT Section .text, 0x10bc bytes at 0x00000020
```

3.6. Cortex-M Security Extensions (CMSE)   **431**

```
...
0405c8:                 __acle_se_wait_on_event:
0405c8:                  .thumb
0405c8:                  :
0405c8: 80B5             PUSH                {R7, LR}
0405ca: FFE7             B                   0x000405CC
0405cc: 00F0FAFC         BL                  receive_signal [0x40fc4]
0405d0: 08B9             CBNZ                R0, 0x000405D6
0405d2: FFE7             B                   0x000405D4
...
04070c:                 __acle_se_set_event_handler:
04070c:                  .thumb
04070c:                  :
04070c: 82B0             ADD                 SP, #-8
04070e: 0190             STR                 R0, [SP, #4]
040710: 0198             LDR                 R0, [SP, #4]
040712: 58B1             CBZ                 R0, 0x0004072C
040714: FFE7             B                   0x00040716
...
```

During the link step of the above **tiarmclang** command, the linker will generate a table of secure gateway veneers:

```
%> cat secure.dis
...
TEXT Section Veneer$$CMSE, 0x10 bytes at 0x00500000
050000:                 set_event_handler:
050000:                  .thumb
050000:                 Veneer$$CMSE:
050000: 7FE97FE9         SG
050004: FEF7EEBA         B.W                 __acle_se_set_event_
 ↪handler [0x72c]
050008:                 wait_on_event:
050008:                  .thumb
050008: 7FE97FE9         SG
05000c: FEF748BA         B.W                 __acle_se_wait_on_event␣
 ↪[0x5e8]
...
```

Note that the linker has redefined the values of the *set_event_handler* and *wait_on_event* labels to the first instruction in the secure gateway veneers for the *set_event_handler()* and *wait_on_event()* functions, respectively.

At link-time, the linker-generated secure gateway veneer functions should be allocated into *non-*

secure callable memory. For the purposes of this discussion, it is assumed that non-secure callable memory starts at address 0x050000. The *secure_lnk.cmd* linker command file used in the above **tiarmclang** command will instruct the linker to place the Veneer$$CMSE section in the non-secure callable memory area:

```
/
↪********************************************************************
↪
/* secure_lnk.cmd                                                  ␣
↪         */
/*                                                                 ␣
↪         */
/* Set aside some target memory for secure and non-secure␣
↪callable areas    */
/* for made-up secure image example in tiarmclang user guide.  ␣
↪         */
/*                                                                 ␣
↪         */
/
↪********************************************************************
↪
-c
-stack  0x8000
-heap   0x2000
--args 0x1000

/* SPECIFY THE SECURE AND NON-SECURE CALLABLE MEMORY AREAS */

MEMORY
{
    SEC_DMEM    : org = 0x0030000   len = 0x10000
    SEC_PMEM    : org = 0x0040000   len = 0x10000
    NSC_MEM     : org = 0x0050000   len = 0x00100
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    .intvecs    : {} > 0x0              /* INTERRUPT VECTORS   ␣
↪         */
    .bss        : {} > SEC_DMEM         /* GLOBAL & STATIC VARS ␣
↪         */
    .data       : {} > SEC_DMEM
```

(continues on next page)

```
    .sysmem      : {} > SEC_DMEM           /* DYNAMIC MEMORY␣
↪ALLOCATION AREA    */
    .stack       : {} > SEC_DMEM           /* SOFTWARE SYSTEM STACK ␣
↪            */

    .text        : {} > SEC_PMEM           /* CODE                ␣
↪            */
    .cinit       : {} > SEC_PMEM           /* INITIALIZATION TABLES ␣
↪            */
    .const       : {} > SEC_PMEM           /* CONSTANT DATA       ␣
↪            */
    .rodata      : {} > SEC_PMEM

    Veneer$$CMSE: {} > NSC_MEM
}
```

Note: The definition of the non-secure callable memory area must be within range of the secure code memory area so that the 24-bit PC-relative branch in each secure gateway veneer can reach its destination without the need for a linker-generated trampoline.

Finally, the *--import_cmse_lib_out=implib.o* option in the above **tiarmclang** command creates an *import library*. An *import library* is simply a linker generated object file that contains a list of absolute symbols that constitute a mapping of each non-secure entry function symbol to its associated secure gateway veneer location in non-secure callable memory.

In the CMSE example under consideration, the linker generated *implib.o* file contains the following list of absolute symbols:

```
%> tiarmofd -v -o implib.ofd implib.o
%> cat implib.ofd
OBJECT FILE:  implib.o
...
 Symbol Table ".symtab"

  ...

  <1> "set_event_handler"
    Value:    0x00050001  Kind:       absolute
    Binding:  global      Type:       function
    Size:     0x0         Visibility: STV_DEFAULT

  <2> "wait_on_event"
```

```
     Value:    0x00050009  Kind:        absolute
     Binding:  global       Type:        function
     Size:     0x0          Visibility:  STV_DEFAULT
...
```

The values assigned to *set_event_handler* and *wait_on_event* correspond to the locations of their secure gateway veneers in non-secure callable memory. When the linker incorporates the *implib.o* object file into the link of the non-secure executable application, references to *set_event_handler* and *wait_on_event* will resolve to these addresses as though they were normal thumb mode functions.

Building/Linking Non-secure Application with an Import Library

When building a non-secure application that relies on functionality provided in the secure code, the non-secure application should include an *import library* object file that was generated during the build of the secure image.

To conclude the CMSE example that is under consideration, assume that we have a simple implementation of a non-secure application that references the *set_event_handler()* and *wait_on_event()* functions that were defined in the *secure.out* image created in the above discussion.

```c
#include <stdio.h>

extern void set_event_handler(void (*fp)(void));
extern void wait_on_event(void);

void my_event_handler(void);

int main() {
  wait_on_event();

  set_event_handler(my_event_handler);
  wait_on_event();

  return 0;
}


void my_event_handler(void) {
  printf("My event handler has been invoked\n");
}
```

The non-secure application can then be compiled and linked with the following **tiarmclang** com-

mand:

```
%> tiarmclang -mcpu=cortex-m33 non_secure_app.c implib.o -o non_
↪secure.out \
    -Xlinker non_secure_lnk.cmd
```

An examination of the disassembler output for *non_secure.out* shows the definition of *main()* has resolved its references to *set_event_handler()* and *wait_on_event()* to the correct locations in the *Veneer$$CMSE* section of the secure image:

```
%> tiarmdis non_secure.out non_secure.dis
%> cat non_secure.dis
...
060e30:                  main:
060e30:                   .thumb
060e30:                  :
060e30: 80B5             PUSH            {R7, LR}
060e32: 82B0             ADD             SP, #-8
060e34: 0020             MOVS            R0, #0
060e36: 0090             STR             R0, [SP]
060e38: 0190             STR             R0, [SP, #4]
060e3a: EFF7E5F8         BL              wait_on_event [0x50008]
060e3e: 40F6F160         MOVW.W          R0, #3825
060e42: C0F20600         MOVT.W          R0, #6
060e46: EFF7DBF8         BL              set_event_handler␣
↪[0x50000]
060e4a: EFF7DDF8         BL              wait_on_event [0x50008]
060e4e: 0098             LDR             R0, [SP]
060e50: 02B0             ADD             SP, #8
060e52: 80BD             POP             {R7, PC}
```

When *non_secure.out* is loaded and run on a system where the *secure.out* has already been loaded, the first call to *wait_on_event()* will trigger an indirect call to *default_event_handler()* and the second call to *wait_on_event()* will trigger an indirect call back from the secure code to *my_event_handler()*.

Summary of CMSE Mechanisms

Pre-Defined Macro Symbols

__ARM_FEATURE_CMSE

> The __ARM_FEATURE_CMSE pre-defined macro symbol describes what CMSE extensions are available in a given compilation. The value associated with __ARM_FEATURE_CMSE is a bit-field where each bit is defined as follows:

Bit Index	Meaning
0	if set, support for TT instructions is available
1	if set, tiarmclang targets the secure state of CMSE

The value of __ARM_FEATURE_CMSE is defined according to what compiler options are specified on the tiarmclang command-line, as described below.

Compiler Options

-mcpu=cortex-m33

Instructs the tiarmclang compiler to target the Arm Cortex-M33 processor. In addition to other pre-defined macro symbols that are relevant to the Cortex-M33 processor, the tiarmclang compiler will also set bit 0 of the __ARM_FEATURE_CMSE symbol's value when the *-mcpu=cortex-m33* option is specified.

-mcmse

The *-mcmse* option can only be used in combination with the *-mcpu=cortex-m33* and will set bit 1 of the __ARM_FEATURE_CMSE symbol's value to indicate that the tiarmclang compiler should target the secure state of CMSE.

The use of this option instructs the tiarmclang compiler to enable the use of CMSE mechanisms that are only relevant for secure state code, like the *cmse_nonsecure_entry* and *cmse_nonsecure_call* attributes, for example.

Attributes

cmse_nonsecure_entry

Syntax

```
__attribute__((cmse_nonsecure_entry))
```

Description

The *cmse_nonsecure_entry* attribute can be applied to a function defined in secure to designate the function as a *non-secure entry function*. This will instruct the compiler to generate code to clear registers and flags that may contain secret information prior to returning to a non-secure caller.

cmse_nonsecure_call

Syntax

```
__attribute__((cmse_nonsecure_call))
```

Description

The *cmse_nonsecure_call* attribute can be applied to a function pointer type to designate that pointer type as a *non-secure function pointer*. This will instruct the compiler to generate a *BLXNS* instruction to effect a call to non-secure code via that function pointer type. In

addition, the compiler will generate code prior to the *BLXNS* instruction to preserve the state of callee- and caller-saved registers and prevent leakage of secret information.

Non-secure Function Pointer Intrinsics

cmse_nsfptr_create

Signature

The *cmse_nsfptr_create* intrinsic is implemented as a macro in *<arm_cmse.h>*:

```
#define cmse_nsfptr_create(p) ((typeof(p))((intptr_t)(p) & ~
↪1))
```

Description

The *cmse_nsfptr_create()* macro will clear the least significant bit of a pointer to a thumb mode function. The pointer value can then be checked via the *cmse_is_nsfptr()* macro to determine whether the function pointer is to be interpreted as the address of a non-secure function.

cmse_is_nsfptr

Signature

The *cmse_is_nsfptr* intrinsic is implemented as a macro in *<arm_cmse.h>*:

```
#define cmse_is_nsfptr(p) (!((intptr_t)(p) & 1))
```

Description

The *cmse_is_nsfptr()* macro can be used to test whether a given function pointer value should be interpreted as a non-secure function address. When such a function pointer is created by the *cmse_nsfptr_create()* macro, the least significant bit of its value will be cleared.

3.7 Introduction to Object Modules

The assembler creates object modules from assembly code, and the linker creates executable object files from object modules. These executable object files can be executed by an Arm device.

Object modules make modular programming easier because they encourage you to think in terms of *blocks* of code and data when you write an assembly language program. These blocks are known as sections. Both the assembler and the linker provide directives that allow you to create and manipulate sections.

This chapter focuses on the concept and use of sections in assembly language programs.

3.7.1 Object File Format Specifications

The object files created by the assembler and linker conform to the ELF (Executable and Linking Format) binary format, which is used by the Embedded Application Binary Interface (EABI). The complete Arm ABI specifications can be found in the Application Binary Interface for the Arm Architecture - The Base Standard.

The ELF object files generated by the assembler and linker conform to the December 17, 2003 snapshot of the System V generic ABI (or gABI). This specification is currently maintained by SCO.

3.7.2 Executable Object Files

The linker produces executable object modules. An executable object module has the same format as object files that are used as linker input. The sections in an executable object module, however, have been combined and placed in target memory, and the relocations are all resolved.

To run a program, the data in the executable object module must be transferred, or loaded, into target system memory. See *Program Loading and Running* for details about loading and running programs.

3.7.3 Introduction to Sections

The smallest unit of an object file is a *section*. A section is a block of code or data that occupies contiguous space in the memory map. Each section of an object file is separate and distinct.

ELF format executable object files contain *segments*. An ELF segment is a meta-section. It represents a contiguous region of target memory. It is a collection of *sections* that have the same property, such as writeable or readable. An ELF loader needs the segment information, but does not need the section information. The ELF standard allows the linker to omit ELF section information entirely from the executable object file.

Object files usually contain three default sections:

.text section	Contains executable code
.data section	Usually contains initialized data
.bss	Usually reserves space for uninitialized variables

Some targets allow content other than text, such as constants, in .text sections.

The assembler and linker allow you to create, name, and link other kinds of sections. The .text, .data, and .bss sections are archetypes for how sections are handled.

There are two basic types of sections:

- **Initialized sections:** Contain data or code. The .text and .data sections are initialized; user-named sections created with the .sect assembler directive are also initialized.

- **Uninitialized sections:** Reserve space in the memory map for uninitialized data. The .bss section is uninitialized; user-named sections created with the .usect assembler directive are also uninitialized.

Several assembler directives allow you to associate various portions of code and data with the appropriate sections. The assembler builds these sections during the assembly process, creating an object file organized as shown in the figure below.

One of the linker's functions is to relocate sections into the target system's memory map; this function is called *placement*. Because most systems contain several types of memory, using sections can help you use target memory more efficiently. All sections are independently relocatable; you can place any section into any allocated block of target memory. For example, you can define a section that contains an initialization routine and then allocate the routine in a portion of the memory map that contains ROM. For information on section placement, see *Section Allocation and Placement*.

The figure below shows the relationship between sections in an object file and a hypothetical target memory. ROM may be EEPROM, FLASH or some other type of physical memory in an actual system.
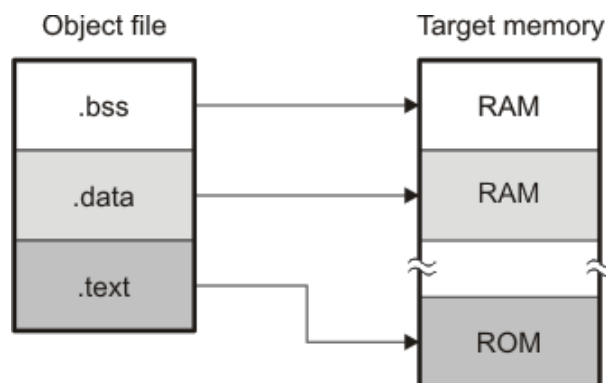
Figure 3.16: Sections in Object File and in Memory

Special Section Names

You can use the .sect and .usect directives to create any section name you like, but certain sections are treated in a special manner by the linker and the compiler's run-time support library. If you create a section with the same name as a special section, you should take care to follow the rules for that special section.

A few common special sections are:

- .text – Used for program code.

- .data – Used for initialized non-const objects (global variables).

- .bss – Used for uninitialized objects (global variables).

- .const – Used for initialized const objects (variables declared const).

- .rodata – Used for initialized const objects (string constants).

- .cinit – Used to initialize C global variables at startup.

- .stack – Used for the function call stack.

- .sysmem – Used for the dynamic memory allocation pool.

For more information on sections, see *Section Allocation and Placement*.

3.7.4 How the Assembler Handles Sections

The assembler identifies the portions of an assembly language program that belong in a given section. The assembler has the following directives that support this function:

- .bss

- .data

- .sect

- .text

- .usect

The .bss and .usect directives create *uninitialized sections*; the .text, .data, and .sect directives create *initialized sections*.

You can create subsections of any section to give you tighter control of the memory map. Subsections are created using the .sect and .usect directives. Subsections are identified with the base section name and a subsection name separated by a colon; see *Subsections*.

Note: If you do not use a section directive, the assembler assembles everything into the .text section.

Uninitialized Sections

Uninitialized sections reserve space in Arm memory; they are usually placed in RAM. These sections have no actual contents in the object file; they simply reserve memory. A program can use this space at run time for creating and storing variables.

Uninitialized data areas are built by using the following assembler directives.

- The .bss directive reserves space in the .bss section.

- The .usect directive reserves space in a specific uninitialized user-named section.

Each time you invoke the .bss or .usect directive, the assembler reserves additional space in the .bss or the user-named section. The syntax is:

	.bss *symbol*,*size in bytes*[**,** *alignment* [**,** *bank offset*]]
symbol	**.usect** "*section name*",*size in bytes*[**,** *alignment*[**,** *bank offset*]]

- *symbol* points to the first byte reserved by this invocation of the .bss or .usect directive. The *symbol* corresponds to the name of the variable for which you are reserving space. It can be referenced by any other section and can also be declared as a global symbol (with the .global directive).

- *size in bytes* is an absolute expression. The .bss directive reserves *size in bytes* bytes in the .bss section. The .usect directive reserves *size in bytes* bytes in *section name*. For both directives, you must specify a size; there is no default value.

- *alignment* is an optional parameter. It specifies the minimum alignment in bytes required by the space allocated. The default value is byte aligned; this option is represented by the value 1. The value must be a power of 2.

- *bank offset* is an optional parameter. It ensures that the space allocated to the symbol occurs on a specific memory bank boundary. The *bank offset* measures the number of bytes to offset from the alignment specified before assigning the symbol to that location.

- *section name* specifies the user-named section in which to reserve space. See *User-Named Sections*.

Initialized section directives (.text, .data, and .sect) change which section is considered the *current* section (see *Current Section*). However, the .bss and .usect directives *do not* change the current section; they simply escape from the current section temporarily. Immediately after a .bss or .usect directive, the assembler resumes assembling into whatever the current section was before the directive. The .bss and .usect directives can appear anywhere in an initialized section without affecting its contents. For an example, see *Using Sections Directives*.

The .usect directive can also be used to create uninitialized subsections. See *Subsections* for more information on creating subsections.

The .common directive is similar to directives that create uninitialized data sections, except that common symbols are created by the linker instead.

Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in Arm memory when the program is loaded. Each initialized section is independently relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these references. The following directives tell the assembler to place code or data into a section. The syntaxes for these directives are:

.text
.data
.sect "*section name*"

The .sect directive can also be used to create initialized subsections. See *Subsections*, for more information on creating subsections.

User-Named Sections

User-named sections are sections that *you* create. You can use them like the default .text, .data, and .bss sections, but each section with a distinct name is kept distinct during assembly.

For example, repeated use of the .text directive builds up a single .text section in the object file. This .text section is allocated in memory as a single unit. Suppose there is a portion of executable code (perhaps an initialization routine) that you want the linker to place in a different location than the rest of .text. If you assemble this segment of code into a user-named section, it is assembled separately from .text, and you can use the linker to allocate it into memory separately. You can also assemble initialized data that is separate from the .data section, and you can reserve space for uninitialized variables that is separate from the .bss section.

These directives let you create user-named sections:

- The **.usect** directive creates uninitialized sections that are used like the .bss section. These sections reserve space in RAM for variables.

- The **.sect** directive creates initialized sections, like the default .text and .data sections, that can contain code or data. The .sect directive creates user-named sections with relocatable addresses.

The syntaxes for these directives are:

symbol	**.usect** "*section name*", *size in bytes*[, *alignment*[, *bank offset*]]
	.sect "*section name*"

The maximum number of sections is 2^{32}-1 (4294967295).

The *section name* parameter is the name of the section. For the .usect and .sect directives, a section name can refer to a subsection; see *Subsections* for details.

Each time you invoke one of these directives with a new name, you create a new user-named section. Each time you invoke one of these directives with a name that was already used, the assembler resumes assembling code or data (or reserves space) into the section with that name. *You cannot use the same names with different directives*. That is, you cannot create a section with the .usect directive and then try to use the same section with .sect.

Current Section

The assembler adds code or data to one section at a time. The section the assembler is currently filling is the *current section*. The .text, .data, and .sect directives change which section is considered the current section. When the assembler encounters one of these directives, it stops assembling into the current section (acting as an implied end of current section command). The assembler sets the designated section as the current section and assembles subsequent code into the designated section until it encounters another .text, .data, or .sect directive.

If one of these directives sets the current section to a section that already has code or data in it from earlier in the file, the assembler resumes adding to the end of that section. The assembler generates only one contiguous section for each given section name. This section is formed by concatenating all of the code or data which was placed in that section.

Section Program Counters

The assembler maintains a separate program counter for each section. These program counters are known as *section program counters*, or *SPCs*.

An SPC represents the current address within a section of code or data. Initially, the assembler sets each SPC to 0. As the assembler fills a section with code or data, it increments the appropriate SPC. If you resume assembling into a section, the assembler remembers the appropriate SPC's previous value and continues incrementing the SPC from that value.

The assembler treats each section as if it began at address 0; the linker relocates the symbols in each section according to the final address of the section in which that symbol is defined. See *Symbolic Relocations* for information on relocation.

Subsections

A subsection is created by creating a section with a colon or period in its name. Subsections are logical subdivisions of larger sections. Subsections are themselves sections and can be manipulated by the assembler and linker.

The assembler has no internal concept of subsections; to the assembler, a colon or period in the name is not special. Subsections named .text:rts and .text.rts are different sections and are considered completely unrelated to the parent section .text. The assembler does not combine such subsections with their parent sections.

In contrast, the linker recognizes both colons and periods as subsection delimiters. To the linker, both .text:rts and .text.rts reference the same subsection of the .text section. See *Using Multi-Level Subsections*.

Subsections are used to keep parts of a section as distinct sections so that they can be separately manipulated. For instance, by placing each function and object in a uniquely-named subsection, the linker gets a finer-grained view of the section for memory placement and unused-function elimination.

By default, when the linker sees a SECTION directive in the linker command file like ".text", it gathers .text and all subsections of .text into one large output section named ".text". You can instead use the SECTION directive to control the subsection independently. See *SECTIONS Directive Syntax* for an example.

You can create subsections in the same way you create other user-named sections: by using the .sect or .usect directive.

The syntaxes for a subsection name are:

symbol	**.usect** "*section_name:subsection_name*", *size in bytes*[, *alignment*[, *bank offset*]]
	.sect "*section_name:subsection_name*"

A subsection is identified by the base section name followed by a colon or period and the name of the subsection. The subsection name may not contain any spaces.

A subsection can be allocated separately or grouped with other sections using the same base name. For example, you create a subsection called _func within the .text section:

```
.sect ".text:_func"
```

Using the linker's SECTIONS directive, you can allocate .text:_func separately, or with all the .text sections.

You can create two types of subsections:

- Uninitialized subsections are created using the .usect directive. See *Uninitialized Sections*.

- Initialized subsections are created using the .sect directive. See *Initialized Sections*.

Subsections are placed in the same manner as sections. See *The SECTIONS Directive* for information on the SECTIONS directive.

Using Sections Directives

The example below shows how you can build sections incrementally, using the sections directives to swap back and forth between the different sections. You can use sections directives to begin assembling into a section for the first time, or to continue assembling into a section that already contains code. In the latter case, the assembler simply appends the new code to the code that is already in the section.

The format shown below is a listing file. The example shows how the SPCs are modified during assembly. A line in a listing file has four fields:

Field 1	contains the source code line counter.
Field 2	contains the section program counter.
Field 3	contains the object code.
Field 4	contains the original source statement.

```
 1                      ************************************************
 2                      ** Assemble an initialized table into .data.  **
 3                      ************************************************
 4 00000000                      .data
 5 00000000 00000011 coeff   .word           011h, 022h, 033h
   00000004 00000022
   00000008 00000033

 6
 7                      ************************************************
 8                      **   Reserve space in .bss for a variable.   **
 9                      ************************************************
10 00000000                      .bss            buffer,10
11                      ************************************************
12                      **              Still in .data.              **
13                      ************************************************
14 0000000c 00000123 ptr     .word           0123h
15                      ************************************************
16                      **   Assemble code into the .text section.   **
17                      ************************************************
18 00000000             .text
19 00000000 E59F14D2 add:    LDR             R1, #1234
20 00000004 E2511001 aloop:  SUBS            R1, R1, #1
21 00000008 1AFFFFFD     BNE             aloop
22                      ************************************************
23                      **   Another initialized table into .data.   **
24                      ************************************************
25 00000010             .data
26 00000010 000000AA ivals   .word           0AAh, 0BBh, 0CCh
   00000014 000000BB
   00000018 000000CC

27                      ************************************************
28                      ** Define another section for more variables.**
29                      ************************************************
30 00000000           var2    .usect          "newvars", 1
31 00000001           inbuf   .usect          "newvars", 7
32                      ************************************************
33                      **       Assemble more code into .text.       **
34                      ************************************************
35 0000000c             .text
36 0000000c E59F3D80 mpy:    LDR             R3, #3456
37 00000010 E0120293 mloop:  MULS            R2, R3, R2
38 00000014 1AFFFFFD     BNE             mloop
39                      ************************************************
40                      ** Define a named section for int. vectors.  **
41                      ************************************************
42 00000000                  .sect           "vectors"
43 00000000 00000011         .word           011h,033h
   00000004 00000033

Field 1  Field 2    Field 3                        Field 4
```

Figure 3.17: Using Sections Directives

As the figure below shows, the file in the example above creates five sections:

.text	contains six 32-bit words of object code.
.data	contains seven 32-bit words of initialized data.
vec-tors	is a user-named section created with the .sect directive; it contains two 32-bit words of initialized data.
.bss	reserves ten bytes in memory.
new-vars	is a user-named section created with the .usect directive; it reserves eight bytes in memory.

3.7. Introduction to Object Modules

The second column shows the object code that is assembled into these sections; the first column shows the source statements that generated the object code.

Line numbers	Object code	Section
19	E59F14D2	.text
20	E2511001	
21	1AFFFFFD	
36	E59F3D80	
37	E0120293	
38	1AFFFFFD	
5	00000011	.data
5	00000022	
5	00000033	
14	00000123	
26	000000AA	
26	000000BB	
26	000000CC	
43	00000011	vectors
43	00000033	
10	No data - ten bytes reserved	.bss
30	No data - eight bytes reserved	newvars
31		

Figure 3.18: Object Code Generated by the Above Assembly Code

3.7.5 How the Linker Handles Sections

The linker has two main functions related to sections. First, the linker uses the sections in object files as building blocks; it combines input sections to create output sections in an executable output module. Second, the linker chooses memory addresses for the output sections; this is called *placement*. Two linker directives support these functions:

- The *MEMORY* directive allows you to define the memory map of a target system. You can name portions of memory and specify their starting addresses and their lengths.

- The *SECTIONS* directive tells the linker how to combine input sections into output sections and where to place these output sections in memory.

Subsections let you manipulate the placement of sections with greater precision. You can specify the location of each subsection with the linker's SECTIONS directive. If you do not specify a subsection, the subsection is combined with the other sections with the same base section name. See *SECTIONS Directive Syntax*.

It is not always necessary to use linker directives. If you do not use them, the linker uses the target processor's default placement algorithm described in *Default Placement Algorithm*. When you *do* use linker directives, you must specify them in a linker command file.

Refer to the following sections for more information about linker command files and linker directives:

- *Linker Command Files*
- *The MEMORY Directive*
- *The SECTIONS Directive*
- *Default Placement Algorithm*

Combining Input Sections

The following figure provides a simplified example of the process of linking two files together. Since this is a simplified example, it does not show all the sections that will be created or the actual sequence of the sections. See *Default Placement Algorithm* for the actual default memory placement map for Arm.
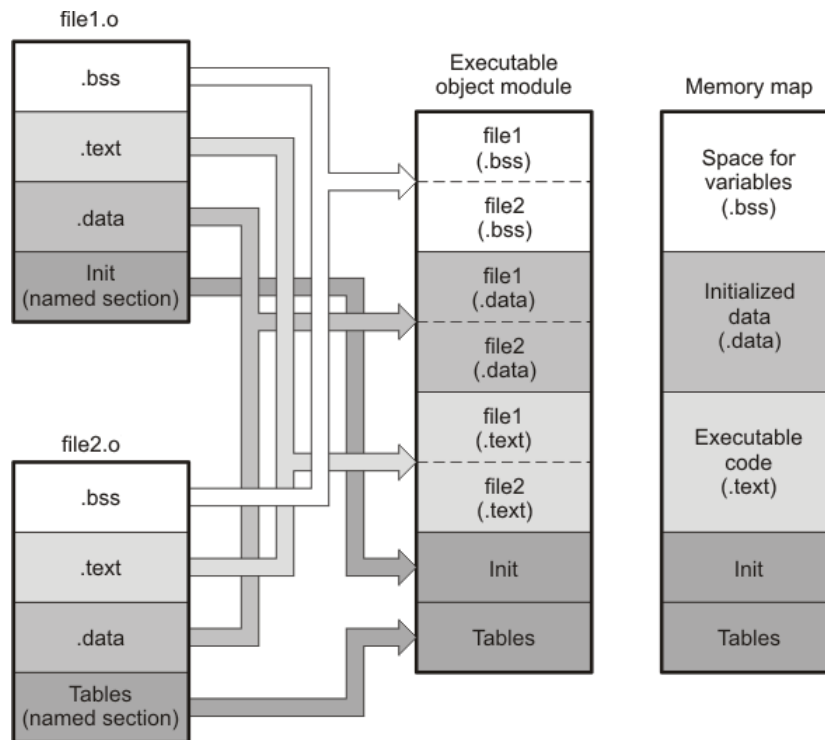
Figure 3.19: Combining Input Sections to Form an Executable Object Module

In the above figure, file1.o and file2.o are object files that are used as linker input. Each contains the .text, .data, and .bss default sections; in addition, each contains a user-named section. The executable object module shows the combined sections. The linker combines the .text section from file1.o and the .text section from file2.o to form one .text section, then combines the two .data sections and the two .bss sections, and finally places the user-named sections at the end. The memory map shows the combined sections to be placed into memory.

Placing Sections

The previous figure illustrates the linker's default method for combining sections. Sometimes you may not want to use the default setup. For example, you may not want all of the .text sections to be combined into a single .text section. Or you may want a user-named section placed where the .data section would normally be allocated. Most memory maps contain various types of memory (RAM, ROM, EEPROM, FLASH, etc.) in varying amounts; you may want to place a section in a specific type of memory.

For further explanation of section placement within the memory map, see the discussions in *The MEMORY Directive* and *The SECTIONS Directive*. See *Default Placement Algorithm* for the actual default memory allocation map for Arm.

3.7.6 Symbols

An object file contains a symbol table that stores information about *symbols* in the object file. The linker uses this table when it performs relocation. See *Symbolic Relocations*.

An object file symbol is a named 32-bit integer value, usually representing an address. A symbol can represent such things as the starting address of a function, variable, section, or an absolute integer (such as the size of the stack).

Symbols are defined in assembly by adding a label or a directive such as .set .equ .bss, or .usect.

Symbols have a *binding*, which is similar to the C standard concept of *linkage*. ELF files may contain symbols bound as *local symbols*, *global symbols*, and *weak symbols*.

- **Global symbols** are visible to the entire program. The linker does not allow more than one global definition of a particular symbol; it issues a multiple-definition error if a global symbol is defined more than once. (The assembler can provide a similar multiple-definition error for local symbols.) A reference to a global symbol from any object file refers to the one and only allowed global definition of that symbol. Assembly code must explicitly make a symbol global by adding a .def, .ref, or .global directive. (See *Global (External) Symbols*.)

- **Local symbols** are visible only within one object file; each object file that uses a symbol needs its own local definition. References to local symbols in an object file are entirely unrelated to local symbols of the same name in another object file. By default, a symbol is local. (See *Local Symbols*.)

- **Weak symbols** are symbols that may be used but not defined in the current module. They may or may not be defined in another module. A weak symbol is intended to be overridden by a strong (non-weak) global symbol definition of the same name in another object file. If a strong definition is available, the weak symbol is replaced by the strong symbol. If no definition is available (that is, if the weak symbol is unresolved), no error is generated, but the weak variable's address is considered to be null (0). For this reason, application code that accesses a weak variable must check that its address is not zero before attempting to access the variable. (See *Weak Symbols*.)

Absolute symbols are symbols that have a numeric value. They may be constants. To the linker, such symbols are unsigned values, but the integer may be treated as signed or unsigned depending on how it is used. The range of legal values for an absolute integer is 0 to 2^32-1 for unsigned treatment and -2^31 to 2^31-1 for signed treatment.

In general, *common symbols* are preferred over weak symbols.

Global (External) Symbols

Global symbols are symbols that are either accessed in the current module but defined in another (an external symbol) or defined in the current module and accessed in another. Such symbols are visible across object modules.

You must use the .def, .ref, or .global directive to identify a symbol as external:

.def	The symbol is defined in the current file and may be used in another file.
.ref	The symbol is referenced in the current file, but defined in another file.
.global	The symbol can be either of the above. The assembler chooses either .def or .ref as appropriate for each symbol.

The following code fragments illustrate the use of the .global directive.

```
x:    ADD        R0, #56h      ; Define x
      .global x ; acts as .def x
```

Because x is defined in this module, the assembler treats ".global x" as ".def x". Now other modules can refer to x.

```
B            y                 ; Reference y
.global y   ; .ref of y
```

Because y is not defined in this module, the assembler treats ".global y" as ".ref y". The symbol y must be defined in another module.

Both the symbols x and y are external symbols and are placed in the object file's symbol table; x as a defined symbol, and y as an undefined symbol. When the object file is linked with other object files, the entry for x is used to resolve references to x in other files. The entry for y causes the linker to look through the symbol tables of other files for y's definition.

The linker attempts to match all references with corresponding definitions. If the linker cannot find a symbol's definition, it prints an error message about the unresolved reference. This type of error prevents the linker from creating an executable object module.

An error also occurs if the same symbol is defined more than once.

Local Symbols

Local symbols are visible within a single object file. Each object file may have its own local definition for a particular symbol. References to local symbols in an object file are entirely unrelated to local symbols of the same name in another object file.

By default, a symbol is local.

Weak Symbols

Weak symbols are symbols that may or may not be defined.

The linker processes symbols that are defined with a "weak" binding differently from symbols that are defined with global binding. Instead of including a weak symbol in the object file's symbol table (as it would for a global symbol), the linker only includes a weak symbol in the output of a "final" link if the symbol is required to resolve an otherwise unresolved reference.

This allows the linker to minimize the number of symbols it includes in the output file's symbol table by omitting those that are not needed to resolve references. Reducing the size of the output file's symbol table reduces the time required to link, especially if there are a large number of pre-loaded symbols to link against.

You can define a weak symbol using either the .weak assembly directive or the weak operator in the linker command file.

- **Using Assembly:** To define a weak symbol in an input object file, the source file can be written in assembly. Use the .weak and .set directives in combination as shown in the following example, which defines a weak symbol "ext_addr_sym".

```
              .weak    ext_addr_sym
ext_addr_sym  .set     0x12345678
```

  Assemble the source file that defines weak symbols, and include the resulting object file in the link. The "ext_addr_sym" in this example is available as a weak symbol in a final link. It is a candidate for removal if the symbol is not referenced elsewhere in the application.

- **Using the Linker Command File:** To define a weak symbol in a linker command file, use the "weak" operator in an assignment expression to designate that the symbol as eligible for removal from the output file's symbol table if it is not referenced. In a linker command file, an assignment expression outside a MEMORY or SECTIONS directive can be used to define a weak linker-defined symbol. For example, you can define "ext_addr_sym" as follows.

```
weak(ext_addr_sym) = 0x12345678;
```

  If the linker command file is used to perform the final link, then "ext_addr_sym" is presented to the linker as a weak symbol; it is not included in the resulting output file

if the symbol is not referenced. See *Declaring Weak Symbols*.

- **Using C/C++ code:** See *weak* for information about the weak GCC-style variable attribute.

If there are multiple definitions of the same symbol, the linker uses certain rules to determine which definition takes precedence. Some definitions may have weak binding and others may have strong binding. "Strong" in this context means that the symbol has *not* been given a weak binding by either of the two methods described above. Some definitions may come from an input object file (that is, using assembly directives) and others may come from an assignment statement in a linker command file.

The linker uses the following guidelines to determine which definition is used when resolving references to a symbol:

- A strongly bound symbol always takes precedence over a weakly bound symbol.

- If two symbols are both strongly bound or both weakly bound, a symbol defined in a linker command file takes precedence over a symbol defined in an input object file.

- If two symbols are both strongly bound and both are defined in an input object file, the linker provides a symbol redefinition error and halts the link process.

The Symbol Table

The assembler generates entries with global (external) binding in the symbol table for each of the following:

- Each .ref, .def, or .global directive (see *Global (External) Symbols*)

- The beginning of each section

The assembler generates entries with local binding for each locally-available function.

For informational purposes, there are also entries in the symbol table for each symbol in a program.

3.7.7 Symbolic Relocations

The assembler treats each section as if it began at address 0. Of course, all sections cannot actually begin at address 0 in memory, so the linker must relocate sections. Relocations are symbol-relative rather than section-relative.

The linker can *relocate* sections by:

- Allocating them into the memory map so that they begin at the appropriate address as defined with the linker's MEMORY directive

- Adjusting symbol values to correspond to the new section addresses

- Adjusting references to relocated symbols to reflect the adjusted symbol values

The linker uses *relocation entries* to adjust references to symbol values. The assembler creates a relocation entry each time a relocatable symbol is referenced. The linker then uses these entries to patch the references after the symbols are relocated. The following example contains a code fragment for an Arm device for which the assembler generates relocation entries.

Example: Code That Generates Relocation Entries

```
 1                           ␣
↪***************************************
 2                    **     Generating Relocation Entries    ␣
↪**
 3                           ␣
↪***************************************
 4                              .ref X
 5                              .def Y
 6 00000000                     .text
 7 00000000 E0921003            ADDS   R1, R2, R3
 8 00000004 0A000001            BEQ    Y
 9 00000008 E1C410BE            STRH   R1, [R4, #14]
10 0000000c EAFFFFFB!           B      X   ; generates a␣
↪relocation entry
11 00000010 E0821003  Y:        ADD    R1, R2, R3
```

In this example, both symbols X and Y are relocatable. Y is defined in the .text section of this module; X is defined in another module. When the code is assembled, X has a value of 0 (the assembler assumes all undefined external symbols have values of 0), and Y has a value of 16 (relative to address 0 in the .text section). The assembler generates two relocation entries: one for X and one for Y. The reference to X is an external reference (indicated by the ! character in the listing). The reference to Y is to an internally defined relocatable symbol (indicated by the ' character in the listing).

After the code is linked, suppose that X is relocated to address 0x10014. Suppose also that the .text section is relocated to begin at address 0x10000; Y now has a relocated value of 0x10010. The linker uses the relocation entry for the reference to X to patch the branch instruction in the object code:

EAFFFFFB! B X	becomes	EA000000

3.7.8 Loading a Program

The linker creates an executable object file which can be loaded in several ways, depending on your execution environment. These methods include using Code Composer Studio or the hex conversion utility. For details, see *Loading*.

3.8 Program Loading and Running

Even after a program is written, compiled, and linked into an executable object file, there are still many tasks that need to be performed before the program does its job. The program must be loaded onto the target, memory and registers must be initialized, and the program must be set to running.

Some of these tasks need to be built into the program itself. *Bootstrapping* is the process of a program performing some of its own initialization. Many of the necessary tasks are handled for you by the compiler and linker, but if you need more control over these tasks, it helps to understand how the pieces are expected to fit together.

This chapter introduces you to the concepts involved in program loading, initialization, and startup.

This chapter does not cover *dynamic loading*.

This chapter currently provides examples for the C6000 device family. Refer to your device documentation for various device-specific aspects of bootstrapping.

3.8.1 Loading

A program needs to be placed into the target device's memory before it may be executed. *Loading* is the process of preparing a program for execution by initializing device memory with the program's code and data. A *loader* might be another program on the device, an external agent (for example, a debugger), or the device might initialize itself after power-on, which is known as *bootstrap loading*, or *bootloading*.

The loader is responsible for constructing the *load image* in memory before the program starts. The load image is the program's code and data in memory before execution. What exactly constitutes loading depends on the environment, such as whether an operating system is present. This section describes several loading schemes for bare-metal devices. This section is not exhaustive.

A program may be loaded in the following ways:

- **A debugger running on a connected host workstation.** In a typical embedded development setup, the device is subordinate to a host running a debugger such as Code Composer Studio (CCS). The device is connected with a communication channel such as a JTAG interface. CCS reads the program and writes the load image directly to target memory through the communications interface.

- **"Burning" the load image onto an EPROM module.**

- **Bootstrap loading from a dedicated peripheral, such as an I2C peripheral.** The device may require a small program called a bootloader to perform the loading from the peripheral.

- **Another program running on the device.** The running program can create the load image and transfer control to the loaded program. If an operating system is present, it may have the ability to load and run programs.

Note: The hex converter (tiarmhex) can assist with creating a bootloader by converting the executable object file into a format suitable for input to an EPROM programmer. The EPROM is placed onto the device itself and becomes a part of the device's memory. See *Hex Conversion Utility Description* for details.

Load and Run Addresses

Consider an embedded device for which the program's load image is burned onto EPROM/ROM. Variable data in the program must be writable, and so must be located in writable memory, typically RAM. However, RAM is *volatile*, meaning it will lose its contents when the power goes out. If this data must have an initial value, that initial value must be stored somewhere else in the load image, or it would be lost when power is cycled. The initial value must be copied from the non-volatile ROM to its run-time location in RAM before it is used. See *Using Linker-Generated Copy Tables* for ways this is done.

The *load address* is the location of an object in the load image.

The *run address* is the location of the object as it exists during program execution.

An *object* is a chunk of memory. It represents a section, segment, function, or data.

The load and run addresses for an object may be the same. This is commonly the case for program code and read-only data, such as the .const section. In this case, the program can read the data directly from the load address. Sections that have no initial value, such as the .bss section, do not have load data and are considered to have load and run addresses that are the same. If you specify different load and run addresses for an uninitialized section, the linker provides a warning and ignores the load address.

The load and run addresses for an object may be different. This is commonly the case for writable data, such as the .data section. The .data section's starting contents are placed in ROM and copied to RAM. This often occurs during program startup, but depending on the needs of the object, it may be deferred to sometime later in the program as described in *Run-Time Relocation*.

Symbols in assembly code and object files almost always refer to the run address. When you look at an address in the program, you are almost always looking at the run address. The load address is rarely used for anything but initialization.

The load and run addresses for a section are controlled by the linker command file and are recorded in the object file metadata.

The load address determines where a loader places the raw data for the section. Any references to the section (such as references to labels in it) refer to its run address. The application must copy the section from its load address to its run address before the first reference of the symbol is encountered at run time; this does *not* happen automatically simply because you specify a separate run address. For examples that specify load and run addresses, see *Specifying Load and Run Addresses*.

For an example that illustrates how to move a block of code at run time, see the example in *Referring to the Load Address by Using the .label Directive*. To create a symbol that lets you refer to the load-time address, rather than the run-time address, see the *Referring to the Load Address by Using the .label Directive*. To use copy tables to copy objects from load-space to run-space at boot time, see *Using Linker-Generated Copy Tables*.

ELF format executable object files contain *segments*. See *Introduction to Sections* for information about sections and segments.

Bootstrap Loading

The details of bootstrap loading (bootloading) vary a great deal between devices. Not every device supports every bootloading mode, and using the bootloader is optional. This section discusses various bootloading schemes to help you understand how they work. Refer to your device's data sheet to see which bootloading schemes are available and how to use them.

A typical embedded system uses bootloading to initialize the device. The program code and data may be stored in ROM or FLASH memory. At power-on, an on-chip bootloader (the *primary bootloader*) built into the device hardware starts automatically.
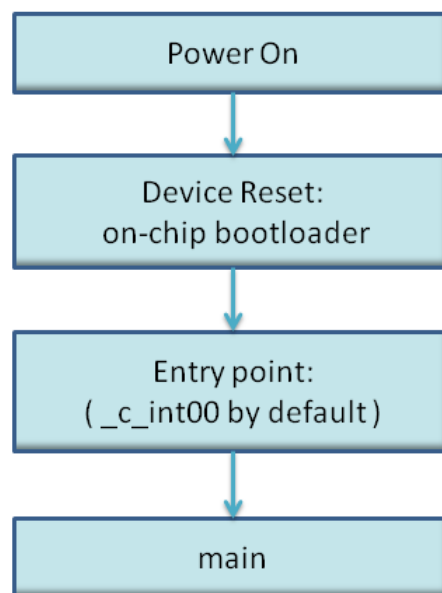
Figure 3.20: Bootloading Sequence (Simplified)

The primary bootloader is typically very small and copies a limited amount of memory from a dedicated location in ROM to a dedicated location in RAM. (Some bootloaders support copying the program from an I/O peripheral.) After the copy is completed, it transfers control to the program.

For many programs, the primary bootloader is not capable of loading the entire program, so these programs supply a more capable secondary bootloader. The primary bootloader loads the secondary bootloader and transfers control to it. Then, the secondary bootloader loads the rest of the program and transfers control to it. There can be any number of layers of bootloaders, each loading a more capable bootloader to which it transfers control.
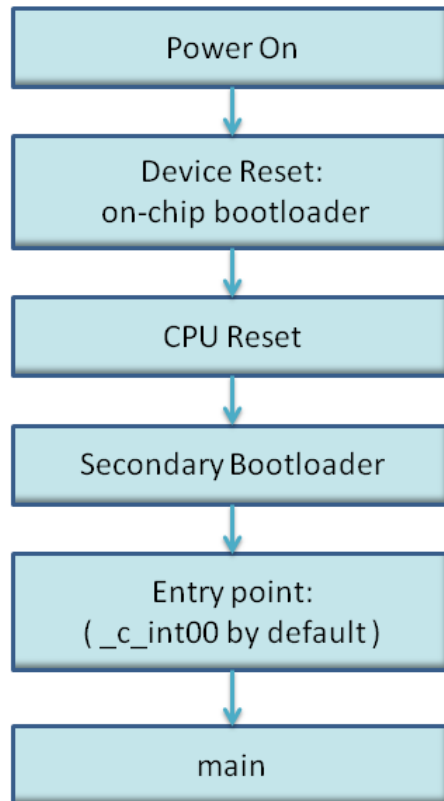
Figure 3.21: Bootloading Sequence with Secondary Bootloader

Boot, Load, and Run Addresses

The *boot address* of a bootloaded object is where its raw data exists in ROM before power-on.

The boot, load, and run addresses for an object may all be the same; this is commonly the case for .const data. If they are different, the object's contents must be copied to the correct location before the object may be used.

The boot address may be different than the load address. The bootloader is responsible for copying the raw data to the load address.

The boot address is not controlled by the linker command file or recorded in the object file; it is strictly a convention shared by the bootloader and the program.

Primary Bootloader

The detailed operation of the primary bootloader is device-specific. Some devices have complex capabilities such as booting from an I/O peripheral or configuring memory controller parameters.

Secondary Bootloader

The hex converter assumes the secondary bootloader is of a particular format. The hex converter's model bootloader uses a *boot table*. You can use whatever format you want, but if you follow this model, the hex converter can create the boot table automatically.

Boot Table

The input for the model secondary bootloader is the *boot table*. The boot table contains records that instruct the secondary bootloader to copy blocks of data contained in the table to specified destination addresses.

The hex conversion utility automatically builds the boot table for the secondary bootloader. Using the utility, you specify the sections you want to initialize, the boot table location, and the name of the section containing the secondary bootloader routine and where it should be located. The hex conversion utility builds a complete image of the table and adds it to the program.

The boot table is target-specific. For C6000, the format of the boot table is simple. A header record contains a 4-byte field that indicates where the boot loader should branch after it has completed copying data. After the header, each section that is to be included in the boot table has the following contents:

- 4-byte field containing the size of the section

- 4-byte field containing the destination address for the copy

- the raw data

- 0 to 3 bytes of trailing padding to make the next field aligned to 4 bytes

More than one section can be entered; a termination block containing an all-zero 4-byte field follows the last section.

See *The Boot Table Format* for details about the boot table format.

Bootloader Routine

The bootloader routine is a normal function, except that it executes before the C environment is set up. For this reason, it can't use the C stack, and it can't call any functions that have yet to be loaded!

The following sample code is for C6000 and is from *Creating a Second-Level Bootloader for FLASH Bootloading on TMS320C6000 Platform With Code Composer Studio* (SPRA999).

Example: Sample Secondary Bootloader Routine

```
;   ======== boot_c671x.s62 ========

; global EMIF symbols defined for the c671x family
        .include        boot_c671x.h62
        .sect ".boot_load"
        .global _boot

_boot:
;
↪******************************************************************************
;* DEBUG LOOP  COMMENT OUT B FOR NORMAL OPERATION
;
↪******************************************************************************
zero B1
_myloop: ;  [!B1] B _myloop
        nop  5
_myloopend: nop
;
↪******************************************************************************
;* CONFIGURE EMIF
;
↪******************************************************************************
        ;
↪******************************************************************************
        ; *EMIF_GCTL = EMIF_GCTL_V;
        ;
↪******************************************************************************
        mvkl  EMIF_GCTL,A4
    ||    mvkl  EMIF_GCTL_V,B4
        mvkh  EMIF_GCTL,A4
    ||    mvkh  EMIF_GCTL_V,B4
        stw   B4,*A4
        ;
↪******************************************************************************
```

(continued from previous page)

```
        ;   *EMIF_CE0 = EMIF_CE0_V
        ;
;****************************************************************
        mvkl  EMIF_CE0,A4
   ||   mvkl  EMIF_CE0_V,B4
        mvkh  EMIF_CE0,A4
   ||   mvkh  EMIF_CE0_V,B4
        stw   B4,*A4
        ;
;****************************************************************
        ;   *EMIF_CE1 = EMIF_CE1_V (setup for 8bit async)
        ;
;****************************************************************
        mvkl  EMIF_CE1,A4
   ||   mvkl  EMIF_CE1_V,B4
        mvkh  EMIF_CE1,A4
   ||   mvkh  EMIF_CE1_V,B4
        stw   B4,*A4
        ;
;****************************************************************
        ;   *EMIF_CE2 = EMIF_CE2_V (setup for 32bit async)
        ;
;****************************************************************
        mvkl  EMIF_CE2,A4
   ||   mvkl  EMIF_CE2_V,B4
        mvkh  EMIF_CE2,A4
   ||   mvkh  EMIF_CE2_V,B4
        stw   B4,*A4
        ;
;****************************************************************
        ;   *EMIF_CE3 = EMIF_CE3_V (setup for 32bit async)
        ;
;****************************************************************
   ||   mvkl  EMIF_CE3,A4
   ||   mvkl  EMIF_CE3_V,B4        ;
        mvkh  EMIF_CE3,A4
   ||   mvkh  EMIF_CE3_V,B4
        stw   B4,*A4
        ;
;****************************************************************
        ;   *EMIF_SDRAMCTL = EMIF_SDRAMCTL_V
        ;
;****************************************************************
```

(continues on next page)

```
       ||     mvkl   EMIF_SDRAMCTL,A4
       ||     mvkl   EMIF_SDRAMCTL_V,B4        ;
              mvkh   EMIF_SDRAMCTL,A4
       ||     mvkh   EMIF_SDRAMCTL_V,B4
              stw    B4,*A4
       ;
↪*************************************************************
       ;  *EMIF_SDRAMTIM = EMIF_SDRAMTIM_V
       ;
↪*************************************************************
       ||     mvkl   EMIF_SDRAMTIM,A4
       ||     mvkl   EMIF_SDRAMTIM_V,B4       ;
              mvkh   EMIF_SDRAMTIM,A4
       ||     mvkh   EMIF_SDRAMTIM_V,B4
              stw    B4,*A4
       ;
↪*************************************************************
       ;  *EMIF_SDRAMEXT = EMIF_SDRAMEXT_V
       ;
↪*************************************************************
       ||     mvkl   EMIF_SDRAMEXT,A4
       ||     mvkl   EMIF_SDRAMEXT_V,B4       ;
              mvkh   EMIF_SDRAMEXT,A4
       ||     mvkh   EMIF_SDRAMEXT_V,B4
              stw    B4,*A4
;
↪*********************************************************************************
; copy sections
;
↪*********************************************************************************
              mvkl   COPY_TABLE, a3 ; load table pointer
              mvkh   COPY_TABLE, a3
              ldw    *a3++, b1      ; Load entry point
copy_section_top:
              ldw    *a3++, b0      ; byte count
              ldw    *a3++, a4      ; ram start address
              nop    3
[!b0]         b copy_done          ; have we copied all sections?
              nop    5
copy_loop:
              ldb    *a3++,b5
              sub    b0,1,b0        ; decrement counter
```

```
[ b0]         b     copy_loop         ; setup branch if not done
[!b0]         b     copy_section_top
              zero  a1
[!b0]         and   3,a3,a1
              stb   b5,*a4++
[!b0]         and   4,a3,a5          ; round address up to next
↪multiple of 4
[ a1]         add   4,a5,a3          ; round address up to next
↪multiple of 4
;
↪****************************************************************************
; jump to entry point
;
↪****************************************************************************
copy_done:
              b     .S2 b1
              nop   5
```

3.8.2 Entry Point

The entry point is the address at which the execution of the program begins. This is the address of the startup routine. The startup routine is responsible for initializing and calling the rest of the program. For a C/C++ program, the startup routine is usually named _c_int00 (see *The _c_int00 Function*). After the program is loaded, the value of the entry point is placed in the PC register and the CPU is allowed to run.

The object file has an entry point field. For a C/C++ program, the linker fills in _c_int00 by default. You can select a custom entry point; see *Define an Entry Point (--entry_point Option)*. The device itself cannot read the entry point field from the object file, so it has to be encoded in the program somewhere.

- If you are using a bootloader, the boot table includes an entry point field. When it finishes running, the bootloader branches to the entry point.

- If you are using an interrupt vector, the entry point is installed as the RESET interrupt handler. When RESET is applied, the startup routine is invoked.

- If you are using a hosted debugger, such as CCS, the debugger may explicitly set the program counter (PC) to the value of the entry point.

3.8.3 Run-Time Initialization

After the load image is in place, the program can run. The subsections that follow describe boot-strap initialization of a C/C++ program. An assembly-only program may not need to perform all of these steps.

The _c_int00 Function

The function _c_int00 is the *startup routine* (also called the *boot routine*) for C/C++ programs. It performs all the steps necessary for a C/C++ program to initialize itself.

The name _c_int00 means that it is the interrupt handler for interrupt number 0, RESET, and that it sets up the C environment. Its name need not be exactly _c_int00, but the linker sets _c_int00 as the entry point for C programs by default. The compiler's run-time-support library provides a default implementation of _c_int00.

The startup routine is responsible for performing the following actions:

1. Switch to user mode and sets up the user mode stack

2. Set up status and configuration registers

3. Set up the stack

4. Process special binit copy table, if present.

5. Process the run-time initialization table to autoinitialize global variables (when using the --rom_model option)

6. Call all global constructors

7. Call the function main

8. Call exit when main returns

RAM Model vs. ROM Model

Choose a startup model based on the needs of your application. The ROM model performs more work during the boot routine. The RAM model performs more work while loading the application.

If your application is likely to need frequent RESETs or is a standalone application, the ROM model may be a better choice, because the boot routine has all the data it needs to initialize RAM variables. However, for a system with an operating system, it may be better to use the RAM model.

In the EABI ROM model, the C boot routine copies data from the .cinit section to the run-time location of the variables to be initialized.

In the EABI RAM model, no .cinit records are generated at startup.

Note that no default startup model is specified to the linker when the tiarmclang compiler runs the linker. Therefore, either the `--rom_model` (`-c`) or `--ram_model` (`-cr`) option must be passed to the linker on the **tiarmclang** command line or in the linker command file. For example:

```
tiarmclang -mcpu=cortex-m4 hello.c -o hello.out -Wl,-c,-llnk.cmd,
 ↪-mhello.map
```

If neither the `-c` or `-cr` option is specified to tiarmclang when running the linker, the linker expects an entry point for the linked application to be identified (using the `-e=<symbol>` linker option). If `-c` or `-cr` is specified, then the linker assumes that the program entry point is _c_int00, which performs any needed auto-initialization and system setup, then calls the user's main() function.

Autoinitializing Variables at Run Time (--rom_model)

Autoinitializing variables at run time is the most common method of autoinitialization. To use this method, invoke the linker with the `--rom_model` option.

The ROM model allows initialization data to be stored in slow non-volatile memory and copied to fast memory each time the program is reset. Use this method if your application runs from code burned into slow memory or needs to survive a reset.

For the ROM model, the .cinit section is loaded into memory along with all the other initialized sections. The linker defines a special symbol called __TI_CINIT_Base that points to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by .cinit) into the run-time location of the variables.

The following figure illustrates autoinitialization at run time using the ROM model.
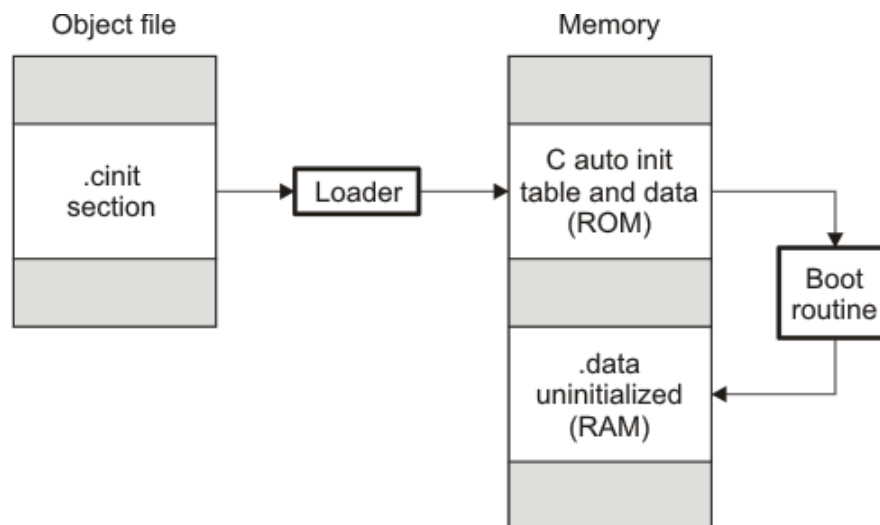
Figure 3.22: Autoinitialization at Run Time

Initializing Variables at Load Time (--ram_model)

The RAM model Initializes variables at load time. To use this method, invoke the linker with the --ram_model option.

This model may reduce boot time and save memory used by the initialization tables.

When you use the --ram_model linker option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.)

The linker sets __TI_CINIT_Base equal to __TI_CINIT_Limit to indicate there are no .cinit records.

The loader copies values directly from the .data section to memory.

The following figure illustrates the initialization of variables at load time.
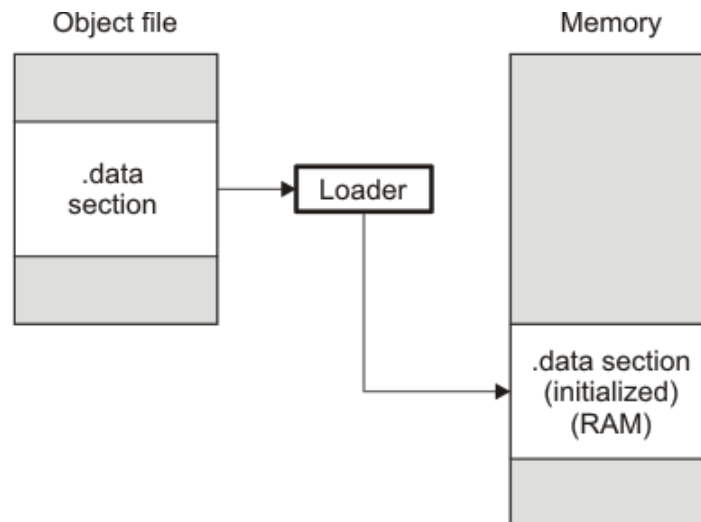
Figure 3.23: Initialization at Load Time

The --rom_model and --ram_model Linker Options

The following list outlines what happens when you invoke the linker with the --ram_model or --rom_model option.

- The symbol _c_int00 is defined as the program entry point. The _c_int00 symbol is the start of the C boot routine in `boot.c.o`. Referencing _c_int00 ensures that `boot.c.o` is automatically linked in from the appropriate run-time-support library.

- *If you use the ROM model* to autoinitialize at run time (--rom_model option), the linker defines a special symbol, __TI_CINIT_Base, to point to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by .cinit) into the run-time location of the variables.

- *If you use the RAM model* to initialize at load time (--ram_model option), the linker sets __TI_CINIT_Base equal to __TI_CINIT_Limit to indicate there are no .cinit records.

About Linker-Generated Copy Tables

The RTS function copy_in can be used at run-time to move code and data around, usually from its load address to its run address. This function reads size and location information from copy tables. The linker automatically generates several kinds of copy tables. Refer to *Using Linker-Generated Copy Tables*.

You can create and control code overlays with copy tables. See *Generating Copy Tables With the table( ) Operator* for details and examples.

Copy tables can be used by the linker to implement run-time relocations as described in *Run-Time Relocation*, however copy tables require a specific table format.

BINIT

The BINIT (boot-time initialization) copy table is special in that the target automatically performs the copying at auto-initialization time. Refer to *Boot-Time Copy Tables* for more about the BINIT copy table name. The BINIT copy table is copied before .cinit processing.

CINIT

EABI .cinit tables are special kinds of copy tables. Refer to *Autoinitializing Variables at Run Time (--rom_model)* for more about using the .cinit section with the ROM model and *Initializing Variables at Load Time (--ram_model)* for more using it with the RAM model.

3.8.4 Arguments to main

Some programs expect arguments to main (argc, argv) to be valid. Normally this isn't possible for an embedded program, but the TI runtime does provide a way to do it. The user must allocate an .args section of an appropriate size using the --args linker option. It is the responsibility of the loader to populate the .args section. It is not specified how the loader determines which arguments to pass to the target. The format of the arguments is the same as an array of pointers to char on the target.

See *Allocate Memory for Use by the Loader to Pass Arguments (–arg_size Option)* for information about allocating memory for argument passing.

3.8.5 Run-Time Relocation

At times you may want to load code into one area of memory and move it to another area before running it. For example, you may have performance-critical code in an external-memory-based system. The code must be loaded into external memory, but it would run faster in internal memory. Because internal memory is limited, you might swap in different speed-critical functions at different times.

The linker provides a way to handle this. Using the SECTIONS directive, you can optionally direct the linker to allocate a section twice: first to set its load address and again to set its run address. Use the *load* keyword for the load address and the *run* keyword for the run address. See *Load and Run Addresses* for more about load and run addresses. If a section is assigned two addresses at link time, all labels defined in the section are relocated to refer to the run-time address so that references to the section (such as branches) are correct when the code runs.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and loads and runs at the same address. If you provide both allocations, the section is actually allocated as if it were two separate sections. The two sections are the same size if the load section is not compressed.

Uninitialized sections (such as .bss) are not loaded, so the only significant address is the run address. The linker allocates uninitialized sections only once; if you specify both run and load addresses, the linker warns you and ignores the load address.

For a complete description of run-time relocation, see *Placing a Section at Different Load and Run Addresses*.

3.8.6 Additional Information

See the following sections and documents for additional information:

- *Allocate Memory for Use by the Loader to Pass Arguments (–arg_size Option)*
- *Define an Entry Point (--entry_point Option)*
- *Specifying Load and Run Addresses*
- *Using Linker-Generated Copy Tables*
- *Linking for Run-Time Initialization*
- *Run-Time Initialization*
- *System Initialization*
- *Hex Conversion Utility Description*

3.9 Archiver Description

The archiver (**tiarmar**) lets you combine several individual files into a single archive file. For example, you can use tiarmar to collect a group of object files into an object library. When this library is specified as part of the link step of an application build, the linker includes members of the object library that resolve external symbol references during the link.

Since there are several different Arm processor variants supported by the tiarmclang compiler tools, it is desirable to have multiple versions of the same object file libraries, each built with different build options. When several versions of the same library are available, the **tiarmlibinfo** library information archiver can be used to create an index library of all the object file library versions. This index library can be used in the link step in place of a particular version of your object library. At link time, the linker finds the version of your object library whose build options are most compatible with the other object files specified as input to the link.

This section of the compiler manual provides details about the usage and available options for the tiarmar and tiarmlibinfo utilities.

Contents:

3.9.1 tiarmar - Archiver

The **tiarmar** command can be used to collect several files, such as object files and LLVM bitcode files, into a single archive library that can be linked into a program. By default, tiarmar generates a symbol table that can be consulted at link-time to aid the linker in determining whether a member of the archive can be pulled into the link to resolve a reference to an unresolved symbol.

When the **tiarmar** command is used to create an archive of LLVM bitcode files, the archive's symbol table will contain both native and bitcode symbols.

Usage

tiarmar [-] *<operation>* [*<modifier>*] {*<relpos>*} {*<count>*} *<archive>* [*<files>* . . .]

- *<operation>* - is an option identifying a single basic operation to be performed on the specified *<archive>*.

- *<modifier>* - is an option that is applicable to the specified *<operation>* and indicates what available modifiers are to applied during the specified *<operation>*.

- *<relpos>* - indicates position in an existing *<archive>* where a file is to be moved or inserted. This argument is only applicable when using the *a*, *b*, or *i* operation-specific *<modifier>* arguments.

- *<count>* - identify an instance of a specified file that the specified *<operation>* applies to. This argument is only applicable in combination with the *d <operation>* and the *N <modifier>*.

- *<archive>* - identifies the archive file for tiarmar to operate on.

- *<files>* - optionally identifies a list of one of more files to be considered as input when operating on the specified *<archive>* file. if no *<files>* are specified, this generally refers to "none" or "all" of the *<archive>* members being the subject of the specified *<operation>*.

The minimal set of arguments to the **tiarmar** command includes at least one *<operation>* and the name of the *<archive>* file.

Operations/Modifiers

d[NT]

Delete specified *<files>* from the *<archive>*. If a specified file does not appear in the *<archive>*, it is simply ignored. If no *<files>* are specified, then the *<archive>* is not modified.

Operation-Specific Modifiers

- *N* - when there are multiple instances of a specified file in the *<archive>*, the *N <modifier>* can be used to identify which instance of a specified file to delete from the *<archive>* via the *<count>* argument, where a *<count>* value of 1 indicates the first instance of the file in the *<archive>*. If the *N <modifier>* is not specified, then the *d <operation>* removes the first instance of a specified file to be deleted. If the *N <modifier>* is used without a *<count>* argument, then the *d <operation>* fails.

- *T* - when the *T <modifier>* is used, the *<archive>* that is created or modified as a result of the *<operation>* will be thin. By default, this behavior is disabled. In the absence of the *T <modifier>*, a newly created archive is always *regular*, and a modified *thin* archive will be converted to *regular*.

m[abi]

Move *<files>* from one location in the *archive>* to another. The specified *<files>* are moved to the location indicated by the specified *<modifier>* options. If no *<modifier>* options are specified, then the specified *<files>* are moved to the end of the *<archive>*. If no *<files>* are specified, then the *<archive>* is not modified.

Operation-Specific Modifiers

- *a* - when the *a <modifier>* is used in combination with the *m <operation>*, the destination of the file to be moved is indicated as **after** the member file identified via the *<relpos>* argument. If a *<relpos>* argument is not specified, then the new file is placed at the end of the *<archive>*.

- *b* - when the *b <modifier>* is used in combination with the *m <operation>*, the destination of the file to be moved is indicated as **before** the member file identified via the

<relpos> argument. If a *<relpos>* argument is not specified, then the new file is placed at the end of the *<archive>*.

- *i* - the *i <modifier>* is a synonym for the *b <modifier>*.

p[v]

Print specified *<files>* to the standard output stream. If no *<files>* are specified, the entire archive is printed. The *p <operation>* does not modify the specified *<archive>*.

Operation-Specific Modifiers

- *v* - print the name of each file in the list of specified *<files>* in addition to the files themselves.

q[LT]

Append specified *<files.* to the end of the *<archive>* without removing duplicate files. If no *<files>* are specified, then the *<archive>* is not modified.

The *L* and *T* modifiers may come into play when using the *q <operation>* to append one archive to another:

- Appending a regular archive to a regular archive appends the archive file. If the *L* modifier is specified, the members are appended instead.

- Appending a regular archive to a thin archive requires the *T* modifier and appends the archive file. The *L* modifier is not supported for this use case.

- Appending a thin archive to a regular archive appends the archive file. If the *L* modifier is specified, the members are appended instead.

- Appending a thin archive to a thin archive always appends its members.

Operation-Specific Modifiers

- *L* - when the *L <modifier>* is used while appending one archive to another, instead of appending the indicated archive, append that archive's members.

- *T* - when the *T <modifier>* is used, the *<archive>* that is created or modified as a result of the *<operation>* will be thin. By default, this behavior is disabled. In the absence of the *T <modifier>*, a newly created archive is always *regular*, and a modified *thin* archive will be converted to *regular*.

r[abTu]

Replace existing *<files>* or insert them at the end of the *<archive>* if they do not exist. If no *<files>* are specified, the *<archive>* is not modified.

Operation-Specific Modifiers

- *a* - when the *a <modifier>* is used in combination with the *r <operation>*, the destination of the new file is indicated as *after* the member file identified via the *<relpos>* argument. If a *<relpos>* argument is not specified, then the new file is placed at the end of the *<archive>*.

- *b* - when the *b <modifier>* is used in combination with the *r <operation>*, the destination of the new file is indicated as *before* the member file identified via the *<relpos>* argument. If a *<relpos>* argument is not specified, then the new file is placed at the end of the *<archive>*.

- *T* - when the *T <modifier>* is used, the *<archive>* that is created or modified as a result of the *<operation>* will be thin. By default, this behavior is disabled. In the absence of the *T <modifier>*, a newly created archive is always *regular*, and a modified *thin* archive will be converted to *regular*.

`t[vO]`

Print the table of contents for the specified *<archive>* file. Without any modifiers, this operation prints the names of the *<archive>* members to *stdout*. If any *<files>* are specified, the operation only applies for those files that are present in the *<archive>*. If no *<files>* are specified, then tiarmar prints the table of contents for the entire *<archive>*.

Operation-Specific Modifiers

- *v* - with this modifier applied to the *t <operation>*, tiarmar prints additional information about the members of the *<archive>* that the operation applies to (e.g. file type, file permissions, file size, and timestamp).

- *O* - with this modifier applied to the *t* "*<operation>*, tiarmar prints *<archive>* member offsets in addition to the names of the *<files>*.

`x[oP]`

Extract *<archive>* members back to *<files>*. This operation retrieves the specified *<files>* from an existing *<archive>* and writes the contents of those files to the operating system's file system. If no *<files>* are specified, then the entire *<archive>* is extracted.

Operation-Specific Modifiers

- *o* - when extracting *<files>*, use the timestamp of the specified *<files>* as they exist in the *<archive>*. Without this modifier, an extracted file is marked with a timestamp corresponding to the time of extraction.

Generic Modifiers

The following *<modifier>* arguments can be applied to any operation:

c

Normally, tiarmar prints a warning message indicating that an *<archive>* is being created if it doesn't already exist. Use of the *c* modifier suppresses this warning.

D

Use zero for timestamps and UIDs/GIDs. This is enabled by default.

P

> Use full paths when matching *<archive>* member names rather than just the file name.

s

> Request that a symbol table (or archive index) be added to the *<archive>*. The symbol table will contain all externally visible functions and global variables defined by all the members of the *<archive>*. This behavior is enabled by default. The *s* generic modifier can also be used as an *<operation>* for an archive that doesn't already contain a symbol table.

S

> Disable generation of the *<archive>* symbol table.

u

> Only update *<archive>* members with *<files>* that have more recent timestamps.

U

> Use actual timestamps and UIDs/GIDs. This overrides the default *D* modifier.

Other Options

-h, --help

> Print a summary of tiarmar usage information to *stdout*.

V, --version

> Display the version of the tiarmar executable.

@<file>

> Read command-line options and commands from specified *<file>*.

Examples

- Creating an object file library:

  Assuming you have the following object files available in your current working directory: *sin.o*, *cos.o*, and *tan.o*, you can create an object file library containing those files with the following command:

```
%> tiarmar rc functions.lib sine.o cos.o tan.o
```

The *r* option instructs the archiver to replace or insert the specified object files into the specified archive file. The *c* option prevents the archiver from printing a warning when creating the archive file.

- Listing the contents of a library:

You can then list the contents of functions.lib using the following command:

```
%> tiarmar tv functions.lib
rw-r--r-- 0/0    1648 Dec 31 18:00 1969 sin.o
rw-r--r-- 0/0    1732 Dec 31 18:00 1969 cos.o
rw-r--r-- 0/0    1716 Dec 31 18:00 1969 tan.o
```

Without the *v* (verbose) option, the above command simply lists the names of the object files contained in functions.lib.

- Adding files to a library:

Assuming you have additional object files in your current working directory that you want to add to the functions.lib object file library, you can do this with the following command:

```
%> tiarmar r functions.lib asin.o acos.o atan.o
```

Now verify the updated contents of functions.lib:

```
%> tiarmar t functions.lib
sin.o
cos.o
tan.o
asin.o
acos.o
atan.o
```

- Replacing an existing library member:

Suppose you've made some improvements to the sin.c file that was used as the source for the compiler generated sin.o file. You can then re-compile the updated source file and replace the previous version of sin.o in the object file library with the new one:

```
%> tiarmclang -mcpu=cortex-m0 -c sin.c
%> tiarmar r functions.lib sin.o
```

Exit Status

If tiarmar execution is successful, it exits with a zero return code. If an error occurs during execution, tiarmar exits with a non-zero return code.

3.9.2 tiarmlibinfo - Library Information Archiver

The **tiarmlibinfo** command allows you to collect multiple versions of the same object file library, each version built with a different set of command-line options, into a single index library file. This index library file can then be used at link-time as a proxy for the actual object file library. The linker considers the build options used to create the input object files to a link and find the matching object file library from among those included in the index library. If successful, the linker incorporates the matching object file library into the link.

Usage

tiarmlibinfo [*<options>*] **-o** = *<index_library>* *<archive1>*[, *<archive2>*, . . .]

- *<options>* - can be used to modify the default behavior.

- -o= *<index_library>* - identify the index library file to be created or updated

- *<archiveN>* - identify a list of one or more object file libraries, each of which is given an entry in the *<index_library>* that is created or updated.

Options

–h, --help

> Print usage information summary to *stdout*.

–o=<index_library>, **--output**=<index_library>

> Identify the *<index_library>* file to be created or updated.

–u, --update

> Update existing information in the specified *<index_library>* file. This option can be used to replace an existing object file library entry in the *index_library>* instead of adding what may be a duplicate.

Example

1 Creating object file libraries

> Compiling each version of *lib_mem.c* and creating an object file library for each containing a single member, *lib_mem.o*:

> As an exploration of how to build up an index library from scratch, consider a simple example where there is a different version of the source file *lib_mem.c* for each of the Arm processor variants that are supported by tiarmclang. Each version contains a definition of a global variable, *mem_global*, that is initialized differently depending on the Arm processor option used to compile *lib_mem.c*.

```
%> tiarmclang <build option> -c lib_mem.c
%> tiarmar r <object library name> lib_mem.o
```

results in the following list of libraries:

Target	mem_global value	<build option>	<object library name>
cortex-m0	10	-mcpu=cortex-m0	cortexm0_def.a
cortex-m3	20	-mcpu=cortex-m3	cortexm3_def.a
cortex-m4	30	-mcpu=cortex-m4	cortexm4_def.a
cortex-m33	40	-mcpu=cortex-m33	cortexm33_def.a
cortex-r4	50	-mcpu=cortex-r4	cortexr4_def.a
cortex-r5	60	-mcpu=cortex-r5	cortexr5_def.a

2 Creating an index library:

An index library called *def.a* can then be constructed with the following command:

```
%> tiarmlibinfo -o def.a cortexm0_def.a cortexm3_def.a␣
 ↪cortexm4_def.a \
   cortexm33_def.a cortexr4_def.a cortexr5_def.a
```

The contents of the *def.a* index library can then be checked via the following tiarmar command:

```
%> tiarmar t def.a cortexm0_def.a.libinfo
cortexm3_def.a.libinfo
cortexm4_def.a.libinfo
cortexm33_def.a.libinfo
cortexr4_def.a.libinfo
cortexr5_def.a.libinfo
__TI_$$LIBINFO
```

3 Using an index library in the link step:

A source file, *print_lib_mem_global.c*, containing a reference to the global variable *mem_global* can then be linked with the index library *def.a*.

```
%> tiarmclang -mcpu=cortex-m4 print_lib_mem_global.c -o␣
 ↪print_mem.out -Wl,-llnk.cmd,def.a,-mprint_mem.map
```

At link-time, the linker selects the object file library in def.a that is most compatible with the object file generated by the compiler for *print_lib_mem_global.c*. In the above case, the linker should pull in the *lib_mem.o* file from the *cortexm4_def.a* object file library and the contents of the linker-generated *print_mem.map* file reveal that this is indeed the case:

```
********************************************************************
TI ARM Clang Linker Unix v1.2.0
********************************************************************
>> Linked Fri Jan 15 15:06:50 2021

OUTPUT FILE NAME:   <print_mem.out>
ENTRY POINT SYMBOL: "_c_int00"  address: 00000e89


...

SECTION ALLOCATION MAP

 output                                       attributes/
section    page    origin      length        input sections
--------   ----   ----------   ----------     ----------------
...
.data       0     2000a020     000001d1       UNINITIALIZED
                  2000a1ec     00000004        cortexm4_def.
 ↪a : lib_mem.o (.data.mem_global)
...
```

If a different -mcpu option had been specified on the above **tiarmclang** command line, then the linker would have pulled in the *lib_mem.o* from a different, appropriate, version of the object file library that matches the specified -mcpu option.

Exit Status

If tiarmlibinfo execution is successful, it exits with a zero return code. If an error occurs during execution, tiarmlibinfo exits with a non-zero return code.

3.10 Linker Description

The Arm linker creates executable modules by combining object modules. This chapter describes the linker options, directives, and statements used to create executable modules. Object libraries, command files, and other key concepts are discussed as well.

The concept of sections is basic to linker operation; see the *Introduction to Object Modules* section for a detailed discussion of sections.

Contents:

3.10.1 Linker Overview

The Arm linker, tiarmlnk, is the proprietary linker provided by Texas Instruments.

This is the same proprietary linker use for the Texas Instruments proprietary Arm compiler.

The linker allows you to allocate output sections efficiently in the memory map. As the linker combines object files, it performs the following tasks:

- Allocates sections into the target system's configured memory
- Relocates symbols and sections to assign them to final addresses
- Resolves undefined external references between input files

The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS, allow you to:

- Allocate sections into specific areas of memory
- Combine object file sections
- Define or redefine global symbols at link time

3.10.2 The Linker's Role in the Software Development Flow

The following figure illustrates the linker's role in the software development process. The linker accepts several types of files as input, including object files, command files, libraries, and partially linked files. The linker creates an executable object module that can be downloaded to one of several development tools or executed by an Arm device.
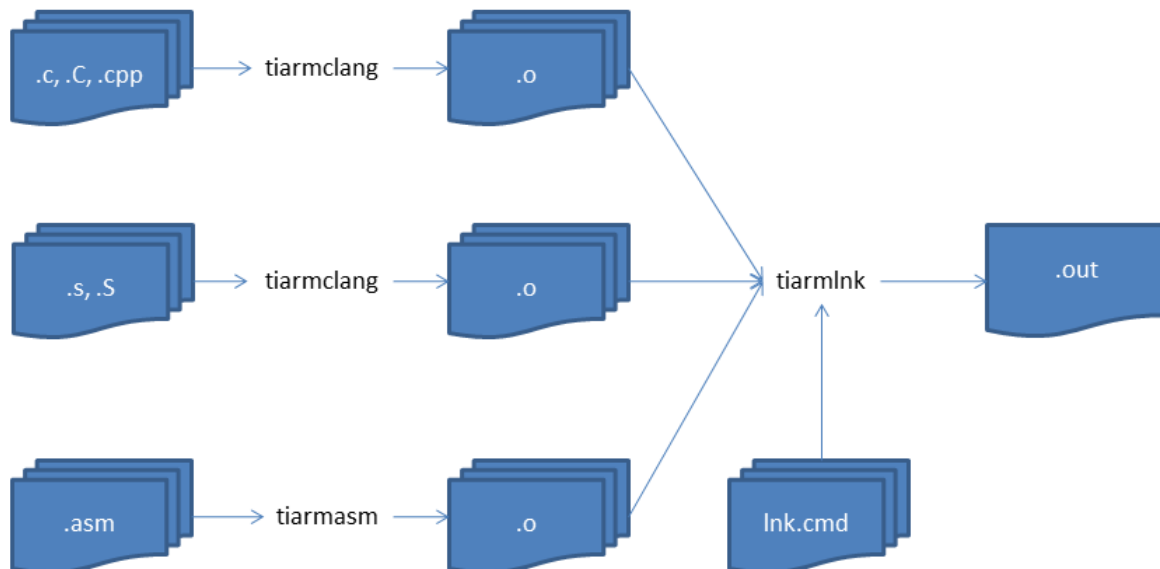
Figure 3.24: Linker's Role in Development Flow

3.10.3 Invoking the Linker

The default behavior of the tiarmclang compiler is to compile specified C, C++, or assembly source files into temporary object files and then pass those object files along with any explicitly specified object files and any specified linker options to the linker.

Alternately, if you specify only object files as input to the tiarmclang compiler, the compiler passes those files to the linker along with any specified options that are applicable to the link.

- *Compile and Link*

- *Link-Only Using tiarmclang*

- *Passing Options to the Linker*

- *File and Path Names Containing Special Characters*

- *Wildcards in File, Section, and Symbol Patterns*

- *Specifying C/C++ Symbols with Linker Options*

Compile and Link

The general syntax for invoking the compiler and linker together is:

```
tiarmclang [options] [source file names] [object file names] [-
↪Wl,<linker options>]
```

In the following example, assume that the C code in **file1.c** references a data object that is defined in an object file named **file2.o**. The specified **tiarmclang** command compiles **file1.c** into a temporary object file. That object file, along with **file2.o** and a linker command file, **link_test.cmd**, is input to the linker and linked with applicable object files from the tiarmclang runtime libraries to create an executable output file named **test.out**:

```
tiarmclang -mcpu=cortex-m0 file1.c file2.o -o test.out -Wl,link_
↪test.cmd
```

Note that there is no mention of the tiarmclang runtime libraries on the **tiarmclang** command line or inside the **link_test.cmd** linker command file. When the linker is invoked from the **tiarmclang** command line, the tiarmclang compiler implicitly tells the linker where to find applicable runtime libraries like the C runtime library (libc.a). In the above **tiarmclang** command line, the **-Wl,** prefix in front of the specification of the **link_test.cmd** file name indicates to the compiler that the

link_test.cmd file should be input directly into the linker. (You can also use the **-Xlinker** prefix for this purpose.)

If you add the verbose (**-v**) option to the above **tiarmclang** command, the output shows exactly how the linker (**tiarmlnk**) was invoked and with what options. For example, this command:

```
tiarmclang -mcpu=cortex-m0 -v file1.c file2.o -o test.out -Wl,
↪link_test.cmd
```

shows the following with regards to how **tiarmlnk** is invoked by the tiarmclang compiler:

```
<install directory>/bin/tiarmlnk -I<install directory>/lib
-o test.out /tmp/file1-98472f.o file2.o link_test.cmd
--start-group -llibc++.a -llibc++abi.a -llibc.a -llibsys.a
-llibsysbm.a -llibclang_rt.builtins.a -llibclang_rt.profile.a --
↪end-group
```

In the above invocation of the linker, the compiler inserts a *-I<install directory>/lib* option that tells the linker where to find the tiarmclang runtime libraries. The compiler also inserts the *--start_group/--end_group* options to specify which runtime libraries are incorporated into the link.

Link-Only Using tiarmclang

When only object files are specified as input to the **tiarmclang** compiler command, the compiler passes those files to the linker along with any other specified options that are applicable to the link.

```
tiarmclang [options] [object file names] [-Wl,<linker options>]
```

As in the default case of "Compile and Link" described above, a **-Wl,** or **-Xlinker** prefix must be specified in front of options that are intended for the linker. For example, this **tiarmclang** command:

```
tiarmclang -mcpu=cortex-m0 file1.o file2.o -o test.out -Wl,link_
↪test.cmd
```

invokes the linker as follows:

```
<install directory>/bin/tiarmlnk -I<install directory>/lib
-o test.out file1.o file2.o link_test.cmd
--start-group -llibc++.a -llibc++abi.a -llibc.a -llibsys.a
-llibsysbm.a -llibclang_rt.builtins.a -llibclang_rt.profile.a --
↪end-group
```

As in the "Compile and Link" case, the compiler inserts a *-I<install directory>/lib* option that tells the linker where to find the tiarmclang runtime libraries. The compiler also inserts the --

start_group/--end_group option list that specifies exactly which runtime libraries are incorporated into the link.

Passing Options to the Linker

The **tiarmclang** command line provides the following ways to pass options to the linker:

- The -Wl option passes a comma-separated list of options to the linker.
- The -Xlinker option passes a single option to the linker and can be used multiple times on the same command line.
- A linker command file can specify options to pass to the linker.

For example, the following command line passes several linker options using the -Wl option:

```
tiarmclang -mcpu=cortex-m0 hello.c -o a.out -Wl,-stack=0x8000,--
↪ram_model,link_test.cmd
```

The following command line passes the same linker options using the -Xlinker option:

```
tiarmclang -mcpu=cortex-m0 hello.c -o a.out -Xlinker -
↪stack=0x8000 -Xlinker --ram_model -Xlinker link_test.cmd
```

The following lines from a linker command file, pass the same linker options to the linker:

```
/
↪*************************************************************/
↪
/* Example Linker Command File                                   ␣
↪   */
/
↪*************************************************************/
↪
-stack  0x8000                        /* SOFTWARE STACK SIZE     ␣
↪    */
--ram_model                           /* INITIALIZE VARIABLES AT LOAD␣
↪TIME */
```

File and Path Names Containing Special Characters

A reference to a normal file name, such as *file.o* in a link command line or in a linker command file is handled as expected by the linker. You can also specify path information or include special characters, like hyphens, in a file name specification. In most cases, a file name specification containing path information should be properly interpreted by the linker, but in some cases, especially when a file name specification contains a special character, like a hyphen, you should enclose the file name specification in double-quotes to ensure that it is properly interpreted.

Specifically in the case of a hyphen, the reason that a file name specification containing a hyphen must be enclosed in double-quotes is because a hyphen can be legitimately interpreted as a subtraction operator.

For example, file or library names containing hyphens referenced in a linker command file without enclosing double-quotes cause problems at link time:

```
SECTIONS
{
  ....

  .mytext1     : { lib-with-dashes.lib(.text) } > 0x00010000
  .mytext2     : { name-with-dashes.o(.text)  } > 0x10000000


  ...
}
```

```
%> tiarmlnk.cmd -mcpu=cortex-m4 name-with-dashes.o -o a.out -Wl,
↪badlnk.cmd,-ma.map
"badlnk.cmd", line 24: error: cannot find file "lib"
"badlnk.cmd", line 24: error: -l must specify a filename
"badlnk.cmd", line 24: error: cannot find file "with"
"badlnk.cmd", line 24: error: cannot find file "dashes.lib"
"badlnk.cmd", line 25: error: cannot find file "name"
"badlnk.cmd", line 25: error: -l must specify a filename
"badlnk.cmd", line 25: error: cannot find file "with"
"badlnk.cmd", line 25: error: cannot find file "dashes.o"
"badlnk.cmd", line 24: warning: no matching section
"badlnk.cmd", line 25: warning: no matching section
"badlnk.cmd", line 24: warning: no matching section
"badlnk.cmd", line 24: warning: no matching section
"badlnk.cmd", line 25: warning: no matching section
"badlnk.cmd", line 25: warning: no matching section


undefined first referenced
 symbol       in file
```

(continues on next page)

```
--------  ----------------
my_func   name-with-dashes.o

error: unresolved symbols remain
error: errors encountered during linking; "a.out" not built
```

If the referenced file names are enclosed in double-quotes, the link succeeds:

```
SECTIONS
{
  ....

  .mytext1    : { "lib-with-dashes.lib"(.text) } > 0x00010000
  .mytext2    : { "name-with-dashes.o"(.text)  } > 0x10000000

  ...
}
```

```
%> tiarmlnk.cmd -mcpu=cortex-m4 name-with-dashes.o -o a.out -Wl,
 ↪goodlnk.cmd,-ma.map
%> cat a.map
...
SECTION ALLOCATION MAP

 output                                     attributes/
section    page    origin      length      input sections
--------   ----   ----------  ----------    ----------------
...

.mytext1   0     00010000    00000018
                  00010000    00000010       lib-with-dashes.lib :␣
↪my-func.o (.text.my_func)
                  00010010    00000008       libc.a : printf.c.obj␣
↪(.tramp.printf.1)

.mytext2   0     10000000    0000001c
                  10000000    00000014       name-with-dashes.o (.
↪text.main)
                  10000014    00000008       lib-with-dashes.lib :␣
↪my-func.o (.tramp.my_func.1)
```

It is recommended that if your file name specification contains unusual or special characters that might not be interpreted by the linker as an obvious part of a file or path name, then you should try

---

3.10. Linker Description   **483**

enclosing your file name specification in double-quotes to ensure that it is properly interpreted.

Wildcards in File, Section, and Symbol Patterns

The linker allows file, section, and symbol names to be specified using the asterisk (*) and question mark (?) wildcards. Using * matches any number of characters. Using ? matches a single character. Wildcards can make it easier to handle related objects, provided they follow a suitable naming convention.

For example:

```
mp3*.o        /* matches anything .o that begins with mp3    */
task?.o*      /* matches task1.o, task2.c.o, taskX.o55, etc. */

SECTIONS
{
    .fast_code: { *.o(*fast*) }                 > FAST_MEM
    .vectors  : { vectors.c.o(.vector:part1:*) > 0xFFFFFF00
    .str_code : { rts*.lib<str*.c.o>(.text) }   > S1ROM
}
```

Specifying C/C++ Symbols with Linker Options

The link-time symbol is the same as the high-level language name.

For more information on symbol names, see *tiarmnm - Name Utility*. For information specifically about C++ symbol naming, see *tiarmdem - C++ Name Demangler Utility*. See *Using Linker Symbols in C/C++ Applications* for information about referring to linker symbols in C/C++ code.

3.10.4 Linker Options

Linker options control linking operations. They can be placed on the command line or in a command file. Linker options must be preceded by a hyphen (-). Options can be separated from arguments (if they have them) by an optional space.

Contents:

Basic Options

The options listed in the subsections below control basic linker behavior. On the **tiarmclang** command line, they should be passed to the linker using the -Wl or -Xlinker option as described in *Passing Options to the Linker*.

- *Option Summary*
- *Name an Output Module (--output_file Option)*
- *Create a Map File (--map_file Option)*
- *Define Stack Size (--stack_size Option)*
- *Define Heap Size (--heap_size Option)*

Option Summary

--output_file (-o)

Names the executable output module. The default filename is a.out. See *Name an Output Module (--output_file Option)*.

--map_file (-m)

Produces a map or listing of the input and output sections, including holes, and places the listing in *filename*. See *Create a Map File (--map_file Option)*.

--stack_size (-stack)

Sets C system stack size to *size* bytes and defines a global symbol that specifies the stack size. Default = 2K bytes. See *Define Stack Size (--stack_size Option)*.

--heap_size (-heap)

Sets heap size (for the dynamic memory allocation in C) to *size* bytes and defines a global symbol that specifies the heap size. Default = 2K bytes. See *Define Heap Size (--heap_size Option)*.

Name an Output Module (--output_file Option)

The linker creates an output module when no errors are encountered. If you do not specify a filename for the output module, the linker gives it the default name a.out. If you want to write the output module to a different file, use the *--output_file* option. The syntax for the *--output_file* option is:

--output_file=ized*filename*

The *filename* is the new output module name.

This example links file1.c.o and file2.c.o and creates an output module named run.out:

```
tiarmclang -Wl,--output_file=run.out file1.c.o file2.c.o
```

Create a Map File (--map_file Option)

The syntax for the *--map_file* option is:

--map_file=_filename_

The linker map describes:

- Memory configuration
- Input and output section allocation
- Linker-generated copy tables
- Trampolines
- The addresses of external symbols after they have been relocated
- Hidden and localized symbols

The map file contains the name of the output module and the entry point; it can also contain up to three tables:

- A table showing the new memory configuration if any non-default memory is specified (memory configuration). This information is generated on the basis of the information in the MEMORY directive in the linker command file. For more about the MEMORY directive, see *The MEMORY Directive*. The table has the following columns;

    - **Name.** This is the name of the memory range specified with the MEMORY directive.

    - **Origin.** This specifies the starting address of a memory range.

    - **Length.** This specifies the length of a memory range.

    - **Unused.** This specifies the total amount of unused (available) memory in that memory area.

    - **Attributes.** This specifies one to four attributes associated with the named range:

        * R specifies that the memory can be read.

        * W specifies that the memory can be written to.

        * X specifies that the memory can contain executable code.

        * I specifies that the memory can be initialized.

- A table showing the linked addresses of each output section and the input sections that make up the output sections (section placement map). This information is generated on the basis of

the information in the SECTIONS directive in the linker command file. For more about the SECTIONS directive, see *The SECTIONS Directive*. This table has the following columns:

- **Output section.** This is the name of the output section specified with the SECTIONS directive.

- **Origin.** The first origin listed for each output section is the starting address of that output section. The indented origin value is the starting address of that portion of the output section.

- **Length.** The first length listed for each output section is the length of that output section. The indented length value is the length of that portion of the output section.

- **Attributes/input sections.** This lists the input file or value associated with an output section. If the input section could not be allocated, the map file indicates this with "FAILED TO ALLOCATE".

- A table showing each external symbol and its address sorted by symbol name.

- A table showing each external symbol and its address sorted by symbol address.

The following example links file1.c.o and file2.c.o and creates a map file called map.out:

```
tiarmclang file1.c.o file2.c.o -Wl,--map_file=a.map
```

Linker Example shows an example of a map file.

Define Stack Size (--stack_size Option)

The Arm C/C++ compiler uses an uninitialized section, .stack, to allocate space for the run-time stack. You can set the size of this section in bytes at link time with the *--stack_size* option. The syntax for the *--stack_size* option is:

--stack_size=_size_

The *size* must be a constant and is in bytes. This example defines a 4K byte stack:

```
tiarmclang -Wl,--stack_size=0x1000 /* defines a 4K heap (.stack␣
↪section)*/
```

If you specified a different stack size in an input section, the input section stack size is ignored. Any symbols defined in the input section remain valid; only the stack size is different.

When the linker defines the .stack section, it also defines a global symbol, __TI_STACK_SIZE, and assigns it a value equal to the size of the section. The default software stack size is 2K bytes. See *Using Linker Symbols in C/C++ Applications* for information about referring to linker symbols in C/C++ code.

To debug issues related to the stack size, we recommend using the CCS Stack Usage view to see the static stack usage of each function in the application. See Stack Usage View in CCS for more

information. Using the Stack Usage View requires that source code be built with *debug enabled*. This feature relies on the *–call_graph* capability provided by the *tiarmofd - Object File Display Utility*.

Define Heap Size (--heap_size Option)

The C/C++ compiler uses an uninitialized section called .sysmem for the C run-time memory pool used by malloc(). You can set the size of this memory pool at link time by using the *--heap_size* option. The syntax for the *--heap_size* option is:

--heap_size=*size*

The *size* must be a constant. This example defines a 4K byte heap:

```
tiarmclang -Wl,--heap_size=0x1000 /* defines a 4k heap (.sysmem␣
↪section)*/
```

The linker creates the .sysmem section only if there is a .sysmem section in an input file.

The linker also creates a global symbol, __TI_SYSMEM_SIZE, and assigns it a value equal to the size of the heap. The default size is 2K bytes. See *Using Linker Symbols in C/C++ Applications* for information about referring to linker symbols in C/C++ code.

File Search Path Options

The options listed in the subsections below control how the linker locates files, such as object libraries. On the **tiarmclang** command line they should be passed to the linker using the -Wl or -Xlinker option as described in *Passing Options to the Linker*.

- *Option Summary*
- *Alter the Library Search Algorithm (--library, --search_path)*
    - *Name an Alternate Library Directory (--search_path Option)*
    - *Exhaustively Read and Search Libraries (--reread_libs and --priority Options)*
- *Automatic Library Selection (--disable_auto_rts Option)*

Option Summary

--library (`-l`)

Names an archive library or link command *filename* as linker input. See *Alter the Library Search Algorithm (--library, --search_path)*.

--disable_auto_rts

Disables the automatic selection of a run-time-support library. See *Automatic Library Selection (--disable_auto_rts Option)*.

--priority (`-priority`)

Satisfies unresolved references by the first library that contains a definition for that symbol. See *Exhaustively Read and Search Libraries (--reread_libs and --priority Options)*.

--reread_libs (`-x`)

Forces rereading of libraries, which resolves back references. See *Exhaustively Read and Search Libraries (--reread_libs and --priority Options)*.

--search_path (`-i`)

Alters library-search algorithms to look in a directory named with *pathname* before looking in the default location. This option must appear before the *--library* option. See *Name an Alternate Library Directory (--search_path Option)*.

Alter the Library Search Algorithm (--library, --search_path)

Usually, when you want to specify a file as linker input, you simply enter the filename; the linker looks for the file in the current directory. For example, suppose the current directory contains the library *object.lib*. If this library defines symbols that are referenced in the file *file1.c.o*, this is how you link the files:

```
tiarmclang file1.c.o object.lib
```

If you want to use a file that is not in the current directory, use the *--library* linker option. The *--library* option's short form is *-l*. The syntax for this option is:

--library=[*pathname*] *filename*

The *filename* is the name of an archive, an object file, or linker command file. You can specify up to 128 search paths.

The *--library* option is not required when one or more members of an object library are specified for input to an output section. For more information about allocating archive members, see *Specifying Library or Archive Members as Input to Output Sections*.

You can adjust the linker's directory search algorithm using the *--search_path* linker option. When the *--library* option is applied to a file name specification, the linker searches for object files, object libraries, and linker command files in this order:

1. It searches directories named with the *--search_path* linker option. The *--search_path* option must appear before the *--library* option on the command line or in a command file.

2. It searches the current directory.

For example, let's suppose you have an object library named *my.lib* in the current work directory, and another version of the library with the same name in a sub-directory called *old_libs*. We can choose which version of *my.lib* is used in a link with the help of the *--search_path* (*-I*) and *--library* (*-l*) options.

In the following **tiarmclang** command, the *my.lib* in the current work directory is incorporated in the link since the reference to *my.lib* is **not** prefixed with the *--library* or *-l* option:

```
%> tiarmclang -mcpu=cortex-m4 use_my_lib.c -o a.out -Wl,-I./old_
↪libs,my.lib,-llnk.cmd
```

If the *-l* option is used as a prefix to the reference to *my.lib*, the linker finds and uses the version of *my.lib* from the *old_lib* directory in the link:

```
%> tiarmclang -mcpu=cortex-m4 use_my_lib.c -o a.out -Wl,-I./old_
↪libs,-lmy.lib,-llnk.cmd
```

Name an Alternate Library Directory (--search_path Option)

The *--search_path* option names an alternate directory that contains input files. The *--search_path* option's short form is *-I*. The syntax for this option is:

--search_path=*pathname*

The *pathname* names a directory that contains input files.

When the linker is searching for input files named with the *--library* option, it searches through directories named with *--search_path* first. Each *--search_path* option specifies only one directory, but you can have several *--search_path* options per invocation. When you use the *--search_path* option to name an alternate directory, it must precede any *--library* option used on the command line or in a command file.

For example, assume that there are two archive libraries called *r.lib* and *lib2.lib* that reside in *ld* and *ld2* directories. The command below shows the directories that *r.lib* and *lib2.lib* reside in and how to use both libraries during a link. (Note that directory paths with forward slashes (/) can be used on both Unix and Windows tiarmclang command lines.)

```
tiarmclang f1.c.o f2.c.o -Wl,--search_path=/ld,--search_path=/
↪ld2,--library=r.lib,--library=lib2.lib
```

Exhaustively Read and Search Libraries (--reread_libs and --priority Options)

There are two ways to exhaustively search for unresolved symbols:

- Reread libraries if you cannot resolve a symbol reference (*--reread_libs*).

- Search libraries in the order that they are specified (*--priority*).

The linker normally reads input files, including archive libraries, only once when they are encountered on the command line or in the command file. When an archive is read, any members that resolve references to undefined symbols are included in the link. If an input file later references a symbol defined in a previously read archive library, the reference is not resolved.

With the *--reread_libs* option, you can force the linker to reread all libraries. The linker rereads libraries until no more references can be resolved. Linking using *--reread_libs* may be slower, so you should use it only as needed. For example, if *a.lib* contains a reference to a symbol defined in *b.lib*, and *b.lib* contains a reference to a symbol defined in *a.lib*, you can resolve the mutual dependencies by listing one of the libraries twice, as in:

```
tiarmclang -Wl,--library=a.lib,--library=b.lib,--library=a.lib
```

or you can force the linker to do it for you.

The *--priority* option provides an alternate search mechanism for libraries. Using *--priority* causes each unresolved reference to be satisfied by the first library that contains a definition for that symbol. For example:

```
objfile   references A
lib1      defines B
lib2      defines A, B; obj defining A references B

% tiarmclang objfile lib1 lib2
```

Under the existing model, objfile resolves its reference to *A* in *lib2*, pulling in a reference to *B*, which resolves to the *B* in *lib2*.

Under *--priority*, objfile resolves its reference to *A* in *lib2*, pulling in a reference to *B*, but now *B* is resolved by searching the libraries in order and resolves *B* to the first definition it finds, namely the one in *lib1*.

The *--priority* option is useful for libraries that provide overriding definitions for related sets of functions in other libraries without having to provide a complete version of the whole library.

For example, suppose you want to override versions of *malloc* and *free* defined in the *libc.a* without providing a full replacement for *libc.a*. Using *--priority* and linking your new library before *libc.a* guarantees that all references to *malloc* and *free* resolve to the new library.

The *--priority* option supports linking programs with a Runtime Operating System (RTOS) where situations like the one illustrated above occur.

Automatic Library Selection (--disable_auto_rts Option)

The *--disable_auto_rts* option disables the automatic selection of a run-time-support (RTS) library. See *Invoking the Compiler* for more on the automatic selection process.

Command File Preprocessing Options

The options listed in the subsections below control how the linker preprocesses linker command files. On the **tiarmclang** command line they should be passed to the linker using the *-Wl* or *-Xlinker* option as described in *Passing Options to the Linker*.

- *Option Summary*
- *Linker Command File Preprocessing (--disable_pp, --define and --undefine Options)*

Option Summary

--define

Predefines *name* as a preprocessor macro. See *Linker Command File Preprocessing (--disable_pp, --define and --undefine Options)*.

--undefine

Removes the preprocessor macro *name*. See *Linker Command File Preprocessing (--disable_pp, --define and --undefine Options)*.

--disable_pp

Disables preprocessing for command files. See *Linker Command File Preprocessing (--disable_pp, --define and --undefine Options)*.

--honor_cmdfile_order

Specify the order of output sections according to the order listed in a linker command file. This also caused other placement constraints, such as placement to a specific memory address, to be honored before falling back to command file order.

Linker Command File Preprocessing (--disable_pp, --define and --undefine Options)

The linker preprocesses linker command files using a standard C preprocessor. Therefore, the command files can contain well-known preprocessing directives such as #define, #include, and #if / #endif.

Three linker options control the preprocessor:

- **--disable_pp** - Disables preprocessing for command files

- **--define**=*name*[=*val*] - Predefines &name* as a preprocessor macro

- **--undefine**=*name* - Removes the macro *name*

The compiler has *--define* and *--undefine* options with the same meanings. However, the linker options are distinct; only *--define* and *--undefine* options passed to the linker with *-Wl* or *-Xlinker* affect linker preprocessing. For example:

```
tiarmclang --define=FOO=1 main.c -Wl,--define=BAR=2 lnk.cmd
```

The linker sees only the *--define* for BAR; the compiler only sees the *--define* for FOO.

When one command file #includes another, preprocessing context is carried from parent to child in the usual way (that is, macros defined in the parent are visible in the child). However, when a command file is invoked other than through #include, either on the command line or by the typical way of being named in another command file, preprocessing context is **not** carried into the nested file. The exception to this is *--define* and *--undefine* options, which apply globally from the point they are encountered. For example:

```
--define GLOBAL
#define LOCAL


#include "incfile.cmd"     /* sees GLOBAL and LOCAL */
nestfile.cmd               /* only sees GLOBAL      */
```

Two cautions apply to the use of *--define* and *--undefine* in command files. First, they have global effect as mentioned above. Second, since they are not actually preprocessing directives themselves, they are subject to macro substitution, probably with unintended consequences. This effect can be defeated by quoting the symbol name. For example:

```
--define MYSYM=123
--undefine MYSYM          /* expands to --undefine 123 (!) */
--undefine "MYSYM"        /* ahh, that's better            */
```

The linker uses the same search paths to find #include files as it does to find libraries. That is, #include files are searched in the following places:

1. If the #include file name is in quotes (rather than <brackets>), in the directory of the current file

2. In the list of directories specified with *--search_path* options or environment variables (see *Alter the Library Search Algorithm (--library, --search_path)*).

There are two exceptions: relative pathnames (such as "../name") always search the current directory; and absolute pathnames (such as "/usr/tools/name") bypass search paths entirely.

The linker provides the built-in macro definitions in the following list. The availability of these macros within the linker is determined by the command-line options used, not the build attributes of

the files being linked. If these macros are not set as expected, confirm that your project's command line uses the correct compiler option settings.

- __DATE__ Expands to the compilation date in the form "*mmm dd yyyy*"

- __FILE__ Expands to the current source filename

- __TI_COMPILER_VERSION__ Defined to a 7-9 digit integer, depending on if X has 1, 2, or 3 digits. The number does not contain a decimal. For example, version 3.2.1 is represented as 3002001. The leading zeros are dropped to prevent the number being interpreted as an octal.

- __TI_EABI__ Defined to 1 if EABI is enabled; otherwise, it is undefined.

- __TIME__ Expands to the compilation time in the form "*hh:mm:ss*"

- __TI_ARM__ Always defined |

- __TI_ARM_V6M0__ Defined to 1 if the v6M0 architecture (Cortex-M0) is targeted (the -mcpu=cortex-m0 option is used); otherwise, it is undefined.

- __TI_ARM_V7__ Defined to 1 if any v7 architecture (Cortex) is targeted; otherwise, it is undefined.

- __TI_ARM_V7M__ Defined to 1 if any Cortex-M architecture is targeted; otherwise, it is undefined.

- __TI_ARM_V7M3__ Defined to 1 if the v7M3 architecture (Cortex-M3) is targeted (the -mcpu=cortex-m3 option is used); otherwise, it is undefined.

- __TI_ARM_V7M4__ Defined to 1 if the v7M4 architecture (Cortex-M4) is targeted (the -mcpu=cortex-m4 option is used); otherwise, it is undefined.

- __TI_ARM_V7R4__ Defined to 1 if the v7R4 architecture (Cortex-R4) is targeted (the -mcpu=cortex-r4 option is used); otherwise, it is undefined.

- __TI_ARM_V7R5__ Defined to 1 if the v7R5 architecture (Cortex-R5) is targeted (the -mcpu=cortex-r5 option is used); otherwise, it is undefined.

Diagnostic Options

The options listed in the subsections below control how the linker generates diagnostic messages. On the **tiarmclang** command line they should be passed to the linker using the *-Wl* or *-Xlinker* option as described in *Passing Options to the Linker*.

- *Option Summary*

- *Control Linker Diagnostics*

- *Disable Name Demangling (--no_demangle)*

> • *Display a Message When an Undefined Output Section Is Created (--warn_sections)*

Option Summary

--diag_error

Categorizes the diagnostic identified by *num* as an error. See *Control Linker Diagnostics*.

--diag_remark

Categorizes the diagnostic identified by *num* as a remark. See *Control Linker Diagnostics*.

--diag_suppress

Suppresses the diagnostic identified by *num*. See *Control Linker Diagnostics*.

--diag_warning

Categorizes the diagnostic identified by *num* as a warning. See *Control Linker Diagnostics*.

--display_error_number

Displays a diagnostic's identifiers along with its text. See *Control Linker Diagnostics*.

--emit_references:file`[=\ *file*]`

Emits a file containing section information. The information includes section size, symbols defined, and references to symbols. See *Control Linker Diagnostics*.

--emit_warnings_as_errors `(-pdew)`

Treats warnings as errors. See *Control Linker Diagnostics*.

--issue_remarks

Issues remarks (non-serious warnings). See *Control Linker Diagnostics*.

--no_demangle

Disables demangling of symbol names in diagnostics. See *Disable Name Demangling (--no_demangle)*.

--no_warnings

Suppresses warning diagnostics (errors are still issued). See *Control Linker Diagnostics*.

--set_error_limit

Sets the error limit to *num*. The linker abandons linking after this number of errors. (The default is 100.) See *Control Linker Diagnostics*.

--verbose_diagnostics

Provides verbose diagnostics that display the original source with line-wrap. See *Control Linker Diagnostics*.

--warn_sections `(-w)`

Displays a message when an undefined output section is created. See *Display a Message When an Undefined Output Section Is Created (--warn_sections)*.

Control Linker Diagnostics

The linker honors certain C/C++ compiler options to control linker-generated diagnostics. The diagnostic options must be specified without passing them directly to the linker with -Wl or -Xlinker.

--diag_error=num

> Categorize the diagnostic identified by *num* as an error. To find the numeric identifier of a diagnostic message, use the *--display_error_number* option first in a separate link. Then use *--diag_error=num* to recategorize the diagnostic as an error. You can only alter the severity of discretionary diagnostics.

--diag_remark=num

> Categorize the diagnostic identified by *num* as a remark. To find the numeric identifier of a diagnostic message, use the *--display_error_number* option first in a separate link. Then use *--diag_remark=num* to recategorize the diagnostic as a remark. You can only alter the severity of discretionary diagnostics.

--diag_suppress=num

> Suppress the diagnostic identified by *num*. To find the numeric identifier of a diagnostic message, use the *--display_error_number* option first in a separate link. Then use *--diag_suppress=num* to suppress the diagnostic. You can only suppress discretionary diagnostics.

--diag_warning=num

> Categorize the diagnostic identified by *num* as a warning. To find the numeric identifier of a diagnostic message, use the *--display_error_number* option first in a separate link. Then use *--diag_warning=num* to recategorize the diagnostic as a warning. You can only alter the severity of discretionary diagnostics.

--display_error_number

> Display a diagnostic message's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (*--diag_suppress*, *--diag_error*, *--diag_remark*, and *--diag_warning*). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix "-D"; otherwise, no suffix is present. See *Diagnostic Options* for more information on controlling diagnostic messages.

--emit_references:file[=filename]

> Emits a file containing section information. The information includes section size, symbols defined, and references to symbols. This information allows you to determine why each section is included in the linked application. The output file is a simple ASCII text file. The *filename* is used as the base name of a file created. For example, *--emit_references:file=myfile* generates a file named myfile.txt in the current directory.

--emit_warnings_as_errors
> Treat all warnings as errors. This option cannot be used with the *--no_warnings* option. The

--diag_remark option takes precedence over this option. This option takes precedence over the *--diag_warning* option.

--issue_remarks

Issue remarks (non-serious warnings), which are suppressed by default.

--no_warnings

Suppress warning diagnostics (errors are still issued).

--set_error_limit=num

Set the error limit to *num*, which can be any decimal value. The linker abandons linking after this number of errors. (The default is 100.)

--verbose_diagnostics

Provide verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line.

Disable Name Demangling (--no_demangle)

By default, the linker uses demangled symbol names in diagnostics. For example:

undefined symbol	first referenced in file
ANewClass::getValue()	test.cpp.o

The *--no_demangle* option instead shows the linkname for symbols in diagnostics. For example:

undefined symbol	first referenced in file
_ZN9ANewClass8getValueEv	test.cpp.o

For information on referencing symbol names, see *tiarmnm - Name Utility*. For information specifically about C++ symbol naming, see *tiarmdem - C++ Name Demangler Utility*.

Display a Message When an Undefined Output Section Is Created (--warn_sections)

In a linker command file, you can set up a SECTIONS directive that describes how input sections are combined into output sections. However, if the linker encounters one or more input sections that do not have a corresponding output section defined in the SECTIONS directive, the linker combines the input sections that have the same name into an output section with that name. By default, the linker does not display a message to tell you that this occurred.

You can use the *--warn_sections* option to cause the linker to display a message when it creates a new output section.

For more information about the SECTIONS directive, see *The SECTIONS Directive*. For more information about the default actions of the linker, see *Default Placement Algorithm*.

Linker Output Options

The options listed in the subsections below control how the linker generates output. On the **tiarm-clang** command line they should be passed to the linker using the *-Wl* or *-Xlinker* option as described in *Passing Options to the Linker*.

- *Option Summary*
- *Relocation Capabilities (--absolute_exe and --relocatable Options)*
    - *Producing an Absolute Output Module (--absolute_exe option)*
    - *Producing a Relocatable Output Module (--relocatable option)*
    - *Producing an Executable, Relocatable Output Module (-ar Option)*
- *Error Correcting Code Testing (--ecc Options)*
- *Managing Map File Contents (--mapfile_contents Option)*
- *Generate XML Link Information File (--xml_link_info Option)*
- *Generate XML Function Hash Table (--gen_xml_func_hash)*

Option Summary

--absolute_exe (-a)

Produces an absolute, executable module. This is the default; if neither *--absolute_exe* nor *--relocatable* is specified, the linker acts as if *--absolute_exe* were specified. See *Producing an Absolute Output Module (--absolute_exe option)*.

--ecc={ on \| off }

Enable linker-generated Error Correcting Codes (ECC). The default is off. See *Error Correcting Code Testing (--ecc Options)* and *Configuring Error Correcting Code (ECC) with the Linker*.

--ecc:data_error

Inject the specified errors into the output file for testing. See *Error Correcting Code Testing (--ecc Options)* and *Configuring Error Correcting Code (ECC) with the Linker*.

--ecc:ecc_error

Inject the specified errors into the Error Correcting Code (ECC) for testing. See *Error Correcting Code Testing (--ecc Options)* and *Configuring Error Correcting Code (ECC) with the Linker*.

--mapfile_contents

Controls the information that appears in the map file. See *Managing Map File Contents (--mapfile_contents Option)*.

--relocatable (-r)

> Produces a nonexecutable, relocatable output module. See *Producing a Relocatable Output Module (--relocatable option)*.

--xml_link_info

> Generates a well-formed XML *file* containing detailed information about the result of a link. See *Generate XML Link Information File (--xml_link_info Option)*.

--gen_xml_func_hash

> When the *–gen_xml_func_hash* linker option is combined with the *–xml_link_info* linker option, the linker includes a function hash table in the *–xml_link_info* output. See *Generate XML Function Hash Table (--gen_xml_func_hash)*.

Relocation Capabilities (--absolute_exe and --relocatable Options)

The linker performs relocation, which is the process of adjusting all references to a symbol when the symbol's address changes (*Symbolic Relocations*).

The linker supports two options (*--absolute_exe* and *--relocatable*) that allow you to produce an absolute or a relocatable output module. The linker also supports a third option (*-ar*) that allows you to produce an executable, relocatable output module.

When the linker encounters a file that contains no relocation or symbol table information, it issues a warning message (but continues executing). Relinking an absolute file can be successful only if each input file contains no information that needs to be relocated (that is, each file has no unresolved references and is bound to the same virtual address that it was bound to when the linker created it).

Producing an Absolute Output Module (--absolute_exe option)

When you use the *--absolute_exe* option without the *--relocatable* option, the linker produces an *absolute, executable output module*. Absolute files contain **no** relocation information. Executable files contain the following:

- Special symbols defined by the linker (see *Symbols Automatically Defined by the Linker*)
- A header that describes information such as the program entry point
- *No* unresolved references

The following example links file1.c.o and file2.c.o and creates an absolute output module called a.out:

```
tiarmclang -Wl,--absolute_exe file1.c.o file2.c.o
```

Note: If you do not use the *--absolute_exe* or the *--relocatable* option, the linker acts as if you specified *--absolute_exe*.

Producing a Relocatable Output Module (--relocatable option)

When you use the *--relocatable* option, the linker retains relocation entries in the output module. If the output module is relocated (at load time) or relinked (by another linker execution), use *--relocatable* to retain the relocation entries.

The linker produces a file that is not executable when you use the *--relocatable* option without the *--absolute_exe* option. A file that is not executable does not contain special linker symbols or an optional header. The file can contain unresolved references, but these references do not prevent creation of an output module.

This example links file1.c.o and file2.c.o and creates a relocatable output module called a.out:

```
tiarmclang -Wl,--relocatable file1.c.o file2.c.o
```

The output file a.out can be relinked with other object files or relocated at load time. (Linking a file that will be relinked with other files is called partial linking. For more information, see *Partial (Incremental) Linking*.)

Producing an Executable, Relocatable Output Module (-ar Option)

If you invoke the linker with both the *--absolute_exe* and *--relocatable* options, the linker produces an *executable, relocatable* object module. The output file contains the special linker symbols, an optional header, and all resolved symbol references; however, the relocation information is retained.

This example links file1.c.o and file2.c.o to create an executable, relocatable output module called xr.out:

```
tiarmclang -Wl,-ar,--output_file=xr.out file1.c.o file2.c.o
```

Error Correcting Code Testing (--ecc Options)

Error Correcting Codes (ECC) can be generated and placed in separate sections through the linker command file.

To enable ECC support, include *--ecc=on* as a linker option on the command line. By default ECC generation is off, even if the ECC directive and ECC specifiers are used in the linker command file. This allows you to fully configure ECC in the linker command file while still being able to quickly turn the code generation on and off via the command line. See *Configuring Error Correcting Code (ECC) with the Linker* for details on linker command file syntax to configure ECC support.

ECC uses extra bits to allow errors to be detected and/or corrected by a device. The ECC support provided by the linker is compatible with the ECC support in TI Flash memory on various TI devices. TI Flash memory uses a modified Hamming(72,64) code, which uses 8 parity bits for every 64 bits. Check the documentation for your Flash memory to see if ECC is supported. (ECC for read-write memory is handled completely in hardware at run time.)

After enabling ECC with the *--ecc=on* option, you can use the following command-line options to test ECC by injecting bit errors into the linked executable. These options let you specify an address where an error should appear and a bitmask of bits in the code/data at that address to flip. You can specify the address of the error absolutely or as an offset from a symbol. When a data error is injected, the ECC parity bits for the data are calculated as if the error were not present. This simulates bit errors that might actually occur and tests ECC's ability to correct different levels of errors.

The **--ecc:data_error option** injects errors into the load image at the specified location. The syntax is:

```
--ecc:data_error=(symbol+offset|address)[,page],bitmask
```

The *address* is the location of the minimum addressable unit where the error is to be injected. A *symbol+offset* can be used to specify the location of the error to be injected with a signed offset from that symbol. The *page* number is needed to make the location non-ambiguous if the address occurs on multiple memory pages. The *bitmask* is a mask of the bits to flip; its width should be the width of an addressable unit.

For example, the following command line flips the least-significant bit in the byte at the address 0x100, making it inconsistent with the ECC parity bits for that byte:

```
tiarmclang test.c -Xlinker --ecc:data_error=0x100,0x01 -Xlinker -
 ↪o=test.out
```

The following command flips two bits in the third byte of the code for main():

```
tiarmclang test.c -Xlinker --ecc:data_error=main+2,0x42 -Xlinker␣
 ↪-o=test.out
```

The **--ecc:ecc_error option** injects errors into the ECC parity bits that correspond to the specified location. Note that the ecc_error option can therefore only specify locations inside ECC input ranges, whereas the data_error option can also specify errors in the ECC output memory ranges. The syntax is:

```
--ecc:ecc_error=(symbol+offset|address)[,page],bitmask
```

The parameters for this option are the same as for *--ecc:data_error*, except that the *bitmask* must be exactly 8 bits. Mirrored copies of the affected ECC byte also contain the same injected error.

An error injected into an ECC byte with *--ecc:ecc_error* may cause errors to be detected at run time in any of the 8 data bytes covered by that ECC byte.

For example, the following command flips every bit in the ECC byte that contains the parity information for the byte at 0x200:

```
tiarmclang test.c -Xlinker --ecc:ecc_error=0x200,0xff -Xlinker -
↪o=test.out
```

The linker disallows injecting errors into memory ranges that are neither an ECC range nor the input range for an ECC range. The compiler can only inject errors into initialized sections.

Managing Map File Contents (--mapfile_contents Option)

The *--mapfile_contents* option assists with managing the content of linker-generated map files. The syntax for the *--mapfile_contents* option is:

--mapfile_contents=filter*[, filter]*

When the *--map_file* option is specified, the linker produces a map file containing information about memory usage, placement information about sections that were created during a link, details about linker-generated copy tables, and symbol values.

The *--mapfile_contents* option provides a mechanism for you to control what information is included in or excluded from a map file. When you specify *--mapfile_contents=help* from the command line, a help screen listing available filter options is displayed. The following filter options are available:

Attribute	Description	Default State
crctables	CRC tables	On
copytables	Copy tables	On
entry	Entry point	On
load_addr	Display load addresses	Off
memory	Memory ranges	On
modules	Module view	On
sections	Sections	On
sym_defs	Defined symbols per file	Off
sym_dp	Symbols sorted by data page	On
sym_name	Symbols sorted by name	On
sym_runaddr	Symbols sorted by run address	On
all	Enables all attributes	
none	Disables all attributes	

The *--mapfile_contents* option controls display filter settings by specifying a comma-delimited list of display attributes. When prefixed with the word no, an attribute is disabled instead of enabled. For example:

```
--mapfile_contents=copytables,noentry
--mapfile_contents=all,nocopytables
--mapfile_contents=none,entry
```

By default, those sections that are currently included in the map file when the *--map_file* option is specified are included. The filters specified in the *--mapfile_contents* options are processed in the order that they appear in the command line. In the third example above, the first filter, none, clears all map file content. The second filter, entry, then enables information about entry points to be included in the generated map file. That is, when *--mapfile_contents=none,entry* is specified, the map file contains *only* information about entry points.

The load_addr and sym_defs attributes are both disabled by default.

If you turn on the load_addr filter, the map file includes the load address of symbols that are included in the symbol list in addition to the run address (if the load address is different from the run address).

You can use the sym_defs filter to include information sorted on a file by file basis. You may find it useful to replace the sym_name, sym_dp, and sym_runaddr sections of the map file with the sym_defs section by specifying the following *--mapfile_contents* option:

```
--mapfile_contents=nosym_name,nosym_dp,nosym_runaddr,sym_defs
```

By default, information about global symbols defined in an application are included in tables sorted

by name, data page, and run address. If you use the *--mapfile_contents=sym_defs* option, static variables are also listed.

Generate XML Link Information File (--xml_link_info Option)

The linker supports the generation of an XML link information file through the *--xml_link_info=file* option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker generated map file.

See *XML Link Information File Description* for details about the contents of the XML link information file.

Generate XML Function Hash Table (--gen_xml_func_hash)

In the Sitara OpTI-Flash multicore context, the ability to identify common functions across multiple executables is desired in order to allow users to abstract these functions out and place them in shared memory in order reduce individual executable size. This is also known as "OpTI-SHARE". In order to identify common functions in a meaningful way (where function name and size are not enough), the tiarmclang linker can generate an MD5 hash based on the function's raw data prior to relocation and emit it within a table of function symbols in the linker-generated XML link info file.

The linker also generates a list of referenced data sections from each global function uniquely identified by their object component IDs. Common read-only data sections can also be allocated in shared memory. However, writes to read-write data sections from common code must be managed through hardware address translation available on the device (aka "RAT"). These referenced data section lists can also be used in conjunction with "Smart Placement" where fast data access from frequently executed functions is desired.

When linking an application, the aforementioned table and referenced section lists are generated when the *--xml_link_info* option is used in conjunction with *--gen_xml_func_hash*. The *--xml_link_info* option can be given a specified file name to use for the output.

The generated table is designated by a `func_symbol_table` XML tag, with each global function represented by a `symbol` tag. The associated MD5 hash is indicated by a `value` tag and the referenced data section lists indicated in `refd_ro_sections` and `refd_rw_sections` tags for read-only (constant) data and read-write data, respectively. For example:

```
<func_symbol_table>
  <symbol>
    <name>func0</name>
    <sectname>.text.main</sectname>
    <value>b6e5b51736000aef4da6e8afb91846e4</value>
```

(continues on next page)

```
    </symbol>
    <symbol>
      <name>func1</name>
      <sectname>.text.foo</sectname>
      <value>b1b9d95dd364df1b53f4e8c571ddaf68</value>
    </symbol>
    <symbol>
      <name>func2</name>
      <refd_ro_sections>
        <object_component_ref idref="oc-92"/>
        <object_component_ref idref="oc-99"/>
      </refd_ro_sections>
      <refd_rw_sections>
        <object_component_ref idref="oc-94"/>
        <object_component_ref idref="oc-96"/>
        <object_component_ref idref="oc-97"/>
        <object_component_ref idref="oc-98"/>
      </refd_rw_sections>
    </symbol>
</func_symbol_table>
```

See *XML Link Information File Description* for details about the contents of the XML link information file.

Symbol Management Options

The options listed in the subsections below control how the linker manages symbols. On the **tiarmclang** command line, they should be passed to the linker using the *-Wl* or *-Xlinker* option as described in *Passing Options to the Linker*.

- *Option Summary*

- *Define an Entry Point (--entry_point Option)*

- *Change Symbol Localization (--globalize and --localize options)*

- *Make All Global Symbols Static (--make_static Option)*

- *Hiding Symbols (--hide and --unhide options)*

- *Disable Merging of Symbolic Debugging Information (--no_sym_merge Option)*

- *Strip Symbolic Information (--no_symtable Option)*

- *Retain Discarded Sections (--retain Option)*

> - *Scan All Libraries for Duplicate Symbol Definitions (--scan_libraries Option)*
>
> - *Mapping of Symbols (--symbol_map Option)*
>
> - *Introduce an Unresolved Symbol (--undef_sym Option)*

Option Summary

--entry_point (-e)

Defines a global symbol that specifies the primary entry point for the output module. See *Define an Entry Point (--entry_point Option)*.

--globalize

Changes the symbol linkage to global for symbols that match *pattern*. See *Hiding Symbols (--hide and --unhide options)*.

--hide

Hides global symbols that match *pattern*. See *Hiding Symbols (--hide and --unhide options)*.

--localize

Changes the symbol linkage to local for symbols that match *pattern*. See *Change Symbol Localization (--globalize and --localize options)*.

--make_global (-g)

Makes *symbol* global (overrides -h). See *Make All Global Symbols Static (--make_static Option)*.

--make_static (-h)

Makes all global symbols static. See *Make All Global Symbols Static (--make_static Option)*.

--no_sym_merge (-s)

Disables merging of symbolic debugging information. See *Disable Merging of Symbolic Debugging Information (--no_sym_merge Option)*.

--no_symtable (-s)

Strips symbol table information and line number entries from the output module. See *Strip Symbolic Information (--no_symtable Option)*.

--retain

Retains a list of sections that otherwise would be discarded. See *Retain Discarded Sections (--retain Option)*.

--scan_libraries (-scanlibs)

Scans all libraries for duplicate symbol definitions. See *Scan All Libraries for Duplicate Symbol Definitions (--scan_libraries Option)*.

`--symbol_map`

> Maps symbol references to a symbol definition of a different name. See *Mapping of Symbols (--symbol_map Option)*.

`--undef_sym` `(-u)`

> Places an unresolved external *symbol* into the output module's symbol table. See *Introduce an Unresolved Symbol (--undef_sym Option)*.

`--unhide`

> Reveals (un-hides) global symbols that match *pattern*. See *Hiding Symbols (--hide and --unhide options)*.

Define an Entry Point (--entry_point Option)

The memory address at which a program begins executing is called the *entry point*. When a loader loads a program into target memory, the program counter (PC) must be initialized to the entry point; the PC then points to the beginning of the program.

The linker can assign one of four values to the entry point. These values are listed below in the order in which the linker tries to use them. If you use one of the first three values, it must be an external symbol in the symbol table.

- The value specified by the *--entry_point* option. The syntax is *--entry_point=global_symbol* where *global_symbol* defines the entry point and must be defined as an external symbol of the input files. The external symbol name of C or C++ objects may be different than the name as declared in the source language. See *Entry Point*.

- The value of symbol _c_int00 (if present). The _c_int00 symbol *must* be the entry point if you are linking code produced by the C compiler.

- The value of symbol _main (if present)

- 0 (default value)

This example links file1.c.o and file2.c.o. The symbol begin is the entry point; begin must be defined as external in file1 or file2.

```
tiarmclang -Wl,--entry_point=begin file1.c.o file2.c.o
```

See *Using Linker Symbols in C/C++ Applications* for information about referring to linker symbols in C/C++ code.

Change Symbol Localization (--globalize and --localize options)

Symbol localization changes symbol linkage from global to local (static). This is used to obscure global symbols that should not be widely visible, but must be global because they are accessed by several modules in the library. The linker supports symbol localization through the *--localize* and *--globalize* linker options.

The syntax for these options are:

--localize=*'pattern'*

--globalize=*'pattern'*

The *pattern* is a "glob" (a string with optional ? or * wildcards). Use ? to match a single character. Use * to match zero or more characters.

The *--localize* option changes the symbol linkage to local for symbols matching the *pattern*.

The *--globalize* option changes the symbol linkage to global for symbols matching the *pattern*. The *--globalize* option only affects symbols that are localized by the *--localize* option. The *--globalize* option excludes symbols that match the pattern from symbol localization, provided the pattern defined by *--globalize* is more restrictive than the pattern defined by *--localize*.

See *Specifying C/C++ Symbols with Linker Options* for information about using C/C++ identifiers in linker options such as *--localize* and *--globalize*.

These options have the following properties:

- The *--localize* and *--globalize* options can be specified more than once on the command line.
- The order of *--localize* and *--globalize* options has no significance.
- A symbol is matched by only one pattern defined by either *--localize* or *--globalize*.
- A symbol is matched by the most restrictive pattern. Pattern A is considered more restrictive than Pattern B, if Pattern A matches a narrower set than Pattern B.
- It is an error if a symbol matches patterns from *--localize* and *--globalize* and if one does not supersede other. Pattern A supersedes pattern B if A can match everything B can, and some more. If Pattern A supersedes Pattern B, then Pattern B is said to more restrictive than Pattern A.
- These options affect final and partial linking.

In map files these symbols are listed under the **Localized Symbols** heading.

Make All Global Symbols Static (--make_static Option)

The *--make_static* option makes all global symbols static. Static symbols are not visible to externally linked modules. By making global symbols static, global symbols are essentially hidden. This allows external symbols with the same name (in different files) to be treated as unique.

The *--make_static* option effectively nullifies all .global assembler directives. All symbols become local to the module in which they are defined, so no external references are possible. For example, assume file1.c.o and file2.c.o both define global symbols called EXT. By using the *--make_static* option, you can link these files without conflict. The symbol EXT defined in file1.c.o is treated separately from the symbol EXT defined in file2.c.o.

```
tiarmclang -Wl,--make_static file1.c.o file2.c.o
```

The *--make_static* option makes all global symbols static. If you have a symbol that you want to remain global and you use the *--make_static* option, you can use the *--make_global* option to declare that symbol to be global. The *--make_global* option overrides the effect of the *--make_static* option for the symbol that you specify. The syntax for the *--make_global* option is:

--make_global=*global_symbol*

Hiding Symbols (--hide and --unhide options)

Symbol hiding prevents the symbol from being listed in the output file's symbol table. While localization is used to prevent name space clashes in a link unit (see *Change Symbol Localization (--globalize and --localize options)*), symbol hiding is used to obscure symbols that should not be visible outside a link unit. Such symbol names appear only as empty strings or "no name" in object file readers. The linker supports symbol hiding through the *--hide* and *--unhide* options.

The syntax for these options are:

--hide=*'pattern'*

--unhide=*'pattern'*

The *pattern* is a "glob" (a string with optional ? or * wildcards). Use ? to match a single character. Use * to match zero or more characters.

The *--hide* option hides global symbols with a linkname matching the *pattern*. It hides symbols matching the pattern by changing the name to an empty string. A global symbol that is hidden is also localized.

The *--unhide* option reveals (un-hides) global symbols that match the *pattern* that are hidden by the *--hide* option. The *--unhide* option excludes symbols that match pattern from symbol hiding provided the pattern defined by *--unhide* is more restrictive than the pattern defined by *--hide*.

These options have the following properties:

- The *--hide* and *--unhide* options can be specified more than once on the command line.

- The order of *--hide* and *--unhide* has no significance.

- A symbol is matched by only one pattern defined by either *--hide* or *--unhide*.

- A symbol is matched by the most restrictive pattern. Pattern A is considered more restrictive than Pattern B, if Pattern A matches a narrower set than Pattern B.

- It is an error if a symbol matches patterns from *--hide* and *--unhide* and one does not supersede the other. Pattern A supersedes pattern B if A can match everything B can and more. If Pattern A supersedes Pattern B, then Pattern B is said to more restrictive than Pattern A.

- These options affect final and partial linking.

In map files these symbols are listed under the **Hidden Symbols** heading.

Disable Merging of Symbolic Debugging Information (--no_sym_merge Option)

By default, the linker eliminates duplicate entries of symbolic debugging information. Such duplicate information is commonly generated when a C program is compiled for debugging. For example:

```
-[ header.h ]-
typedef struct
{
    <define some structure members>
} XYZ;

-[ f1.c ]-
#include "header.h"
...

-[ f2.c ]-
#include "header.h"
...
```

When these files are compiled for debugging, both f1.c.o and f2.c.o have symbolic debugging entries to describe type XYZ. For the final output file, only one set of these entries is necessary. The linker eliminates the duplicate entries automatically.

Strip Symbolic Information (--no_symtable Option)

The *--no_symtable* option creates a smaller output module by omitting symbol table information and line number entries. The *--no_symtable* option is useful for production applications when you do not want to disclose symbolic information to the consumer.

This example links file1.c.o and file2.c.o and creates an output module, stripped of line numbers and symbol table information, named nosym.out:

```
tiarmclang -Wl,--output_file=nosym.out,--no_symtable file1.c.o␣
 ↪file2.c.o
```

Using the *--no_symtable* option limits later use of a symbolic debugger.

Note: Stripping Symbolic Information The *--no_symtable* option is deprecated. To remove symbol table information, use the tiarmstrip utility as described in *tiarmstrip - Object File Stripping Tool*.

Retain Discarded Sections (--retain Option)

When *--unused_section_elimination* is on, the ELF linker does not include a section in the final link if it is not needed in the executable to resolve references. The *--retain* option tells the linker to retain a list of sections that would otherwise not be retained. This option accepts the wildcards * and ?. When wildcards are used, the argument should be in quotes. The syntax for this option is:

--retain=*sym_or_scn_spec*

The *--retain* option takes one of the following forms:

--retain=symbol_spec

> Specifying the symbol format retains sections that define *symbol_spec*. For example, this code retains sections that define symbols that start with *init*:

```
--retain="init*"
```

> You cannot specify *--retain="*"*.

--retain=file_spec(scn_spec[, **scn_spec, ...**])

> Specifying the file format retains sections that match one or more *scn_spec* from files matching the *file_spec*. For example, this code retains *.intvec* sections from all input files:

```
--retain="*(.int*)"
```

You can specify *--retain="*(*)"* to retain all sections from all input files. However, this does not prevent sections from library members from being optimized out.

--retain=ar_spec<mem_spec, **[mem_spec, ...]>(scn_spec[, scn_spec, . ..])**

Specifying the archive format retains sections matching one or more *scn_spec* from members matching one or more *mem_spec* from archive files matching *ar_spec*. For example, this code retains the *.text* sections from printf.c.o in the libc.a library:

```
--retain=-llibc.a<printf.c.o>(.text)
```

If the library is specified with the *--library* or *-l* option (*-llibc.a*) the library search path is used to search for the library.

Note: Using *"*<*>(scn_spec)"* or *"*<*>(*)"* as the argument to *--retain* will be ignored

You cannot specify *"*<*>(scn_spec)"* or *"*<*>(*)"* as the argument to a *--retain* option. Either of these arguments are detected and ignored with a warning diagnostic by the linker. If allowed, the linker would try to scan all object file members of all libraries referenced in the linker invocation, including any that are mentioned in linker command files that are referenced. This would also include all C++, C, and compiler runtime libraries that are implicitly referenced from the **tiarmclang** command line during a link.

Scan All Libraries for Duplicate Symbol Definitions (--scan_libraries Option)

The *--scan_libraries* option scans all libraries during a link looking for duplicate symbol definitions to those symbols that are actually included in the link. The scan does not consider absolute symbols or symbols defined in COMDAT sections. The *--scan_libraries* option helps determine those symbols that were actually chosen by the linker over other existing definitions of the same symbol in a library.

The library scanning feature can be used to check against unintended resolution of a symbol reference to a definition when multiple definitions are available in the libraries.

Mapping of Symbols (--symbol_map Option)

Symbol mapping allows a symbol reference to be resolved by a symbol with a different name. Symbol mapping allows functions to be overridden with alternate definitions. This feature can be used to patch in alternate implementations, which provide patches (bug fixes) or alternate functionality. The syntax for the *--symbol_map* option is:

--symbol_map=*refname=defname*

For example, the following code makes the linker resolve any references to foo by the definition foo_patch:

```
--symbol_map=foo=foo_patch
```

The *--symbol_map* option is now supported even if *-flto* was used when compiling and linking.

The string passed with the *--symbol_map* option should contain no spaces and not be surrounded by quotes. This allows the same linker option syntax to work on the command line, in a linker command file, and in an options file.

Introduce an Unresolved Symbol (--undef_sym Option)

The *--undef_sym* option introduces the linkname for an unresolved symbol into the linker's symbol table. This forces the linker to search a library and include the member that defines the symbol. The linker must encounter the *--undef_sym* option **before** it links in the member that defines the symbol. The syntax for the *--undef_sym* option is:

--undef_sym=symbol

For example, suppose a library named rtsv4_A_be_eabi.lib contains a member that defines the symbol symtab; none of the object files being linked reference symtab. However, suppose you plan to relink the output module and you want to include the library member that defines symtab in this link. Using the *--undef_sym* option as shown below forces the linker to search rtsv4_A_be_eabi.lib for the member that defines symtab and to link in the member.

```
tiarmclang -Wl,--undef_sym=symtab file1.c.o file2.c.o rtsv4_A_be_
↪eabi.lib
```

If you do not use *--undef_sym*, this member is not included, because there is no explicit reference to it in file1.c.o or file2.c.o.

Run-Time Environment Options

The options listed in the subsections below control how the linker manages the run-time environment. See *Linking for Run-Time Initialization* for more about the run-time environment.

On the **tiarmclang** command line they should be passed to the linker using the -Wl or -Xlinker option as described in *Passing Options to the Linker*.

- *Option Summary*
- *Allocate Memory for Use by the Loader to Pass Arguments (–arg_size Option)*
- *Set Default Fill Value (--fill_value Option)*

- *C Language Options (--ram_model and --rom_model Options)*

Option Summary

--arg_size (`--args`)

Allocates memory to be used by the loader to pass arguments. See *Allocate Memory for Use by the Loader to Pass Arguments (–arg_size Option)*.

--cinit_hold_wdt=`{on|off}`

Hold (on) or do not hold (off) watchdog timer during cinit auto-initialization. See *Initialization of Cinit and Watchdog Timer Hold*.

--fill_value (`-f`)

Sets default fill values for holes within output sections; *fill_value* is a 32-bit constant. See *Set Default Fill Value (--fill_value Option)*.

--minimize_trampolines

Places sections to minimize number of far trampolines required. See *Minimizing the Number of Trampolines Required (--minimize_trampolines Option)*.

--ram_model (`-cr`)

Initializes variables at load time. See *C Language Options (--ram_model and --rom_model Options)*.

--rom_model (`-c`)

Autoinitializes variables at run time. See *C Language Options (--ram_model and --rom_model Options)*.

--trampolines

Generates far call trampolines; on by default. See *Generate Far Call Trampolines (--trampolines Option)*.

--trampoline_min_spacing

When trampoline reservations are spaced more closely than the specified limit, tries to make them adjacent. See *Making Trampoline Reservations Adjacent (--trampoline_min_spacing Option)*.

Allocate Memory for Use by the Loader to Pass Arguments (–arg_size Option)

The --arg_size option instructs the linker to allocate memory to be used by the loader to pass arguments from the command line of the loader to the program. The syntax of the --arg_size option is:

--arg_size = *size*

The *size* is the number of bytes to be allocated in target memory for command-line options.

By default, the linker creates the __c_args__ symbol and sets it to -1. When you specify --arg_size = *size*, the following occur:

- The linker creates an uninitialized section named .args of *size* bytes.

- The __c_args__ symbol contains the address of the .args section.

The loader and the target boot code use the .args section and the __c_args__ symbol to determine whether and how to pass arguments from the host to the target program. See *Arguments to main* for more information.

Set Default Fill Value (--fill_value Option)

The --fill_value option fills the holes formed within output sections. The syntax for the option is:

--fill_value=value

The argument *value* is a 32-bit constant (up to eight hexadecimal digits). If you do not use --fill_value, the linker uses 0 as the default fill value.

This example fills holes with the hexadecimal value ABCDABCD:

```
tiarmclang -Wl,--fill_value=0xABCDABCD file1.c.o file2.c.o
```

C Language Options (--ram_model and --rom_model Options)

The --ram_model and --rom_model options cause the linker to use linking conventions that are required by the C compiler. Both options inform the linker that the program is a C program and requires a boot routine.

- The --ram_model option tells the linker to initialize variables at load time.

- The --rom_model option tells the linker to autoinitialize variables at run time.

No default startup model is specified to the linker when the tiarmclang compiler runs the linker. Therefore, either the `--rom_model` (`-c`) or `--ram_model` (`-cr`) option must be passed to the linker on the **tiarmclang** command line using the `-Wl` or `-Xlinker` option or in the linker command file. For example:

```
tiarmclang -mcpu=cortex-m4 hello.c -o hello.out -Wl,-c,-llnk.cmd,
↪-mhello.map
```

If neither the `-c` or `-cr` option is specified to tiarmclang when running the linker, the linker expects an entry point for the linked application to be identified (using the `-e=<symbol>` linker option). If `-c` or `-cr` is specified, then the linker assumes that the program entry point is _c_int00, which performs any needed auto-initialization and system setup, then calls the user's main() function.

For more information, see *Linking C/C++ Code*, *Autoinitializing Variables at Run Time (--rom_model)*, and *Initializing Variables at Load Time (--ram_model)*.

Generate Far Call Trampolines (--trampolines Option)

The Arm device has PC-relative call and PC-relative branch instructions whose range is smaller than the entire address space. When these instructions are used, the destination address must be near enough to the instruction that the difference between the call and the destination fits in the available encoding bits. If the called function is too far away from the calling function, the linker generates an error or generates a trampoline, depending on the setting of the --trampolines option (on or off).

The alternative to a PC-relative call is an absolute call, which is often implemented as an indirect call: load the called address into a register, and call that register. This is often undesirable because it takes more instructions (speed- and size-wise) and requires an extra register to contain the address.

By default, the compiler generates calls that may require a trampoline if the destination is too far away. On some architectures, this type of call is called a "near call."

The --trampolines option allows you to control the generation of trampolines. When set to "on", this option causes the linker to generate a trampoline code section for each call that is linked out-of-range of its called destination. The trampoline code section contains a sequence of instructions that performs a transparent long branch to the original called address. Each calling instruction that is out-of-range from the called function is redirected to the trampoline.

The syntax for this option is:

--trampolines[=on|off]

The default setting is on. For Arm, trampolines are turned on by default.

For example, in a section of C code the bar function calls the foo function. The compiler generates this code for the function:

```
bar:
        ...
      call    foo     ; call the function "foo"
        ...
```

If the foo function is placed out-of-range from the call to foo that is inside of bar, then with --trampolines the linker changes the original call to foo into a call to foo_trampoline as shown:

```
bar:
      ...
      call    foo_trampoline  ; call a trampoline for foo
      ...
```

The above code generates a trampoline code section called foo_trampoline, which contains code that executes a long branch to the original called function, foo. For example:

```
foo_trampoline:
      branch_long    foo
```

Trampolines can be shared among calls to the same called function. The only requirement is that all calls to the called function be linked near the called function's trampoline.

When the linker produces a map file (the --map_file option) and it has produced one or more trampolines, then the map file contains statistics about what trampolines were generated to reach which functions. A list of calls for each trampoline is also provided in the map file.

Note: The Linker Assumes R13 Contains the Stack Pointer Assembly language programmers must be aware that the linker assumes R13 contains the stack pointer. The linker must save and restore values on the stack in trampoline code that it generates. If you do not use R13 as the stack pointer, you should use the linker option that disables trampolines, --trampolines=off. Otherwise, trampolines could corrupt memory and overwrite register values.

Advantages and Disadvantages of Using Trampolines

The advantage of using trampolines is that you can treat all calls as near calls, which are faster and more efficient. You only need to modify those calls that don't reach. In addition, there is little need to consider the relative placement of functions that call each other. Cases where calls must go through a trampoline are less common than near calls.

While generating far call trampolines provides a more straightforward solution, trampolines have the disadvantage that they are somewhat slower than directly calling a function. They require both a call and a branch. Additionally, while inline code could be tailored to the environment of the call, trampolines are generated in a more general manner, and may be slightly less efficient than inline code.

An alternative method to creating a trampoline code section for a call that cannot reach its called function is to actually modify the source code for the call. In some cases this can be done without affecting the size of the code. However, in general, this approach is extremely difficult, especially when the size of the code is affected by the transformation.

Minimizing the Number of Trampolines Required (--minimize_trampolines Option)

The --minimize_trampolines option attempts to place sections so as to minimize the number of far call trampolines required, possibly at the expense of optimal memory packing. The syntax is:

--minimize_trampolines=postorder

The argument selects a heuristic to use. The postorder heuristic attempts to place functions before their callers, so that the PC-relative offset to the callee is known when the caller is placed. By placing the callee first, its address is known when the caller is placed so the linker can definitively know if a trampoline is required.

Making Trampoline Reservations Adjacent (--trampoline_min_spacing Option)

When a call is placed and the callee's address is unknown, the linker must provisionally reserve space for a far call trampoline in case the callee turns out to be too far away. Even if the callee ends up being close enough, the trampoline reservation can interfere with optimal placement for very large code sections.

When trampoline reservations are spaced more closely than the specified limit, use the --trampoline_min_spacing option to try to make them adjacent. The syntax is:

--trampoline_min_spacing=_size_

A higher value minimizes fragmentation, but may result in more trampolines. A lower value may reduce trampolines, at the expense of fragmentation and linker running time. Specifying 0 for this option disables coalescing. The default is 16K.

Carrying Trampolines From Load Space to Run Space

It is sometimes useful to load code in one location in memory and run it in another. The linker provides the capability to specify separate load and run allocations for a section. The burden of actually copying the code from the load space to the run space is left to you.

A copy function must be executed before the real function can be executed in its run space. To facilitate this copy function, the assembler provides the .label directive, which allows you to define a load-time address. These load-time addresses can then be used to determine the start address and size of the code to be copied. However, this mechanism does _not_ work if the code contains a call that requires a trampoline to reach its called function. This is because the trampoline code is generated at link time, after the load-time addresses associated with the .label directive have been defined. If the linker detects the definition of a .label symbol in an input section that contains a trampoline call, then a warning is generated.

To solve this problem, you can use the START(), END(), and SIZE() operators (see _Address and Dimension Operators_. These operators allow you to define symbols to represent the load-time start address and size inside the linker command file. These symbols can be referenced by the

copy code, and their values are not resolved until link time, after the trampoline sections have been allocated.

Here is an example of how you could use the START() and SIZE() operators in association with an output section to copy the trampoline code section along with the code containing the calls that need trampolines:

```
SECTIONS
{    .foo : load = ROM, run = RAM, start(foo_start), size(foo_
 ↪size)
           { x.o(.text) }

    .text: {} > ROM

    .far : { --library=rts.lib(.text) } > FAR_MEM
}
```

A function in x.c.o contains an run-time-support call. The run-time-support library is placed in far memory and so the call is out-of-range. A trampoline section is added to the .foo output section by the linker. The copy code can refer to the symbols foo_start and foo_size as parameters for the load start address and size of the entire .foo output section. This allows the copy code to copy the trampoline section along with the original x.c.o code in .text from its load space to its run space.

See *Using Linker Symbols in C/C++ Applications* for information about referring to linker symbols in C/C++ code.

Link-Time Compression and Specialization Options

The options listed in the subsections below control how the linker handles optimization. On the **tiarmclang** command line they should be passed to the linker using the -Wl or -Xlinker option as described in *Passing Options to the Linker*.

- *Option Summary*
- *Compression (--cinit_compression and --copy_compression Option)*
- *Compress DWARF Information (--compress_dwarf Option)*
- *RTS Optimization (--use_memcpy and --use_memset Options)*
- *Printf Support Optimization (no option)*
- *Do Not Remove Unused Sections (--unused_section_elimination Option)*

Option Summary

`--cinit_compression` `[=compression_kind]`

Specifies the type of compression to apply to the C auto initialization data. The default if this option is used with no kind specified is lzss for Lempel-Ziv-Storer-Szymanski compression. Alternately, specify --cinit_compression=rle to use Run Length Encoded compression, which generally provides less efficient compression. See *Compression (--cinit_compression and --copy_compression Option)*.

`--compress_dwarf`

Aggressively reduces the size of DWARF information from input object files. See *Compress DWARF Information (--compress_dwarf Option)*.

`--copy_compression` `[=compression_kind]`

Compresses data copied by linker copy tables. See *Compression (--cinit_compression and --copy_compression Option)*.

`--use_memcpy` `[=small | fast]`

Select the optimization goal for the RTS memcpy() function. See *RTS Optimization (--use_memcpy and --use_memset Options)*.

`--use_memset` `[=small | fast]`

Select the optimization goal for the RTS memset() function. See *RTS Optimization (--use_memcpy and --use_memset Options)*.

`--unused_section_elimination`

Eliminates sections that are not needed in the executable module; on by default. See *Do Not Remove Unused Sections (--unused_section_elimination Option)*.

In addition, the version of the C RTS printf function used is optimized at link-time based on the format strings used in the application. See *Printf Support Optimization (no option)*.

Compression (--cinit_compression and --copy_compression Option)

By default, the linker does not compress copy table (*About Linker-Generated Copy Tables* and *Using Linker-Generated Copy Tables*) source data sections. The --cinit_compression and --copy_compression options specify compression through the linker.

The --cinit_compression option specifies the compression type the linker applies to the C autoinitialization copy table source data sections. The default is lzss.

Overlays can be managed by using linker-generated copy tables. To save ROM space the linker can compress the data copied by the copy tables. The compressed data is decompressed during copy. The --copy_compression option controls the compression of the copy data tables.

The syntax for the options are:

--cinit_compression[=*compression_kind*]

--copy_compression[=*compression_kind*]

The *compression_kind* can be one of the following types:

- **off**. Don't compress the data.

- **rle**. Compress data using Run Length Encoding.

- **lzss**. Compress data using Lempel-Ziv-Storer-Szymanski compression(the default if no *compression_kind* is specified).

Compressed sections within initialization tables are byte aligned in order to reduce the occurrence of holes in the .cinit table.

See *Compression* for more information about compression.

Compress DWARF Information (--compress_dwarf Option)

The --compress_dwarf option aggressively reduces the size of DWARF information by eliminating duplicate information from input object files.

For ELF object files, which are used with EABI, the --compress_dwarf option eliminates duplicate information that could not be removed through the use of ELF COMDAT groups. (See the ELF specification for information on COMDAT groups.)

RTS Optimization (--use_memcpy and --use_memset Options)

There are two versions of the memcpy and memset functions available in the C RTS library. One version is designed for efficient performance in terms of speed. The other version is much smaller than the first, but slower in comparison, especially if large blocks of data are to be handled by the memcpy or memset functions. The linker chooses one of these two versions of the C RTS memcpy and memset functions according to the optimization goals of the application.

The **tiarmclang** command line influences the selection of the memcpy and memset function implementations used. If the specified optimization option favors generating smaller code (as with the -Oz option), the linker chooses the smaller implementation of memcpy and memset. If the specified optimization option favors generating faster code (as with the -O3 option), the linker chooses the faster implementations.

The selection of the memcpy and memset function implementations can also be set explicitly using the following linker options:

- -–use_memcpy={small|fast}

- -–use_memset={small|fast}

These options override any influence that the optimization option has on link-time selection of the memcpy or memset implementation. If neither the -–use_memcpy/-–use_memset options nor an optimization option is specified, then the linker selects the smaller implementation of the memcpy and memset functions by default.

Printf Support Optimization (no option)

There are three different versions of the __TI_printfi function in the C RTS library. This function supports processing of format strings and format specifiers for the C RTS family of printf-like functions (printf, sprintf, fprintf, etc.).

Each version of __TI_printfi provides a different level of support for processing format strings. The linker chooses the smallest version of the underlying printf support function to that meets the needs of the application. This choice is based on what format specifiers are used in format strings in the application.

The three version of the function can then be characterized in terms of the format specifiers they support:

- **minimal**. The smallest version of __TI_printfi is chosen if there are no calls to any variation of printf with format strings that contain any of the following format specifiers: l, u, p, x, field width with precision, h, i, a, A, g, G, e, E, f, F, and L.

- **nofloat**. This version of __TI_printfi is larger than the minimal version, but still quite a bit smaller than the full version of __TI_printfi. It is chosen if there are no calls to any variation of printf with format strings that contain any of the floating-point related format specifiers: a, A, g, G, e, E, f, F, and L.

- **full**. This version of __TI_printfi is chosen if any calls are made to any variation of printf with format strings that contain any of the floating-point format specifier: a, A, g, G, e, E, f, F, or L. Additionally, if the linker is unable to determine whether a smaller version of __TI_printfi can be safely used, the full version of __TI_printfi is included in the link by default.

If your application uses printf-style functions from the C RTS library, but it does not use format specifiers that require more involved code to support, then you may realize a code size savings if the linker can determine it is safe to use a smaller version of __TI_printfi.

Do Not Remove Unused Sections (--unused_section_elimination Option)

To minimize the footprint, the ELF linker does not include sections that are not needed to resolve any references in the final executable. Use --unused_section_elimination=off to disable this optimization. The linker default behavior is equivalent to --unused_section_elimination=on.

Miscellaneous Options

The options listed in the subsections below control how the linker handles other behaviors. On the **tiarmclang** command line they should be passed to the linker using the -Wl or -Xlinker option as described in *Passing Options to the Linker*.

- *Option Summary*

- *Prioritizing Function Placement (--preferred_order Option)*

- *Zero Initialization (--zero_init Option)*

Option Summary

--linker_help (`-help`)

Displays information about syntax and available options.

--preferred_order

Prioritizes placement of functions. See *Prioritizing Function Placement (--preferred_order Option)*.

--zero_init

Controls preinitialization of uninitialized variables. Default is on. Always off if --ram_model is used. See *Zero Initialization (--zero_init Option)*.

Prioritizing Function Placement (--preferred_order Option)

The compiler prioritizes the placement of a function relative to others based on the order in which --preferred_order options are encountered during the linker invocation. The syntax is:

--preferred_order=*function specification*

Zero Initialization (--zero_init Option)

The C and C++ standards require that global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler supports preinitialization of uninitialized variables by default. To turn this off, specify the linker option --zero_init=off.

The syntax for the --zero_init option is:

--zero_init[={*on*|*off*}]

Zero initialization takes place only if the --rom_model linker option, which causes autoinitialization to occur, is used. If you use the --ram_model option for linking, the linker does not generate initialization records, and the loader must handle both data and zero initialization.

Note: Disabling Zero Initialization Not Recommended In general, disabling zero initialization is not recommended. If you turn off zero initialization, automatic initialization of uninitialized global and static objects to zero will not occur. You are then expected to initialize these variables to zero in some other manner.

3.10.5 Linker Command Files

Linker command files allow you to put linker options and directives in a file; this is useful when you invoke the linker often with the same options and directives. Linker command files are also useful because they allow you to use the MEMORY and SECTIONS directives to customize your application. You must use these directives in a command file; you cannot use them on the command line.

Linker command files are ASCII files that contain one or more of the following:

- Input filenames, which specify object files, archive libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from called command files.)

- Linker options, which can be used in the command file in the same manner that they are used on the command line.

- The MEMORY and SECTIONS linker directives. The MEMORY directive defines the target memory configuration (see *The MEMORY Directive*). The SECTIONS directive controls how sections are built and allocated (see *The SECTIONS Directive*).

- Assignment statements, which define and assign values to global symbols.

To invoke the linker with a command file, enter the **tiarmclang** command and follow it with the name of the command file:

```
tiarmclang command_filename
```

The linker processes input files in the order that it encounters them. If the linker recognizes a file as an object file, it links the file. Otherwise, it assumes that a file is a command file and begins reading and processing commands from it. Command filenames are case sensitive, regardless of the system used.

The following sample linker command file, link.cmd, specifies two object files to link and two command line options to use:

```
a.c.o                   /*  First input filename      */
b.c.o                   /*  Second input filename      */
--output_file=prog.out  /*  Option to specify output file */
--map_file=prog.map     /*  Option to specify map file    */
```

This sample linker command file contains only filenames and options. (You can place comments in a command file by delimiting them with /* and */.) To invoke the linker with this command file, enter:

```
tiarmclang link.cmd
```

You can place other parameters on the command line when you use a command file:

```
tiarmclang -Wl,--relocatable link.cmd x.c.o y.c.o
```

The linker processes the command file as soon as it encounters the filename, so a.c.o and b.c.o are linked into the output module before x.c.o and y.c.o.

You can specify multiple command files. If, for example, you have a file called names.lst that contains filenames and another file called dir.cmd that contains linker directives, you could enter:

```
tiarmclang names.lst dir.cmd
```

One command file can call another command file; this type of nesting is limited to 16 levels. If a command file calls another command file as input, this statement must be the *last* statement in the calling command file.

Blanks and blank lines are insignificant in a command file except as delimiters. This also applies to the format of linker directives in a command file. The following example linker command file that contains linker directives.

```
a.o b.o c.o                    /* Input filenames      */
--output_file=prog.out         /* Options              */
--map_file=prog.map

MEMORY                         /* MEMORY directive     */
```

```
{
    FAST_MEM:   origin = 0x0100     length = 0x0100
    SLOW_MEM:   origin = 0x7000     length = 0x1000
}


SECTIONS                             /* SECTIONS directive    */
{
    .text:   > SLOW_MEM
    .data:   > SLOW_MEM
    .bss:    > FAST_MEM
}
```

For more information, see *The MEMORY Directive* for the MEMORY directive, and *The SEC-TIONS Directive* for the SECTIONS directive.

Contents:

Reserved Names in Linker Command Files

The following names (in both uppercase and lowercase) are reserved as keywords for linker directives. Do not use them as symbol or section names in a command file.

- ADDRESS_MASK

- ALGORITHM

- ALIAS

- ALIGN

- ATTR

- BLOCK

- COMPRESSION

- COPY

- CRC_TABLE

- DSECT

- ECC

- END

- f

- FILL

- GROUP

- HAMMING_MASK

- HIGH

- INPUT_PAGE

- INPUT_RANGE

- l (lowercase L)

- LAST

- LEN

- LENGTH

- LOAD

- LOAD_END

- LOAD_SIZE

- LOAD_START

- MEMORY

- MIRRORING

- NOINIT

- NOLOAD

- o

- ORG

- ORIGIN

- PAGE

- PALIGN

- PARITY_MASK

- RUN

- RUN_END

- RUN_SIZE

- RUN_START

- SECTIONS

- SIZE

- START

- TABLE

- TYPE

- UNION

- UNORDERED

- VFILL

In addition, any section names used by the TI tools are reserved from being used as the prefix for other names, unless the section will be a subsection of the section name used by the TI tools. For example, section names may not begin with .debug.

Constants in Linker Command Files

You can specify constants with either of two syntax schemes: the scheme used for specifying decimal, octal, or hexadecimal constants (but not binary constants) used in the assembler (see *Accessing Assembly Language Constants* and *Character String Constants*) or the scheme used for integer constants in C syntax.

Examples:

Format	Decimal	Octal	Hexadecimal
Assembler format	32	40q	020h
C format	32	040	0x20

Accessing Files and Libraries from a Linker Command File

Many applications use custom linker command files (or LCFs) to control the placement of code and data in target memory. For example, you may want to place a specific data object from a specific file into a specific location in target memory. This is simple to do using the available LCF syntax to reference the desired object file or library. However, a problem that many developers run into when they try to do this is a linker generated "file not found" error when accessing an object file or library from inside the LCF that has been specified earlier in the command-line invocation of the linker. Most often, this error occurs because the syntax used to access the file on the linker command line does not match the syntax that is used to access the same file in the LCF.

Consider a simple example. Imagine that you have an application that requires a table of constants called "app_coeffs" to be defined in a memory area called "DDR". Assume also that the "app_coeffs" data object is defined in a .data section that resides in an object file, app_coeffs.c.o. The app_coeffs.c.o file is then included in the object file library app_data.lib. In your LCF, you can control the placement of the "app_coeffs" data object as follows:

```
SECTIONS
{
    ...
```

```
    .coeffs: { app_data.lib<app_coeffs.c.o>(.data) } > DDR
    ...
}
```

Now assume that the app_data.lib object library resides in a sub-directory called "lib" relative to where you are building the application. In order to gain access to app_data.lib from the build command line, you can use a combination of the –i and –l options to set up a directory search path which the linker can use to find the app_data.lib library:

```
%> tiarmclang <compile options/files> -Wl,-i=./lib,-lapp_data.
↪lib mylnk.cmd <link files>
```

The –i option adds the lib sub-directory to the directory search path and the –l option instructs the linker to look through the directories in the directory search path to find the app_data.lib library. However, if you do not update the reference to app_data.lib in mylnk.cmd, the linker fails to find the app_data.lib library and generate a "file not found" error. The reason is that when the linker encounters the reference to app_data.lib inside the SECTIONS directive, there is no –l option preceding the reference. Therefore, the linker tries to open app_data.lib in the current working directory.

In essence, the linker has a few different ways of opening files:

- If there is a path specified, the linker looks for the file in the specified location. For an absolute path, the linker tries to open the file in the specified directory. For a relative path, the linker follows the specified path starting from the current working directory and try to open the file at that location.

- If there is no path specified, the linker tries to open the file in the current working directory.

- If a –l option precedes the file reference, then the linker tries to find and open the referenced file in one of the directories in the directory search path. The directory search path is set up via –i options.

As long as a file is referenced in a consistent manner on the command line and throughout any applicable LCFs, the linker is able to find and open your object files and libraries.

Returning to the earlier example, you can insert a –l option in front of the reference to app_data.lib in mylnk.cmd to ensure that the linker can find and open the app_data.lib library when the application is built:

```
SECTIONS
{
    ...
    .coeffs: { -l app_data.lib<app_coeffs.c.o>(.data) } > DDR
    ...
}
```

Another benefit to using the –l option when referencing a file from within an LCF is that if the location of the referenced file changes, you can modify the directory search path to incorporate the new location of the file (using –i option on the command line, for example) without having to modify the LCF.

The MEMORY Directive

The linker determines where output sections are allocated into memory; it must have a model of target memory to accomplish this. The MEMORY directive allows you to specify a model of target memory so that you can define the types of memory your system contains and the address ranges they occupy. The linker maintains the model as it allocates output sections and uses it to determine which memory locations can be used for object code.

The memory configurations of Arm systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations. After you use MEMORY to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

For more information, see *How the Linker Handles Sections*.

- *Default Memory Model*
- *MEMORY Directive Syntax*
- *Expressions and Address Operators*

Default Memory Model

If you do not use the MEMORY directive, the linker uses a default memory model that is based on the Arm architecture. This model assumes that the full 32-bit address space (2^{32} locations) is present in the system and available for use. For more information about the default memory model, see *Default Placement Algorithm*.

MEMORY Directive Syntax

The MEMORY directive identifies ranges of memory that are physically present in the target system and can be used by a program. Each range has several characteristics:

- Name
- Starting address
- Length
- Optional set of attributes

- Optional fill specification

The MEMORY directive also allows you to use the GROUP keyword to create logical groups of memory ranges for use with Cyclic Redundancy Checks (CRC). See *Using the crc_table() Operator in the MEMORY Directive* for how to compute CRCs over memory ranges using the GROUP syntax.

When you use the MEMORY directive, be sure to identify all memory ranges that are available for the program to access at run time. Memory defined by the MEMORY directive is configured; any memory that you do not explicitly account for with MEMORY is unconfigured. The linker does not place any part of a program into unconfigured memory. You can represent nonexistent memory spaces by simply not including an address range in a MEMORY directive statement.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. The MEMORY directive in the following example defines a system that has 4K bytes of fast external memory at address 0x00000000, 2K bytes of slow external memory at address 0x00001000 and 4K bytes of slow external memory at address 0x10000000. It also demonstrates the use of memory range expressions as well as start/end/size address operators (see *Expressions and Address Operators*).

```
/********************************************************/
/* Sample command file with MEMORY directive            */
/********************************************************/
file1.c.o   file2.c.o              /* Input files */
--output_file=prog.out             /* Options     */

MEMORY
{
    FAST_MEM (RX): origin = 0x00000000 length = 0x00001000
    SLOW_MEM (RW): origin = 0x00001000 length = 0x00000800
    EXT_MEM (RX): origin = 0x10000000 length = 0x00001000
```

The general syntax for the MEMORY directive is:

```
MEMORY
{
    name_1 [( attr )] : origin = expr, length = expr [, fill =␣
↪constant] [LAST( sym )]
    ...
    ...
    name_n [( attr )] : origin = expr, length = expr [, fill =␣
↪constant] [LAST( sym )]
}
```

- *name* names a memory range. A memory name can be one to 64 characters; valid characters include A-Z, a-z, $, ., and _. The names have no special significance to the linker; they simply identify memory ranges. Memory range names are internal to the linker and are not

retained in the output file or in the symbol table. All memory ranges must have unique names and must not overlap.

- *attr* specifies one to four attributes associated with the named range. Attributes are optional; when used, they must be enclosed in parentheses. Attributes restrict the allocation of output sections into certain memory ranges. If you do not use any attributes, you can allocate any output section into any range with no restrictions. Any memory for which no attributes are specified (including all memory in the default model) has all four attributes. Valid attributes are:

  - **R** specifies that the memory can be read.

  - **W** specifies that the memory can be written to.

  - **X** specifies that the memory can contain executable code.

  - **I** specifies that the memory can be initialized.

- **origin** specifies the starting address of a memory range; enter as *origin*, *org*, or *o*. The value, specified in bytes, is a 32-bit integer constant expression, which can be decimal, octal, or hexadecimal.

- **length** specifies the length of a memory range; enter as *length*, *len*, or *l*. The value, specified in bytes, is a 32-bit integer constant expression, which can be decimal, octal, or hexadecimal.

- **fill** specifies a fill character for the memory range; enter as *fill* or *f*. Fills are optional. The value is an integer constant and can be decimal, octal, or hexadecimal. The fill value is used to fill areas of the memory range that are not allocated to a section. (See *Using the VFILL Specifier in the Memory Map* for virtual filling of memory ranges when using Error Correcting Code (ECC).)

- **LAST** optionally specifies a symbol that can be used at run-time to find the address of the last allocated byte in the memory range. See *LAST Operator*.

Note: Filling Memory Ranges

If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with 0s) causes raw data to be generated for all unallocated blocks of memory in the range.

The following example specifies a memory range with the R and W attributes and a fill constant of 0FFFFFFFFh:

```
MEMORY
{
    RFILE (RW) : o = 0x0020, l = 0x1000, f = 0xFFFF
}
```

You normally use the MEMORY directive in conjunction with the SECTIONS directive to control placement of output sections. For more information about the SECTIONS directive, see *The SECTIONS Directive*.

Expressions and Address Operators

Memory range origin and length can use expressions of integer constants with the following operators:

Type	Operators
Binary operators:	* / % + - << >> == = < <= > >= & | && ||
Unary operators:	- ~ !

Expressions are evaluated using standard C operator precedence rules.

No checking is done for overflow or underflow, however, expressions are evaluated using a larger integer type.

Preprocess directive #define constants can be used in place of integer constants. Global symbols cannot be used in Memory Directive expressions.

Three address operators reference memory range properties from prior memory range entries:

Operators	Description
START(MR)	Returns start address for previously defined memory range MR.
SIZE(MR)	Returns size of previously defined memory range MR.
END(MR)	Returns end address for previously defined memory range MR.

The following example uses an expression to specify an origin and a length:

```
/******************************************************/
/* Sample command file with MEMORY directive          */
/******************************************************/
file1.c.o file2.c.o                     /* Input files */
--output_file=prog.out                  /* Options */
#define ORIGIN 0x00000000
#define BUFFER 0x00000200
#define CACHE 0x0001000


MEMORY
{
    FAST_MEM (RX): origin = ORIGIN + CACHE length = 0x00001000 +
↪BUFFER
    SLOW_MEM (RW): origin = end(FAST_MEM)  length = 0x00001800 -
↪size(FAST_MEM)
```

```
    EXT_MEM (RX):   origin = 0x10000000    length = size(FAST_
↪MEM) - CACHE
}
```

The ALIAS Statement

Certain devices, such as the MSP432 Cortex M4, have a region of RAM that can be addressed by two different memory buses--a system bus and an instruction bus. This RAM region, which is located in the DATA region of the memory map (usually at 0x20000000), is internally aliased to the CODE region (usually at 0x01000000). This aliasing takes advantage of the instruction bus to fetch code from RAM while freeing the other system buses. On such devices, your linker command file should use the ALIAS statement so that placements to CODE and DATA are made with no collisions.

In order to use the above capability, the linker must be aware of the two addresses that point to the same memory. Use the following syntax within a MEMORY directive to create an ALIAS for a memory range. ALIAS regions must have the same length.

```
MEMORY
{
    ...
    ALIAS
    {
        SRAM_CODE (RWX) : origin = 0x01000000
        SRAM_DATA (RW)  : origin = 0x20000000
    } length = 0x0001000
    ...
}
```

The SECTIONS Directive

After you use MEMORY to specify the target system's memory model, you can use SECTIONS to allocate output sections into specific named memory ranges or into memory that has specific attributes. For example, you could allocate the .text and .data sections into the area named FAST_MEM and allocate the .bss section into the area named SLOW_MEM.

The SECTIONS directive controls your sections in the following ways:

- Describes how input sections are combined into output sections

- Defines output sections in the executable program

- Allows you to control where output sections are placed in memory in relation to each other and to the entire memory space (Note that the memory placement order is *not* simply the se-

quence in which sections occur in the SECTIONS directive unless the *–honor_cmdfile_order* option is used.)

- Permits renaming of output sections

For more information, see *How the Linker Handles Sections*, *Symbolic Relocations*, and *Subsections*. Subsections allow you to manipulate sections with greater precision.

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. *Default Placement Algorithm* describes this algorithm in detail.

- *SECTIONS Directive Syntax*
- *Section Allocation and Placement*
    - *Binding*
    - *Named Memory*
    - *Controlling Placement Using The HIGH Location Specifier*
    - *Alignment and Blocking*
    - *Alignment With Padding*
- *Specifying Input Sections*
- *Using Multi-Level Subsections*
- *Specifying Library or Archive Members as Input to Output Sections*
- *Allocation Using Multiple Memory Ranges*
- *Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges*

SECTIONS Directive Syntax

The SECTIONS directive is specified in a command file by the word SECTIONS (uppercase), followed by a list of output section specifications enclosed in braces.

The general syntax for the SECTIONS directive is:

```
SECTIONS
{
    name : [property [, property] [, property] ... ]
    name : [property [, property] [, property] ... ]
    name : [property [, property] [, property] ... ]
}
```

Each section specification, beginning with *name*, defines an output section. (An output section is

a section in the output file.) Section names can refer to sections, subsections, or archive library members. (See *Using Multi-Level Subsections* for information on multi-level subsections.) After the section name is a list of properties that define the section's contents and how the section is allocated. The properties can be separated by optional commas. Possible properties for a section are as follows:

Load allocation

>   Defines where in memory the section is to be loaded. See *Run-Time Relocation*, *Load and Run Addresses*, and *Placing a Section at Different Load and Run Addresses*.

>   Syntax:

```
load = allocation
    or
> allocation
```

Run allocation

>   Defines where in memory the section is to be run.

>   Syntax:

```
run = allocation
    or
run > allocation
```

Input sections

>   Defines the input sections (object files) that constitute the output section. See *Specifying Input Sections*.

>   Syntax:

```
{ input_sections }
```

Section type

>   Defines flags for special section types. See *Special Section Types (DSECT, COPY, NOLOAD, and NOINIT)*.

>   Syntax:

```
type = COPY
    or
type = DSECT
    or
type = NOLOAD
```

Fill value

>   Defines the value used to fill uninitialized holes. See *Creating and Filling Holes*.

Syntax:

```
fill = value
```

The following example shows a SECTIONS directive in a sample linker command file.

```
/************************************************/
/* Sample command file with SECTIONS directive  */
/************************************************/
file1.c.o file2.c.o          /* Input files */
--output_file=prog.out       /* Options */

SECTIONS
{
    .text: load = EXT_MEM, run = 0x00000800
    .const: load = FAST_MEM
    .rodata: load = FAST_MEM
    .bss: load = SLOW_MEM
    .vectors: load = 0x00000000
        {
            t1.c.o(.intvec1)
            t2.c.o(.intvec2)
            endvec = .;
        }
    .data:alpha: align = 16
    .data:beta: align = 16
}
```

The following figure shows the output sections defined by the SECTIONS directive in the previous example (.vectors, .text, .const, .rodata, .bss, .data:alpha, and .data:beta) and shows how these sections are allocated in memory using the MEMORY directive given in *MEMORY Directive Syntax*.
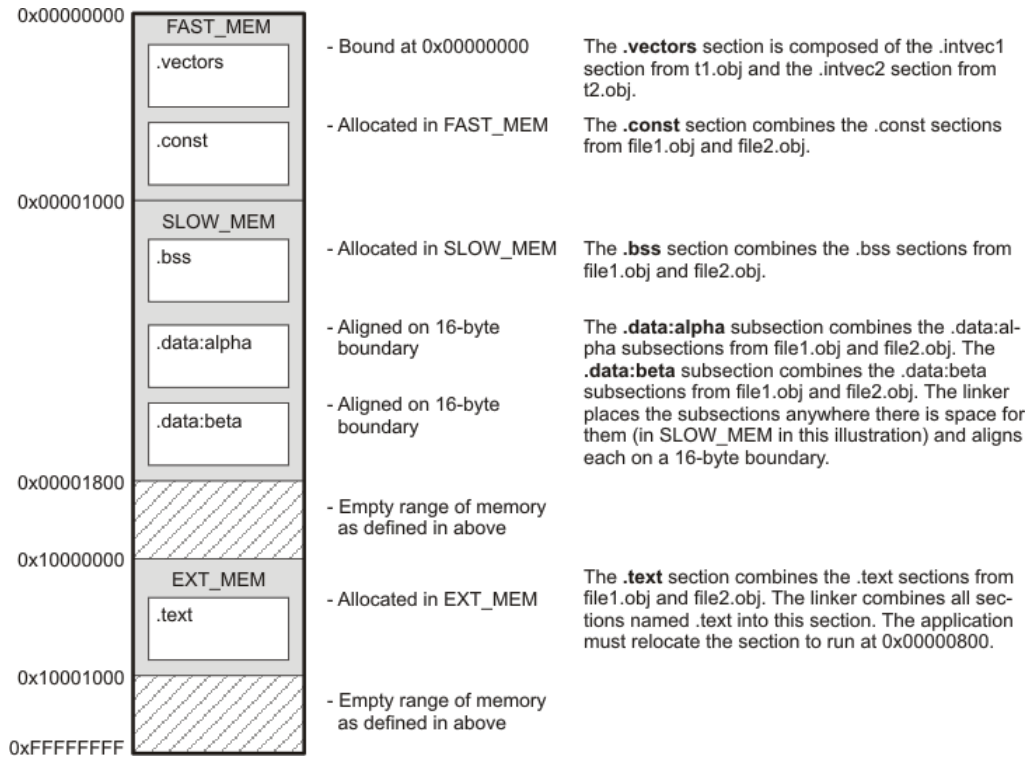
Figure 3.25: Output Sections Defined by the SECTIONS Directive

Section Allocation and Placement

The linker assigns each output section two locations in target memory: the location where the section will be loaded and the location where it will be run. Usually, these are the same, and you can think of each section as having only a single address. The process of locating the output section in the target's memory and assigning its address(es) is called placement. For more information about using separate load and run placement, see *Placing a Section at Different Load and Run Addresses*.

If you do not tell the linker how a section is to be allocated, it uses a default algorithm to place the section. Generally, the linker puts sections wherever they fit into configured memory. You can override this default placement for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control placement by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equal sign or greater-than sign, and a value optionally enclosed in parentheses. If load and run placement are separate, all parameters following the keyword LOAD apply to load placement, and those following the keyword RUN apply to run placement. The allocation parameters are:

Bind-ing	allocates a section at a specific address. `.text:  load = 0x1000`
Named mem-ory	allocates the section into a range defined in the MEMORY directive with the specified name (like SLOW_MEM) or attributes. `.text:  load > SLOW_MEM`
Align-ment	uses the align or palign keyword to specify the section must start on an address boundary. `.text:  align = 0x100`
Block-ing	uses the block keyword to specify the section must fit between two address aligned to the blocking factor. If a section is too large, it starts on an address boundary. `.text: block(0x100)`

For the load (usually the only) allocation, use a greater-than sign and omit the load keyword:

```
.text: > SLOW_MEM.text: {...} > SLOW_MEM   .text: > 0x4000
```

If more than one parameter is used, you can string them together as follows:

```
.text: > SLOW_MEM align 16
```

Or if you prefer, use parentheses for readability:

```
.text: load = (SLOW_MEM align(16))
```

You can also use an input section specification to identify the sections from input files that are combined to form an output section. See *Specifying Input Sections*.

Binding

You can set the starting address for an output section by following the section name with an address:

```
.text: 0x00001000
```

This example specifies that the .text section must begin at location 0x1000. The binding address must be a 32-bit constant.

Output sections can be bound anywhere in configured memory (assuming there is enough space), but they cannot overlap. If there is not enough space to bind a section to a specified address, the linker issues an error message.

Note: **Binding is Incompatible With Alignment and Named Memory** You cannot bind a section to an address if you use alignment or named memory. If you try to do this, the linker issues an error message.

Named Memory

You can allocate a section into a memory range that is defined by the MEMORY directive (see *The MEMORY Directive*). This example names ranges and links sections into them:

```
MEMORY
{
    SLOW_MEM (RIX)  : origin = 0x00000000,  length = 0x00001000
    FAST_MEM (RWIX) : origin = 0x03000000,  length = 0x00000300
}

SECTIONS
{
    .text  :> SLOW_MEM
    .data  :> FAST_MEM ALIGN(128)
    .bss:> FAST_MEM
}
```

In this example, the linker places .text into the area called SLOW_MEM. The .data and .bss output sections are allocated into FAST_MEM. You can align a section within a named memory range; the .data section is aligned on a 128-byte boundary within the FAST_MEM range.

Similarly, you can link a section into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parentheses) instead of a memory name. Using the same MEMORY directive declaration, you can specify:

```
SECTIONS
{
    .text: > (X)    /* .text --> executable memory*/
    .data: > (RI)   /* .data --> read or init memory */
    .bss : > (RW)   /* .bss --> read or write memory */
}
```

In this example, the .text output section can be linked into either the SLOW_MEM or FAST_MEM area because both areas have the X attribute. The .data section can also go into either SLOW_MEM or FAST_MEM because both areas have the R and I attributes. The .bss output section, however, must go into the FAST_MEM area because only FAST_MEM is declared with the W attribute.

You cannot control where in a named memory range a section is allocated, although the linker uses lower memory addresses first and avoids fragmentation when possible. In the preceding examples, assuming no conflicting assignments exist, the .text section starts at address 0. If a section must start on a specific address, use binding instead of named memory.

Controlling Placement Using The HIGH Location Specifier

The linker allocates output sections from low to high addresses within a designated memory range by default. Alternatively, you can cause the linker to allocate a section from high to low addresses within a memory range by using the HIGH location specifier in the SECTION directive declaration. You might use the HIGH location specifier in order to keep RTS code separate from application code, so that small changes in the application do not cause large changes to the memory map.

For example, given this MEMORY directive:

```
MEMORY
{
    RAM                 : origin = 0x0200, length = 0x0800
    FLASH               : origin = 0x1100, length = 0xEEE0
    VECTORS             : origin = 0xFFE0, length = 0x001E
    RESET               : origin = 0xFFFE, length = 0x000
}
```

and an accompanying SECTIONS directive:

```
SECTIONS
{
    .bss     : {} > RAM
    .sysmem  : {} > RAM
    .stack   : {} > RAM (HIGH)
}
```

The HIGH specifier used on the .stack section placement causes the linker to attempt to allocate .stack into the higher addresses within the RAM memory range. The .bss and .sysmem sections are allocated into the lower addresses within RAM. The following example shows a portion of a map file that shows where the given sections are allocated within RAM for a typical program.

```
.bss          0       00000200      00000270      UNINITIALIZED
                      00000200      0000011a      rtsxxx.lib  : defs.
↪c.o  (.bss)
                      0000031a      00000088                  :␣
↪trgdrv.c.o  (.bss)
                      000003a2      00000078                  :␣
↪lowlev.c.o  (.bss)
                      0000041a      00000046                  : exit.
↪c.o  (.bss)
                      00000460      00000008                  :␣
↪memory.c.o  (.bss)
                      00000468      00000004                  : _lock.
↪c.o  (.bss)
```

```
                        0000046c     00000002                        : fopen.
↪c.o (.bss)
                        0000046e     00000002     hello.c.o (.bss)

.sysmem     0     00000470     00000120     UNINITIALIZED
                  00000470     00000004     rtsxxx .lib      :␣
↪memory.c.o (.sysmem)

.stack      0     000008c0     00000140     UNINITIALIZED
                  000008c0     00000002     rtsxxx .lib      :␣
↪boot.c.o (.stack)
```

As shown in the previous example, the .bss and .sysmem sections are allocated at the lower ad-
dresses of RAM (0x0200 - 0x0590) and the .stack section is allocated at address 0x08c0, even
though lower addresses are available.

Without using the HIGH specifier, the linker allocation would result in the code shown in the
following map file contents. The HIGH specifier is ignored if it is used with specific address
binding or automatic section splitting (>> operator).

```
.bss        0     00000200     00000270     UNINITIALIZED
                  00000200     0000011a     rtsxxx.lib  : defs.
↪c.o (.bss)
                  0000031a     00000088                       :␣
↪trgdrv.c.o (.bss)
                  000003a2     00000078                       :␣
↪lowlev.c.o (.bss)
                  0000041a     00000046                       : exit.
↪c.o (.bss)
                  00000460     00000008                       :␣
↪memory.c.o (.bss)
                  00000468     00000004                       : _lock.
↪c.o (.bss)
                  0000046c     00000002                       : fopen.
↪c.o (.bss)
                  0000046e     00000002     hello.c.o (.bss)

.stack      0     00000470     00000140     UNINITIALIZED
                  00000470     00000002     rtsxxx .lib      :␣
↪boot.c.o (.stack)

.sysmem     0     000005b0     00000120     UNINITIALIZED
                  000005b0     00000004     rtsxxx .lib      :␣
↪memory.c.o (.sysmem)
```

3.10. Linker Description   **542**

Alignment and Blocking

align(n) operator

You can tell the linker to place an output section at an address that falls on an n-byte boundary, where n is a power of 2, by using the align keyword. For example, the following code allocates .text so that it falls on a 32-byte boundary:

```
.text: load = align(32)
```

align(power2) operator

The align operator can also take the *power2* keyword as a parameter. This parameter tells the linker to align a section to the next power of two boundary that is equal to or greater than the section's size. For example, consider the following section specification:

```
.mytext: align(power2) {} > PMEM
```

Assume that the size of the .mytext section is 120 bytes and PMEM starts at address 0x10020. After applying the align(power2) operator, the .mytext output section will have the following properties:

```
name        addr         size    align
-------     ----------   -----   -----
.mytext     0x00010080   0x78    128
```

block(n) operator

Blocking is a weaker form of alignment that allocates a section anywhere *within* a block of size n. The specified block size must be a power of 2. For example, the following code allocates .bss so that the entire section is contained in a single 128-byte block or begins on that boundary:

```
.bss: load = block(0x0080)
```

You can use alignment or blocking alone or in conjunction with a memory area, but alignment and blocking cannot be used together.

Alignment With Padding

palign(n) operator

As with align, you can tell the linker to place an output section at an address that falls on an n-byte boundary, where n is a power of 2, by using the palign keyword. In addition, palign ensures that the size of the section is a multiple of its placement alignment restrictions, padding the section size up to such a boundary, as needed.

For example, the following code lines allocate .text on a 2-byte boundary within the PMEM area. The .text section size is guaranteed to be a multiple of 2 bytes. Both statements are equivalent:

```
.text: palign(2) {} > PMEM

.text: palign = 2 {} > PMEM
```

If the linker adds padding to an initialized output section then the padding space is also initialized. By default, padding space is filled with a value of 0 (zero). However, if a fill value is specified for the output section then any padding for the section is also filled with that fill value. For example, consider the following section specification:

```
.mytext: palign(8), fill = 0xffffffff {} > PMEM
```

In this example, the length of the .mytext section is 6 bytes before the palign operator is applied. The contents of .mytext are as follows:

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
```

After the palign operator is applied, the length of .mytext is 8 bytes, and its contents are as follows:

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
0006 0xffff
```

The size of .mytext has been bumped to a multiple of 8 bytes and the padding created by the linker has been filled with 0xff.

The fill value specified in the linker command file is interpreted as a 16-bit constant. If you specify this code:

```
.mytext: palign(8), fill = 0xff {} > PMEM
```

The fill value assumed by the linker is 0x00ff, and .mytext will then have the following contents:

```
addr content
---- -------
0000 0x1234
0002 0x1234
```

3.10. Linker Description

```
0004 0x1234
0006 0x00ff
```

If the palign operator is applied to an uninitialized section, then the size of the section is bumped to the appropriate boundary, as needed, but any padding created is not initialized.

palign(power2) operator

The palign operator can also take the *power2* keyword as a parameter. This parameter tells the linker to add padding to increase the section's size to the next power of two boundary. In addition, the section is aligned on that power of 2 as well. For example, consider the following section specification:

```
.mytext: palign(power2) {} > PMEM
```

Assume that the size of the .mytext section is 120 bytes and PMEM starts at address 0x10020. After applying the palign(power2) operator, the .mytext output section will have the following properties:

```
name        addr          size      align
-------     ----------    -----     -----
.mytext     0x00010080    0x80      128
```

Specifying Input Sections

An input section specification identifies the sections from input files that are combined to form an output section. In general, the linker combines input sections by concatenating them in the order in which they are specified. However, if alignment or blocking is specified for an input section, all of the input sections within the output section are ordered as follows:

- All aligned sections, from largest to smallest
- All blocked sections, from largest to smallest
- All other sections, from largest to smallest

The size of an output section is the sum of the sizes of the input sections that it comprises.

The following example shows the most common type of section specification; note that no input sections are listed.

```
SECTIONS
{
    .text:
    .data:
```

```
    .bss:
}
```

In the example above, the linker takes all the .text sections from the input files and combines them into the .text output section. The linker concatenates the .text input sections in the order that it encounters them in the input files. The linker performs similar operations with the .data and .bss sections. You can use this type of specification for any output section.

You can explicitly specify the input sections that form an output section. Each input section is identified by its filename and section name. If the filename is hyphenated (or contains special characters), enclose it within quotes:

```
SECTIONS {
    .text :                     /* Build .text output section       ␣
↪                  */
    {
        f1.c.o(.text)        /* Link .text section from f1.c.o      ␣
↪              */     f2.c.o(sec1)          /* Link sec1 section␣
↪from f2.c.o                        */     "f3-new.c.o"          /*␣
↪Link ALL sections from f3-new.c.o                      */     f4.c.
↪o(.text,sec2) /* Link .text and sec2 from f4.c.o                  ␣
↪    */       f5.c.o(.task??)     /* Link .task00, .task01, .taskXX,
↪ etc. from f5.c.o */
        f6.c.o(*_ctable)    /* Link sections ending in "_ctable"␣
↪from f6.c.o     */
        X*.c.o(.text)        /* Link .text section for all files␣
↪starting with    */
                            /*                  "X" and ending in ".
↪c.o"            */
    }
}
```

It is not necessary for input sections to have the same name as each other or as the output section they become part of. If a file is listed with no sections, *all* of its sections are included in the output section. If any additional input sections have the same name as an output section but are not explicitly specified by the SECTIONS directive, they are automatically linked in at the end of the output section. For example, if the linker found more .text sections in the preceding example and these .text sections *were not* specified anywhere in the SECTIONS directive, the linker would concatenate these extra sections after f4.c.o(sec2).

The specifications in the first example above are actually a shorthand method for the following:

```
SECTIONS
{
```

```
    .text: { *(.text) }
    .data: { *(.data) }
    .bss:  { *(.bss)  }
}
```

The specification *(.text) means *the unallocated .text sections from all input files.* This format is useful if:

- You want the output section to contain all input sections that have a specified name, but the output section name is different from the input sections' name.

- You want the linker to allocate the input sections before it processes additional input sections or commands within the braces.

The following example illustrates the two purposes above:

```
SECTIONS
{
    .text  :  {
                abc.c.o(xqt)
                *(.text)
            }
    .data  :  {
                *(.data)
                fil.c.o(table)
            }
}
```

In this example, the .text output section contains a named section xqt from file abc.c.o, which is followed by all the .text input sections. The .data section contains all the .data input sections, followed by a named section table from the file fil.c.o. This method includes all the unallocated sections. For example, if one of the .text input sections was already included in another output section when the linker encountered *(.text), the linker could not include that first .text input section in the second output section.

Each input section acts as a prefix and gathers longer-named sections. For example, the pattern *(.data) matches .dataspecial. This mechanism enables the use of subsections, which are described in the following section.

Using Multi-Level Subsections

Subsections can be identified with the base section name and one or more subsection names separated by colons or periods. For example, A:B and A:B:C name subsections of the base section A. Likewise, A.B and A.B.C name the same subsections of the base section A. In certain places in a linker command file specifying a base name, such as A, selects the section A as well as any subsections of A, such as A:B or A:C:D.

A name such as A:B can specify a (sub)section of that name as well as any (multi-level) subsections beginning with that name, such as A:B:C, A:B:OTHER, etc. All subsections of A:B are also subsections of A. A and A:B are supersections of A:B:C. Among a group of supersections of a subsection, the nearest supersection is the supersection with the longest name. Thus, among {A, A:B} the nearest supersection of A:B:C:D is A:B. With multiple levels of subsections, the constraints are the following:

1. When specifying **input** sections within a file (or library unit) the section name selects an input section of the same name and any subsections of that name.

2. Input sections that are not explicitly allocated are allocated in an existing **output** section of the same name or in the nearest existing supersection of such an output section. An exception to this rule is that during a partial link (specified by the --relocatable linker option) a subsection is allocated only to an existing output section of the same name.

3. If no such output section described in 2) is defined, the input section is put in a **newly created output** section with the same name as the base name of the input section

Consider linking input sections with the following names:

- europe:north:norway
- europe:central:france
- europe:south:spain
- europe:north:sweden
- europe:central:germany
- europe:south:italy
- europe:north:finland
- europe:central:denmark
- europe:south:malta
- europe:north:iceland

This SECTIONS specification allocates the input sections as indicated in the comments:

```
SECTIONS
{
```

```
    nordic:  {*(europe:north)
                *(europe:central:denmark)} /* the nordic␣
↪countries */
    central: {*(europe:central)}                /* france, germany    ␣
↪*/
    therest: {*(europe)}                        /* spain, italy, malta ␣
↪*/
}
```

This SECTIONS specification allocates the input sections as indicated in the comments:

```
SECTIONS
{
    islands: {*(europe:south:malta)
                *(europe:north:iceland)}  /* malta, iceland   */
    europe:north:finland : {}             /* finland          */
    europe:north         : {}             /* norway, sweden   */
    europe:central       : {}             /* germany, denmark */
    europe:central:france: {}             /* france           */

    /* (italy, spain) go into a linker-generated output section
↪"europe" */
}
```

Note: Upward Compatibility of Multi-Level Subsections

Existing linker commands that use the existing single-level subsection features and which do not contain section names containing multiple colon characters continue to behave as before. However, if section names in a linker command file or in the input sections supplied to the linker contain multiple colon characters, some change in behavior could be possible. You should carefully consider the impact of the rules for multiple levels to see if it affects a particular system link.

Specifying Library or Archive Members as Input to Output Sections

You can specify one or more members of an object library or archive for input to an output section. Consider this SECTIONS directive:

```
SECTIONS
{
    boot    >       BOOT1
    {
```

```
        -l rtsXX.lib<boot.c.o> (.text)
        -l rtsXX.lib<exit.c.o strcpy.c.o> (.text)
    }

    .rts    >       BOOT2
    {
        -l rtsXX.lib (.text)
    }

    .text   >       RAM
    {
        * (.text)
    }
}
```

In *Example 9*, the .text sections of boot.c.o, exit.c.o, and strcpy.c.o are extracted from the run-time-support library and placed in the .boot output section. The remainder of the run-time-support library object that is referenced is allocated to the .rts output section. Finally, the remainder of all other .text sections are to be placed in section .text.

An archive member or a list of members is specified by surrounding the member name(s) with angle brackets < and > after the library name. Any object files separated by commas or spaces from the specified archive file are legal within the angle brackets.

The --library option (which normally implies a library path search be made for the named file following the option) listed before each library in *Example 9* is optional when listing specific archive members inside < >. Using < > implies that you are referring to a library.

To collect a set of the input sections from a library in one place, use the --library option within the SECTIONS directive. For example, the following collects all the .text sections from rtsv4_A_be_eabi.lib into the .rtstest section:

```
SECTIONS
{
    .rtstest { -l rtsv4_A_be_eabi.lib(.text) } > RAM
}
```

Note: SECTIONS Directive Effect on --priority

Specifying a library in a SECTIONS directive causes that library to be entered in the list of libraries that the linker searches to resolve references. If you use the --priority option, the first library specified in the command file will be searched first.

Allocation Using Multiple Memory Ranges

The linker allows you to specify an explicit list of memory ranges into which an output section can be allocated. Consider the following example:

```
MEMORY
{
    P_MEM1 :  origin = 0x02000,  length = 0x01000
    P_MEM2 :  origin = 0x04000,  length = 0x01000
    P_MEM3 :  origin = 0x06000,  length = 0x01000
    P_MEM4 :  origin = 0x08000,  length = 0x01000
}
SECTIONS
{
    .text  : { } > P_MEM1 | P_MEM2 | P_MEM4
}
```

The | operator is used to specify the multiple memory ranges. The .text output section is allocated as a whole into the first memory range in which it fits. The memory ranges are accessed in the order specified. In this example, the linker first tries to allocate the section in P_MEM1. If that attempt fails, the linker tries to place the section into P_MEM2, and so on. If the output section is not successfully allocated in any of the named memory ranges, the linker issues an error message.

With this type of SECTIONS directive specification, the linker can seamlessly handle an output section that grows beyond the available space of the memory range in which it is originally allocated. Instead of modifying the linker command file, you can let the linker move the section into one of the other areas.

Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges

The linker can split output sections among multiple memory ranges for efficient allocation. Use the >> operator to indicate that an output section can be split, if necessary, into the specified memory ranges:

```
MEMORY
{
    P_MEM1 :  origin = 0x2000,  length = 0x1000
    P_MEM2 :  origin = 0x4000,  length = 0x1000
    P_MEM3 :  origin = 0x6000,  length = 0x1000
    P_MEM4 :  origin = 0x8000,  length = 0x1000
}

SECTIONS
{
```

(continues on next page)

```
      .text: { *(.text) } >> P_MEM1 | P_MEM2 | P_MEM3 | P_MEM4
}
```

In this example, the >> operator indicates that the .text output section can be split among any of the listed memory areas. If the .text section grows beyond the available memory in P_MEM1, it is split on an input section boundary, and the remainder of the output section is allocated to P_MEM2 | P_MEM3 | P_MEM4.

The | operator is used to specify the list of multiple memory ranges.

You can also use the >> operator to indicate that an output section can be split within a single memory range. This functionality is useful when several output sections must be allocated into the same memory range, but the restrictions of one output section cause the memory range to be partitioned. Consider the following example:

```
MEMORY
{
    RAM :  origin = 0x1000,  length = 0x8000
}

SECTIONS
{
    .special: { f1.c.o(.text) } load = 0x4000
    .text: { *(.text) } >> RAM
}
```

The .special output section is allocated near the middle of the RAM memory range. This leaves two unused areas in RAM: from 0x1000 to 0x4000, and from the end of f1.c.o(.text) to 0x8000. The specification for the .text section allows the linker to split the .text section around the .special section and use the available space in RAM on either side of .special.

The >> operator can also be used to split an output section among all memory ranges that match a specified attribute combination. For example:

```
MEMORY
{
    P_MEM1 (RWX) : origin = 0x1000,  length = 0x2000
    P_MEM2 (RWI) : origin = 0x4000,  length = 0x1000
}

SECTIONS
{
    .text: { *(.text) } >> (RW)
}
```

The linker attempts to allocate all or part of the output section into any memory range whose attributes match the attributes specified in the SECTIONS directive.

This SECTIONS directive has the same effect as:

```
SECTIONS
{
    .text: { *(.text) } >> P_MEM1 | P_MEM2}
}
```

Certain sections should not be split:

- Certain sections created by the compiler, including:

    - The .cinit section, which contains the auto-initialization table for C/C++ programs

    - The .pinit section, which contains the list of global constructors for C++ programs

- An output section with an input section specification that includes an expression to be evaluated. The expression may define a symbol that is used in the program to manage the output section at run time.

- An output section that has a START(), END(), OR SIZE() operator applied to it. These operators provide information about a section's load or run address, and size. Splitting the section may compromise the integrity of the operation.

- The run allocation of a UNION. (Splitting the load allocation of a UNION is allowed.)

If you use the >> operator on any of these sections, the linker issues a warning and ignores the operator.

Placing a Section at Different Load and Run Addresses

At times, you may want to load code into one area of memory and run it in another. For example, you may have performance-critical code in slow external memory. The code must be loaded into slow external memory, but it would run faster in fast external memory.

The linker provides a simple way to accomplish this. You can use the SECTIONS directive to direct the linker to allocate a section twice: once to set its load address and again to set its run address. For example:

```
.fir: load = SLOW_MEM, run = FAST_MEM
```

Use the *load* keyword for the load address and the *run* keyword for the run address.

See *Run-Time Relocation* for an overview on run-time relocation.

The application must copy the section from its load address to its run address; this does *not* happen automatically when you specify a separate run address. (The TABLE operator instructs the linker to produce a copy table; see *The table() Operator*.)

> • *Specifying Load and Run Addresses*

Specifying Load and Run Addresses

The load address determines where a loader places the raw data for the section. Any references to the section (such as labels in it) refer to its run address. See *Load and Run Addresses* for an overview of load and run addresses.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and loads and runs at the same address. If you provide both allocations, the section is allocated as if it were two sections of the same size. This means that both allocations occupy space in the memory map and cannot overlay each other or other sections. (The UNION directive provides a way to overlay sections; see *Overlaying Sections With the UNION Statement*.)

If either the load or run address has additional parameters, such as alignment or blocking, list them after the appropriate keyword. Everything related to allocation after the keyword *load* affects the load address until the keyword *run* is seen, after which, everything affects the run address. The load and run allocations are completely independent, so any qualification of one (such as alignment) has no effect on the other. You can also specify run first, then load. Use parentheses to improve readability.

The examples that follow specify load and run addresses.

In this example, align applies only to load:

```
.data: load = SLOW_MEM, align = 32, run = FAST_MEM
```

The following example uses parentheses, but has effects that are identical to the previous example:

```
.data: load = (SLOW_MEM align 32), run = FAST_MEM
```

The following example aligns FAST_MEM to 32 bits for run allocations and aligns all load allocations to 16 bits:

```
.data: run  = FAST_MEM, align 32, load = align 16
```

For more information on run-time relocation see *Run-Time Relocation*.

Uninitialized sections (such as .bss) are not loaded, so their only significant address is the run address. The linker allocates uninitialized sections only once: if you specify both run and load addresses, the linker warns you and ignores the load address. Otherwise, if you specify only one address, the linker treats it as a run address, regardless of whether you call it load or run.

This example specifies load and run addresses for an uninitialized section:

```
.bss: load = 0x1000, run = FAST_MEM
```

A warning is issued, load is ignored, and space is allocated in FAST_MEM. All of the following examples have the same effect. The .bss section is allocated in FAST_MEM.

```
.dbss: load = FAST_MEM
.bss: run = FAST_MEM
.bss: > FAST_MEM
```

See *Using Linker Symbols in C/C++ Applications* for information about referring to linker symbols in C/C++ code.

Referring to the Load Address by Using the .label Directive

Normally, any reference to a symbol refers to its run-time address. However, it may be necessary at run time to refer to a load-time address. Specifically, the code that copies a section from its load address to its run address must have access to the load address. The .label directive defines a special symbol that refers to the section's load address. Thus, whereas normal symbols are relocated with respect to the run address, .label symbols are relocated with respect to the load address.

The following examples show assembly code that uses the .label directive to copy a section from its load address in SLOW_MEM to its run address in FAST_MEM. A linker command file to use with this assembly code is also provided. The figure that follows the examples illustrates the run-time execution of this code.

If you use the table operator, the .label directive is not needed. See *The table( ) Operator*.

```
;------------------------------------------------------------
;    define a section to be copied from SLOW_MEM to FAST_MEM
;------------------------------------------------------------
        .sect ".fir"
        .label fir_src     ; load address of section
fir:                       ; run address of section
        <code here>        ; code for section
        .label fir_end     ; load address of section end
;------------------------------------------------------------
; copy .fir section from SLOW_MEM to FAST_MEM
;------------------------------------------------------------
        .text

        LDR     r4, fir_s   ; get fir load address start
        LDR     r5, fir_e   ; get fir load address stop
        LDR     r3, fir_a   ; get fir run address
$1:     CMP     r4, r5
```

(continues on next page)

```
        LDRCC   r0, [r4], #4 ; copy fir routine to its
                             ; run address
        STRCC   r0, [r3], #4
        BCC     $1
;------------------------------------------------------------
; jump to fir routine, now in FAST_MEM
;------------------------------------------------------------
        B       fir
fir_a   .word   fir
fir_s   .word   fir_start
fir_e   .word   fir_end
```

The following linker command file is used with the code above:

```
/****************************************************/
/* PARTIAL LINKER COMMAND FILE FOR FIR EXAMPLE      */
/****************************************************/
MEMORY
{
    FAST_MEM : origin = 0x00001000, length = 0x00001000
    SLOW_MEM : origin = 0x10000000, length = 0x00001000
}

SECTIONS
{
    .text: load = FAST_MEM
    .fir: load = SLOW_MEM, run FAST_MEM
}
```

The following figure shows the run-time execution of the code in the assembly example above that moves a function from slow to fast memory.
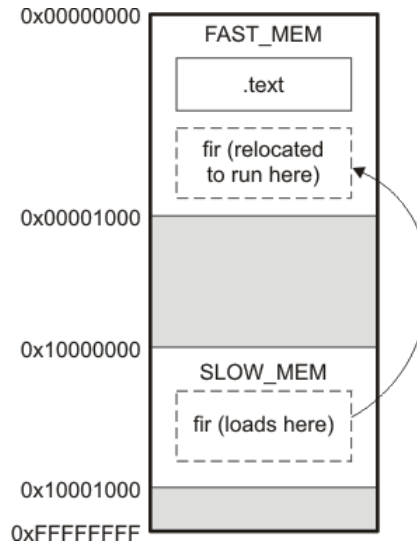
Figure 3.26: Run-Time Code Execution

Using GROUP and UNION Statements

Two SECTIONS statements allow you to organize or conserve memory: GROUP and UNION. Grouping sections causes the linker to allocate them contiguously in memory. Unioning sections causes the linker to allocate them to the same run address.

- *Grouping Output Sections Together*

- *Overlaying Sections With the UNION Statement*

- *Using Memory for Multiple Purposes*

- *Nesting UNIONs and GROUPs*

- *Checking the Consistency of Allocators*

- *Naming UNIONs and GROUPs*

Grouping Output Sections Together

The SECTIONS directive's GROUP option forces several output sections to be allocated contiguously and in the order listed, unless the UNORDERED operator is used. For example, assume that a section named term_rec contains a termination record for a table in the .data section. You can force the linker to allocate .data and term_rec together:

```
SECTIONS
{
```

```
    .text               /* Normal output section           */
    .bss                /* Normal output section           */
    GROUP 0x00001000 :  /* Specify a group of sections      */
    {
        .data           /* First section in the group       */
        term_rec        /* Allocated immediately after .data */
    }
}
```

You can use binding, alignment, or named memory to allocate a GROUP in the same manner as a single output section. In the preceding example, the GROUP is bound to address 0x1000. This means that .data is allocated at 0x1000, and term_rec follows it in memory.

Note: You Cannot Specify Addresses for Sections Within a GROUP

When you use the GROUP option, binding, alignment, or allocation into named memory can be specified for the group only. You cannot use binding, named memory, or alignment for sections within a group.

The MEMORY directive also allows you to use the GROUP keyword to create logical groups of memory ranges for use with Cyclic Redundancy Checks (CRC). See *Using the crc_table() Operator in the MEMORY Directive* for how to compute CRCs over memory ranges using the GROUP syntax.

Overlaying Sections With the UNION Statement

For some applications, you may want to allocate more than one section that occupies the same address during run time. For example, you may have several routines you want in fast external memory at different stages of execution. Or you may want several data objects that are not active at the same time to share a block of memory. The UNION statement within the SECTIONS directive provides a way to allocate several sections at the same run-time address.

In the following example, the .bss sections from file1.c.o and file2.c.o are allocated at the same address in FAST_MEM. In the memory map, the union occupies as much space as its largest component. The components of a union remain independent sections; they are simply allocated together as a unit.

```
SECTIONS
{
    .text: load = SLOW_MEM
    UNION: run  = FAST_MEM
    {
```

```
        .bss:part1: { file1.c.o(.bss) }
        .bss:part2: { file2.c.o(.bss) }
    }

        .bss:part3: run = FAST_MEM { globals.c.o(.bss) }
}
```

Allocation of a section as part of a union affects only its *run address.* Under no circumstances can sections be overlaid for loading. If an initialized section is a union member (an initialized section, such as .text, has raw data), its load allocation *must* be separately specified as shown in the following example. (There is an exception to this rule when combining an initialized section with uninitialized sections; see *Using Memory for Multiple Purposes*.)

```
UNION run = FAST_MEM
{
    .text:part1: load = SLOW_MEM, { file1.c.o(.text) }
    .text:part2: load = SLOW_MEM, { file2.c.o(.text) }
}
```

The following figure shows the memory allocation for the first example above (left) and the second example above (right)
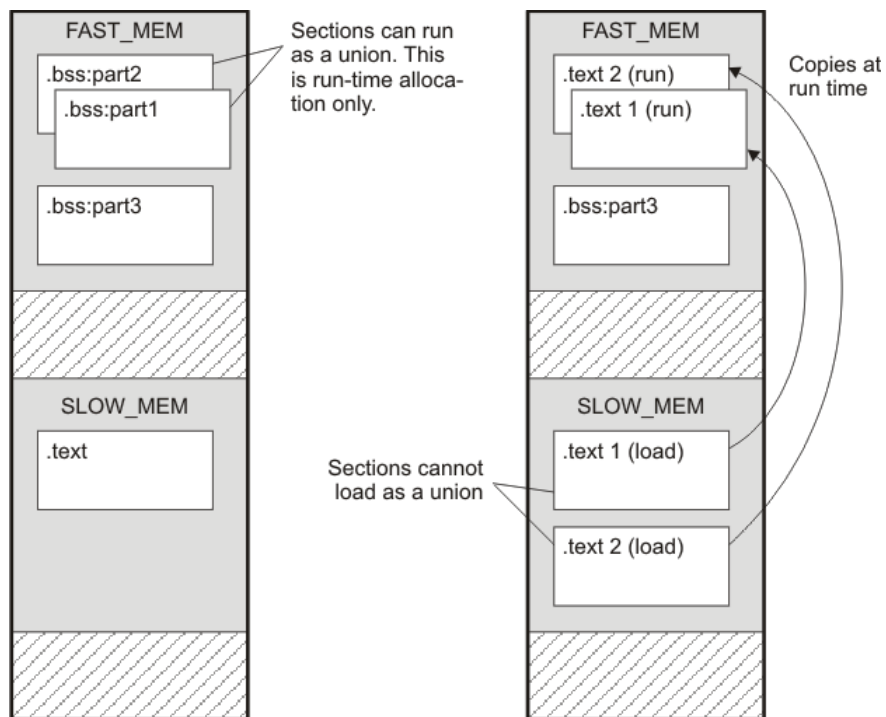
Figure 3.27: Memory Allocation for First (left) and Second (right) Examples

Since the .text sections contain raw data, they cannot *load* as a union, although they can be *run* as a

---

union. Therefore, each requires its own load address. If you fail to provide a load allocation for an initialized section within a UNION, the linker issues a warning and allocates load space anywhere it can in configured memory.

Uninitialized sections are not loaded and do not require load addresses.

The UNION statement applies only to allocation of run addresses, so it is meaningless to specify a load address for the union itself. For purposes of allocation, the union is treated as an uninitialized section: any one allocation specified is considered a run address, and if both run and load addresses are specified, the linker issues a warning and ignores the load address.

Using Memory for Multiple Purposes

One way to reduce an application's memory requirement is to use the same range of memory for multiple purposes. You can first use a range of memory for system initialization and startup. Once that phase is complete, the same memory can be repurposed as a collection of uninitialized data variables or a heap. To implement this scheme, use the following variation of the UNION statement to allow one section to be initialized and the remaining sections to be uninitialized.

Generally, an initialized section (one with raw data, such as .text) in a union must have its load allocation specified separately. However, one and only one initialized section in a union can be allocated at the union's run address. By listing it in the UNION statement with no load allocation at all, it will use the union's run address as its own load address.

For example:

```
UNION run = FAST_MEM { .cinit .bss }
```

In this example, the .cinit section is an initialized section. It will be loaded into FAST_MEM at the run address of the union. In contrast, .bss is an uninitialized section. Its run address will also be that of the union.

Nesting UNIONs and GROUPs

The linker allows arbitrary nesting of GROUP and UNION statements with the SECTIONS directive. By nesting GROUP and UNION statements, you can express hierarchical overlays and groupings of sections. The following example shows how two overlays can be grouped together.

```
SECTIONS
{
    GROUP 0x1000 : run = FAST_MEM
    {
        UNION:
        {
```

```
            mysect1: load = SLOW_MEM
            mysect2: load = SLOW_MEM
        }
        UNION:
        {
            mysect3: load = SLOW_MEM
            mysect4: load = SLOW_MEM
        }
    }
}
```

For this example, the linker performs the following allocations:

- The four sections (mysect1, mysect2, mysect3, mysect4) are assigned unique, non-overlapping load addresses. The name you defined with the .label directive is used in the SLOW_MEM memory region. This assignment is determined by the particular load allocations given for each section.

- Sections mysect1 and mysect2 are assigned the same run address in FAST_MEM.

- Sections mysect3 and mysect4 are assigned the same run address in FAST_MEM.

- The run addresses of mysect1/mysect2 and mysect3/mysect4 are allocated contiguously, as directed by the GROUP statement (subject to alignment and blocking restrictions).

To refer to groups and unions, linker diagnostic messages use the notation:

GROUP_n UNION_n

where n is a sequential number (beginning at 1) that represents the lexical ordering of the group or union in the linker control file without regard to nesting. Groups and unions each have their own counter.

Checking the Consistency of Allocators

The linker checks the consistency of load and run allocations specified for unions, groups, and sections. The following rules are used:

- Run allocations are only allowed for top-level sections, groups, or unions (sections, groups, or unions that are not nested under any other groups or unions). The linker uses the run address of the top-level structure to compute the run addresses of the components within groups and unions.

- The linker does not accept a load allocation for UNIONs.

- The linker does not accept a load allocation for uninitialized sections.

3.10. Linker Description   **561**

- In most cases, you must provide a load allocation for an initialized section. However, the linker does not accept a load allocation for an initialized section that is located within a group that already defines a load allocator.

- As a shortcut, you can specify a load allocation for an entire group, to determine the load allocations for every initialized section or subgroup nested within the group. However, a load allocation is accepted for an entire group only if all of the following conditions are true:

  – The group is initialized (that is, it has at least one initialized member).

  – The group is not nested inside another group that has a load allocator.

  – The group does not contain a union containing initialized sections.

- If the group contains a union with initialized sections, it is necessary to specify the load allocation for each initialized section nested within the group. Consider the following example:

```
SECTIONS
{
    GROUP: load = SLOW_MEM, run = SLOW_MEM
    {
        .text1:
        UNION:
        {
            .text2:
            .text3:
        }
    }
}
```

The load allocator given for the group does not uniquely specify the load allocation for the elements within the union: .text2 and .text3. In this case, the linker issues a diagnostic message to request that these load allocations be specified explicitly.

Naming UNIONs and GROUPs

You can give a name to a UNION or GROUP by entering the name in parentheses after the declaration. For example:

```
GROUP(BSS_SYSMEM_STACK_GROUP)
{
    .bss    :{}
    .sysmem :{}
    .stack  :{}
} load=D_MEM,  run=D_MEM
```

The name you defined is used in diagnostics for easy identification of the problem LCF area. For example:

```
warning: LOAD placement  ignored for "BSS_SYSMEM_STACK_GROUP":
 ↪object is uninitialized

UNION(TEXT_CINIT_UNION)
{
    .const  :{}load=D_MEM, table(table1)
    .rodata :{}load=D_MEM, table(table1)
    .pinit  :{}load=D_MEM, table(table1)
}run=P_MEM

warning:table(table1) operator ignored: table(table1) has
 ↪already been applied to a section
in the "UNION(TEXT_CINIT_UNION)" in which ".pinit" is a
 ↪descendant
```

Special Section Types (DSECT, COPY, NOLOAD, and NOINIT)

You can assign the following special types to output sections: DSECT, COPY, NOLOAD, and NOINIT. These types affect the way that the program is treated when it is linked and loaded. You can assign a type to a section by placing the type after the section definition. For example:

```
SECTIONS
{
    sec1: load = 0x00002000, type =  DSECT  {f1.c.o}
    sec2: load = 0x00004000, type =  COPY   {f2.c.o}
    sec3: load = 0x00006000, type =  NOLOAD {f3.c.o}
    sec4: load = 0x00008000, type =  NOINIT {f4.c.o}
}
```

- The DSECT type creates a dummy section with the following characteristics:

  - It is not included in the output section memory allocation. It takes up no memory and is not included in the memory map listing.

  - It can overlay other output sections, other DSECTs, and unconfigured memory.

  - Global symbols defined in a dummy section are relocated normally. They appear in the output module's symbol table with the same value they would have if the DSECT had actually been loaded. These symbols can be referenced by other input sections.

  - Undefined external symbols found in a DSECT cause specified archive libraries to be searched.

- The section's contents, relocation information, and line number information are not placed in the output module.

  In the preceding example, none of the sections from f1.c.obj are allocated, but all the symbols are relocated as though the sections were linked at address 0x2000. The other sections can refer to any of the global symbols in sec1.

- A COPY section is similar to a DSECT section, except that its contents and associated information are written to the output module. The .cinit section that contains initialization tables for the Arm C/C++ compiler has this attribute under the run-time initialization model.

- A NOLOAD section differs from a normal output section in one respect: the section's contents, relocation information, and line number information are not placed in the output module. The linker allocates space for the section, and it appears in the memory map listing.

- A NOINIT section is not C auto-initialized by the linker. It is your responsibility to initialize this section as needed.

Configuring Error Correcting Code (ECC) with the Linker

Error Correcting Codes (ECC) can be generated and placed in separate sections through the linker command file. ECC uses extra bits to allow errors to be detected and/or corrected by a device. To enable ECC generation, you must include **--ecc=on** as a linker option on the command line. By default ECC generation is off, even if the ECC directive and ECC specifiers are used in the linker command file. This allows you to fully configure ECC in the linker command file while still being able to quickly turn the code generation on and off via the command line.

The ECC support provided by the linker is compatible with the ECC support in TI Flash memory on various TI devices. TI Flash memory uses a modified Hamming(72,64) code, which uses 8 parity bits for every 64 bits. Check the documentation for your Flash memory to see if ECC is supported. (ECC for read-write memory is handled completely in hardware at run time.)

You can control the details of ECC generation using the ECC specifier in the memory map (*Using the ECC Specifier in the Memory Map*) and the ECC directive (*Using the ECC Directive*).

See *Error Correcting Code Testing (--ecc Options)* for command-line options that introduce bit errors into code that has a corresponding ECC section or into the ECC parity bits themselves. Use these options to test ECC error handling code.

ECC can be generated during linking. The ECC data is included in the resulting object file, alongside code and data, as a data section located at the appropriate address. No extra ECC generation step is required after compilation, and the ECC can be uploaded to the device along with everything else.

- *Using the ECC Specifier in the Memory Map*

- *Using the ECC Directive*

> • *Using the VFILL Specifier in the Memory Map*

Using the ECC Specifier in the Memory Map

To generate ECC, add a separate memory range to your memory map to hold ECC data and to indicate which memory range contains the Flash data that corresponds to this ECC data. If you have multiple memory ranges for Flash data, you should add a separate ECC memory range for each Flash data range.

The definition of an ECC memory range can also provide parameters for how to generate the ECC data.

The memory map for a device supporting Flash ECC may look something like this:

```
MEMORY {
    VECTORS  : origin=0x00000000 length=0x000020
    FLASH0   : origin=0x00000020 length=0x17FFE0
    FLASH1   : origin=0x00180000 length=0x180000
    STACKS   : origin=0x08000000 length=0x000500
    RAM      : origin=0x08000500 length=0x03FB00
    ECC_VEC  : origin=0xf0400000 length=0x000004 ECC={ input_
↪range=VECTORS }
    ECC_FLA0 : origin=0xf0400004 length=0x02FFFC ECC={ input_
↪range=FLASH0  }
    ECC_FLA1 : origin=0xf0430000 length=0x030000 ECC={ input_
↪range=FLASH1  }
}
```

The specification syntax for ECC memory ranges is as follows:

```
MEMORY {
    <memory specifier1> : <memory attributes> [ vfill=<fill␣
↪value> ]
    <memory specifier2> : <memory attributes> ECC = {
        input_range = <memory specifier1>
        [ algorithm   = <algorithm name> ]
        [ fill        = [ true, false ]  ]
    }
}
```

The "ECC" specifier attached to the ECC memory ranges indicates the data memory range that the ECC range covers. The ECC specifier supports the following parameters:

in-put_range = *<range>*	The data memory range covered by this ECC data range. Required.
algo-rithm = *<ECC alg name>*	The name of an ECC algorithm defined later in the command file using the ECC directive. Optional if only one algorithm is defined. (See *Using the ECC Directive*).
fill = true | false	Whether to generate ECC data for holes in the initialized data of the input range. The default is "true". Using fill=false produces behavior similar to the nowECC tool. The input range can be filled normally or using a virtual fill (see *Using the VFILL Specifier in the Memory Map*).

Using the ECC Directive

In addition to specifying ECC memory ranges in the memory map, the linker command file must specify parameters for the algorithm that generates ECC data. You might need multiple ECC algorithm specifications if you have multiple Flash devices.

Each TI device supporting Flash ECC has exactly one set of valid values for these parameters. The linker command files provided with Code Composer Studio include the ECC parameters necessary for ECC support on the Flash memory accessible by the device. Documentation is provided here for completeness.

You specify algorithm parameters with the top-level ECC directive in the linker command file. The specification syntax is as follows:

```
ECC {
   <algorithm name> : parity_mask  = <8-bit integer>
                      mirroring    = [ F021, F035 ]
                      address_mask = <32-bit mask>
}
```

For example:

```
MEMORY {
    FLASH0 : origin=0x00000020 length=0x17FFE0
    ECC_FLA0 : origin=0xf0400004 length=0x02FFFC ECC={ input_
→range=FLASH0 algorithm=F021 }
}


ECC { F021 : parity_mask = 0xfc
             mirroring = F021 }
```

This ECC directive accepts the following attributes:

algorithm_name	Specify the name you would like to use for referencing the algorithm.
address_mask = <32-bit mask>	This mask determines which bits of the address of each 64-bit piece of memory are used in the calculation of the ECC byte for that memory. Default is 0xffffffff, so that all bits of the address are used. (Note that the ECC algorithm itself ignores the lowest bits, which are always zero for a correctly-aligned input block.)
parity_mask = <8-bit mask>	This mask determines which ECC bits encode even parity and which bits encode odd parity. Default is 0, meaning that all bits encode even parity.
mirroring = F021 | F035	This setting determines the order of the ECC bytes and their duplication pattern for redundancy. Default is F021.

Using the VFILL Specifier in the Memory Map

Normally, specifying a fill value for a MEMORY range creates initialized data sections to cover any previously uninitialized areas of memory. To generate ECC data for an entire memory range, the linker either needs to have initialized data in the entire range, or needs to know what value uninitialized memory areas will have at run time.

In cases where you want to generate ECC for an entire memory range, but do not want to initialize the entire range by specifying a fill value, you can use the "vfill" specifier instead of a "fill" specifier to virtually fill the range:

```
MEMORY {
    FLASH : origin=0x0000  length=0x4000  vfill=0xffffffff
}
```

The vfill specifier is functionally equivalent to omitting a fill specifier, except that it allows ECC data to be generated for areas of the input memory range that remain uninitialized. This has the benefit of reducing the size of the resulting object file.

The vfill specifier has no effect other than in ECC data generation. It cannot be specified along with a fill specifier, since that would introduce ambiguity.

If fill is specified in the ECC specifier, but vfill is not specified, vfill defaults to 0xff.

Assigning Symbols at Link Time

Linker assignment statements allow you to define external (global) symbols and assign values to them at link time. You can use this feature to initialize a variable or pointer to an allocation-dependent value. See *Using Linker Symbols in C/C++ Applications* for information about referring to linker symbols in C/C++ code.

- *Syntax of Assignment Statements*
- *Assigning the SPC to a Symbol*
- *Assignment Expressions*
- *Symbols Automatically Defined by the Linker*
- *Assigning Exact Start, End, and Size Values of a Section to a Symbol*
- *Why the Dot Operator Does Not Always Work*
- *Address and Dimension Operators*
    - *Input Items*
    - *Output Section*
    - *GROUPs*
    - *UNIONs*
- *LAST Operator*

Syntax of Assignment Statements

The syntax of assignment statements in the linker is similar to that of assignment statements in the C language:

Assignment Statement Syntax in Linker Command Files

symbol	=	*expression;*	assigns the value of expression to symbol
symbol	+=	*expression;*	adds the value of expression to symbol
symbol	-=	*expression;*	subtracts the value of expression from symbol
symbol	*=	*expression;*	multiplies symbol by expression
symbol	/=	*expression;*	divides symbol by expression

The symbol should be defined externally. If it is not, the linker defines a new symbol and enters it into the symbol table. The expression must follow the rules defined in *Assignment Expressions*. Assignment statements *must* terminate with a semicolon.

The linker processes assignment statements *after* it allocates all the output sections. Therefore, if an expression contains a symbol, the address used for that symbol reflects the symbol's address in the executable output file.

For example, suppose a program reads data from one of two tables identified by two external symbols, Table1 and Table2. The program uses the symbol cur_tab as the address of the current table. The cur_tab symbol must point to either Table1 or Table2. You could accomplish this in the assembly code, but you would need to reassemble the program to change tables. Instead, you can use a linker assignment statement to assign cur_tab at link time:

```
prog.c.o          /* Input file                        */
cur_tab = Table1;  /* Assign cur_tab to one of the tables */
```

Assigning the SPC to a Symbol

A special symbol, denoted by a dot (.), represents the current value of the section program counter (SPC) during allocation. The SPC keeps track of the current location within a section. The linker's . symbol is analogous to the assembler's $ symbol. The . symbol can be used only in assignment statements within a SECTIONS directive because . is meaningful only during allocation and SECTIONS controls the allocation process. (See *The SECTIONS Directive*.)

The . symbol refers to the current run address, not the current load address, of the section.

For example, suppose a program needs to know the address of the beginning of the .data section. By using the .global directive (see *Global (External) Symbols*), you can create an external undefined variable called Dstart in the program. Then, assign the value of . to Dstart:

```
SECTIONS
{
    .text:     {}
    .data:     {Dstart = .;}
    .bss :     {}
}
```

This defines Dstart to be the first linked address of the .data section. (Dstart is assigned *before* .data is allocated.) The linker relocates all references to Dstart.

A special type of assignment assigns a value to the . symbol. This adjusts the SPC within an output section and creates a hole between two input sections. Any value assigned to . to create a hole is relative to the beginning of the section, not to the address actually represented by the . symbol. Holes and assignments to . are described in *Creating and Filling Holes*.

Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in the table below.

- All numbers are treated as long (32-bit) integers.

- Constants are identified by the linker in the same way as by the assembler. That is, numbers are recognized as decimal unless they have a suffix (H or h for hexadecimal and Q or q for octal). C language prefixes are also recognized (0 for octal and 0x for hex). Hexadecimal constants must begin with a digit. No binary constants are allowed.

- Symbols within an expression have only the value of the symbol's *address*. No type-checking is performed.

- Linker expressions can be absolute or relocatable. If an expression contains *any* relocatable symbols (and 0 or more constants or absolute symbols), it is relocatable. Otherwise, the expression is absolute. If a symbol is assigned the value of a relocatable expression, it is relocatable; if it is assigned the value of an absolute expression, it is absolute.

The linker supports the C language operators listed in the following table in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in this table, the linker also has an align operator that allows a symbol to be aligned on an n-byte boundary within an output section (n is a power of 2). For example, the following expression aligns the SPC within the current section on the next 16-byte boundary. Because the align operator is a function of the current SPC, it can be used only in the same context as . —that is, within a SECTIONS directive.

```
. = align(16);
```

Groups of Operators Used in Expressions for highest to lowest precedence:

Precedence Group	Operator	Description
Group 1	!	Logical NOT
	~	Bitwise NOT
	-	Negation
Group 2	*	Multiplication
	/	Division
	%	Modulus
Group 3	+	Addition
	-	Subtraction

continues on next page

Table 3.5 – continued from previous page

Precedence Group	Operator	Description
Group 4	>>	Arithmetic right shift
	<<	Arithmetic left shift
Group 5	==	Equal to
	!=	Not equal to
	>	Greater than
	<	Less than
	<=	Less than or equal to
	>=	Greater than or equal to
Group 6	&	Bitwise AND
Group 7	|	Bitwise OR
Group 8	&&	Logical AND
Group 9	||	Logical OR
Group 10	=	Assignment
	+=	A += B is equivalent to A = A + B
	-=	A -= B is equivalent to A = A - B
	*=	A *= B is equivalent to A = A * B
	/=	A /= B is equivalent to A = A / B

Symbols Automatically Defined by the Linker

The linker automatically defines the following symbols:

- .text is assigned the first address of the .text output section. (It marks the beginning of executable code.)

- etext is assigned the first address following the .text output section. (It marks the end of executable code.)

- .data is assigned the first address of the .data output section. (It marks the beginning of initialized data tables.)

- edata is assigned the first address following the .data output section. (It marks the end of initialized data tables.)

- .bss is assigned the first address of the .bss output section. (It marks the beginning of unini-
tialized data.)

- end is assigned the first address following the .bss output section. (It marks the end of
uninitialized data.)

The linker automatically defines the following symbols for C/C++ support when the --ram_model
or --rom_model option is used.

__TI_STACK_SIZE	is assigned the size of the .stack section.
__TI_STACK_END	is assigned the end of the .stack section.
__TI_SYSMEM_SIZE	is assigned the size of the .sysmem section.

These linker-defined symbols can be accessed in any assembly language module if they are de-
clared with a .global directive (see *Global (External) Symbols*).

See *Using Linker Symbols in C/C++ Applications* for information about referring to linker symbols
in C/C++ code.

Assigning Exact Start, End, and Size Values of a Section to a Symbol

The code generation tools currently support the ability to load program code in one area of (slow)
memory and run it in another (faster) area. This is done by specifying separate load and run
addresses for an output section or group in the linker command file. Then execute a sequence of
instructions (the copying code in *Referring to the Load Address by Using the .label Directive*) that
moves the program code from its load area to its run area before it is needed.

There are several responsibilities that a programmer must take on when setting up a system with
this feature. One of these responsibilities is to determine the size and run-time address of the
program code to be moved. The current mechanisms to do this involve use of the .label directives
in the copying code. A simple example is illustrated in *Referring to the Load Address by Using the
.label Directive*.

This method of specifying the size and load address of the program code has limitations. While
it works fine for an individual input section that is contained entirely within one source file, this
method becomes more complicated if the program code is spread over several source files or if the
programmer wants to copy an entire output section from load space to run space.

Another problem with this method is that it does not account for the possibility that the section
being moved may have an associated far call trampoline section that needs to be moved with it.

Why the Dot Operator Does Not Always Work

The dot operator (.) is used to define symbols at link-time with a particular address inside of an output section. It is interpreted like a PC. Whatever the current offset within the current section is, that is the value associated with the dot. Consider an output section specification within a SECTIONS directive:

```
outsect:
{
    s1.c.o(.text)
    end_of_s1   = .;
    start_of_s2 = .;
    s2.c.o(.text)
    end_of_s2 = .;
}
```

This statement creates three symbols:

- end_of_s1—the end address of .text in s1.c.o

- start_of_s2—the start address of .text in s2.c.o

- end_of_s2—the end address of .text in s2.c.o

Suppose there is padding between s1.c.o and s2.c.o created as a result of alignment. Then start_of_s2 is not really the start address of the .text section in s2.c.o, but it is the address before the padding needed to align the .text section in s2.c.o. This is due to the linker's interpretation of the dot operator as the current PC. It is also true because the dot operator is evaluated independently of the input sections around it.

Another potential problem in the above example is that end_of_s2 may not account for any padding that was required at the end of the output section. You cannot reliably use end_of_s2 as the end address of the output section. One way to get around this problem is to create a dummy section immediately after the output section in question. For example:

```
GROUP
{
    outsect:
    {
        start_of_outsect = .;
        ...
    }
        dummy: { size_of_outsect = . - start_of_outsect; }
    }
```

Address and Dimension Operators

Six operators allow you to define symbols for load-time and run-time addresses and sizes:

LOAD_START(*sym*) **START**(*sym*)	Defines *sym* with the load-time start address of related allocation unit
LOAD_END(*sym*) **END**(*sym*)	Defines *sym* with the load-time end address of related allocation unit
LOAD_SIZE(*sym*) **SIZE**(*sym*)	Defines *sym* with the load-time size of related allocation unit
RUN_START(*sym*)	Defines *sym* with the run-time start address of related allocation unit
RUN_END(*sym*)	Defines *sym* with the run-time end address of related allocation unit
RUN_SIZE(*sym*)	Defines *sym* with the run-time size of related allocation unit
LAST(*sym*)	Defines *sym* with the run-time address of the last allocated byte in the related memory range.

Note: **Linker Command File Operator Equivalencies:** LOAD_START() and START() are equivalent, as are LOAD_END()/END() and LOAD_SIZE()/SIZE(). The LOAD names are recommended for clarity.

These address and dimension operators can be associated with several different kinds of allocation units, including input items, output sections, GROUPs, and UNIONs. The following sections provide some examples of how the operators can be used in each case.

Symbols defined by the linker can be accessed in C/C++ code using various techniques. See *Using Linker Symbols in C/C++ Applications* for more information about referring to linker symbols in C/C++ code.

Input Items

Consider an output section specification within a SECTIONS directive:

```
outsect:
{
    s1.c.o(.text)
    end_of_s1   = .;
    start_of_s2 = .;
    s2.c.o(.text)
    end_of_s2 = .;
}
```

This can be rewritten using the START and END operators as follows:

```
outsect:
{
    s1.c.o(.text) { END(end_of_s1) }
    .c.o(.text)   { START(start_of_s2), END(end_of_s2) }
}
```

The values of end_of_s1 and end_of_s2 will be the same as if you had used the dot operator in the original example, but start_of_s2 would be defined after any necessary padding that needs to be added between the two .text sections. Remember that the dot operator would cause start_of_s2 to be defined before any necessary padding is inserted between the two input sections.

The syntax for using these operators in association with input sections calls for braces { } to enclose the operator list. The operators in the list are applied to the input item that occurs immediately before the list.

Output Section

The START, END, and SIZE operators can also be associated with an output section. Here is an example:

```
outsect: START(start_of_outsect), SIZE(size_of_outsect)
{
    <list of input items>
}
```

In this case, the SIZE operator defines size_of_outsect to incorporate any padding that is required in the output section to conform to any alignment requirements that are imposed.

The syntax for specifying the operators with an output section does not require braces to enclose the operator list. The operator list is simply included as part of the allocation specification for an output section.

GROUPs

Here is another use of the START and SIZE operators in the context of a GROUP specification:

```
GROUP
{
    outsect1: { ... }
    outsect2: { ... }
} load = ROM, run = RAM, START(group_start), SIZE(group_size);
```

This can be useful if the whole GROUP is to be loaded in one location and run in another. The copying code can use group_start and group_size as parameters for where to copy from and how much is to be copied. This makes the use of .label in the source code unnecessary.

UNIONs

The RUN_SIZE and LOAD_SIZE operators provide a mechanism to distinguish between the size of a UNION's load space and the size of the space where its constituents are going to be copied before they are run. Here is an example:

```
UNION: run = RAM, LOAD_START(union_load_addr),
       LOAD_SIZE(union_ld_sz), RUN_SIZE(union_run_sz)
{
    .text1: load = ROM, SIZE(text1_size) { f1.c.o(.text) }
    .text2: load = ROM, SIZE(text2_size) { f2.c.o(.text) }
}
```

Here union_ld_sz is going to be equal to the sum of the sizes of all output sections placed in the union. The union_run_sz value is equivalent to the largest output section in the union. Both of these symbols incorporate any padding due to blocking or alignment requirements.

LAST Operator

The LAST operator is similar to the START and END operators that were described previously. However, LAST applies to a memory range rather than to a section. You can use it in a MEMORY directive to define a symbol that can be used at run-time to learn how much memory was allocated when linking the program. See *MEMORY Directive Syntax* for syntax details.

For example, a memory range might be defined as follows:

```
D_MEM : org = 0x20000020 len = 0x20000000 LAST(dmem_end)
```

Your C/C++ code can then access this symbol at runtime as described in *Using Linker Symbols in C/C++ Applications*.

Creating and Filling Holes

The linker provides you with the ability to create areas *within output sections* that have nothing linked into them. These areas are called *holes*. In special cases, uninitialized sections can also be treated as holes. This section describes how the linker handles holes and how you can fill holes (and uninitialized sections) with values.

> • *Initialized and Uninitialized Sections*
>
> • *Creating Holes*
>
> • *Filling Holes*
>
> • *Explicit Initialization of Uninitialized Sections*

Initialized and Uninitialized Sections

There are two rules to remember about the contents of output sections. An output section contains either:

- Raw data for the *entire* section

- *No* raw data

A section that has raw data is referred to as *initialized*. This means that the object file contains the actual memory image contents of the section. When the section is loaded, this image is loaded into memory at the section's specified starting address. The .text and .data sections *always* have raw data if anything was assembled into them. Named sections defined with the .sect assembler directive also have raw data.

By default, the .bss section (see *Uninitialized Sections*) and sections defined with the .usect directive (see *User-Named Sections*) have no raw data (they are *uninitialized*). They occupy space in the memory map but have no actual contents. Uninitialized sections typically reserve space in fast external memory for variables. In the object file, an uninitialized section has a normal section header and can have symbols defined in it; no memory image, however, is stored in the section.

Creating Holes

You can create a hole in an initialized output section. A hole is created when you force the linker to leave extra space between input sections within an output section. When such a hole is created, *the linker must supply raw data for the hole.*

Holes can be created only *within* output sections. Space can exist *between* output sections, but such space is not a hole. To fill the space between output sections, see *MEMORY Directive Syntax*.

To create a hole in an output section, you must use a special type of linker assignment statement within an output section definition. The assignment statement modifies the SPC (denoted by .) by adding to it assigning a greater value to it, or aligning it on an address boundary. The operators, expressions, an syntaxes of assignment statements are described in *Assigning Symbols at Link Time*.

The following example uses assignment statements to create holes in output sections:

```
SECTIONS
{
    outsect:
    {
        file1.c.o(.text)
        . += 0x0100       /* Create a hole with size 0x0100 */
        file2.c.o(.text)
        . = align(16);     /* Create a hole to align the SPC */
        file3.c.o(.text)
    }
}
```

The output section outsect is built as follows:

1. The .text section from file1.c.o is linked in.

2. The linker creates a 256-byte hole.

3. The .text section from file2.c.o is linked in after the hole.

4. The linker creates another hole by aligning the SPC on a 16-byte boundary.

5. Finally, the .text section from file3.c.o is linked in.

All values assigned to the . symbol within a section refer to the *relative address within the section*. The linker handles assignments to the . symbol as if the section started at address 0 (even if you have specified a binding address). Consider the statement . = align(16) in the example. This statement effectively aligns the file3.c.o .text section to start on a 16-byte boundary within outsect. If outsect is ultimately allocated to start on an address that is not aligned, the file3.c.o .text section will not be aligned either.

The . symbol refers to the current run address, not the current load address, of the section.

Expressions that decrement the . symbol are illegal. For example, it is invalid to use the -= operator in an assignment to the . symbol. The most common operators used in assignments to the . symbol are += and align.

If an output section contains all input sections of a certain type (such as .text), you can use the following statements to create a hole at the beginning or end of the output section.

```
.text:    {   .+= 0x0100; }      /* Hole at the beginning */
.data:    {   *(.data)
              . += 0x0100; }     /* Hole at the end        */
```

Another way to create a hole in an output section is to combine an uninitialized section with an initialized section to form a single output section. *In this case, the linker treats the uninitialized section as a hole and supplies data for it.* The following example illustrates this method:

3.10. Linker Description 578

```
SECTIONS
{
    outsect:
    {
        file1.c.o(.text)
        file1.c.o(.bss)        /* This becomes a hole */
    }
}
```

Because the .text section has raw data, all of outsect must also contain raw data. Therefore, the uninitialized .bss section becomes a hole.

Uninitialized sections become holes only when they are combined with initialized sections. If several uninitialized sections are linked together, the resulting output section is also uninitialized.

Filling Holes

When a hole exists in an initialized output section, the linker must supply raw data to fill it. The linker fills holes with a 32-bit fill value that is replicated through memory until it fills the hole. The linker determines the fill value as follows:

1. If the hole is formed by combining an uninitialized section with an initialized section, you can specify a fill value for the uninitialized section. Follow the section name with an = sign and a 32-bit constant. For example:

```
SECTIONS
    { outsect:
        {
            file1.c.o(.text)
            file2.c.o(.bss)= 0xFF00FF00  /* Fill this hole with
 →0xFF00FF00 */
        }
    }
```

2. You can also specify a fill value for all the holes in an output section by supplying the fill value after the section definition:

```
SECTIONS
{ outsect:fill = 0xFF00FF00 /* Fills holes with 0xFF00FF00 */
    {
        . += 0x0010;          /* This creates a hole */
        file1.c.o(.text)
        file1.c.o(.bss)       /* This creates another hole */
```

```
        }
}
```

3. If you do not specify an initialization value for a hole, the linker fills the hole with the value specified with the --fill_value option (see *Set Default Fill Value (--fill_value Option)*). For example, suppose the command file link.cmd contains the following SECTIONS directive:

```
SECTIONS { .text: { .= 0x0100; } /* Create a 100 word hole */
 ↪ }
```

Now invoke the linker with the --fill_value option:

```
tiarmclang -Wl,--fill_value=0xFFFFFFFF link.cmd
```

This fills the hole with 0xFFFFFFFF.

4. If you do not invoke the linker with the --fill_value option or otherwise specify a fill value, the linker fills holes with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map along with the value the linker uses to fill it.

Explicit Initialization of Uninitialized Sections

You can force the linker to initialize an uninitialized section by specifying an explicit fill value for it in the SECTIONS directive. This causes the entire section to have raw data (the fill value). For example:

```
SECTIONS
{
    .bss: fill = 0x12341234 /* Fills .bss with 0x12341234 */
}
```

Note: Filling Sections

Because filling a section (even with 0s) causes raw data to be generated for the entire section in the output file, your output file will be very large if you specify fill values for large sections or holes.

3.10.6 Linker Symbols

This section provides information about using and resolving linker symbols.

Contents:

Using Linker Symbols in C/C++ Applications

Linker symbols have a name and a value. The value is a 32-bit unsigned integer, even if it represents a pointer value on a target that has pointers smaller than 32 bits.

The most common kind of symbol is generated by the compiler for each function and variable. The value represents the target address where that function or variable is located. When you refer to the symbol by name in the linker command file or in an assembly file, you get that 32-bit integer value.

However, in C and C++ names mean something different. If you have a variable named x that contains the value Y, and you use the name "x" in your C program, you are actually referring to the contents of variable x. If "x" is used on the right-hand side of an expression, the compiler fetches the value Y. To realize this variable, the compiler generates a linker symbol named x with the value &x. Even though the C/C++ variable and the linker symbol have the same name, they don't represent the same thing. In C, x is a variable name with the address &x and content Y. For linker symbols, x is an address, and that address contains the value Y.

Because of this difference, there are some tricks to referring to linker symbols in C code. The basic technique is to cause the compiler to create a "fake" C variable or function and take its address. The details differ depending on the type of linker symbol.

Linker symbols that represent a function address: In C code, declare the function as an extern function. Then, refer to the value of the linker symbol using the same name. This works because function pointers "decay" to their address value when used without adornment. For example:

```
extern void _c_int00(void);

printf("_c_int00 %lx\n", (unsigned long)&_c_int00);
```

Suppose your linker command file defines the following linker symbol:

```
func_sym=printf+100;
```

Your C application can refer to this symbol as follows:

```
extern void func_sym(void);

printf("func_sym %lx\n", (unsigned long)&func_sym);
```

Linker symbols that represent a data address: In C code, declare the variable as an extern variable. Then, refer to the value of the linker symbol using the & operator. Because the variable is at a valid data address, we know that a data pointer can represent the value.

Suppose your linker command file defines the following linker symbols:

```
data_sym=.data+100; xyz=12345
```

Your C application can refer to these symbols as follows:

```
extern char data_sym;
extern int xyz;

printf("data_sym %p\n", &data_sym);  myvar = &xyz;
```

Linker symbols for an arbitrary address: In C code, declare the linker symbol as an extern symbol. The type does not matter. If you are using GCC extensions, declare it as "extern void". If you are not using GCC extensions, declare it as "extern char". Then, refer to the value of the linker symbol mySymbol as &mySymbol.

Suppose your linker command file defines the following linker symbol:

```
abs_sym=0x12345678;
```

Your C application can refer to this symbol as follows:

```
extern char abs_sym;

printf("abs_sym %lx\n", &abs_sym);
```

Note: This technique assumes that the pointer to the symbol is 32 bits, which matches the 32-bit value of the linker symbol.

Declaring Weak Symbols

In a linker command file, an assignment expression outside a MEMORY or SECTIONS directive can be used to define a linker-defined symbol. To define a weak symbol in a linker command file, use the "weak" operator in an assignment expression to designate that the symbol as eligible for removal from the output file's symbol table if it is not referenced. For example, you can define "ext_addr_sym" as follows:

```
weak(ext_addr_sym) = 0x12345678;
```

When the linker command file is used to perform the final link, then "ext_addr_sym" is presented to the linker as a weak absolute symbol; it will not be included in the resulting output file if the symbol is not referenced.

See *Weak Symbols* for details about how weak symbols are handled by the linker.

Resolving Symbols with Object Libraries

An object library is a partitioned archive file that contains object files as members. Usually, a group of related modules are grouped together into a library. When you specify an object library as linker input, the linker includes any members of the library that define existing unresolved symbol references. You can use the archiver to build and maintain libraries. *Archiver Description* contains more information about the archiver.

Using object libraries can reduce link time and the size of the executable module. Normally, if an object file that contains a function is specified at link time, the file is linked whether the function is used or not; however, if that same function is placed in an archive library, the file is included only if the function is referenced.

The order in which libraries are specified is important, because the linker includes only those members that resolve symbols that are undefined at the time the library is searched. The same library can be specified as often as necessary; it is searched each time it is included. Alternatively, you can use the --reread_libs option to reread libraries until no more references can be resolved (see *Exhaustively Read and Search Libraries (--reread_libs and --priority Options)*. A library has a table that lists all external symbols defined in the library; the linker searches through the table until it determines that it cannot use the library to resolve any more references.

The following examples link several files and libraries, using these assumptions:

- Input files f1.c.o and f2.c.o both reference an external function named *clrscr*.
- Input file f1.c.o references the symbol *origin*.
- Input file f2.c.o references the symbol *fillclr*.
- Member 0 of library libc.lib contains a definition of *origin*.
- Member 3 of library liba.lib contains a definition of *fillclr*.
- Member 1 of both libraries defines *clrscr*.

If you enter:

```
tiarmclang f1.c.o f2.c.o liba.lib libc.lib
```

then:

- Member 1 of liba.lib satisfies the f1.c.o and f2.c.o references to *clrscr* because the library is searched and the definition of *clrscr* is found.
- Member 0 of libc.lib satisfies the reference to *origin*.

- Member 3 of liba.lib satisfies the reference to *fillclr*.

If, however, you enter:

```
tiarmclang f1.c.o f2.c.o libc.lib liba.lib
```

then the references to *clrscr* are satisfied by member 1 of libc.lib.

If none of the linked files reference symbols defined in a library, you can use the --undef_sym option to force the linker to include a library member. (See *Introduce an Unresolved Symbol (--undef_sym Option)*.) The next example creates an undefined symbol rout1 in the linker's global symbol table:

```
tiarmclang -Wl,--undef_sym=rout1 libc.lib
```

If any member of libc.lib defines rout1, the linker includes that member.

Library members are allocated according to the SECTIONS directive default allocation algorithm; see *The SECTIONS Directive*.

Alter the Library Search Algorithm (--library, --search_path) describes methods for specifying directories that contain object libraries.

3.10.7 Default Placement Algorithm

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. However, any memory locations or sections you choose *not* to specify must still be handled by the linker. The linker uses algorithms to build and allocate sections in coordination with any specifications you do supply.

If you do not use the MEMORY and SECTIONS directives, the linker allocates output sections as though the memory map and section definitions shown in the following example were specified.

```
{
    RAM       : origin = 0x00000000, length = 0xFFFFFFFF
}

SECTIONS
{
    .text :  ALIGN(4)    {} > RAM
    .const:  ALIGN(4)    {} > RAM
    .rodata: ALIGN(4)    {} > RAM
    .data :  ALIGN(4)    {} > RAM
    .bss  :  ALIGN(4)    {} > RAM
    .cinit:  ALIGN(4)    {} > RAM      /* -c option only */
    .pinit:  ALIGN(4)    {} > RAM      /* -c option only */
}
```

See *Combining Input Sections* for information about default memory allocation.

All .text input sections are concatenated to form a .text output section in the executable output file, and all .data input sections are combined to form a .data output section.

If you use a SECTIONS directive, the linker performs *no part* of this default allocation. Instead, allocation is performed according to the rules specified by the SECTIONS directive and the general algorithm described next in *How the Allocation Algorithm Creates Output Sections*.

Contents:

How the Allocation Algorithm Creates Output Sections

An output section can be formed in one of two ways:

- **Method 1:** As the result of a SECTIONS directive definition.
- **Method 2:** By combining input sections with the same name into an output section that is not defined in a SECTIONS directive.

If an output section is formed as a result of a SECTIONS directive, this definition completely determines the section's contents. (See *The SECTIONS Directive* for examples of how to define an output section's content.)

If an output section is formed by combining input sections not specified by a SECTIONS directive, the linker combines all such input sections that have the same name into an output section with that name. For example, suppose the files f1.c.o and f2.c.o both contain named sections called Vectors and that the SECTIONS directive does not define an output section for them. The linker combines the two Vectors sections from the input files into a single output section named Vectors, allocates it into memory, and includes it in the output file.

By default, the linker does not display a message when it creates an output section that is not defined in the SECTIONS directive. You can use the --warn_sections linker option (see *Display a Message When an Undefined Output Section Is Created (--warn_sections)*) to cause the linker to display a message when it creates a new output section.

After the linker determines the composition of all output sections, it must allocate them into configured memory. The MEMORY directive specifies which portions of memory are configured. If there is no MEMORY directive, the linker uses the default configuration as shown in *Default Placement Algorithm*. (See *The MEMORY Directive* for more information on configuring memory.)

Reducing Memory Fragmentation

The linker's allocation algorithm attempts to minimize memory fragmentation. This allows memory to be used more efficiently and increases the probability that your program will fit into memory. The algorithm comprises these steps:

1. Each output section for which you supply a specific binding address is placed in memory at that address.

2. Each output section that is included in a specific, named memory range or that has memory attribute restrictions is allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary, unless the *–honor_cmdfile_order* option is used, in which case the output section is placed with respect to its sequence order as defined by the linker command file.

3. Any remaining sections are allocated in the order in which they are defined. Sections not defined in a SECTIONS directive are allocated in the order in which they are encountered. Each output section is placed into the first available memory space, considering alignment where necessary.

3.10.8 Using Linker-Generated Copy Tables

The linker supports extensions to the linker command file syntax that enable the following:

- Make it easier for you to copy objects from load-space to run-space at boot time

- Make it easier for you to manage memory overlays at run time

- Allow you to split GROUPs and output sections that have separate load and run addresses

For an introduction to copy tables and their use, see *About Linker-Generated Copy Tables*.

Contents:

Using Copy Tables for Boot Loading

In some embedded applications, there is a need to copy or download code and/or data from one location to another at boot time before the application actually begins its main execution thread. For example, an application may have its code and/or data in FLASH memory and need to copy it into on-chip memory before the application begins execution.

One way to develop such an application is to create a copy table in assembly code that contains three elements for each block of code or data that needs to be moved from FLASH to on-chip memory at boot time:

- The load address

- The run address

- The size

The process you follow to develop such an application might look like this:

1. Build the application to produce a .map file that contains the load and run addresses of each section that has a separate load and run placement.

2. Edit the copy table (used by the boot loader) to correct the load and run addresses as well as the size of each block of code or data that needs to be moved at boot time.

3. Build the application again, incorporating the updated copy table.

4. Run the application.

This process puts a heavy burden on you to maintain the copy table (by hand, no less). Each time a piece of code or data is added or removed from the application, you must repeat the process in order to keep the contents of the copy table up to date.

Using Built-in Link Operators in Copy Tables

You can avoid some of this maintenance burden by using the LOAD_START(), RUN_START(), and SIZE() operators that are already part of the linker command file syntax . For example, instead of building the application to generate a .map file, the linker command file can be annotated:

```
SECTIONS
{
    .flashcode: { app_tasks.c.o(.text) }
        load = FLASH, run = PMEM,
        LOAD_START(_flash_code_ld_start),
        RUN_START(_flash_code_rn_start),
        SIZE(_flash_code_size)
    ...
}
```

In this example, the LOAD_START(), RUN_START(), and SIZE() operators instruct the linker to create three symbols:

Symbol	Description
_flash_code_ld_start	Load address of .flashcode section
_flash_code_rn_start	Run address of .flashcode section
_flash_code_size	Size of .flashcode section

These symbols can then be referenced from the copy table. The actual data in the copy table will be updated automatically each time the application is linked. This approach removes step 1 of the process described in *Using Copy Tables for Boot Loading*.

While maintenance of the copy table is reduced markedly, you must still carry the burden of keeping the copy table contents in sync with the symbols that are defined in the linker command file. Ideally, the linker would generate the boot copy table automatically. This would avoid having to build the application twice *and* free you from having to explicitly manage the contents of the boot copy table.

For more information on the LOAD_START(), RUN_START(), and SIZE() operators, see *Address and Dimension Operators*.

Overlay Management Example

Consider an application that contains a memory overlay that must be managed at run time. The memory overlay is defined using a UNION in the linker command file as illustrated in the following example:

```
SECTIONS
{
    ...
    UNION
    {
        GROUP
        {
            .task1: { task1.c.o(.text) }
            .task2: { task2.c.o(.text) }
        } load = ROM, LOAD_START(_task12_load_start), SIZE(_
↪task12_size)

        GROUP
        {
            .task3: { task3.c.o(.text) }
            .task4: { task4.c.o(.text) }
        } load = ROM, LOAD_START(_task34_load_start), SIZE(_task_
↪34_size)
    } run = RAM, RUN_START(_task_run_start)
    ...
}
```

The application must manage the contents of the memory overlay at run time. That is, whenever any services from .task1 or .task2 are needed, the application must first ensure that .task1 and .task2 are resident in the memory overlay. Similarly for .task3 and .task4.

To affect a copy of .task1 and .task2 from ROM to RAM at run time, the application must first gain access to the load address of the tasks (_task12_load_start), the run address (_task_run_start), and the size (_task12_size). Then this information is used to perform the actual code copy.

Generating Copy Tables With the table() Operator

The linker supports extensions to the linker command file syntax that enable you to do the following:

- Identify any object components that may need to be copied from load space to run space at some point during the run of an application

- Instruct the linker to automatically generate a copy table that contains (at least) the load address, run address, and size of the component that needs to be copied

- Instruct the linker to generate a symbol specified by you that provides the address of a linker-generated copy table.

For instance, *Overlay Management Example* can be written as shown in the following example:

```
SECTIONS
{
    ...
    UNION
    {
        GROUP
        {
            .task1: { task1.c.o(.text) }
            .task2: { task2.c.o(.text) }
        } load = ROM, table(_task12_copy_table)

        GROUP
        {
            .task3: { task3.c.o(.text) }
            .task4: { task4.c.o(.text) }
        } load = ROM, table(_task34_copy_table)
    } run = RAM
    ...
}
```

Using the SECTIONS directive from this example linker command file, the linker generates two copy tables named: _task12_copy_table and _task34_copy_table. Each copy table provides the load address, run address, and size of the GROUP that is associated with the copy table. This information is accessible from application source code using the linker-generated symbols, _task12_copy_table and _task34_copy_table, which provide the addresses of the two copy tables, respectively.

Using this method, you need not worry about the creation or maintenance of a copy table. You can reference the address of any copy table generated by the linker in C/C++ or assembly source code, passing that value to a general-purpose copy routine, which will process the copy table and affect the actual copy.

> - *The table( ) Operator*
>
> - *Boot-Time Copy Tables*
>
> - *Using the table( ) Operator to Manage Object Components*
>
> - *Linker-Generated Copy Table Sections and Symbols*
>
> - *Splitting Object Components and Overlay Management*

The table() Operator

You can use the table() operator to instruct the linker to produce a copy table. A table() operator can be applied to an output section, a GROUP, or a UNION member. The copy table generated for a particular table() specification can be accessed through a symbol specified by you that is provided as an argument to the table() operator. The linker creates a symbol with this name and assigns it the address of the copy table as the value of the symbol. The copy table can then be accessed from the application using the linker-generated symbol.

Each table() specification you apply to members of a given UNION must contain a unique name. If a table() operator is applied to a GROUP, then none of that GROUP's members may be marked with a table() specification. The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the table() specification. The linker does not generate a copy table for erroneous table() operator specifications.

Copy tables can be generated automatically; see *Generating Copy Tables With the table( ) Operator*. The table operator can be used with compression; see *Compression*.

Boot-Time Copy Tables

The linker supports a special copy table name, BINIT (or binit), that you can use to create a boot-time copy table. This table is handled before the .cinit section is used to initialize variables at startup. For example, the linker command file for the boot-loaded application described in *Using Built-in Link Operators in Copy Tables* can be rewritten as follows:

```
SECTIONS
{
    .flashcode: { app_tasks.c.o(.text) }
    load = FLASH, run = PMEM,
        table(BINIT)
    ...
}
```

For this example, the linker creates a copy table that can be accessed through a special linker-generated symbol, __binit__, which contains the list of all object components that need to be copied from their load location to their run location at boot-time. If a linker command file does not contain any uses of table(BINIT), then the __binit__ symbol is given a value of -1 to indicate that a boot-time copy table does not exist for a particular application.

You can apply the table(BINIT) specification to an output section, GROUP, or UNION member. If used in the context of a UNION, only one member of the UNION can be designated with table(BINIT). If applied to a GROUP, then none of that GROUP's members may be marked with table(BINIT). The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the table(BINIT) specification.

Using the table() Operator to Manage Object Components

If you have several pieces of code that need to be managed together, then you can apply the same table() operator to several different object components. In addition, if you want to manage a particular object component in multiple ways, you can apply more than one table() operator to it. Consider the linker command file excerpt in the following example:

```
SECTIONS
{
    UNION
    {
        .first: { a1.c.o(.text), b1.c.o(.text), c1.c.o(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_
↪ctbl)
        .second: { a2.c.o(.text), b2.c.o(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
    }
    .extra: load = EMEM, run = PMEM, table(BINIT)
    ...
}
```

In this example, the output sections .first and .extra are copied from external memory (EMEM) into program memory (PMEM) at boot time while processing the BINIT copy table. After the application has started executing its main thread, it can then manage the contents of the overlay using the two overlay copy tables named: _first_ctbl and _second_ctbl.

Linker-Generated Copy Table Sections and Symbols

The linker creates and allocates a separate input section for each copy table that it generates. Each copy table symbol is defined with the address value of the input section that contains the corresponding copy table.

The linker generates a unique name for each overlay copy table input section. For example, table(_first_ctbl) would place the copy table for the .first section into an input section called .ovly:_first_ctbl. The linker creates a single input section, .binit, to contain the entire boot-time copy table.

The following example shows how you can control the placement of the linker-generated copy table sections using the input section names in the linker command file.

```
SECTIONS
{
    UNION
    {
        .first: { a1.c.o(.text), b1.c.o(.text), c1.c.o(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_
↪ctbl)
        .second: { a2.c.o(.text), b2.c.o(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
    }
    .extra: load = EMEM, run = PMEM, table(BINIT)
    ...
    .ovly: { } > BMEM
    .binit: { } > BMEM
}
```

For the linker command file in this example, the boot-time copy table is generated into a .binit input section, which is collected into the .binit output section, which is mapped to an address in the BMEM memory area. The _first_ctbl is generated into the .ovly:_first_ctbl input section and the _second_ctbl is generated into the .ovly:_second_ctbl input section. Since the base names of these input sections match the name of the .ovly output section, the input sections are collected into the .ovly output section, which is then mapped to an address in the BMEM memory area.

If you do not provide explicit placement instructions for the linker-generated copy table sections, they are allocated according to the linker's default placement algorithm.

The linker does not allow other types of input sections to be combined with a copy table input section in the same output section. The linker does not allow a copy table section that was created from a partial link session to be used as input to a succeeding link session.

Splitting Object Components and Overlay Management

It is possible to split sections that have separate load and run placement instructions. The linker can access both the load address and run address of every piece of a split object component. Using the table() operator, you can tell the linker to generate this information into a copy table. The linker gives each piece of the split object component a COPY_RECORD entry in the copy table object.

For example, consider an application which has seven tasks. Tasks 1 through 3 are overlaid with tasks 4 through 7 (using a UNION directive). The load placement of all of the tasks is split among four different memory areas (LMEM1, LMEM2, LMEM3, and LMEM4). The overlay is defined as part of memory area PMEM. You must move each set of tasks into the overlay at run time before any services from the set are used.

You can use table() operators in combination with splitting operators, >>, to create copy tables that have all the information needed to move either group of tasks into the memory overlay as shown the following example:

```
SECTIONS
{
    UNION
    {
        .task1to3: { *(.task1), *(.task2), *(.task3) }
            load >> LMEM1 | LMEM2 | LMEM4, table(_task13_ctbl)

        GROUP
        {
            .task4: { *(.task4) }
            .task5: { *(.task5) }
            .task6: { *(.task6) }
            .task7: { *(.task7) }
        } load >> LMEM1 | LMEM3 | LMEM4, table(_task47_ctbl)
    } run = PMEM
    ...
    .ovly: > LMEM4
}
```

The following example illustrates a possible driver for such an application.

```
#include <cpy_tbl.h>


extern far COPY_TABLE task13_ctbl;
extern far COPY_TABLE task47_ctbl;


extern void task1(void);
...
```

```
extern void task7(void);

main() {
    ...
    copy_in(&task13_ctbl);
    task1();
    task2();
    task3();
    ...

    copy_in(&task47_ctbl);
    task4();
    task5();
    task6();
    task7();
    ...
}
```

You must declare a COPY_TABLE object as *far* to allow the overlay copy table section placement to be independent from the other sections containing data objects (such as .bss).

The contents of the .task1to3 section are split in the section's load space and contiguous in its run space. The linker-generated copy table, _task13_ctbl, contains a separate COPY_RECORD for each piece of the split section .task1to3. When the address of _task13_ctbl is passed to copy_in(), each piece of .task1to3 is copied from its load location into the run location.

The contents of the GROUP containing tasks 4 through 7 are also split in load space. The linker performs the GROUP split by applying the split operator to each member of the GROUP in order. The copy table for the GROUP then contains a COPY_RECORD entry for every piece of every member of the GROUP. These pieces are copied into the memory overlay when the _task47_ctbl is processed by copy_in().

The split operator can be applied to an output section, GROUP, or the load placement of a UNION or UNION member. The linker does not permit a split operator to be applied to the run placement of either a UNION or of a UNION member. The linker detects such violations, emits a warning, and ignores the offending split operator usage.

Compression

When automatically generating copy tables, the linker provides a way to compress the load-space data. This can reduce the read-only memory foot print. This compressed data can be decompressed while copying the data from load space to run space.

You can specify compression in two ways:

- The linker command line option --copy_compression= *compression_kind* can be used to apply the specified compression to any output section that has a table() operator applied to it.

- The table() operator accepts an optional compression parameter. The syntax is:

table(*name***, compression=***compression_kind***)**

The *compression_kind* can be one of the following types:

- **off**. Don't compress the data.

- **rle**. Compress data using Run Length Encoding.

- **lzss**. Compress data using Lempel-Ziv-Storer-Szymanski compression.

A table() operator without the compression keyword uses the compression kind specified using the command line option --copy_compression.

When you choose compression, it is not guaranteed that the linker will compress the load data. The linker compresses load data only when such compression reduces the overall size of the load space. In some cases even if the compression results in smaller load section size the linker does not compress the data if the decompression routine offsets for the savings.

For example, assume RLE compression reduces the size of section1 by 30 bytes. Also assume the RLE decompression routine takes up 40 bytes in load space. By choosing to compress section1 the load space is increased by 10 bytes. Therefore, the linker will not compress section1. On the other hand, if there is another section (say section2) that can benefit by more than 10 bytes from applying the same compression then both sections can be compressed and the overall load space is reduced. In such cases the linker compresses both the sections.

You cannot force the linker to compress the data when doing so does not result in savings.

You cannot compress the decompression routines or any member of a GROUP containing .cinit.

- *Compressed Copy Table Format*

- *Compressed Section Representation in the Object File*

- *Compressed Data Layout*

- *Run-Time Decompression*

- *Compression Algorithms*

Compressed Copy Table Format

The copy table format is the same irrespective of the *compression_kind*. The size field of the copy record is overloaded to support compression. The following figure shows the compressed copy table layout.

Rec size	Rec cnt			
Load address			Run address	Size (0 if load data is compressed)

Figure 3.28: Compressed Copy Table Layout

If the rec_size in the copy record is non-zero it represents the size of the data to be copied, and also means that the size of the load data is the same as the run data. When the size is 0, it means that the load data is compressed.

Compressed Section Representation in the Object File

The linker creates a separate input section to hold the compressed data. Consider the following table() operation in the linker command file.

```
SECTIONS
{
    .task1: load = ROM, run = RAM, table(_task1_table)
}
```

The output object file has one output section named .task1 which has different load and run addresses. This is possible because the load space and run space have identical data when the section is not compressed.

Alternatively, consider the following:

```
SECTIONS
{
    .task1: load = ROM, run = RAM, table(_task1_table,
↪compression=rle)
}
```

If the linker compresses the .task1 section then the load space data and the run space data are different. The linker creates the following two sections:

- **.task1**: This section is uninitialized. This output section represents the run space image of section task1.

- **.task1.load**: This section is initialized. This output section represents the load space image of the section task1. This section usually is considerably smaller in size than .task1 output section.

The linker allocates load space for the .task1.load input section in the memory area that was specified for load placement for the .task1 section. There is only a single load section to represent the load placement of .task1 - .task1.load. If the .task1 data had not been compressed, there would be two allocations for the .task1 input section: one for its load placement and another for its run placement.

Compressed Data Layout

The compressed load data has the following layout:

8-bit index : **compressed data**

The first 8 bits of the load data are the handler index. This handler index is used to index into a handler table to get the address of a handler function that knows how to decode the data that follows. The handler table is a list of 32-bit function pointers as shown in the following figure:
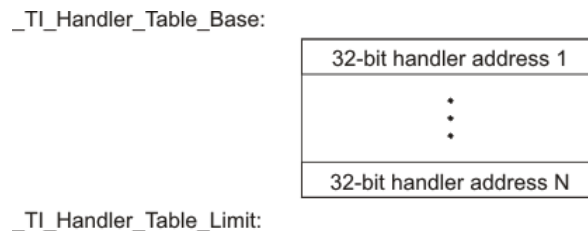
Figure 3.29: Handler Table

The linker creates a separate output section for the load and run space. For example, if .task1.load is compressed using RLE, the handler index points to an entry in the handler table that has the address of the run-time-support routine __TI_decompress_rle().

Run-Time Decompression

During run time you call the run-time-support routine copy_in() to copy the data from load space to run space. The address of the copy table is passed to this routine. First the routine reads the record count. Then it repeats the following steps for each record:

1. Read load address, run address and size from record.

2. If size is zero go to step 5.

3. Call memcpy passing the run address, load address and size.

4. Go to step 1 if there are more records to read.

5. Read the first byte from the load address. Call this index.

6. Read the handler address from (&__TI_Handler_Base)[index].

7. Call the handler and pass load address + 1 and run address.

8. Go to step 1 if there are more records to read.

The routines to handle the decompression of load data are provided in the run-time-support library.

Compression Algorithms

The following subsections provide information about decompression algorithms for the RLE and LZSS formats. To see example decompression algorithms, refer to the following functions in the Run-Time Support library:

- RLE: The __TI_decompress_rle() function in the copy_decompress_rle.c file.

- LZSS: The __TI_decompress_lzss() function in the copy_decompress_lzss.c file.

Run Length Encoding (RLE):

8-bit index : Initialization data compressed using RLE

The data following the 8-bit index is compressed using run length encoded (RLE) format. Arm uses a simple run length encoding that can be decompressed using the following algorithm. See copy_decompress_rle.c for details.

1. Read the first byte, Delimiter (D).

2. Read the next byte (B).

3. If B != D, copy B to the output buffer and go to step 2.

4. Read the next byte (L).

   a. If L == 0, then length is either a 16-bit or 24-bit value or we've reached the end of the data, read the next byte (L).

      1. If L == 0, length is a 24-bit value or the end of the data is reached, read next byte (L).

         a. If L == 0, the end of the data is reached, go to step 7.

         b. Else L <<= 16, read next two bytes into lower 16 bits of L to complete 24-bit value for L.

      2. Else L <<= 8, read next byte into lower 8 bits of L to complete 16-bit value for L.

   b. Else if L > 0 and L < 4, copy D to the output buffer L times. Go to step 2.

   c. Else, length is 8-bit value (L).

5. Read the next byte (C); C is the repeat character.

6. Write C to the output buffer L times; go to step 2.

7. End of processing.

The Arm run-time support library has a routine __TI_decompress_rle24() to decompress data compressed using RLE. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

Note: **RLE Decompression Routine** The previous decompression routine, __TI_decompress_rle(), is included in the run-time-support library for decompressing RLE encodings that are generated by older versions of the linker.

Lempel-Ziv-Storer-Szymanski Compression (LZSS):

8-bit index	Data compressed using LZSS
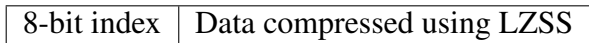
The data following the 8-bit index is compressed using LZSS compression. The Arm run-time-support library has the routine __TI_decompress_lzss() to decompress the data compressed using LZSS. The first argument to this function is the address pointing to the byte after the 8-bit Index, and the second argument is the run address from the C auto initialization record.

See copy_decompress_lzss.c for details on the LZSS algorithm.

Copy Table Contents

To use a copy table generated by the linker, you must know the contents of the copy table. This information is included in a run-time-support library header file, cpy_tbl.h, which contains a C source representation of the copy table data structure that is generated by the linker. The following example shows the copy table header file.

```
/
↪*****************************************************************************************
↪
/* cpy_tbl.h v#####                                                    ␣
↪                */
/* Copyright (c) 2003 Texas Instruments Incorporated                   ␣
↪             */
/*                                                                     ␣
↪                */
/* Specification of copy table data structures which can be␣
↪automatically    */
/* generated by the linker (using the table() operator in the␣
↪LCF).         */
/
↪*****************************************************************************************
↪
#ifndef _CPY_TBL
```

(continues on next page)

```
#define _CPY_TBL

#ifdef __cplusplus
extern "C" namespace std {
#endif /* __cplusplus */

/
 ↪********************************************************************
 ↪
/* Copy Record Data Structure                                      ␣
 ↪            */
/
 ↪********************************************************************
 ↪
typedef struct copy_record
{
    unsigned int load_addr;
    unsigned int run_addr;
    unsigned int size;
} COPY_RECORD;

/
 ↪********************************************************************
 ↪
/* Copy Table Data Structure                                       ␣
 ↪           */
/
 ↪********************************************************************
 ↪
typedef struct copy_table
{
    unsigned short rec_size;
    unsigned short num_recs;
    COPY_RECORD recs[1];
} COPY_TABLE;

/
 ↪********************************************************************
 ↪
/* Prototype for general-purpose copy routine.                     ␣
 ↪           */
/
 ↪********************************************************************
 ↪
```

```
extern void copy_in(COPY_TABLE *tp);

#ifdef __cplusplus
} /* extern "C" namespace std */

#ifndef _CPP_STYLE_HEADER
using std::COPY_RECORD;
using std::COPY_TABLE;
using std::copy_in;
#endif /* _CPP_STYLE_HEADER */
#endif /* __cplusplus */
#endif /* !_CPY_TBL */
```

For each object component that is marked for a copy, the linker creates a COPY_RECORD object for it. Each COPY_RECORD contains at least the following information for the object component:

- The load address

- The run address

- The size

The linker collects all COPY_RECORDs that are associated with the same copy table into a COPY_TABLE object. The COPY_TABLE object contains the size of a given COPY_RECORD, the number of COPY_RECORDs in the table, and the array of COPY_RECORDs in the table. For instance, in the BINIT example in *Boot-Time Copy Tables*, the .first and .extra output sections will each have their own COPY_RECORD entries in the BINIT copy table. The BINIT copy table will then look like this:

```
COPY_TABLE __binit__ = { 12, 2,
                        { <load address of .first>,
                          <run address of .first>,
                          <size of .first> },
                        { <load address of .extra>,
                          <run address of .extra>,
                          <size of .extra> } };
```

General-Purpose Copy Routine

The cpy_tbl.h file in *Copy Table Contents* also contains a prototype for a general-purpose copy routine, copy_in(), which is provided as part of the run-time-support library. The copy_in() routine takes a single argument: the address of a linker-generated copy table. The routine then processes the copy table data object and performs the copy of each object component specified in the copy table.

The copy_in() function definition is provided in the cpy_tbl.c run-time-support source file shown in the following example.

```
/
↪********************************************************************
↪
/* cpy_tbl.c v#####                                                ␣
↪            */
/*                                                                 ␣
↪            */
/* General-purpose copy routine. Given the address of a linker-
↪generated    */
/* COPY_TABLE data structure, effect the copy of all object␣
↪components       */
/* that are designated for copy via the corresponding LCF␣
↪table() operator.  */
/
↪********************************************************************
↪
#include <cpy_tbl.h>
#include <string.h>

typedef void (*handler_fptr)(const unsigned char *in, unsigned␣
↪char *out)

/
↪********************************************************************
↪
/* COPY_IN()                                                       ␣
↪            */
/
↪********************************************************************
↪
void copy_in(COPY_TABLE *tp)
{
    unsigned short I;
```

```
    for (I = 0; I < tp->num_recs; I++)
    {
        COPY_RECORD crp = tp->recs[i];
        unsigned char *ld_addr = (unsigned char *)crp.load_addr;
        unsigned char *rn_addr = (unsigned char *)crp.run_addr;

        if (crp.size)
        {
            /*------------------------------------------------
↪--------------*/
            /* Copy record has a non-zero size so the data is␣
↪not compressed.   */
            /* Just copy the data.                          ␣
↪                */
            /*------------------------------------------------
↪--------------*/
            memcpy(rn_addr, ld_addr, crp.size);
        }
    }
}
```

3.10.9 Linker-Generated CRC Tables and CRC Over Memory Ranges

The linker supports an extension to the linker command file syntax that enables the verification of code or data by means of Cyclic Redundancy Code (CRC). The linker computes a CRC value for the specified region at link time, and stores that value in target memory such that it is accessible at boot or run time. The application code can then compute the CRC for that region and ensure that the value matches the linker-computed value.

In a linker command file, you can cause CRC values to be generated for the following:

- **CRC for a section:** Use the crc_table() operator within the SECTIONS directive. See *Using the crc_table() Operator in the SECTIONS Directive*.

- **CRC for memory range:** Use the crc() operator for a GROUP in a MEMORY directive. See *Using the crc_table() Operator in the MEMORY Directive*.

The run-time-support library does not supply a routine to calculate CRC values at boot or run time. Examples that perform cyclic redundancy checking using linker-generated CRC tables are provided in the Tools Insider blog in TI's E2E community.

Contents:

Using the crc_table() Operator in the SECTIONS Directive

For any section that should be verified with a CRC, the linker command file must be modified to include the crc_table() operator. The specification of a CRC algorithm is optional. The syntax is:

crc_table(*user_specified_table_name*[**, algorithm=***xxx*]**)**

The linker uses the CRC algorithm from any specification given in a crc_table() operator. If that specification is omitted, the TMS570_CRC64_ISO algorithm is used. The linker includes CRC table information in the map file. This includes the CRC value as well as the algorithm used for the calculation.

The CRC table generated for a particular crc_table() instance can be accessed through the table name provided as an argument to the crc_table() operator. The linker creates a symbol with this name and assigns the address of the CRC table as the value of the symbol. The CRC table can then be accessed from the application using the linker-generated symbol.

The crc_table() operator can be applied to an output section, a GROUP, a GROUP member, a UNION, or a UNION member. In a GROUP or UNION, the operator is applied to each member.

You can include calls in your application to a routine that will verify CRC values for relevant sections. You must provide this routine. See below for more details on the data structures and suggested interface.

Restrictions when using the crc_table() Operator

It is important to note that the CRC generator used by the linker is parameterized as described in the crc_tbl.h header file (see *Interface When Using the crc_table() Operator*). Any CRC calculation routine employed outside of the linker must function in the same way to ensure matching CRC values. The linker cannot detect a mismatch in the parameters. To understand these parameters, see A Painless Guide to CRC Error Detection Algorithms by Ross Williams.

Only CRC algorithm names and identifiers in crc_tbl.h are supported. All other names and ID values are reserved for future use. Systems may not include built-in hardware that computes these CRC algorithms. Consult documentation for your hardware for details. These CRC algorithms are supported:

- CRC8_PRIME
- CRC16_ALT
- CRC16_802_15_4
- CRC_CCITT
- CRC24_FLEXRAY
- CRC32_PRIME
- CRC32_C

- CRC64_ISO

The default is the TMS570_CRC64_ISO algorithm, which has an initial value of 0. Additional information about the algorithm can be found in *A Note on the TMS570_CRC64_ISO Algorithm*.

There are also restrictions that will be enforced by the linker:

- CRC can only be requested at final link time.

- CRC can only be applied to initialized sections.

- CRC can be requested for load addresses only.

- Certain restrictions also apply to CRC table names. For example, BINIT may not be used as a CRC table name.

Examples When Using the crc_table() Operator

The crc_table() operator is similar in syntax to the table() operator used for copy tables. A few simple examples of linker command files follow.

The following example defines a section named ".section_to_be_verified", which contains the .text data from the a1.c.o file. The crc_table() operator requests that the linker compute the CRC value for the .text data and store that value in a table named "my_crc_table_for_a1".

```
SECTIONS
{
    ...
    .section_to_be_verified: {a1.c.o(.text)} crc_table(_my_crc_
→table_for_a1)
}
```

This table will contain all the information needed to invoke a user-supplied CRC calculation routine, and verify that the CRC calculated at run time matches the linker-generated CRC. The table can be accessed from application code using the symbol my_crc_table_for_a1, which should be declared of type "extern CRC_TABLE". This symbol will be defined by the linker. The application code might resemble the following.

```
#include "crc_tbl.h"

extern CRC_TABLE my_crc_table_for_a1;

verify_a1_text_contents()
{
    ...
    /* Verify CRC value for .text sections of a1.c.o. */
```

(continues on next page)

```
    if (my_check_CRC(&my_crc_table_for_a1)) puts("OK");
}
```

The my_check_CRC() routine is shown in detail in *Interface When Using the crc_table() Operator*.

In the following example, the CRC algorithm is specified in the crc_table() operator. The specified algorithm is used to compute the CRC of the text data from b1.c.o. The CRC tables generated by the linker are created in the special section .TI.crctab, which can be placed in the same manner as other sections. In this case, the CRC table _my_crc_table_for_b1 is created in section .TI.crctab:_my_crc_table_for_b1, and that section is placed in the CRCMEM memory region.

```
SECTIONS
{
    ...
    .section_to_be_verified_2: {b1.c.o(.text)} load=SLOW_MEM,␣
→run=FAST_MEM,
        crc_table(_my_crc_table_for_b1, algorithm=TMS570_CRC64_
→ISO)

.TI.crctab: > CRCMEM
}
```

In the following example, the same identifier, _my_crc_table_for_a1_and_c1, is specified for both a1.c.o and c1.c.o. The linker creates a single table that contains entries for both text sections. Multiple CRC algorithms can occur in a single table. In this case, _my_crc_table_for_a1_and_c1 contains an entry for the text data from a1.c.obj using the default CRC algorithm, and an entry for the text data from c1.c.obj using the TMS570_CRC64_ISO algorithm. The order of the entries is unspecified.

```
SECTIONS
{
    .section_to_be_verified_1: {a1.c.o(.text)}
        crc_table(_my_crc_table_for_a1_and_c1)
    .section_to_be_verified_3: {c1.c.o(.text)}
        crc_table(_my_crc_table_for_a1_and_c1, algorithm=TMS570_
→CRC64_ISO)
}
```

When the crc_table() operator is applied to a GROUP or a UNION, the linker applies the table specification to the members of the GROUP or UNION.

In the following example, the linker creates two CRC tables, table1 and table2. table1 contains one entry for section1. Because both sections are members of the UNION, table2 contains entries for section1 and section2. The order of the entries in table2 is unspecified.

```
SECTIONS
{
    UNION
    {
        section1: {} crc_table(table1)
        section2:
    } crc_table(table2)
}
```

Interface When Using the crc_table() Operator

The CRC generation function uses a mechanism similar to the copy table functionality. Using the syntax shown above in the linker command file allows specification of code/data sections that have CRC values computed and stored in the run time image. This section describes the table data structures created by the linker, and how to access this information from application code.

The CRC tables contain entries as detailed in the run-time-support header file crc_tbl.h, as shown in the following figure:
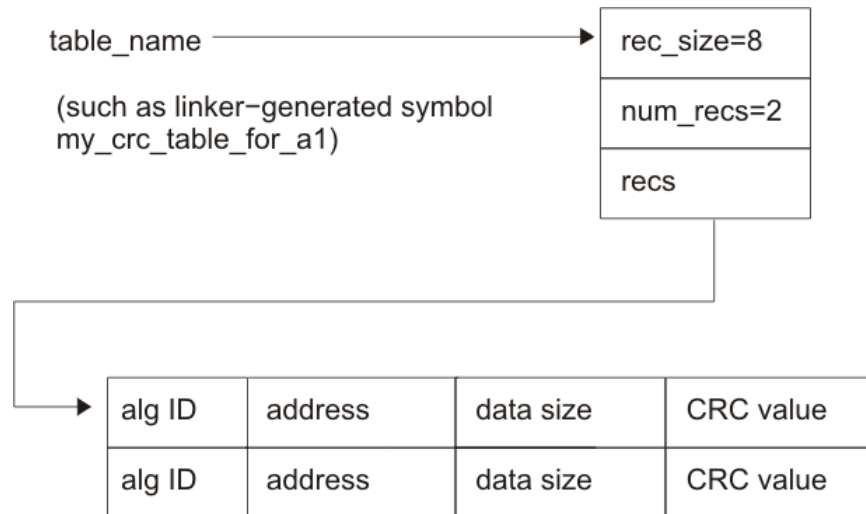
Figure 3.30: CRC Table Format

The crc_tbl.h header file is included below. This file specifies the C structures created by the linker to manage CRC information. It also includes the specifications of the supported CRC algorithms. A full discussion of CRC algorithms is beyond the scope of this document, and the interested reader should consult the referenced document for a description of the fields shown in the table. The following fields are relevant to this document.

- Name – text identifier of the algorithm, used by the programmer in the linker command file.

- ID – the numeric identifier of the algorithm, stored by the linker in the crc_alg_ID member of each table entry.

- Order – the number of bits used by the CRC calculation.

- Polynomial – used by the CRC computation engine.

- Initial Value – the initial value given to the CRC computation engine.

```
/
↪*******************************************************************
↪
/* crc_tbl.h                                                      ␣
↪           */
/*                                                                ␣
↪           */
/* Specification of CRC table data structures which can be␣
↪automatically    */
/* generated by the linker (using the crc_table() operator in␣
↪the linker     */
/* command file).                                                ␣
↪           */
/
↪*******************************************************************
↪
/*                                                                ␣
↪           */
/* The CRC generator used by the linker is based on concepts␣
↪from the        */
/* document:                                                      ␣
↪           */
/* "A Painless Guide to CRC Error Detection Algorithms"           ␣
↪           */
/*                                                                ␣
↪           */
/* Author : Ross Williams (ross@guest.adelaide.edu.au.).          ␣
↪           */
/* Date : 3 June 1993.                                            ␣
↪           */
/* Status : Public domain (C code).                               ␣
↪           */
/*                                                                ␣
↪           */
/* Description : For more information on the Rocksoft^tm Model␣
↪CRC            */
/* Algorithm, see the document titled "A Painless Guide to CRC␣
↪Error          */
```

```
/* Detection Algorithms" by Ross Williams (ross@guest.adelaide.
↪edu.au.).       */
/
↪****************************************************************************
↪
#include <stdint.h>        /* For uintXX_t */


/
↪****************************************************************************
↪
/* CRC Algorithm Specifiers                                              ␣
↪            */
/*                                                                      ␣
↪            */
/* The following specifications, based on the above cited␣
↪document, are used */
/* by the linker to generate CRC values.                                ␣
↪            */
/*
 ID Name                   Order Polynomial  Initial     Ref Ref␣
↪CRC XOR     Zero
                                              Value       In  Out␣
↪Value       Pad
 -----------------------------------------------------------------
↪------------
 10 "TMS570_CRC64_ISO", 64,   0x0000001b, 0x00000000, 0,  0, ␣
↪0x00000000, 1

                                                                   ␣
↪            */
/* Users should specify the name, such as TMS570_CRC64_ISO, in␣
↪the linker    */
/* command file. The resulting CRC_RECORD structure will contain␣
↪the        */
/* corresponding ID value in the crc_alg_ID field.                     ␣
↪            */
/
↪****************************************************************************
↪
#define TMS570_CRC64_ISO 10


/***************************************************************/
/* CRC Record Data Structure                                 */
```

3.10. Linker Description

```
/* NOTE: The list of fields and the size of each field    */
/* varies by target and memory model.                     */
/**************************************************************/
typedef struct crc_record
{
    uint64_t    crc_value;
    uint32_t    crc_alg_ID; /* CRC algorithm ID        */
    uint32_t    addr;       /* Starting address        */
    uint32_t    size;       /* size of data in bytes */
    uint32_t    padding;    /* explicit padding so layout is the
 ↪same for ELF */
} CRC_RECORD;
```

In the CRC_TABLE struct, the array recs[1] is dynamically sized by the linker to accommodate the number of records contained in the table (num_recs). A user-supplied routine to verify CRC values should take a table name and check the CRC values for all entries in the table. An outline of such a routine is shown in the following example:

```
/**************************************************************/
/* General-purpose CRC check routine. Given the address of a  */
/* linker-generated CRC_TABLE data structure, verify the CRC  */
/* of all object components that are designated with the      */
/* corresponding LCF crc_table() operator.                    */
/**************************************************************/
#include <crc_tbl.h>

/**************************************************************/
/* MY_CHECK_CRC() - returns 1 if CRCs match, 0 otherwise */
/**************************************************************/
unsigned int my_check_CRC(CRC_TABLE *tp)
{
    int i;

    for (i = 0; i < tp-> num_recs; i++)
    {
        CRC_RECORD crc_rec = tp->recs[i];

        /****************************************************/
        /* COMPUTE CRC OF DATA STARTING AT crc_rec.addr     */
        /* FOR crc_rec.size UNITS. USE                      */
        /* crc_rec.crc_alg_ID to select algorithm.          */
        /* COMPARE COMPUTED VALUE TO crc_rec.crc_value.     */
        /****************************************************/
```

```
    }
    if all CRCs match, return 1;
    else return 0;
}
```

Using the crc_table() Operator in the MEMORY Directive

Along with generating CRC Tables, the linker can also generate CRCs over memory ranges as well. To do this, instead of using the crc_table() operator in a SECTIONS directive, you use the crc() operator in a MEMORY directive. Within the MEMORY directive, you specify a GROUP of memory regions to have a CRC value computed. The memory ranges in the GROUP must be continuous.

The syntax is as follows:

```
MEMORY
{
    GROUP(FLASH)
    {
        RANGE1 :...
        RANGE2 :...
    } crc(_table_name, algorithm=xxx)
}
```

This syntax causes the linker to compute a single CRC over both RANGE1 and RANGE2. The CRC is based on the algorithm specified, taking into account all output sections that have been placed in those ranges. The result is stored in a table in the format described in *Interface When Using the crc() Operator*. This table is placed in an output section called .TI.memcrc, which is accessible through the table name as a linker symbol.

The algorithm argument for crc() may be any algorithm listed in *Restrictions when using the crc_table() Operator*. The algorithm is required in the current version, and linking will fail without it. In future releases, the algorithm specification will be optional, and the default is specified. If no algorithm is specified, the default algorithm will be chosen, which is TMS570_CRC64_ISO.

Specifying the GROUP name is optional. For example:

```
MEMORY
{
    GROUP
    {
        RANGE1 :...
        RANGE2 :...
```

```
        } crc(_table_name, algorithm=CRC8_PRIME)
}
```

When GROUP is used inside a MEMORY block, the syntax options are limited to the functionality described here and in the subsections that follow. The full functionality described in *Using GROUP and UNION Statements* for GROUP within the SECTIONS directive is not available within the MEMORY directive.

Restrictions when Using the crc() Operator

The crc() operator can only be applied to a GROUP within a MEMORY directive. It cannot be applied to individual memory ranges in a MEMORY directive or to groups in the SECTIONS directive.

Along with the restrictions described in *Restrictions when using the crc_table() Operator*, the following additional restrictions apply:

- Memory range groups cannot contain any gaps between the ranges.

- All of the memory ranges must be on the same page.

- Memory ranges that contain sections that would not otherwise be eligible for CRC table generation cannot have a CRC computed. That is, memory ranges for which a CRC value is generated must correspond only to load addresses of initialized sections.

- The .TI.memcrc section may not be placed in a range that itself is having a CRC value computed. This would result in a circular reference; the CRC result would depend upon the result of the CRC. See *Generate CRC for Most or All of Flash Memory* for ways to generate CRCs for most or all of Flash memory without violating this restriction.

Using the VFILL Specifier within a GROUP

In addition to specifying the origin and length of a memory range within a GROUP, you can also use the VFILL specifier, as described in *Using the VFILL Specifier in the Memory Map*, to allow ECC data to be generated for areas of the input memory range that remain uninitialized.

The load image will have gaps between output sections, and how these bits are set depends on your device. Most devices count empty spaces as 0x1 values, but if your device counts empty space as 0x0 values, the result of the CRC will be different. Thus, if the CRC result does not line up, make sure that you specify the empty space byte with the VFILL parameter, as shown in the following example:

```
MEMORY
{
```

```
    GROUP
    {
        FLASH : origin = 0x0000, length = 0x1000,
                VFILL = 0x0 /* Fill gaps with zeroes */
    } crc(_table_name, CRC8_PRIME)
}
```

If no VFILL parameter is specified, it defaults to 0x1, which fills everything with ones. Remember to update every memory range that has a fill value other than 0x1 for CRCs.

Generate CRC for Most or All of Flash Memory

If you are trying to generate a CRC value for the entire FLASH memory, place the table in a separate memory range, which .TI.memcrc will be placed in by default. For example:

```
MEMORY
{
    /* Carve out a section of FLASH to store the CRC result */
    CRC_PRELUDE : origin=0x0, length=0x10
    GROUP
    {
        FLASH : origin=0x10, length=0xFFFF
    } crc(_flash_crc, algorithm=CRC8_PRIME)
    /* Other memory ranges... */
}
SECTION
{
    .TI.memcrc > CRC_PRELUDE
}
```

In the above example, a small section of flash has been cut out of the whole, to allow the .TI.memcrc section to reside there, while everything else that is eligible for CRC generation is placed in FLASH. This avoids placing the CRC result in the CRC range.

In some cases, you may want to generate a CRC for all of Flash memory and read back the CRC result via the linker-generated map file (see *Create a Map File (--map_file Option)*). However, there is no memory location to place the CRC result for the memory range covering all of Flash memory. If you place it in Flash, then you violate the rule that the result cannot be placed within the input range. Thus, if there's no good place to put the CRC result, you can mark the .TI.memcrc section as a COPY section like so:

```
.TI.memcrc : type=COPY
```

This prevents the CRC result for a memory range from being placed anywhere. Marking .TI.memcrc as a DSECT section has the same result.

Computing CRCs for Both Memory Ranges and Sections

You can run a CRC on both memory ranges and output sections together. In the following example, a CRC is computed over the memory range FLASH2, which is used only by the .text section. A CRC table is also generated for only the .text output section, which does not include the rest of the memory range.

```
MEMORY
{
    FLASH1 : origin = 0x0000, length = 0x1000
    GROUP
    {
        FLASH2 : origin = 0x1000, length=0x1000
    } crc(_memrange_flash_crc, algorithm=CRC8_PRIME)
}
SECTION
{
    .TI.memcrc > CRC_PRELUDE
    .text > FLASH2, crc_table(_crc_table, algorithm=CRC8_PRIME)
}
```

Example Specifying Memory Range CRCs

Here is a full linker command file that uses the crc() operator to generate a memory range CRC:

```
-c                 /* Use C linking conventions: auto-init vars at␣
↪runtime */
-stack 0x1400      /* Stack size */
-heap  0x0c00      /* Heap size */

MEMORY
{
    GROUP(FLASH)
    {
        MEM(RW)          : origin = 0x1200, length = 0x9DE0, VFILL␣
↪= 0x0
    } crc(_ext_memrange_crc, algorithm=CRC32_PRIME)

    MEM2                 : origin = 0xAFE0, length = 0x5000
```

```
    VECTORS(R)          : origin = 0xFFE0, length = 0x001E
    RESET               : origin = 0xFFFE, length = 0x0002
}

SECTIONS
{
    .intvecs            : {} > VECTORS

    /* These sections are uninitialized */
    .bss                : {} > MEM2
    .sysmem             : {} > MEM2
    .stack              : {} > MEM2

    /* These sections will be CRC'd */
    .text               : {} > MEM
    .const              : {} > MEM
    .rodata             : {} > MEM
    .cinit              : {} > MEM
    .switch             : {} > MEM

    .reset              : > RESET
}
```

Interface When Using the crc() Operator

CRCs over memory ranges are stored in a table format similar to that shown in *Interface When Using the crc_table() Operator* for the CRCs over sections. However, the table format is different than that of CRC tables.

The following figure shows the storage format for CRCs over memory ranges with example values:
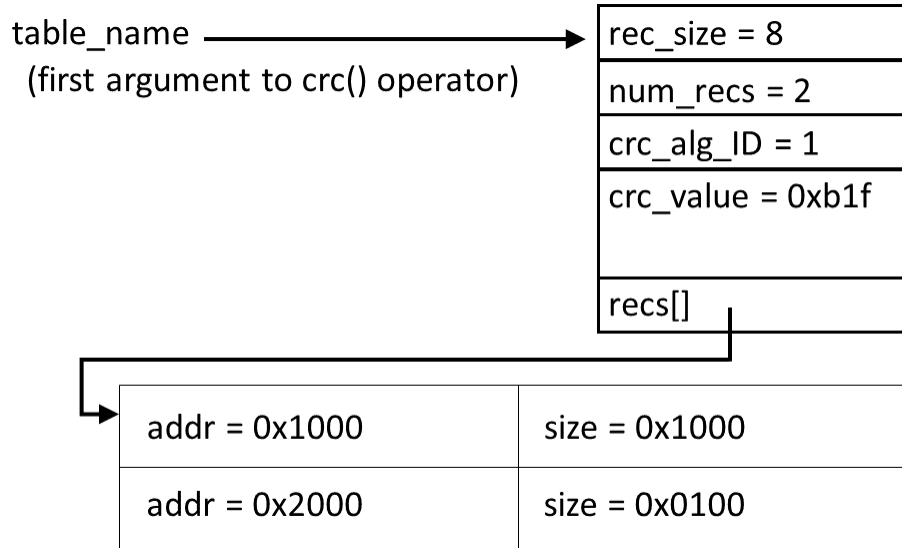
Figure 3.31: CRC Storage Format

The table header stores the record count and size, as well as the algorithm type and the CRC result. Each table entry encodes the start address and length of a memory range that was used to compute the CRC.

The following header file excerpt shows the C structures the linker creates to manage the CRC information:

```
typedef struct memrange_crc_record {
    uintptr_t           addr;      /* Starting address */
#if defined(__LARGE_CODE_MODEL__) || defined(__LARGE_DATA_MODEL__)
    uint32_t            size;      /* size of data in 8-bit
addressable units */
#else
    uint16_t            size;      /* size of data in 8-bit
addressable units */
#endif
} MEMRANGE_CRC_RECORD;
typedef struct memrange_crc_table {
    uint16_t            rec_size;  /* 8-bit addressable units
*/
    uint16_t            num_recs;  /* how many records are in
the table */
    uint16_t            crc_alg_ID; /* CRC algorithm ID */
    uint32_t            crc_value; /* result of crc */
    MEMRANGE_CRC_RECORD recs[1];
} MEMRANGE_CRC_TABLE;
```

A Note on the TMS570_CRC64_ISO Algorithm

The MCRC module calculates CRCs on 64-bit chunks of data. This is accomplished by writing a long long value to two memory mapped registers. In C this looks like a normal write of a long long to memory. The code generated to read/write a long long to memory is something like the following, where R2 contains the most significant word and R3 contains the least significant word. So the most significant word is written to the low address and the least significant word is written to the high address:

```
LDM R0, {R2, R3}
STM R1, {R2, R3}
```

The CRC memory mapped registers are in the reverse order from how the compiler performs the store. The least significant word is mapped to the low address and the most significant word is mapped to the high address.

This means that the words are actually swapped before performing the CRC calculation. It also means that the calculated CRC value has the words swapped. The TMS570_CRC64_ISO algorithm takes these issues into consideration and performs the swap when calculating the CRC value. The computed CRC value stored in the table has the words swapped so the value is the same as it is in memory.

For the end user, these details should be transparent. If the run-time CRC routine is written in C, the long long loads and stores will be generated correctly. The DMA mode of the MCRC module will also work correctly.

Another issue with the algorithm is that it requires the run-time CRC calculation to be done with 64-bit chunks. The MCRC module allows smaller chunks of data, but the values are padded to 64-bits. The TMS570_CRC64_ISO algorithm does not perform any padding, so all CRC computations must be done with 64-bit values. The algorithm will automatically pad the end of the data with zeros if it does not end on a 64-bit boundary.

3.10.10 Partial (Incremental) Linking

An output file that has been linked can be linked again with additional modules. This is known as *partial linking* or *incremental linking*. Partial linking allows you to partition large applications, link each part separately, and then link all the parts together to create the final executable program.

Follow these guidelines for producing a file that you will relink:

- The intermediate files produced by the linker *must* have relocation information. Use the --relocatable option when you link the file the first time. (See *Producing a Relocatable Output Module (--relocatable option)*.)

- Intermediate files *must* have symbolic information. By default, the linker retains symbolic information in its output. Do not use the --no_sym_table option if you plan to relink a file,

because --no_sym_table strips symbolic information from the output module. (See *Strip Symbolic Information (--no_symtable Option)*.)

- Intermediate link operations should be concerned only with the formation of output sections and not with allocation. All allocation, binding, and MEMORY directives should be performed in the final link.

- If the intermediate files have global symbols that have the same name as global symbols in other files and you want them to be treated as static (visible only within the intermediate file), you must link the files with the --make_static option (see *Make All Global Symbols Static (--make_static Option)*).

- If you are linking C code, do not use --ram_model or --rom_model until the final link. Every time you invoke the linker with the --ram_model or --rom_model option, the linker attempts to create an entry point. (See *C Language Options (--ram_model and --rom_model Options)*, *Autoinitializing Variables at Run Time (--rom_model)*, and *Initializing Variables at Load Time (--ram_model)*.)

The following example shows how you can use partial linking:

Step 1: Link the file file1.com; use the --relocatable option to retain relocation information in the output file tempout1.out.

```
tiarmclang -Wl,--relocatable,--output_file=tempout1 file1.com
```

file1.com contains:

```
SECTIONS
{
    ss1: {
        f1.c.o
        f2.c.o
        .
        .
        .
        fn.c.o
    }
}
```

Step 2: Link the file file2.com; use the --relocatable option to retain relocation information in the output file tempout2.out.

```
tiarmclang -Wl,--relocatable,--output_file=tempout2 file2.com
```

file2.com contains:

```
SECTIONS
{
    ss2: {
        g1.c.o
        g2.c.o
        .
        .
        .
        gn.c.o
    }
}
```

Step 3: Link tempout1.out and tempout2.out.

```
tiarmclang -Wl,--map_file=final.map,--output_file=final.out␣
↪tempout1.out tempout2.out
```

3.10.11 Linking C/C++ Code

The C/C++ compiler produces assembly language source code that can be assembled and linked. For example, a C program consisting of modules prog1, prog2, etc., can be assembled and then linked to produce an executable file called prog.out:

```
tiarmclang -Wl,--rom_model,--output_file=prog.out prog1.c.o␣
↪prog2.c.o ...
```

The --rom_model option tells the linker to use special conventions that are defined by the C/C++ environment.

The archive libraries shipped by TI contain C/C++ run-time-support functions and are brought it automatically.

C, C++, and mixed C and C++ programs can use the same run-time-support library. Run-time-support functions and variables that can be called and referenced from both C and C++ have the same linkage.

For more information about the Arm C/C++ language, including the run-time environment and run-time-support functions, see *C/C++ Language Implementation*.

Contents:

Linking for Run-Time Initialization

All C/C++ programs must be linked with code to initialize and execute the program, called a *bootstrap* routine, also known as the *boot.c.o* object module. The symbol _c_int00 is defined as the program entry point and is the start of the C boot routine in boot.c.o; referencing _c_int00 ensures that boot.c.o is automatically linked in from the run-time-support library. When a program begins running, it executes boot.c.o first. The boot.c.o symbol contains code and data for initializing the run-time environment and performs the following tasks:

- Changes from system mode to user mode

- Sets up the user mode stack

- Processes the run-time *.cinit* initialization table and autoinitializes global variables (when the linker is invoked with the --rom_model option)

- Calls main

The run-time-support object libraries contain boot.c.o. You can:

- Use the archiver to extract boot.c.o from the library and then link the module in directly.

- Include the appropriate run-time-support library as an input file (the linker automatically extracts boot.c.o when you use the --ram_model or --rom_model option).

Object Libraries and Run-Time Support

The *Built-In Functions* section describes additional built-in functions that are predefined by the compiler. If your program uses any of these functions, you must link the appropriate run-time-support library with your object files. See also *Library Naming Conventions*.

You can also create your own object libraries and link them. The linker includes and links only those library members that resolve undefined references.

If you want to link object files created with the TI CodeGen tools with object files generated by other compiler tool chains, the Arm standard specifies that you should define the _AE-ABI_PORTABILITY_LEVEL preprocessor symbol as follows before #including any standard header files, such as <stdlib.h>.

```
#define _AEABI_PORTABILITY_LEVEL 1
```

This definition enables full portability. Defining the symbol to 0 specifies that the "C standard" portability level should be used.

Setting the Size of the Stack and Heap Sections

The C/C++ language uses two uninitialized sections called .sysmem and .stack for the memory
pool used by the malloc( ) functions and the run-time stacks, respectively. You can set the size
of these by using the --heap_size or --stack_size option and specifying the size of the section as
a 4-byte constant immediately after the option. If the options are not used, the default size of the
heap is 2K bytes and the default size of the stack is 2K bytes.

See *Define Heap Size (--heap_size Option)* for setting heap sizes and *Define Stack Size (--stack_size
Option)* for setting stack sizes.

To debug issues related to the stack size, we recommend using the CCS Stack Usage view to see
the static stack usage of each function in the application. See Stack Usage View in CCS for more
information. Using the Stack Usage View requires that source code be built with *debug enabled*.
This feature relies on the *–call_graph* capability provided by the *tiarmofd - Object File Display
Utility*.

Initializing and AutoInitializing Variables at Run Time

Autoinitializing variables at run time is the typical method of autoinitialization. To use this method,
invoke the linker with the --rom_model option. See *Autoinitializing Variables at Run Time (--
rom_model)* for details.

Initialization of variables at load time enhances performance by reducing boot time and by saving
the memory used by the initialization tables. To use this method, invoke the linker with the --
ram_model option. See *Initializing Variables at Load Time (--ram_model)* for details.

See *The --rom_model and --ram_model Linker Options* for information about the steps that are
performed when you invoke the linker with the --ram_model or --rom_model option. See *RAM
Model vs. ROM Model* for further information.

Initialization of Cinit and Watchdog Timer Hold

You can use the --cinit_hold_wdt option on some devices to specify whether the watchdog timer
should be held (on) or not held (off) during cinit auto-initialization. Setting this option causes an
RTS auto-initialization routine to be linked in with the program to handle the desired watchdog
timer behavior.

3.10.12 Linker Example

This example links three object files named demo.c.o, ctrl.c.o, and tables.c.o and creates a program called demo.out.

Assume that target memory has the following program memory configuration:

Address Range	Contents
0x00000000 to 0x00001000	SLOW_MEM
0x00001000 to 0x00002000	FAST_MEM
0x08000000 to 0x08000400	EEPROM

The output sections are constructed in the following manner:

- Executable code, contained in the .text sections of demo.c.o, ctrl.c.o, and tables.c.o, must be linked into FAST_MEM.

- A set of interrupt vectors, contained in the .intvecs section of tables.c.o, must be linked at address FAST_MEM.

- A table of coefficients, contained in the .data section of tables.c.o, must be linked into EEPROM. The remainder of block FLASH must be initialized to the value 0xFF00FF00.

- A set of variables, contained in the .bss section of ctrl.c.o, must be linked into SLOW_MEM and preinitialized to 0x00000100.

- The .bss sections of demo.c.o and tables.c.o must be linked into SLOW_MEM.

The following example shows the linker command file for this example. After the linker command file, the map file is shown.

```
/
↪****************************************************************************
↪
/*** Specify Link Options ***/
/
↪****************************************************************************
↪
--entry_point SETUP /* Define the program entry point */
--output_file=demo.out /* Name the output file */
--map_file=demo.map /* Create an output map file */
/
↪****************************************************************************
↪
/*** Specify the Input Files ***/
/
↪****************************************************************************
↪
```
(continues on next page)

```
demo.c.o
ctrl.c.o
tables.c.o
/
↪********************************************************************
↪
/*** Specify the Memory Configurations ***/
/
↪********************************************************************
↪
MEMORY
{
    FAST_MEM : org = 0x00000000 len = 0x00001000 /* PROGRAM␣
↪MEMORY (ROM) */
    SLOW_MEM : org = 0x00001000 len = 0x00001000 /* DATA MEMORY␣
↪(RAM) */
    EEPROM : org = 0x08000000 len = 0x00000400 /* COEFFICIENTS␣
↪(EEPROM) */
}
/
↪********************************************************************
↪
/* Specify the Output Sections */
/
↪********************************************************************
↪
SECTIONS
{
    .text : {} > FAST_MEM /* Link all .text sections into ROM */
    .intvecs : {} > 0x0 /* Link interrupt vectors at 0x0 */
    .data : /* Link .data sections */
    {
        tables.c.o(.data)
        . = 0x400; /* Create hole at end of block */
    } > EEPROM, fill = 0xFF00FF00 /* Fill and link into EEPROM */
    ctrl_vars: /* Create new sections for ctrl variables */
    {
        ctrl.c.o(.bss)
    } > SLOW_MEM, fill = 0x00000100 /* Fill with 0x100 and link␣
↪into RAM */
    .bss : {} > SLOW_MEM /* Link remaining .bss sections into␣
↪RAM */
```

3.10. Linker Description   **623**

```
}
/
↪********************************************************************
↪
/*** End of Command File ***/
/
↪********************************************************************
↪
```

Invoke the linker by entering the following command:

```
tiarmclang demo.cmd
```

This creates the following map file and an output file called demo.out that can be run on an Arm device.

```
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: "SETUP"  address: 000000d4
MEMORY CONFIGURATION


name        origin      length      attributes      fill
--------    --------    ---------   ----------      --------
FAST_MEM    00000000    000001000       RWIX
SLOW_MEM    00001000    000001000       RWIX
EEPROM      08000000    000000400       RWIX


SECTION ALLOCATION MAP


output                                            attributes/
section    page    origin      length      input sections
--------   ----    ---------   ---------   ----------------
.text      0       00000020    00000138
                   00000020    000000a0    ctrl.c.o (.text)
                   000000c0    00000000    tables.c.o (.text)
                   000000c0    00000098    demo.c.o (.text)


.intvecs   0       00000000    00000020
                   00000000     00000020     tables.c.o (.intvecs)


.data      0       08000000    00000400
                   08000000    00000168    tables.c.o (.data)
                   08000168    00000298    --HOLE-- [fill =␣
↪ff00ff00]
```

```
                      08000400      00000000        ctrl.c.o (.data)
                      08000400      00000000        demo.c.o (.data)


ctrl_var    0     00001000      00000500
                  00001000      00000500        ctrl.c.o (.bss) [fill␣
 ↪= 00000100]


.bss        0     00001500      00000100        UNINITIALIZED
                  00001500      00000100         demo.c.o (.bss)
                  00001600      00000000         tables.c.o (.bss)


GLOBAL SYMBOLS
address   name                                   address   name
--------- ----                                   --------- ----
000000d4 SETUP                                   00000020 clear
00000020 clear                                   000000b8 set
000000b8 set                                     000000c0 x42
000000c0 x42                                     000000d4 SETUP


[4 symbols]
```

3.10.13 XML Link Information File Description

The linker supports the generation of an XML link information file via the --xml_link_info file option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information in this file includes all of the information that is produced in a linker-generated map file.

See *Generate XML Link Information File (--xml_link_info Option)* for information about using the --xml_link_info option.

XML Information File Element Types

These types of elements are generated by the linker:

- **Container elements** represent an object that contains other elements that describe the object. Container elements have an *id* attribute that makes them accessible from other elements.

- **String elements** contain a string representation of their value.

- **Constant elements** contain a 64-bit unsigned long representation of their value (with a `0x` prefix).

- **Reference elements** are empty elements that contain an *idref* attribute that specifies a link to another container element.

Document Elements

The root element, also called the document element, is **<link_info>**. All other elements contained in the XML link information file are children of the <link_info> element.

The following sections describe the elements that an XML information file can contain. In the following sections, the data type of each element value is specified in parentheses following the element description. For example: The <address> is the entry point address (constant).

Header Elements

Within the **<link_info>** element, the first elements in the XML link information file provide general information about the linker and the link session:

- The **<banner>** element lists the name of the executable and the version information (string).

- The **<copyright>** element lists the TI copyright information (string).

- The **<link_time>** is a timestamp representation of the link time (unsigned 32-bit int).

- The **<output_file>** element lists the absolute path and name of the linked output file generated (string).

- The **<entry_point>** element specifies the program entry point, as determined by the linker (container) with two entries:

  - The **<name>** is the entry point symbol name, if any (string).

  - The **<address>** is the entry point address (constant).

Example Header Elements in the hi.out Output File

```
<banner>Linker Version x.xx (Mar 15 2024)</banner>
<copyright>Copyright (c) 1996-2024 Texas Instruments Incorporated
 ↪</copyright>
<link_time>0x65f8a4ec</link_time>
<output_file>/usr/mycode/hi.out</output_file>
<entry_point>
    <name>_c_int00</name>
    <address>0xaf80</address>
</entry_point>
```

Input File List

After the header elements, the next section in the XML link information file is the input file list, which is delimited with an **<input_file_list>** container element. The <input_file_list> can contain any number of <input_file> elements.

Each **<input_file>** instance specifies the input file involved in the link. Each <input_file> has an *id* attribute that can be referenced by other elements, such as an <object_component>. An <input_file> is a container element enclosing the following elements:

- The <path> element names an absolute directory path, if applicable (string).
- The <kind> element specifies a file type, either archive or object (string).
- The <file> element specifies an archive name or filename (string).
- The <name> element specifies an object file name, or archive member name (string).

Input File List for the hi.out Output File

```
<input_file_list>
    <input_file id="fl-1">
        <kind>object</kind>
        <file>hi.obj</file>
        <name>hi.obj</name>
    </input_file>
    <input_file id="fl-2">
        <path>/usr/tools/lib/</path>
        <kind>archive</kind>
        <file>rtsxxx.lib</file>
        <name>boot.obj</name>
    </input_file>
    <input_file id="fl-3">
        <path>/usr/tools/lib/</path>
        <kind>archive</kind>
        <file>rtsxxx.lib</file>
        <name>exit.obj</name>
    </input_file>
    <input_file id="fl-4">
        <path>/usr/tools/lib/</path>
        <kind>archive</kind>
        <file>rtsxxx.lib</file>
        <name>printf.obj</name>
    </input_file>
    ...
</input_file_list>
```

Object Component List

The next section of the XML link information file contains a specification of all of the object components that are involved in the link. An example of an object component is an input section. In general, an object component is the smallest piece of object that can be manipulated by the linker.

The **<object_component_list>** is a container element enclosing any number of <object_component> elements.

Each **<object_component>** specifies a single object component. Each <object_component> has an *id* attribute so that it can be referenced directly from other elements, such as a <logical_group>. An <object_component> is a container element enclosing the following elements:

- The **<name>** element names the object component (string).

- The **<load_address>** element specifies the load-time address of the object component (constant).

- The **<run_address>** element specifies the run-time address of the object component (constant).

- The **<alignment>** element specifies the alignment of the object component (unsigned int).

- The **<size>** element specifies the size of the object component (constant).

- The **<executable>** element specifies whether the object component allows execute access (string). If an object has executable access, it cannot also have read-write access. While "false" is a valid value, the linker emits this element only if it is "true" for this object component.

- The **<readonly>** element specifies whether the object component allows read-only access (string). If an object has read-only access, it cannot also have read-write access. While "false" is a valid value, the linker emits this element only if it is "true" for this object component.

- The **<readwrite>** element specifies whether the object component allows read-write access (string). If an object has read-write access, it cannot also have read-only access or executable access. While "false" is a valid value, the linker emits this element only if it is "true" for this object component.

- The **<uninitialized>** element specifies whether the object component is initialized (string). While "false" is a valid value, the linker emits this element only if it is "true" for this object component.

- The **<input_file_ref>** element specifies the source file where the object component originated (reference).

Object Component List for the fl-4 Input File

```
<object_component id="oc-20">
    <name>.text</name>
    <load_address>0xac00</load_address>
    <run_address>0xac00</run_address>
    <alignment>0x1</alignment>
    <size>0xc0</size>
    <readonly>true</readonly>
    <executable>false</executable>
    <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-21">
    <name>.data</name>
    <load_address>0x80000000</load_address>
    <run_address>0x80000000</run_address>
    <alignment>0x1</alignment>
    <size>0x0</size>
    <readwrite>true</readwrite>
    <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-22">
    <name>.bss</name>
    <load_address>0x80000000</load_address>
    <run_address>0x80000000</run_address>
    <alignment>0x1</alignment>
    <size>0x0</size>
    <readwrite>true</readwrite>
    <uninitialized>true</uninitialized>
    <input_file_ref idref="fl-4"/>
</object_component>
```

Logical Group List

The **<logical_group_list>** section of the XML link information file is similar to the output section listing in a linker-generated map file. However, the XML link information file contains a specification of GROUP and UNION output sections, which are not represented in a map file. There are several kinds of list items that can occur in a <logical_group_list>:

- The **<logical_group>** is the specification of a section or GROUP that contains a list of object components or logical group members. Each <logical_group> element is given an *id* so that it may be referenced from other elements. Each <logical_group> is a container element enclosing the following elements:

    - The **<name>** element names the logical group (string).

- – The **<load_address>** element specifies the load-time address of the logical group (constant).

- – The **<run_address>** element specifies the run-time address of the logical group (constant).

- – The **<size>** element specifies the size of the logical group (constant).

- – The **<output_section_group>** specifies whether the logical group is a GROUP of output sections (string). While "false" is a valid value, the linker emits this element only if it is "true" for this logical group.

- – The **<contents>** element lists elements contained in this logical group (container). These elements refer to each of the member objects contained in this logical group:

  - ∗ The **<object_component_ref>** is an object component that is contained in this logical group (reference).

  - ∗ The **<logical_group_ref>** is a logical group that is contained in this logical group (reference).

- The **<overlay>** is a special kind of logical group that represents a UNION, or a set of objects that share the same memory space (container). Each <overlay> element is given an *id* so that it may be referenced from other elements (like from an <allocated_space> element in the placement map). Each <overlay> contains the following elements:

  - – The **<name>** element names the overlay (string).

  - – The **<run_address>** element specifies the run-time address of overlay (constant).

  - – The **<size>** element specifies the size of logical group (constant).

  - – The **<contents>** container element lists elements contained in this overlay. These elements refer to each of the member objects contained in this logical group:

    - ∗ The **<object_component_ref>** is an object component that is contained in this logical group (reference).

    - ∗ The **<logical_group_ref>** is a logical group that is contained in this logical group (reference).

- The **<split_section>** is another special kind of logical group that represents a collection of logical groups that is split among multiple memory areas. Each <split_section> element is given an *id* so that it may be referenced from other elements. Each <<split_section> contains the following elements:

  - – The **<name>** element names the split section (string).

  - – The **<contents>** container element lists elements contained in this split section. The <logical_group_ref> elements refer to each of the member objects contained in this split section, and each element referenced is a logical group that is contained in this split section (reference).

Logical Group List for the fl-4 Input File

```
<logical_group_list>
...
<logical_group id="lg-7">
    <name>.text</name>
    <output_section_group>true</output_section_group>
    <load_address>0x20</load_address>
    <run_address>0x20</run_address>
    <size>0xb240</size>
    <contents>
        <object_component_ref idref="oc-34"/>
        <object_component_ref idref="oc-108"/>
        <object_component_ref idref="oc-e2"/>
        ...
    </contents>
</logical_group>
...
<overlay id="lg-b">
    <name>UNION_1</name>
    <run_address>0xb600</run_address>
    <size>0xc0</size>
    <contents>
        <object_component_ref idref="oc-45"/>
        <logical_group_ref idref="lg-8"/>
    </contents>
</overlay>
...
<split_section id="lg-12">
    <name>.task_scn</name>
    <size>0x120</size>
    <contents>
        <logical_group_ref idref="lg-10"/>
        <logical_group_ref idref="lg-11"/>
    </contents>
</split_section>
...
</logical_group_list>
```

Placement Map

The **\<placement_map\>** element describes the memory placement details of all named memory areas in the application, including unused spaces between logical groups that have been placed in a particular memory area.

The **\<memory_area\>** is a description of the placement details within a named memory area (container). The description consists of these items:

- The **\<name\>** names the memory area (string).

- The **\<page_id\>** gives the id of the memory page in which this memory area is defined (constant).

- The **\<origin\>** specifies the beginning address of the memory area (constant).

- The **\<length\>** specifies the length of the memory area (constant).

- The **\<used_space\>** specifies the amount of allocated space in this area (constant).

- The **\<unused_space\>** specifies the amount of available space in this area (constant).

- The **\<attributes\>** lists the RWXI attributes that are associated with this area, if any (string).

- The **\<fill_value\>** specifies the fill value that is to be placed in unused space, if the fill directive is specified with the memory area (constant).

- The **\<usage_details\>** lists details of each allocated or available fragment in this memory area. If the fragment is allocated to a logical group, then a \<logical_group_ref\> element is provided to facilitate access to the details of that logical group. All fragment specifications include \<start_address\> and \<size\> elements.

    - The **\<allocated_space\>** element provides details of an allocated fragment within this memory area (container):

        * The **\<start_address\>** specifies the address of the fragment (constant).

        * The **\<size\>** specifies the size of the fragment (constant).

        * The **\<logical_group_ref\>** provides a reference to the logical group that is allocated to this fragment (reference).

- The **\<available_space\>** element provides details of an available fragment within this memory area (container):

    - The **\<start_address\>** specifies the address of the fragment (constant).

    - The **\<size\>** specifies the size of the fragment (constant).

Placement Map for the fl-4 Input File

```
<placement_map>
    <memory_area>
```
(continues on next page)

```
        <name>PMEM</name>
        <page_id>0x0</page_id>
        <origin>0x20</origin>
        <length>0x100000</length>
        <used_space>0xb240</used_space>
        <unused_space>0xf4dc0</unused_space>
        <attributes>RWXI</attributes>
        <usage_details>
            <allocated_space>
                <start_address>0x20</start_address>
                <size>0xb240</size>
                <logical_group_ref idref="lg-7"/>
            </allocated_space>
            <available_space>
                <start_address>0xb260</start_address>
                <size>0xf4dc0</size>
            </available_space>
        </usage_details>
    </memory_area>
    ...
</placement_map>
```

Far Call Trampoline List

The **<far_call_trampoline_list>** is a list of <far_call_trampoline> elements. The linker supports the generation of far call trampolines to help a call site reach a destination that is out of range. A far call trampoline function is guaranteed to reach the called function (callee) as it may utilize an indirect call to the called function.

The <far_call_trampoline_list> enumerates all of the far call trampolines that are generated by the linker for a particular link. The <far_call_trampoline_list> can contain any number of <far_call_trampoline> elements. Each **<far_call_trampoline>** is a container enclosing the following elements:

- The <callee_name> element names the destination function (string).

- The <callee_address> is the address of the called function (constant).

- The <trampoline_object_component_ref> is a reference to an object component that contains the definition of the trampoline function (reference).

- The <trampoline_address> is the address of the trampoline function (constant).

- The <caller_list> enumerates all call sites that utilize this trampoline to reach the called function (container).

- The <trampoline_call_site> provides the details of a trampoline call site (container) and consists of these items:

    - The <caller_address> specifies the call site address (constant).

    - The <caller_object_component_ref> is the object component where the call site resides (reference).

Fall Call Trampoline List for the fl-4 Input File

```
<far_call_trampoline_list>
    ...
    <far_call_trampoline>
        <callee_name>_foo</callee_name>
        <callee_address>0x08000030</callee_address>
        <trampoline_object_component_ref idref="oc-123"/>
        <trampoline_address>0x2020</trampoline_address>
        <caller_list>
            <call_site>
                <caller_address>0x1800</caller_address>
                <caller_object_component_ref idref="oc-23"/>
            </call_site>
            <call_site>
                <caller_address>0x1810</caller_address>
                <caller_object_component_ref idref="oc-23"/>
            </call_site>
        </caller_list>
    </far_call_trampoline>
    ...
</far_call_trampoline_list>
```

Symbol Table

The **<symbol_table>** contains a list of all of the global symbols that are included in the link. The list provides information about a symbol's name and value. In the future, the symbol_table list may provide type information, the object component in which the symbol is defined, storage class, etc.

The **<symbol>** is a container element that specifies the name and value of a symbol with these elements:

- The **<name>** element specifies the symbol name (string).

- The **<value>** element specifies the symbol value (constant).

- The **<local>** element specifies whether the symbol has local binding (string). While "false" is a valid value, the linker emits this element only if it is "true" for this symbol.

Symbol Table for the fl-4 Input File

```
<symbol_table>
    <symbol>
        <name>_c_int00</name>
        <value>0xaf80</value>
    </symbol>
    <symbol>
        <name>_main</name>
        <value>0xb1e0</value>
    </symbol>
    <symbol>
        <name>_printf</name>
        <value>0xac00</value>
        <local>true</local>
    </symbol>
    ...
</symbol_table>
```

3.11 Link Time Optimization - LTO

In the world of embedded systems, getting the maximum performance out of an application can be challenging, particularly when the available system memory is limited. A robust and effective C/C++ optimizing compiler can be an invaluable resource in helping achieve higher-level performance for embedded applications that must run within a limited memory space, especially if that compiler is able to perform inter-module optimizations on whole-programs.

The **tiarmclang** compiler tools, starting with version 2.1.0.LTS, enable inter-module optimization over a whole-program at link-time. This feature is commonly referred to as *Link-Time Optimization* or *LTO*.

Contents:

3.11.1 Benefits of Using LTO - Enabling Inter-Module Optimizations

A Simple Example

Consider a simple example application that demonstrates just one of the potential benefits of using LTO to enable inter-module optimization . . .

Suppose we have a series of source files in which many of the same string constants are referenced repeatedly and across multiple source files.

If we compile and link without LTO turned on:

```
%> tiarmclang -mcpu=cortex-m4 -Oz constant_merge_test.c ic_s10.c
↪ic_s20.c ic_s30.c ic_s40.c s10.c s20.c s30.c s40.c -o no_lto.
↪out -Wl,-llnk.cmd,-mno_lto.map
```

The linker generated map file, no_lto.map, reveals that the size of the .rodata section where all of
the string constants are defined is reasonably large:

```
...
SEGMENT ALLOCATION MAP

run origin  load origin   length    init length attrs members
----------  -----------  ----------  ----------- ----- -------
00000020    00000020     00007a4c    00007a4c     r-x
  00000020    00000020     00004ad2    00004ad2     r-- .rodata
  ...
...
```

But if we then compile with LTO enabled:

```
%> tiarmclang -mcpu=cortex-m4 -flto -Oz constant_merge_test.c ic_
↪s10.c ic_s20.c ic_s30.c ic_s40.c s10.c s20.c s30.c s40.c -o
↪with_lto.out -Wl,-llnk.cmd,-mwith_lto.map
```

Then the map file, with_lto.map, shows that the .rodata output section is significantly smaller in
the LTO-enabled build:

```
...
SEGMENT ALLOCATION MAP

run origin  load origin   length    init length attrs members
----------  -----------  ----------  ----------- ----- -------
00000020    00000020     00005b84    00005b84     r-x
  ...
  00004530    00004530     00001674    00001674     r-- .rodata
...
```

The use of LTO in this example enables the compiler to perform an inter-module constant merging
optimization that results in a savings of 0x4ad2 - 0x1674 -> 0x345e (13406) bytes in the .rodata
section. Note that in this example, the savings in the size of the .rodata section is offset somewhat
by increased code size in other sections like .text. The net savings is 0x7a4c - 0x5b84 -> 0x1ec8
(7880) bytes.

Code Size Reduction Due to Use of LTO

Significant code size savings can be realized by simply enabling the LTO feature in the build of an application. Comparisons between compiler-generated code size over a collection of Cortex-M0+, Cortex-M4, and Cortex-R5 example applications demonstrated that building these applications with LTO enabled resulted in significant code size reductions versus building the applications without LTO enabled.

Table 1: Code Size Reduction Due to Use of LTO

TI Arm Processor	% Code Size Reduction	Example Applications (count)
Cortex-M0+	23-25%	M0+ Driver Libraries (191)
Cortex-M4	6-11%	M4 SDK Examples (362)
Cortex-R5	4-8%	EEMBC AutoBench (15)

- % Code Size Reduction = (1-(LTO code size / non-LTO code size))*100

These code size measurements were taken over 568 Cortex-M0+, Cortex-M4, and Cortex-R5 example applications. All of these applications were compiled with the *-Oz* compiler option to prioritize code size reduction optimizations. The *-flto* compiler option was used to enable LTO.

Performance Improvement Due to Use of LTO

Enabling LTO during an application build can also provide significant speedup. With LTO enabled, example applications built on Cortex-M0+, Cortex-M4, and Cortex-R5 ran significantly faster than when the same applications were built without LTO enabled.

Table 2: Performance Improvement Due to Use of LTO

TI Arm Processor	Speedup Factor	% Reduction in Time/Cycles
Cortex-M0+	1.20	3.8%
Cortex-M4	1.12	4.4%
Cortex-R5	1.17	9.3%

- Speedup Factor = (non-LTO cycles/LTO cycles)

- % Reduction in Cycles = ((non-LTO cycles - LTO cycles)/non-LTO cycles)*100

The performance results shown here were derived from measurements over 15 EEMBC AutoBench applications running on Cortex-M0+, Cortex-M4, and Cortex-R5 hardware. For the Cortex-M0+ measurements, an application's execution time was calculated using a SysTick interrupt service routine. For the Cortex-M4 and COrtex-R5 measurements, a simple cycle count for each application's execution time was collected. All example applications were compiled with the *-O3* compiler option to prioritize performance improvement optimizations. The *-flto* compiler option was used to enable LTO.

Note: Increased Function Inlining

Using LTO may result in increased function inlining, which may improve performance as well as code size generally but may result in larger stack frames. This may require the user to either *increase the size of the of the stack* or else *prevent certain functions from being inlined* that are known to require large stack frames.

To debug this, it is recommended that users use the CCS Stack View to see a view of the static stack usage of each function in the application. See Stack Usage View in CCS for more information. Using the Stack Usage View requires that source code be built with *debug enabled*. This feature relies on the *–call_graph* capability provided by the *tiarmofd - Object File Display Utility*.

3.11.2 LTO Development Flow

How LTO Works

A key advantage to using LTO in an application build is that the linker is able to provide the compiler with the ability to optimize across C/C++ compilation unit boundaries. If LTO is *not* enabled during the build of an application, the compiler's visibility into the application's source code is limited to the C/C++ source file that is currently being compiled. Consequently, the compiler must make conservative assumptions about functions and variables that are referenced from the C/C++ source file, but are defined elsewhere. Hence, the compiler uses constraint with regards to what optimizations can be applied during the compilation of a given C/C++ source file.

When LTO is enabled, the linker combines *internal representation* (IR) modules from the incoming object files that were compiler-generated from C/C++ source files into a single, merged IR module representing all of the functions and variables from all of the C/C++ source files in an application. The compiler then has visibility across all C/C++ source files via this merged IR module and is able to apply inter-module optimizations, such as aggressive inlining, constant merging, and aggressive machine outlining. For example, if multiple source files require access to the same string constant, with LTO enabled, they can all access a single instance of storage for the string constant as opposed to each source file requiring their own copy of the string constant.

LTO Development Flow Overview

LTO is easy to incorporate into an application build. An overview of the LTO development flow is shown in figure 1 below. The LTO development flow can be divided into two phases:

Compilation Phase

- Compile C/C++ source files with the tiarmclang **-flto** compiler option

  The *-flto* option instructs the compiler to embed a bitcode encoding of the *internal representation* (IR) into each object file that is generated from a C/C++ source file by the compiler. A compiler-generated object that contains an bitcode IR encoding of a given C/C++ source module enables that module to participate in link-time optimization. A compiler-generated object file that does not contain a bitcode IR representation of a C/C++ source module is treated as an ordinary object file at link time. It does not participate in link-time optimization, but is incorporated into the traditional link step that occurs after link-time optimization and produces a linked ELF executable.

- Compile as much C/C++ source as possible with the **-flto** option

  Every C/C++ source file that is compiled with the *-flto* option is able to participate in link-time optimization. This includes C/C++ source files that are used to build libraries as well as an application's C/C++ source files whose compiler-generated object files are input directly into the linker. An application reaps a greater benefit from link-time optimization if a higher percentage of the C/C++ source files that are incorporated into the build of the application are compiled with the *-flto* option.

  The C/C++ source files for all runtime libraries that are provided with the tiarmclang installation are compiled with the *-flto* option and participate in link-time optimization when LTO is enabled during the link step of an application build.

Note: No additional compile-time is incurred when compiling with the *-flto* option

---

Terminology:
"IR" – refers to an *intermediate representation* of a C/C++ source file

Build user, runtime, TI, 3rd party libraries

Compile application sources
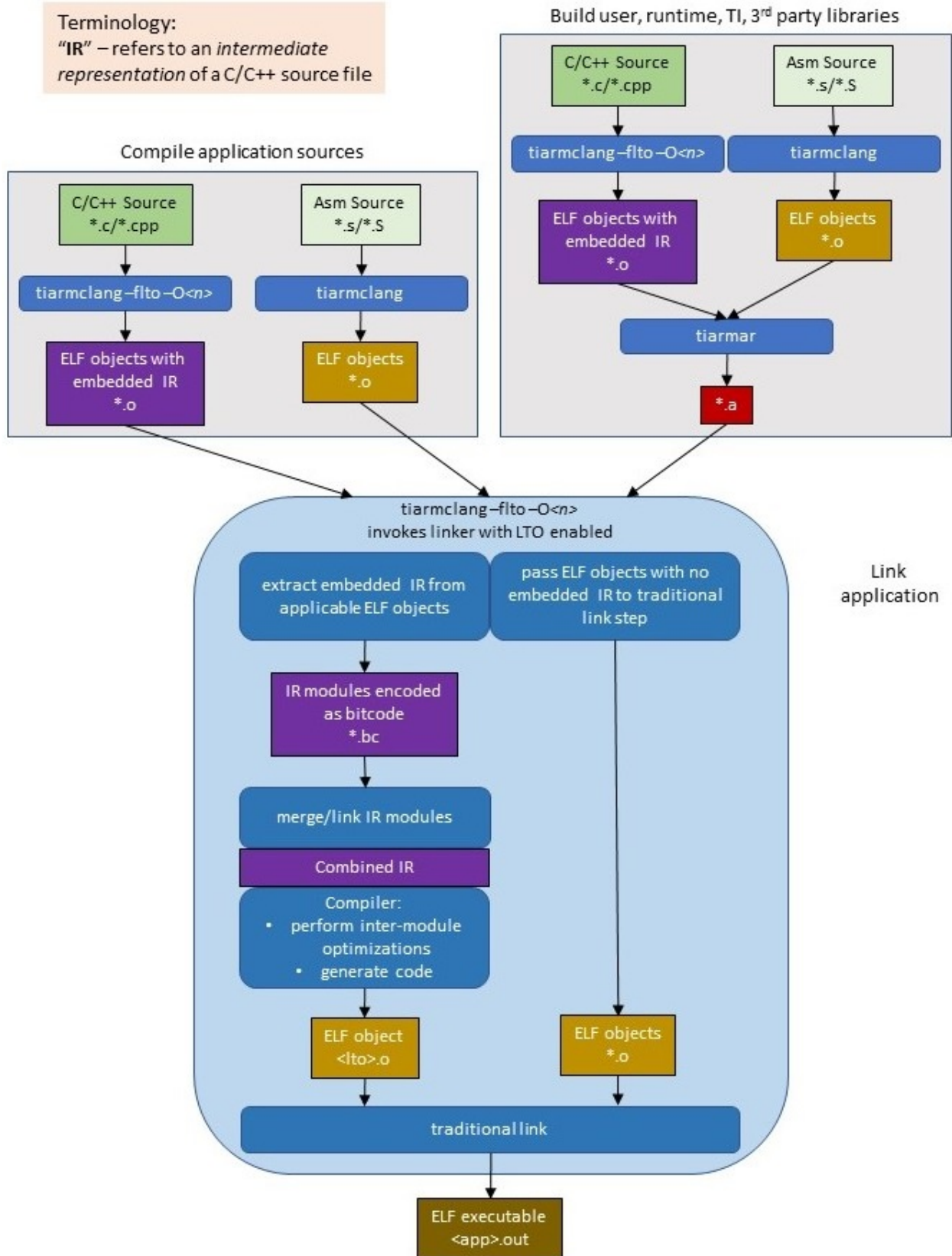
Figure 3.32: Link-time optimization's role in development flow

Link Phase

- Link incoming object files and object libraries with the tiarmclang **-flto** option

  LTO is enabled when the linker is invoked from tiarmclang with the *-flto* option. The linker is instructed to extract embedded bitcode IR from any object file that is submitted directly to the linker or pulled in from an object library to resolve a symbol reference.

  Extracted bitcode IR modules are merged / linked together to form a combined IR representation of the whole application.

  Object files that do not contain embedded bitcode IR are forwarded to the final link step as indicated in Figure 1 above.

  The linker invokes the compiler to perform inter-module optimizations on the combined IR, producing a single object file as a result of the link-time optimizations.

- The linker performs a traditional link step to generate an ELF executable

  The linker incorporates the single object file result of the inter-module optimizations with input object files that did not contain embedded bitcode IR in a traditional link to produce a linked ELF executable.

For more details about how to build your application with LTO enabled from a command-line interface or from within a tiarmclang CCS project, please see *Building an Application with LTO*.

3.11.3 Building an Application with LTO

Use the -flto Option to Enable LTO

The LTO feature can be enabled using the **-flto** option on the **tiarmclang** command line.

Building an LTO-Enabled Application from the Command-Line Interface (CLI)

- Compiling and Linking an Application from the CLI

  If compiling and linking from a single **tiarmclang** command, the *-flto* option can be inserted among the other compiler options. A typical **tiarmclang** command line that turns on the LTO feature looks like this:

```
%> tiarmclang -mcpu=cortex-m4 -Oz -flto hello.c -o hello.out␣
 ↪-Wl,lnk.cmd,-mhello.map
```

- Compiling and Linking an Application in Separate Steps with tiarmclang

  If compiling and linking in separate steps, the *-flto* option should be specified on both the tiarmclang compilation and linking commands, like so:

```
%> tiarmclang -mcpu=cortex-m4 -Oz -flto -c hello.c
%> tiarmclang -mcpu=cortex-m4 -Oz -flto hello.o -o hello.out␣
 ↪-Wl,-llnk.cmd,-mhello.map
```

Building an LTO-Enabled Application in a Code Composer Studio (CCS) Project

A tiarmclang CCS project that has been imported into or created in a workspace can be built with LTO enabled by checking the *Select Link-Time Optimization (LTO) (-flto)* box in the *Build->|family_name| Compiler->Optimization* tab in the project build settings dialog pop-up window. This capability is available starting in CCS 12.0.

For example, given a simple "Hello World!" CCS project as the project in focus in a workspace, you can click on *Project->Build Settings* to bring up the *Properties* pop-up window. Assuming that version 2.1.0.LTS or later of the TI Arm Clang Compiler Tools has been selected in the *General->Compiler version* and other settings besides *-flto* have been accounted for, then:

1. Choose *Build->* Arm *Compiler->Optimization*

2. Click to enable the check-box next to *Select Link-Time Optimization (LTO) (-flto)*
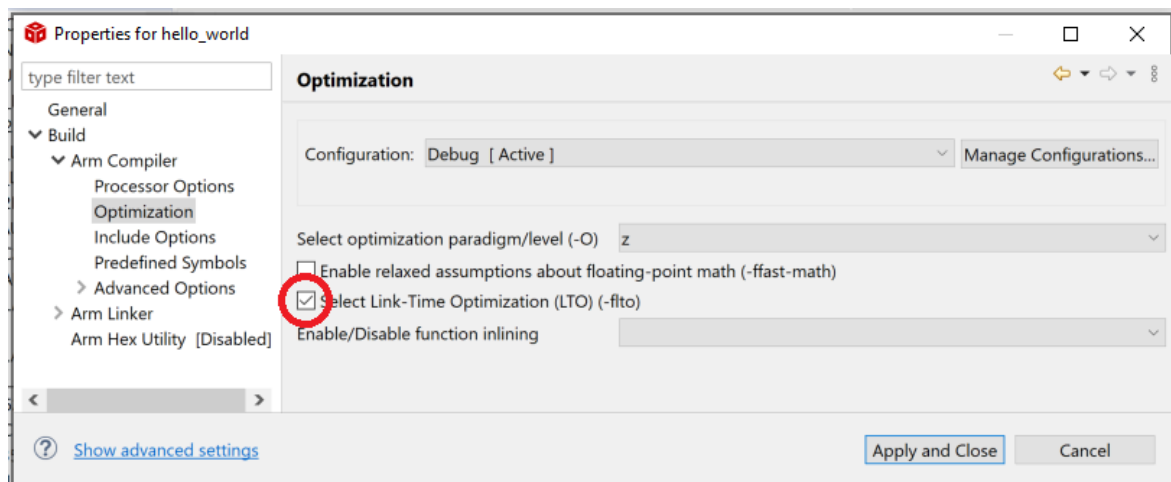
3. Click the *Apply and Close* button.

4. Build your project.

The *-flto* option is used in both the compile and link steps of the project build.

3.11. Link Time Optimization - LTO

If you are using a version of CCS prior to 12.0, then you can still enable LTO for the build of your application. Assuming the same "Hello World!" CCS project with all other settings accounted for, you can enable LTO by inserting the *-flto* option into both the *Build-> |family_name| Compiler* and *Build->|family_name| Linker* tabs in the *Project->Build Settings* pop-up window as follows:

1. Choose *Build-> |family_name| Compiler* and edit the *Command-line pattern* contents:
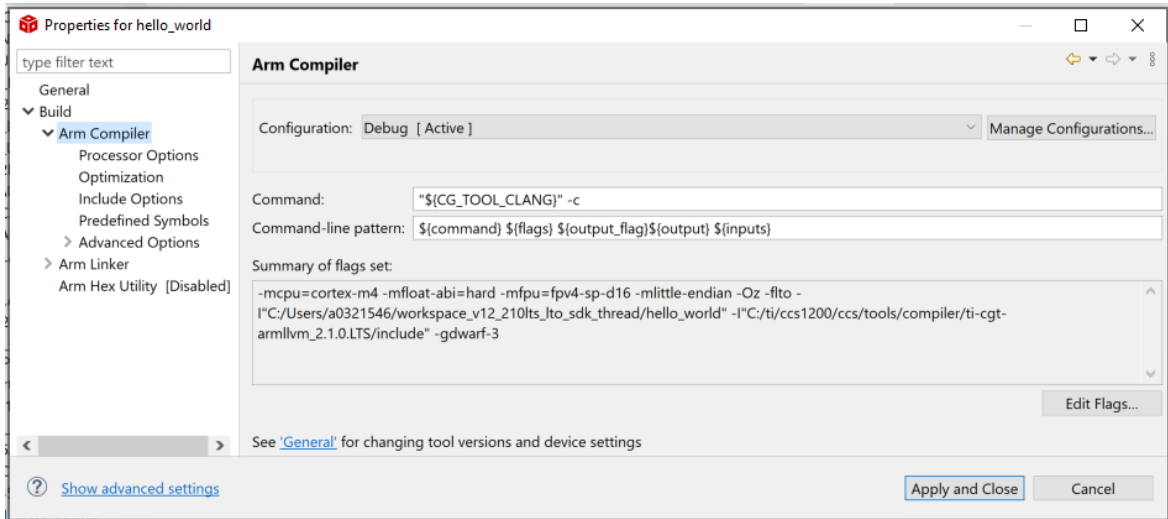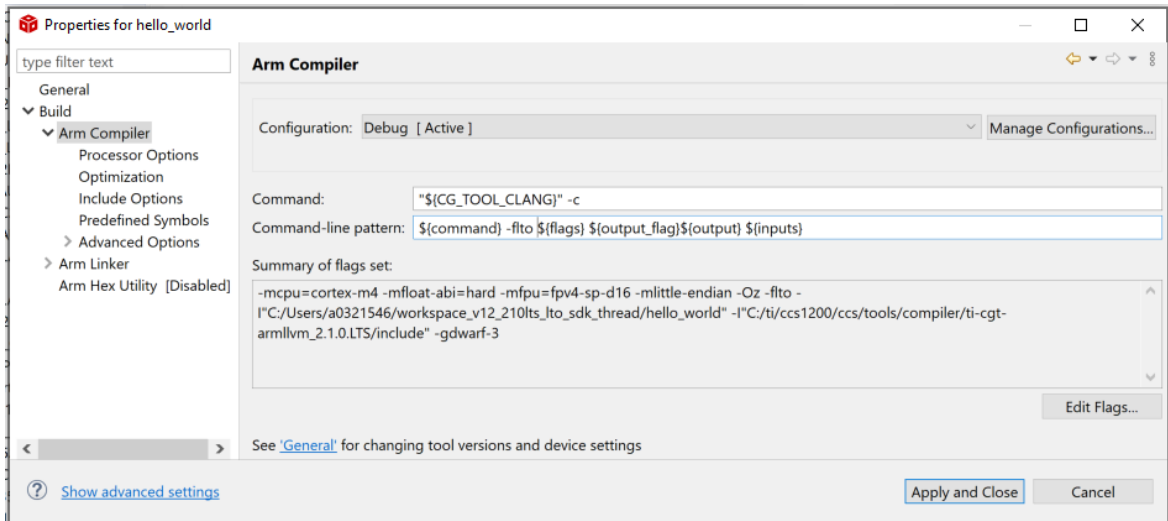
   Before:

   After:

2. Similarly, for the link step, click on *Build->|family_name| Linker* and edit the *Command-line pattern* contents:

   Before:
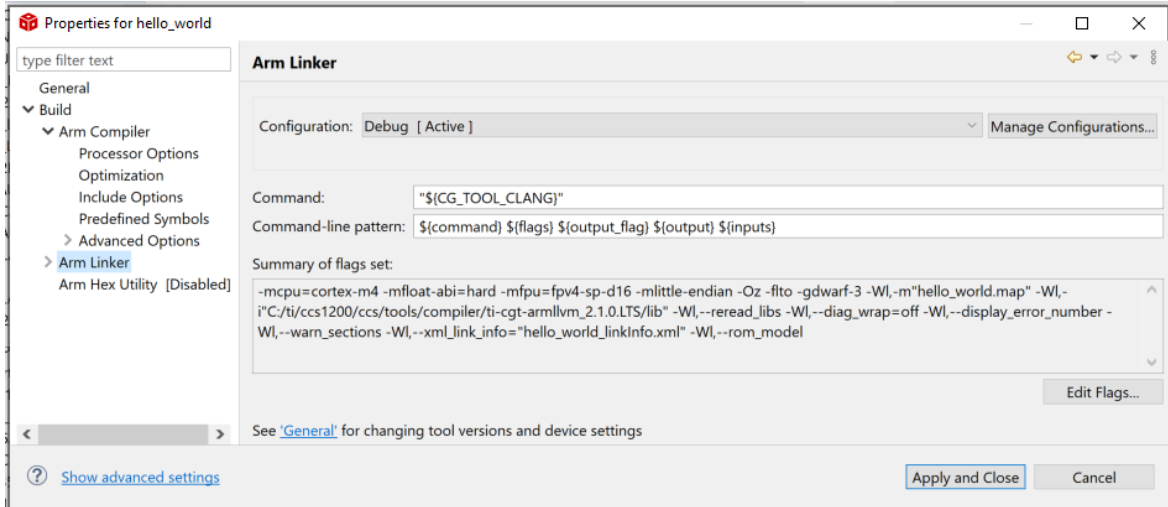
After:
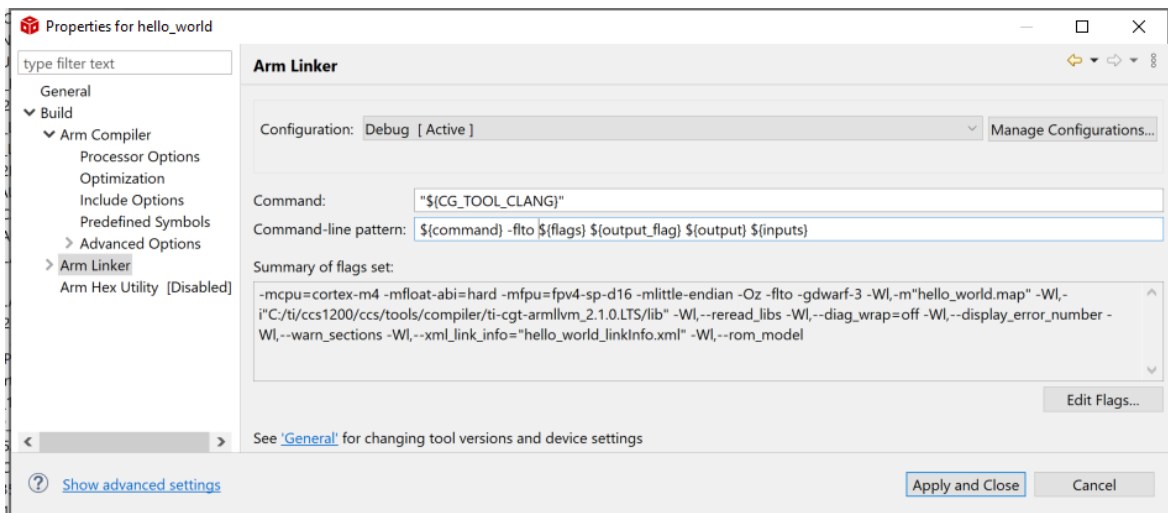
3. Click the *Apply and Close* button.

4. Build your project.

3.12 Code Coverage

Contents:

3.12.1 Source-Based Code Coverage in tiarmclang

The TI Arm Clang Compiler Tools (tiarmclang) support Source-Based Code Coverage that is particularly suited for embedded applications. In addition to being generally useful for thorough application development, code coverage is required by internal and external developers in the Industrial and Automotive markets for Functional Safety.

The following forms of Code Coverage are supported:

- **Function** coverage is the percentage of functions which have been executed at least once. A function is considered to be executed if any of its instantiations are executed.

    - **Instantiation** coverage is the percentage of function instantiations which have been executed at least once. Template functions and static inline functions from headers are two kinds of functions which may have multiple instantiations.

- **Line** coverage is the percentage of code lines which have been executed at least once. Only executable lines within function bodies are considered to be code lines.

- **Region** coverage is the percentage of code regions which have been executed at least once. A code region may span multiple lines (e.g in a large function body with no control flow). However, it is also possible for a single line to contain multiple code regions (e.g in "return x || y && z"). *Region* coverage is equivalent to *Statement* coverage provided by other vendors.

    - **Call Region** coverage is the percentage of code regions *containing function calls* which have been executed at least once. A code region may contain multiple function calls when there is no control flow. This metric is a subset of Region coverage. *Call Region* coverage is equivalent to *Call* coverage provided by other vendors.

- **Branch** coverage is the percentage of source-condition-based branches that have been taken at least once. The new tiarmclang tools' support for *Branch* coverage (also known as *Branch Condition* coverage) provides a finer level of coverage than that which is provided by other vendors, allowing users to track "True/False" execution coverage across *leaf-level Boolean expressions* used in conditional statements. This makes it much more informative and useful than *Decision* coverage that some other vendors support, which only tracks execution counts for a single control flow decision point, which may be a Boolean expression comprised of conditions and zero or more Boolean logical operators.

- **Modified Condition/Decision Coverage** (MC/DC) is the percentage of all single condition outcomes that independently affect a decision outcome that have been exercised in the control flow. MC/DC builds on top of branch coverage, and as such, it too requires that all code blocks and all execution paths have been tested. MC/DC pertains to complex Boolean

expressions involving more than one single condition where each condition has been shown to affect that decision outcome independently.

Note: Coverage across Function Instantiations

If a function has multiple instantiations, as in the case of C++ function templates, the instantiation reflecting the *maximum* coverage of lines, regions, or branches is used for the final coverage tally. In other words, a function definition is considered fully covered if any one of its instantiations is fully covered with respect to lines, regions, or branches.

Support for Embedded Use Cases

The tiarmclang compiler tools minimize the data size requirements of Code Coverage by allocating memory space for *only* the counters and keeping all other coverage related information in non-allocatable sections preserved in the object file itself. This ensures that target memory is only utilized for incrementing counters. In addition, the runtime support only supports writing counters to a file as part of a "bare-metal" profiling model and nothing else. Support for writing a full raw profile file, merging counters, etc., is not included as part of tiarmclang.

Note that instrumentation that is inserted to track the counters will introduce cycle performance and codesize overhead, depending on the size of the program. This is due to the additional instructions needed, number of counters needed, and impact to existing code optimization. Reducing the size of counters will be addressed as a future enhancement for the compiler to decrease the memory footprint introduced by code coverage to better mitigate the codesize overhead.

Effects of Code Optimization

The tiarmclang compiler derives instruction-to-source mappings through the Abstract Syntax Trees during the compilation's Code Generation (CodeGen) phase that are eventually lowered to an intermediate representation, which is where counter instrumentation occurs. Counter instrumentation is completed prior to optimization passes that operate on the intermediate representation, so this means that coverage data is very accurate with respect to the source code. Counter increments that would have occurred in an unoptimized program occur in the optimized variant. For example, counter mapping regions for an inlined function are created with instrumentation prior to inlining. If inlining is performed, the instrumentation is inlined along with it. The resulting execution counts map back to the original source as though the function had never been inlined.

While counter instrumentation is not obstructed by optimization, the presence of counter instrumentation may inhibit certain optimizations

Tool Usage

In addition to the tiarmclang compiler, which is used to produce counter instrumentation, the tools used to produce and visualize code coverage data are **tiarmprofdata** and **tiarmcov**.

Useful Coverage Profile Merging Options (tiarmprofdata)

```
USAGE: tiarmprofdata merge [options] <filename ...>
```

Format of output profile

- **--binary** - binary encoding (default)
- **--text** - output in text mode

Profile kind

- **--instr** - instrumentation profile (default)
- **--obj-file=<string>** - object file
- **--output=<file>** - output file
- **--remapping-file=<file>** - symbol remapping file
- **--sparse** - generate a sparse profile (only meaningful for --instr)

Useful Coverage Visualization Options (tiarmcov)

```
USAGE: tiarmcov {subcommand} [OPTION]...
```

Subcommands

- **export** - Export instrprof file to structured format either as text (JSON) or as LCOV.
    - JSON: tiarmcov export --format=text
    - LCOV: tiarmcov export --format=lcov
    - CSV: tiarmcov export --format=csv
- **report** - Summarize instrprof style coverage information.
- **show** - Annotate source files using instrprof style coverage.

Function Filtering Options

- **--filename-allowlist=<file>** - Show code coverage only for files that match a regular expression listed in the given file.

- **--ignore-filename-regex=<string>** - Skip source code files with file paths that match the given regular expression.

- **--line-coverage-gt=<number>** - Show code coverage only for functions with line coverage greater than the given threshold.

- **--line-coverage-lt=<number>** - Show code coverage only for functions with line coverage less than the given threshold.

- **--name=<string>** - Show code coverage only for functions with the given name.

- **--name-regex=<string>** - Show code coverage only for functions that match the given regular expression.

- **--name-allowlist=<file>** - Show code coverage only for functions listed in the given file.

- **--region-coverage-gt=<number>** - Show code coverage only for functions with region coverage greater than the given threshold.

- **--region-coverage-lt=<number>** - Show code coverage only for functions with region coverage less than the given threshold.

General Options

- **--instr-profile=<string>** - File with the profile data obtained after an instrumented run.

- **--num-threads=<uint>** - Number of merge threads to use (default: autodetect).

- **--object=<string>** - Coverage executable or object file.

- **--output-dir=<string>** - Directory in which coverage information is written out.

- **--path-equivalence=<string>** - <from>,<to> Map coverage data paths to local source file paths.

- **--project-title=<string>** - Set project title for the coverage report.

- **--show-mcdc-summary** - Show MC/DC condition statistics in summary table. Data will only appear if code was compiled with the *-fmcdc* option.

- **--show-branch-summary** - Show branch condition statistics in summary table.

- **--show-instantiation-summary** - Show instantiation statistics in summary table.

- **--show-region-summary** - Show region statistics in summary table.

- **--show-call-region-summary** - Show call region statistics in summary table.

- **--summary-only** - Export only summary information for each source file.

Source-Based Viewing Options (for "tiarmcov show")

- **--show-mcdc** - Show coverage for MC/DC conditions in each Boolean expression. Data will only appear if code was compiled with the *-fmcdc* option.

- **--show-branches=<value>** - Show coverage for branch conditions.

- – =*count* - show True/False counts

- – =*percent* - show True/False percent

- **--show-expansions** - Show expanded source regions.

- **--show-instantiations** - Show function instantiations.

- **--show-branches=<value>** - Show coverage for branch conditions.

- **--show-line-counts-or-regions** - Show the execution counts for each line, or the execution counts for each region on lines that have multiple regions.

- **--show-regions** - Show the execution counts for each region.

Generating Instrumented Binaries

Source code must be built using **tiarmclang** with *-fprofile-instr-generate -fcoverage-mapping* options. The *-fmcdc* option may be used if measuring MC/DC is desired (if MC/DC-level coverage is not desired, please do not use *-fmcdc* in order to minimize the level of instrumentation required). For example:

```
tiarmclang -fprofile-instr-generate -fcoverage-mapping {-fmcdc}
 ↪foo.cc -o foo
```

Note: Instrumented binaries comprised of object files instrumented using version 1.3.x.LTS of the compiler tools aren't supported

Due to format changes added after version 1.3.x.LTS of the compiler tools, instrumented binaries that include object files instrumented with version 1.3.x.LTS of the compiler should not be linked with binaries built using version 2.1.x.LTS (or later) of the compiler tools.

The following options are available to help reduce the size of the instrumentation code and data footprint that is added to an application build to enable computation and visualization of code coverage information.

Reduce Size of Profile Counter

```
-fprofile-counter-size=[64|32]
```

The default size for the compiler generated profile counters that annotate an application when code coverage is enabled is 64-bits. using the option with *-fprofile-counter-size=32* instructs the compiler to use 32-bit integer values to record the execution count associated with a basic block (a sequence of executable code that can potentially be the destination of a call or branch) where applicable.

Limit Generation of Code Coverage Information to Functions

```
-ffunction-coverage-only
```

Normally when compiler generated code coverage is enabled in tiarmclang, the compiler will annotate an application with execution counters for basic blocks. This option can be used to reduce the code coverage instrumentation footprint by limiting compiler generated code coverage information to function entry execution counts.

Use Profile Function Groups to Limit Coverage Overhead

```
-fprofile-function-groups=N
-fprofile-selected-function-group=i
```

Reduce instrumented size overhead by spreading the overhead across 'N' total executable builds, where 'i' refers to an individual executable build between '0' and 'N-1'. Raw profiles from different groups can be merged as described below: *Merging Multiple Indexed Profile Data Files from Multiple Executables*.

Retrieving the Counters From Memory

Once the executable has been loaded and executed one or more times, the counters should be retrieved from memory and written to a raw profile data file on the host. Counters are stored in an allocated memory section named **__llvm_prf_cnts**, and this section is demarcated with the start and stop symbols, **__start___llvm_prf_cnts** and **__stop___llvm_prf_cnts**, which can allow the target memory to be read from the host.

The data retrieved in memory should be saved to a file, and this file is the *raw profile counter file*.

If MC/DC-level coverage is enabled using *-fmcdc* at compile-time, additional coverage data is stored in an allocated memory section named **__llvm_prf_bits**, and this section is demarcated with the start and stop symbols, **__start___llvm_prf_bits** and **__stop___llvm_prf_bits**. This data must be saved in the same *raw profile counter file* immediately following the counter data. *This data must be read as bytes!*

Note: It is critically important that these sections used to track coverage counters (**__llvm_prf_cnts** and **__llvm_prf_bits**) be placed in memory that is writable during runtime ("RAM" instead of "FLASH"). By default, the linker will attempt to place the sections next to the **.bss** section, but users may also manually place the sections using a *linker command file*.

Retrieving Counters Using the CCS Scripting Console

Retrieving counters from memory can be done in Code Composer Studio (CCS) using the following example script, which can be pasted into the CCS scripting console:

```
1  var scriptEnv = Packages.com.ti.ccstudio.scripting.environment.
   ↪ScriptingEnvironment.instance();
2  var server = scriptEnv.getServer("DebugServer.1");
3  var session = server.openSession("Texas Instruments XDS110 USB
   ↪DebugProbe_0/CORTEX_M4_0");
4
5  var cntStart = session.symbol.getAddress("__start___llvm_prf_cnts
   ↪");
6  var cntStop = session.symbol.getAddress("__stop___llvm_prf_cnts
   ↪");
7
8  var cntContent = session.memory.readData(0, cntStart, 8, cntStop
   ↪- cntStart);
9
10 var executable = session.symbol.getSymbolFileName();
11 var outFile = new Packages.java.io.RandomAccessFile(executable +
   ↪".cnt" , "rw");
12
13 outFile.setLength(0);
14 for each (var val in cntContent) {
15     outFile.writeByte(Number(val));
16 }
17
18 var mcdcStart = session.symbol.getAddress("__start___llvm_prf_
   ↪bits");
19 var mcdcStop = session.symbol.getAddress("__stop___llvm_prf_bits
   ↪");
20
21 var mcdcContent = session.memory.readData(0, mcdcStart, 8,
   ↪mcdcStop - mcdcStart);
22 for each (var val in mcdcContent) {
23     outFile.writeByte(Number(val));
24 }
25 outFile.close();
```

This example script produces a *raw profile counter file* named after the executable using the ".cnt" suffix.

Retrieving Counters Using Compiler Runtime Support

Alternatively, the counter data can also be retrieved from memory using a function that is provided as part of the compiler runtime support, **__llvm_profile_write_file()**. This function writes the counters from the target to the host using runtime routines (**fwrite()**). Any other means of downloading the data may also be used. This produces a *raw profile counter file* using the default filename **default.profraw**.

```c
int test_main(int argc, const char *argv[]) {
  // Call into an important routine
  important_func1();

  // Call into an important routine
  important_func2();

  // Write out counter details to file
  __llvm_profile_write_file();

  // Exit
  return 0;
                 }
```

Processing the Raw Profile Counter Data Into an Indexed Profile Data File

An *indexed profile data file* should be produced for each executable that is run; it is produced based on a *raw profile counter file* that has the runtime counter data retrieved from memory (see *Retrieving the Counters From Memory* section above).

This is done by invoking the **tiarmprofdata** utility and indicating the *raw profile counter file* as well as the executable used to produce it. This is required since in order to support embedded use cases, pertinent code coverage information must be extracted from non-allocatable sections in the executable. The result is an *indexed profile data file*. In the example below, the *raw profile counter files* used as input are **app1.profcnts**, **app2.profcnts**, and **app3.profcnts**. The resulting *indexed profile data file* produced for each is **app1.profdata**, **app2.profdata**, and **app3.profdata**, respectively.

```
tiarmprofdata merge -sparse -obj-file=app1.out app1.profcnts -o
 →app1.profdata
tiarmprofdata merge -sparse -obj-file=app2.out app2.profcnts -o
 →app2.profdata
tiarmprofdata merge -sparse -obj-file=app3.out app3.profcnts -o
 →app3.profdata
```

Merging Multiple Indexed Profile Data Files from Multiple Executables

An *indexed profile data file* for each executable must be produced before any profile data from multiple executables can be merged. If multiple executables have been run based on the same source code base, the corresponding *indexed profile data files* for each of the executables can then be merged into a single *indexed profile data file*.

```
tiarmprofdata merge -sparse app1.profdata app2.profdata app3.
↪profdata -o app_merged.profdata
```

Wildcards can be used to identify the range of *indexed profile data files* used as input.

Visualization

In order to visualize the code coverage, the *single merged indexed profile data file* along with each of the corresponding executables must be given as input to the tiarmcov visualization tool. The visualization tool can be used to generate a dump of the source file along with a summary report in either HTML or Text format. The names of each executable must be specified individually by name using the *--object=<executable>* option.

HTML Format

When generating HTML output, a summary coverage report is also generated at the root of a directory tree that contains coverage data for each of the files. For the source-based coverage views, it is recommended to use *--show-expansions* and *--show-instantiations* options to see the full view of all macro expansions and function template instantiations, respectively. In addition, branch coverage information can be included in the source-based view, and it can be represented in terms of execution count or percentage.

The following example visualizes coverage in HTML with macros and templates expanded; it also includes detailed branch coverage in terms of execution count.

```
tiarmcov show --format=html --show-expansions --show-
↪instantiations --show-branches=count
   --object=./app1.out --object=./app2.out --object=./app3.out -
↪instr-profile=app-merged.profdata
   --output-dir=/example/directory
```

Coverage Report

Created: 2020-06-11 09:23

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
scratch/aphipps/llvmtest/cov/demo/demo.c	100.00% (1/1)	96.36% (53/55)	85.71% (24/28)	73.33% (22/30)
Totals	100.00% (1/1)	96.36% (53/55)	85.71% (24/28)	73.33% (22/30)

Generated by llvm-cov -- llvm version 11.0.0git

1/1

Coverage Report

Created: 2020-06-11 09:23

/scratch/aphipps/llvmtest/cov/demo/demo.c

Line	Count	Source ([jump to first uncovered line](#))
1		
2		`#include <stdio.h>`
3		`#include <stdlib.h>`
4		
5		`#ifdef __clang__`
6		`extern void __llvm_profile_write_file(void);`
7		`#endif`
8		
9	2	`#define BRANCH_MACRO(x, y) (x == y)`
10		
11		`int main(int argc, char *argv[])`
12	3	`{`
13	3	`    if (argc == 1)`

> Branch ([13:9](#)): [True: 1, False: 2]

Line	Count	Source
14	1	`    {`
15	1	`#ifdef __clang__`
16	1	`        __llvm_profile_write_file();`
17	1	`#endif`
18	1	`        return 0;`
19	1	`    }`
20	2	
21	2	`    int arg1 = atoi(argv[1]);`
22	2	`    int arg2 = atoi(argv[2]);`
23	2	`    int cnt  = atoi(argv[3]);`
24	2	
25	2	`    int x = arg2 == 0 || arg1 == 0;`

> Branch ([25:13](#)): [True: 0, False: 2]
> Branch ([25:26](#)): [True: 1, False: 1]

Line	Count	Source
26	2	
27	2	`    printf("Hello, World! %u\n", x);`
28	2	
29	2	`    int i;`
30	22	`    for (i = 0; i < cnt; i++)`

```
32      20              if (arg1 == 0 || arg2 == 2 || arg2 == 34)
        ┌─────────────────────────────────────────────────────┐
        │ Branch (32:13): [True: 10, False: 10]               │
        │ Branch (32:26): [True: 10, False: 0]                │
        │ Branch (32:39): [True: 0, False: 0]                 │
        └─────────────────────────────────────────────────────┘
33      20          {
34      20              printf("Hello from the loop!\n");
35      20          }
36      20      }
37       2
38       2      if ((arg1 == 3) && 1)
        ┌─────────────────────────────────────────────────────┐
        │ Branch (38:9): [True: 0, False: 2]                  │
        │ Branch (38:24): [Folded - Ignored]                  │
        └─────────────────────────────────────────────────────┘
39       0          printf("This never executes\n");
40       2
41       2      if (BRANCH_MACRO(arg1, arg1))
        ┌──────┬───────┬────────────────────────────────────────┐
        │ Line │ Count │ Source                                 │
        │  9   │   2   │ #define BRANCH_MACRO(x, y) (x == y)    │
        │      │       │   Branch (9:28): [True: 2, False: 0]   │
        └──────┴───────┴────────────────────────────────────────┘
42       2          printf("This executes on a macro expansion\n");
43       2
44       2      // Explicit Default Case
45       2      switch (arg2) {
46       1          case 1: printf("Case 1\n");
        ┌─────────────────────────────────────────────────────┐
        │ Branch (46:7): [True: 1, False: 1]                  │
        └─────────────────────────────────────────────────────┘
47       1                  break;
48       1          case 2: printf("Case 2\n");
        ┌─────────────────────────────────────────────────────┐
        │ Branch (48:7): [True: 1, False: 1]                  │
        └─────────────────────────────────────────────────────┘
49       1                  break;
50       0          default: break;
        ┌─────────────────────────────────────────────────────┐
        │ Branch (50:7): [True: 0, False: 2]                  │
        └─────────────────────────────────────────────────────┘
51       2      }
52       2
53       2      // Implicit Default Case
54       2      switch (arg2) {
        ┌─────────────────────────────────────────────────────┐
        │ Branch (54:13): [True: 0, False: 2]                 │
        └─────────────────────────────────────────────────────┘
55       1          case 1: printf("Case 1\n");
        ┌─────────────────────────────────────────────────────┐
        │ Branch (55:7): [True: 1, False: 1]                  │
        └─────────────────────────────────────────────────────┘
56       1                  break;
57       1          case 2: printf("Case 2\n");
        ┌─────────────────────────────────────────────────────┐
        │ Branch (57:7): [True: 1, False: 1]                  │
        └─────────────────────────────────────────────────────┘
58       1                  break;
```

3.12. Code Coverage   **656**

7/6/2020

```
59    2        }
60    2
61    2    #ifdef __clang__
62    2        __llvm_profile_write_file();
63    2    #endif
64    2
65    2        return 0;
66    2    }
```

3/3

Figure 3.33: Code Coverage HTML Output Format

When generating HTML output with *--show-mcdc-summary*, the summary coverage report includes an additional column with the MC/DC coverage data.

Coverage Report

Created: 2022-05-09 16:46

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage	MC/DC
scratch/aphipps/llvmtest/cov/demo/demo.c	100.00% (1/1)	95.35% (41/43)	85.71% (24/28)	73.33% (22/30)	16.67% (1/6)
Totals	100.00% (1/1)	95.35% (41/43)	85.71% (24/28)	73.33% (22/30)	16.67% (1/6)

Generated by llvm-cov -- llvm version 15.0.0git

Figure 3.34: Code Coverage HTML Summary Report

Text Format

When generating Text output, the summary coverage report is generated using a separate **tiarmcov report** option. For example, to view the source-based coverage view:

```
tiarmcov show --show-expansions --show-branches=count --object=./
↪app1.out --object=./app2.out --object=./app3.out -instr-
↪profile=app-merged.profdata
```

```
Text Format Output Example

    2|        |
```

(continues on next page)

```
 2|         |#include <stdio.h>
 3|         |#include <stdlib.h>
 4|         |
 5|         |#ifdef __clang__
 6|         |extern void __llvm_profile_write_file(void);
 7|         |#endif
 8|         |
 9|       2|#define BRANCH_MACRO(x, y) (x == y)
10|         |
11|         |int main(int argc, char *argv[])
12|       3|{
13|       3|    if (argc == 1)
------------------
|  Branch (13:9): [True: 1, False: 2]
------------------
14|       1|    {
15|       1|#ifdef __clang__
16|       1|        __llvm_profile_write_file();
17|       1|#endif
18|       1|        return 0;
19|       1|    }
20|       2|
21|       2|    int arg1 = atoi(argv[1]);
22|       2|    int arg2 = atoi(argv[2]);
23|       2|    int cnt  = atoi(argv[3]);
24|       2|
25|       2|    int x = arg2 == 0 || arg1 == 0;
------------------
|  Branch (25:13): [True: 0, False: 2]
|  Branch (25:26): [True: 1, False: 1]
------------------
26|       2|
27|       2|    printf("Hello, World! %u\n", x);
28|       2|
29|       2|    int i;
30|      22|    for (i = 0; i < cnt; i++)
------------------
|  Branch (30:17): [True: 20, False: 2]
------------------
31|      20|    {
32|      20|        if (arg1 == 0 || arg2 == 2 || arg2 == 34)
------------------
```

3.12. Code Coverage 658

(continued from previous page)

```
|  Branch (32:13): [True: 10, False: 10]
|  Branch (32:26): [True: 10, False: 0]
|  Branch (32:39): [True: 0, False: 0]
------------------
33|     20|         {
34|     20|             printf("Hello from the loop!\n");
35|     20|         }
36|     20|     }
37|      2|
38|      2|     if ((arg1 == 3) && 1)
------------------
|  Branch (38:9): [True: 0, False: 2]
|  Branch (38:24): [Folded - Ignored]
------------------
39|      0|         printf("This never executes\n");
40|      2|
41|      2|     if (BRANCH_MACRO(arg1, arg1))
------------------
|  |     9|      2|#define BRANCH_MACRO(x, y) (x == y)
|  |  ------------------
|  |  |  Branch (9:28): [True: 2, False: 0]
|  |  ------------------
------------------
42|      2|         printf("This executes on a macro expansion\n
↪");
43|      2|
44|      2|     // Explicit Default Case
45|      2|     switch (arg2) {
46|      1|       case 1: printf("Case 1\n");
------------------
|  Branch (46:7): [True: 1, False: 1]
------------------
47|      1|           break;
48|      1|       case 2: printf("Case 2\n");
------------------
|  Branch (48:7): [True: 1, False: 1]
------------------
49|      1|           break;
50|      0|     default: break;
------------------
|  Branch (50:7): [True: 0, False: 2]
------------------
```

(continues on next page)

```
   51|      2|    }
   52|      2|
   53|      2|    // Implicit Default Case
   54|      2|    switch (arg2) {
  ------------------
  | Branch (54:13): [True: 0, False: 2]
  ------------------
   55|      1|        case 1: printf("Case 1\n");
  ------------------
  | Branch (55:7): [True: 1, False: 1]
  ------------------
   56|      1|                break;
   57|      1|        case 2: printf("Case 2\n");
  ------------------
  | Branch (57:7): [True: 1, False: 1]
  ------------------
   58|      1|                break;
   59|      2|    }
   60|      2|
   61|      2|#ifdef __clang__
   62|      2|    __llvm_profile_write_file();
   63|      2|#endif
   64|      2|
   65|      2|    return 0;
   66|      2|}
```

To view the report:

```
tiarmcov report --object=./app1.out --object=./app2.out --
↪object=./app3.out -instr-profile=app-merged.profdata
```

```
File '/scratch/aphipps/llvmtest/cov/demo/demo.c':
Name                            Regions   Miss   Cover      Lines   ␣
↪Miss   Cover  Branches    Miss   Cover
-----------------------------------------------------------------
↪-------------------------------------
main                                28      4  85.71%         55   ␣
↪    2  96.36%        30       8  73.33%
-----------------------------------------------------------------
↪-------------------------------------
TOTAL                               28      4  85.71%         55   ␣
↪    2  96.36%        30       8  73.33%
```

The overall MC/DC coverage percentage is also shown as part of the report if *--show-mcdc-summary* is used when generating it:

```
tiarmcov report --show-mcdc-summary --object=./app1.out --
↪object=./app2.out --object=./app3.out -instr-profile=app-
↪merged.profdata
```

```
File '/scratch/aphipps/llvmtest/cov/demo/demo.c':
Name                             Regions    Miss    Cover      Lines    ␣
↪Miss    Cover  Branches    Miss    Cover    MC/DC Conditions    ␣
↪Miss    Cover
------------------------------------------------------------------
↪------------------------------------------------------------------
↪-----------
main                                28      4  85.71%         55    ␣
↪   2  96.36%         30      8  73.33%                       6    ␣
↪   5  16.67%
------------------------------------------------------------------
↪------------------------------------------------------------------
↪-----------
TOTAL                               28      4  85.71%         55    ␣
↪   2  96.36%         30      8  73.33%                       6    ␣
↪   5  16.67%
```

Visualization as TI CSV for Excel

The visualization capability supports exporting the data as a TI-specific CSV format that contains an aggregation of data corresponding to the TI-specified Dynamic Analysis Guidelines.

```
tiarmcov export --format=csv ./multidemo.out -instr-
↪profile=default.profdata c-main.c c-gen1.c c-gen2.c c-gen3.c c-
↪gen4.c c-gen5.c > multidemo.csv
```

The output can be imported and saved as an Excel spreadsheet where it can be manually adjusted:

	A	B	C	D	E	F	G	H	I	J	K	L
1	Coverage Summary											
2	Product Name											
3	Product Version											
4	Applicable Platforms / SoC / EVMs											
5	Tool Used (with Version no)											
6												
7	Total number of files	6										
8	Total number of functions	12										
9	Number of functions NOT covered	0										
10	Percentage of functions covered	100.00%										
11	Total number of Calls	1										
12	Number of Calls NOT covered	0										
13	Percentage of Calls covered	100.00%										
14												
15												
16	Note: Rather than track each individual statement or function call, Clang/LLVM-based Code Coverage measures the execution of code regions that are created to match a function's control flow. This facilitates more efficient counter allocation and instrumentation but with the same level of accuracy. A single region may include multiple statements or call sites. Therefore, if a region is executed, then it is known that every statement or function call in that region is also executed.											
17												
18		Statement Coverage	Branch/Decision Coverage	MC/DC Coverage	Function Coverage	Call Coverage						
19	Expected Coverage											
20	Average Coverage Achieved	88.38%	77.53%	80.00%	100.00%	100.00%						
21												
22												
23	Coverage File List											
24	S No.	File Name	Total Statements	Executed Statements	Total Branch/Decisions	Executed Branch/Decisions	Total MC/DC	Executed MC/DC	Total Functions	Executed Functions	Total Calls	Executed Calls
25	1	c-main.c	5	5	2	2	0	0	2	2	1	1
26	2	c-gen1.c	32	32	22	20	2	2	2	2	0	0
27	3	c-gen2.c	59	43	44	28	0	0	2	2	0	0
28	4	c-gen3.c	52	47	62	46	0	0	2	2	0	0
29	5	c-gen4.c	35	35	38	34	18	14	2	2	0	0
30	6	c-gen5.c	15	13	10	8	0	0	2	2	0	0
31												

This corresponds to the TI-specified Coverage Report Excel Template (Dynamic Analysis Summary) produced by LDRA:

Dynamic Analysis Summary

Product Name	
Product Version	
Applicable Platforms / SoC / EVMs	
Tool Used (with Version no)	LDRA tool suite v10.1.1

Total number of files	
Total number of functions	
Number of functions NOT covered	
Percentage of functions covered	
Total number of Calls	
Number of Calls NOT covered	
Percentage of Calls covered	

	Statement Coverage	Branch/Decision Coverage	MC/DC Coverage	Function Coverage	Call Coverage
Expected Coverage					
Average Coverage Achieved					

Because a CSV is generated, all of the metric categories are produced on the same sheet. This is different from the TI Excel template which separates data out across multiple sheets.

Note: When exporting to TI-CSV with multiple binaries using *--object*, one of the binaries must be specified without the *--object* option. E.g.:

```
tiarmcov export --format=csv ./app1.out --object=./app2.out --
→object=./app3.out -instr-profile=app-merged.profdata
```

This behavior will be made consistent in a future release.

Important Considerations for MC/DC

MC/DC is a *new feature* added to Source-Based Code Coverage supported by the TI Arm Clang Compiler.

- Boolean expressions with **more than six** individual conditions are not supported and will result in a compilation error when *-fmcdc* is used. This restriction is intended to keep the instrumentation footprint optimal.

- Boolean expressions that **include statements with additional nested Boolean expressions** are not supported and will result in a compilation error when *-fmcdc* is used. For example, the Boolean expression "(x > 3) && my_func((y > 3) || (y < 10))" contains a nested Boolean expression as an argument to the function call to my_func() and is not supported.

- Boolean expressions containing **strongly-coupled conditions** are not handled in a special manner. Each strongly-coupled condition is treated as an independent condition and must be rewritten in order to achieve full MC/DC. For example, "((x > 3) && (y > 2)) || ((x > 3) && (y < 10))" is a Boolean expression that is comprised of a condition "(x > 3)" that is strongly coupled. In order to achieve full MC/DC, the Boolean expression must be rewritten as "(x > 3) && ((y > 2) || (y < 10))"

- Foldable constant conditions that comprise Boolean expressions *are not counted as measurable conditions* for MC/DC and are effectively ignored. Note that a condition that is *always true* or *always false* may impact your ability to achieve full MC/DC for other conditions that are not constant.

- Some conditions may be *unevaluatable* due to short-circuit language semantics and don't actually affect the decision outcome. They are *masked* by the tooling in the test vector and are considered to have an *effective* Boolean value of either True or False when compared against other test vectors.

- When showing MC/DC data using *--show-mcdc*, each Boolean expression is annotated in a similar way to the example below. Each leaf-level condition is mapped to a "C" condition name (e.g. C1, C2, etc) for the purposes of visualizing the test vectors, and if it can be shown for a condition that changing its value independently affects the decision outcome while holding all other conditions fixed, the *Independence Pair* of test vectors that cover the condition is shown. Unevaluatable, *short-circuited* conditions are rendered using ' - ' in the test vector.

```
 12|      5|  if ((a && b) || (c && d))
|---> MC/DC Decision Region (12:7) to (12:27)
|
|  Number of Conditions: 4
|     Condition C1 --> (12:8)
|     Condition C2 --> (12:13)
|     Condition C3 --> (12:20)
|     Condition C4 --> (12:25)
```

(continues on next page)

```
|
|   Executed MC/DC Test Vectors:
|
|      C1, C2, C3, C4     Result
|   1 { F,  -,   F,   -  = F      }
|   2 { T,  F,   F,   -  = F      }
|   3 { T,  F,   T,   F  = F      }
|   4 { T,  T,   -,   -  = T      }
|   5 { T,  F,   T,   T  = T      }
|
|   C1-Pair: covered: (1,4)
|   C2-Pair: covered: (2,4)
|   C3-Pair: covered: (2,5)
|   C4-Pair: covered: (3,5)
|   MC/DC Coverage for Decision: 100.00%
|
------------------
```

Important Considerations for Branch Coverage

- Some other vendors define Branch Coverage as only covering *Decisions* that may include one or more logical operators. However, Branch Coverage in the tiarmclang compiler supports coverage for all leaf-level Boolean expressions (expressions that cannot be broken down into simpler Boolean expressions). For example, "x = (y == 2) || (z < 10)" is a Boolean expression that is comprised of two conditions, each of which evaluates to either TRUE or FALSE. This support is functionally closer to GCC GCOV/LCOV support.

- When showing branch coverage, each TRUE and FALSE condition represents a branch that is tied to *how many times* its corresponding condition evaluated to TRUE or FALSE. This can also be shown in terms of percentage.

```
44|      3|      if ((VAR1 == 0 && VAR2 == 2) || VAR3 == 34 ||␣
→VAR1 == VAR3)
------------------
|   Branch (44:10): [True: 1, False: 2]
|   Branch (44:20): [True: 0, False: 1]
|   Branch (44:31): [True: 0, False: 3]
|   Branch (44:42): [True: 0, False: 3]
------------------
```

- When viewing branch coverage details in a source-based visualization, it is recommended that users show all macro expansions (using option *--show-expansions*), particularly since macros may contain hidden Boolean expressions. In addition, macro expansions can be

nested (macros are often defined in terms of other macros), as demonstrated in the following example. The coverage summary report always includes these macro-based Boolean expressions in the overall branch coverage count for a function or source file.

```
58|       3|         MACRO2;
------------------
|   |      7|      5|#define MACRO2( MACRO)
|   |   ------------------
|   |   |   |      6|      2|#define MACRO (MACRO_CONDITION ? VAR2 :␣
 ↪VAR1)
|   |   |   |   ------------------
|   |   |   |   |   |      5|      2|#define MACRO_CONDITION (VAR1 != 9)
|   |   |   |   |   |   ------------------
|   |   |   |   |   |   |  Branch (5:16): [True: 2, False: 0]
|   |   |   |   |   |   ------------------
|   |   |   |   ------------------
|   |   ------------------
|   |   |  Branch (7:17): [True: 2, False: 0]
|   |   ------------------
------------------
```

- Coverage is not tracked for branch conditions that the compiler can fold to TRUE or FALSE since for these cases, branches are not generated. This matches the behavior of other code coverage vendors. In the source-based visualization, these branches are displayed as **[Folded - Ignored]**, so that users are informed about what happened.

```
38|       2|     if ((VAR1 == 3) && TRUE)
------------------
|  Branch (38:9): [True: 0, False: 2]
|  Branch (38:24): [Folded — Ignored]
------------------
```

- Branch coverage is tied directly to branch-generating conditions in the source code. As such (unlike with GCOV), users should not see *hidden branches* that aren't actually tied to the source code.

- For switch statements, a branch region is generated for each switch case, including the default case. If there is no *explicitly* defined default case, a branch region is generated to correspond to the *implicit* default case that is generated by the compiler. The *implicit* branch region is tied to the line and column number of the switch statement condition (since no source code for the implicit case exists). In the example below, no explicit default case exists, and so a corresponding branch region for the implicit default case is created and tied to the switch condition on line 65.

```
65|       3|     switch (condition)
```

```
------------------
|  Branch (65:13): [True: 2, False: 1]
------------------
66|        3|     {
67|        1|         case 0:
------------------
|  Branch (67:9): [True: 1, False: 2]
------------------
68|        1|             printf("case0\n"); // fallthrough
69|        1|         case 2:
------------------
|  Branch (69:9): [True: 0, False: 3]
------------------
70|        1|                                 // fallthrough
71|        1|
72|        1|         case 3:
------------------
|  Branch (72:9): [True: 0, False: 3]
------------------
73|        1|             printf("case3\n"); // fallthrough
74|        3|
75|        3|     }
```

Known Limitations

- Counter Initialization After Some Startup Routines

  - For functions that are part of special boot/reset routines that get called prior to C runtime initialization, counter information for these functions are clobbered. If code coverage data is needed for functions like these, a special startup sequence may be required in your system to ensure the counters are properly initialized to zero and not re-initialized later unless the counter data can be extracted first.

- Code Composer Studio Integration

  - Presently, CCS doesn't have direct support for tiarmclang compiler Code Coverage, though support will be added soon. This support will make it very straightforward for users to build projects for code coverage, download counter data from memory, and visualize the data.

- Counter Size

  - Counters are 64bits in size, which may be too large for some embedded use cases.

  - Counter size can be reduced to 32bits by compiling with *-fprofile-counter-size=32*.

- Counters that have large counts may overflow either during execution or when counter data is merged together by the tiarmprofdata tool. When the counter data is merged, tiarmprofdata uses *saturating addition*, so the final value reflects the largest possible value. This affects the accuracy of the visualization.

- Unexpected Function Instantiations of the Same Function

  - The tiarmcov tool uses a function hash to distinguish between functions. This hash is based on the function name, source filename, as well as all included header filenames *as well as their filepaths*. For functions that have the same name across multiple binaries, if any of the filepaths are different, then a different function hash is used, and functions that have the same name are treated by tiarmcov as separate function instantiations of the same function. In the source-based visualization, these instantiations show up as subviews preceded by a general summary view of the function.

  - If a build system *regenerates* the constituent header files for a source file across different builds such that the header filepaths end up being different from build to build, then even if the header files are identical across builds, the function is represented as multiple instantiations of the same function. If these functions are actually identical, then there will only exist one final set of merged counters for the function, and the coverage will be identical across all instantiations. This will *not* negatively impact the final coverage summary of covered lines, regions, or branches.

- Line Coverage Summary Report shows more Executable Lines than are Actually Executable

  - Header files that define static inline functions are counted as separate function instantiations of those functions. If a header file is included and one or more of its static inline functions are not invoked, they will show up in the code coverage report as an *unexecuted instantiation* of the function. Because these functions are not invoked, they won't be instrumented, and so the code coverage tooling only knows that they exist and how many lines in size they are. The tiarmcov tool will therefore assume that all lines in the function are potentially executable, even though they may contain blank lines or lines with comments that cannot be executed.

  - Because of this, an unexecuted instantiation that has blank lines and comments may appear to the coverage reporting tools as having more lines to cover than an executed instantiation has, and the line coverage will report less than 100%. While all that is necessary to cover a static inline function is a single executed instantiation, the presence of unexecuted instantiations can make it seem like a subset of lines are uncovered. Note that this is only true for line coverage and not region coverage, branch coverage, or MC/DC.

  - When this happens, the recommendation is to document the line coverage gap but ignore it, focusing on ensuring that function coverage, region coverage, branch coverage, and MC/DC (if applicable) are covered.

- Function Differences

  - Different function definitions across multiple executables that *have the same function*

name will likely be reported as having "mismatched data". This is a known issue in code coverage for common function names like **main()**. Care should be taken to filter out cases like this using the tiarmcov filtering mechanism since each instance clearly represents a different function.

- Two or more functions that have the same code base but built different such that they contain different macro expansions will be visualized as multiple instantiations of the same function. This doesn't impede coverage.

• Visualization Tool unable to find Source Code

- When a project is built with code coverage enabled, paths to the source code are embedded within the executable file and are extracted by the tiarmcov visualization tool in order to locate the source files. If the system or machine used to build the project is different from what is used to run the visualization tool, the tool will not be able to locate the source files, and it will behave as though no source code is specified.

- You can *change the embedded source code paths* using the *--path-equivalence=<from>,<to>* option, which will enable the visualization tool to find the source files are a new location. This option allows you to map the paths in the coverage data to local source file paths. This allows you to generate the coverage data on one machine, and then use tiarmcov on a different machine where you have the same files on a different path.

• Source Filtering

- The source filtering facility implemented by tiarmcov isn't as fully featured as it is for other vendors, like LCOV. Specifically, embedded filter tags aren't supported (e.g. LCOV_EXCL_[START|STOP]). Please see the filtering options for more information (tiarmcov **--help**).

• Branch Coverage

- Future compiler enhancements will likely be implemented to minimize the number of counters actually used in nested Boolean expressions like "((A || B) && C)", for example.

3.12.2 tiarmprofdata - Profile Data Tool

The **tiarmprofdata** tool can be used to work with profile data files.

Usage

> **tiarmprofdata** *command* [*options*] <*filenames*>

- **tiarmprofdata** - Command to invoke the profile data tool.
- *command* - One of the available tiarmprofdata modes of operation: *merge* or *show*
- *options* - one or more options arguments appropriate for the specified *command* mode
- <*filenames*> - one or more input profile data files

Commands

merge

The **tiarmprofdata merge** command takes several profile data files generated by tiarmclang instrumentation options and merges them together into a single indexed profile data file.

By default profile data is merged without modification. This means that the relative importance of each input file is proportional to the number of samples or counts it contains. In general, the input from a longer training run will be interpreted as relatively more important than a shorter run. Depending on the nature of the training runs it may be useful to adjust the weight given to each input file by using the *-weighted-input* option.

Profiles passed in via *-weighted-input*, *-input-files*, or via positional arguments are processed once for each time they are seen.

Options

–help
> Print a summary of command line options.

–output=<filename>, **–o**=<filename>
> Specify the output <filename>.

–weighted-input=<weight>,<filename>
> Specify an input <filename> along with a <weight>. The profile counts of the supplied <filename> will be scaled (multiplied) by the supplied <weight>, where <weight> is an integer >= 1. Input files specified with using this option are assigned a default <weight> of 1.

−input−files=<path>, **−f**=<path>

Specify a file which contains a list of files to merge. The entries in this file are newline-separated. Lines starting with '#' are skipped. Entries may be of the form <filename> or <weight>,<filename>.

−remapping−file=<path>, **−r**=<path>

Specify a file which contains a remapping from symbol names in the input profile to the symbol names that should be used in the output profile. The file should consist of lines of the form <input-symbol> <output-symbol>. Blank lines and lines starting with '#' are skipped.

The **llvm-cxxmap** tool can be used to generate the symbol remapping file.

−instr

Specify that the input profile is an instrumentation-based profile (default).

−sample

Specify that the input profile is a sample-based profile.

The format of the output file can be generated in one of three ways:

−binary (default)

Emit the profile using a binary encoding. For instrumentation-based profile the output format is the indexed binary format.

−extbinary

Emit the profile using an extensible binary encoding. This option can only be used with sample-based profile. The extensible binary encoding can be more compact with compression enabled and can be loaded faster than the default binary encoding.

−text

Emit the profile in text mode. This option can also be used with both sample-based and instrumentation-based profile. When this option is used the profile will be dumped in the text format that is parsable by the profile reader.

−sparse=[true|false]

Do not emit function records with 0 execution count. This can only be used in conjunction with the *-instr* option. Defaults to *false*, since it can inhibit compiler optimization during profile guided optimization.

−num−threads=<N>, **−j**=<N>

Use <N> threads to perform profile merging. When <N>=0, tiarmprofdata auto-detects an appropriate number of threads to use. This is the default.

−failure−mode=[any|all]

Set the failure mode. There are two options:

- *any* causes the merge command to fail if any profiles are invalid, and

- *all* causes the merge command to fail only if all profiles are invalid.

If *all* is set, information from any invalid profiles is excluded from the final merged product. The default failure mode is *any*.

-prof-sym-list=<path>

Specify a file which contains a list of symbols to generate profile symbol list in the profile. This option can only be used with sample-based profile in extensible binary format. The entries in this file are newline-separated.

-compress-all-sections=[true|false]

Compress all sections when writing the profile. This option can only be used with sample-based profile in extensible binary format.

-use-md5=[true|false]

Use MD5 to represent string in name table when writing the profile. This option can only be used with sample-based profile in extensible binary format.

-gen-partial-profile=[true|false]

Mark the profile to be a partial profile which only provides partial profile coverage for the optimized target. This option can only be used with sample-based profile in extensible binary format.

-supplement-instr-with-sample=<path to sample profile>

Supplement an instrumentation profile with sample profile. The sample profile is the input of the flag. Output will be in instrumentation format (this only works in combination with the *-instr* option).

show

The **tiarmprofdata show** command takes a profile data file and displays the information about the profile counters for the specified input file and for any of the specified functions.

If the input file is omitted or is '-', then **tiarmprofdata show** reads its input from standard input.

Options

-all-functions

Print details for every function.

-counts

Print the counter values for the displayed functions.

-function=<string>

Print details for a function if the function's name contains the given <string>.

-help

Print a summary of command line options.

–output=<filename>, **–o**=<filename>

>   Specify the output <filename>. If <filename> is '-' or it is not specified, then the output is
>   sent to standard output.

–instr

>   Specify that the input profile is an instrumentation-based profile.

–text

>   Instruct the profile dumper to show profile counts in the text format of the instrumentation-
>   based profile data representation. By default, the profile information is dumped in a more
>   human readable form (also in text) with annotations.

–topn=<n>

>   Instruct the profile dumper to show the top <n> functions with the hottest basic blocks in the
>   summary section. By default, the topn functions are not dumped.

–sample

>   Specify that the input profile is a sample-based profile.

–memop-sizes

>   Show the profiled sizes of the memory intrinsic calls for shown functions.

–value-cutoff=<n>

>   Show only those functions whose max count values are greater or equal to <n>. By default,
>   the value-cutoff is set to 0.

–list-below-cutoff

>   Only output names of functions whose max count value are below the cutoff value.

–showcs

>   Only show context sensitive profile counts. The default is to filter all context sensitive profile
>   counts.

–show-prof-sym-list=[true|false]

>   Show profile symbol list if it exists in the profile. This option is only meaningful for sample-
>   based profile in extensible binary format.

–show-sec-info-only=[true|false]

>   Show basic information about each section in the profile. This option is only meaningful for
>   sample-based profile in extensible binary format.

Exit Status

tiarmprofdata returns 1 if the command is omitted or is invalid, if it cannot read input files, or if there is a mismatch between their data.

3.12.3 tiarmcov - Emit Coverage Information

The **tiarmcov** tool is used to show code coverage information for programs that are instrumented to emit profile data.

Usage

>   **tiarmcov** *command [arguments]*

- **tiarmcov** - Command used to invoke the code coverage display tool.

- *command* - One of the available tiarmcov modes of operation: *show*, *report*, or *export*

- *arguments*

Commands

show

>   **tiarmcov show** [*options*] -instr-profile *profile binary* [*sources*]

The **tiarmcov show** command shows line by line coverage of one or more *binary* files using the *profile* data file. It can optionally be filtered to only show the coverage for the files listed in *sources*.

A *binary* can be an executable, an object file, or an archive.

To use **tiarmcov show**, you need a program that is compiled with instrumentation to emit profile and coverage data. To build such a program with **tiarmclang** use the *-fprofile-instr-generate* and *-fcoverage-mapping* flags. If linking with using the **tiarmclang** command, the *-fprofile-instr-generate* option will be passed to the linker to make sure the necessary runtime libraries are linked in.

The coverage information is stored in the built executable or library itself, and this is what you should pass to **tiarmcov show** as a *binary* argument. The profile data is generated by running this instrumented program normally. When the program exits it will write out a raw profile file, typically called *default.profraw*, which can be converted to a format that is suitable for the *profile* argument using the **tiarmprofdata merge** tool.

Options

–show-line-counts

Show the execution counts for each line. Defaults to true, unless another -show option is used.

–show-expansions

Expand inclusions, such as preprocessor macros or textual inclusions, inline in the display of the source file. Defaults to false.

–show-instantiations

For source regions that are instantiated multiple times, such as templates in C++, show each instantiation separately as well as the combined summary. Defaults to true.

–show-regions

Show the execution counts for each region by displaying a caret that points to the character where the region starts. Defaults to false.

–show-line-counts-or-regions

Show the execution counts for each line if there is only one region on the line, but show the individual regions if there are multiple on the line. Defaults to false.

–use-color

Enable or disable color output. By default this is autodetected.

–arch=[*names*]

Specify a list of architectures such that the Nth entry in the list corresponds to the Nth specified binary. If the covered object is a universal binary, this specifies the architecture to use. It is an error to specify an architecture that is not included in the universal binary or to use an architecture that does not match a non-universal binary.

–name=*<function>*

Show code coverage only for named *function*

–name-allowlist=*<file>*

Show code coverage only for functions listed in the given *<file>*. Each line in the file should start with "*allowlist_fun:*", immediately followed by the name of the function to accept. This name can be a wildcard expression.

–filename-allowlist=*<file>*

Show code coverage only for files that match a regular expression listed in the given *<file>*. Each line in the file should start with "*allowlist_file:*".

–name-regex=*<pattern>*

Show code coverage only for functions that match the given regular expression *<pattern>*.

–ignore-filename-regex=*<pattern>*

Skip source code files with file paths that match the given regular expression *<pattern>*.

–format=*\<format>*

>   Use the specified output *\<format>*. The supported formats are: *text* or *html*.

–tab-size=*\<size>*

>   Replace tabs with *\<size>* spaces when preparing reports. Currently, this is only supported for the html format.

–output-dir=*\<path>*

>   Specify a directory *\<path>* to write coverage reports into. If the directory does not exist, it is created. When used in function view mode (i.e when *-name* or *-name-regex* are used to select specific functions), the report is written to *\<path>*/functions.EXTENSION. When used in file view mode, a report for each file is written to *\<path>*/REL_PATH_TO_FILE.EXTENSION.

–Xdemangler=*\<tool>*|*\<tool-option>*

>   Specify a symbol demangler. This can be used to make reports more human-readable. This option can be specified multiple times to supply arguments to the demangler. The demangler is expected to read a newline-separated list of symbols from *stdin* and write a newline-separated list of the same length to *stdout*.

–num-threads=*\<N>*, **-j**=*\<N>*

>   Use *\<N>* threads to write file reports (only applicable when *-output-dir* is specified). When N=0, **tiarmcov** auto-detects an appropriate number of threads to use. This is the default.

–line-coverage-gt=*\<N>*

>   Show code coverage only for functions with line coverage greater than the given threshold *\<N>*.

–line-coverage-lt=*\<N>*

>   Show code coverage only for functions with line coverage less than the given threshold *\<N>*.

–region-coverage-gt=*\<N>*

>   Show code coverage only for functions with region coverage greater than the given threshold *\<N>*.

–region-coverage-lt=*\<N>*

>   Show code coverage only for functions with region coverage less than the given threshold *\<N>*.

–path-equivalence=*\<from>*,*\<to>*

>   Map the paths in the coverage data to local source file paths. This allows you to generate the coverage data on one machine, and then use **tiarmcov** on a different machine where you have the same files on a different path.

report

> **tiarmcov report** [*options*] -instr-profile *profile binary* [*sources*]

The **tiarmcov report** command displays a summary of the coverage of one or more *binary* files, using the profile data *profile*. It can optionally be filtered to only show the coverage for the files listed in *sources*.

A *binary* may be an executable, an object file, or an archive.

If no source files are provided, a summary line is printed for each file in the coverage data. If any files are provided, summaries can be shown for each function in the listed files if the *-show-functions* option is enabled.

Options

-use-color[=*<value>*]

> Enable or disable color output. By default this is autodetected.

-arch=*<name>*

> If the covered binary is a universal binary, select the architecture to use. It is an error to specify an architecture that is not included in the universal binary or to use an architecture that does not match a non-universal binary.

-show-functions

> Show coverage summaries for each function. Defaults to false.

-show-instantiation-summary

> Show statistics for all function instantiations. Defaults to false.

-show-call-region-summary

> Show statistics for all regions containing function calls instantiations. Defaults to false.

-ignore-filename-regex=*<pattern>*

> Skip source code files with file paths that match the given regular expression *<pattern>*.

export

> **tiarmcov export** [*options*] -instr-profile *profile binary* [*sources*]

The **tiarmcov export** command exports coverage data of one or more *binary* files, using the profile data *profile* in either JSON, csv, or lcov trace file format.

When exporting JSON, the regions, functions, expansions, and summaries of the coverage data will be exported. When exporting an lcov trace file, the line-based coverage and summaries will be exported. When exporting a csv trace file, an aggregation of data, including summaries as well as individual file and function metrics will be generated.

The exported data can optionally be filtered to only export the coverage for the files listed in *sources*.

Options

–arch=*<name>*

> If the covered binary is a universal binary, select the architecture to use. It is an error to specify an architecture that is not included in the universal binary or to use an architecture that does not match a non-universal binary.

–format=*<format>*

> Use the specified output *<format>*. The supported formats are: *text*, *csv*, or *lcov*. The *csv* format generates an aggregation of data in a comma-separated format that can be easily imported as an Excel document and saved to correspond with a TI-specific Code Coverage report format.

–summary–only

> Export only summary information for each file in the coverage data. This mode will not export coverage information for smaller units such as individual functions or regions. The result will contain the same information as produced by the **tiarmcov report** command, but presented in JSON or lcov format rather than text.

–ignore–filename–regex=*<pattern>*

> Skip source code files with file paths that match the given regular expression *<pattern>*.

–skip–expansions

> Skip exporting macro expansion coverage data.

–skip–functions

> Skip exporting per-function coverage data.

–num–threads=*<N>*, **–j**=*<N>*

> Use *<N>* threads to export coverage data. When N=0, tiarmcov auto-detects an appropriate number of threads to use. This is the default.

Exit Status

tiarmcov returns 1 if the command is omitted or is invalid, if it cannot read input files, or if there is a mismatch between their data.

3.12.4 Code Coverage for Functional Safety

The TI Arm Clang Compiler can be used for functional safety development as a tool for generating and collecting structural code coverage as required by functional safety standards by applying the TI Compiler Qualification Kit.

The Code Coverage capability supports statement coverage, call coverage, branch coverage, and MC/DC (Modified Condition/Decision Coverage), as documented in the user guide: *Source-Based Code Coverage in tiarmclang*.

For more information:

- Simplify your functionally safe system development with the TI Arm® Clang compiler (TI E2E Article)

- How to apply the TI Compiler Qualification Kit for functional safety development (TI App Note)

- Download the TI Arm Clang Compiler Qualification Kit

3.13 Compiler Security

Contents:

3.13.1 Control Flow Integrity

- *Introduction*
- *Available Schemes*
- *Forward-Edge CFI for Virtual Calls*
- *Bad Cast Checking*
- *Non-Virtual Member Function Call Checking*
    - *Strictness*
- *Indirect Function Call Checking*
    - `-fsanitize-cfi-icall-generalize-pointers`
    - `-fsanitize-cfi-canonical-jump-tables`
- *Member Function Pointer Call Checking*
- *Ignorelist*

- *Publications*

Introduction

The TI Arm Clang Compiler Tools (tiarmclang) includes an implementation of a number of control flow integrity (CFI) schemes, which are designed to abort the program upon detecting certain forms of undefined behavior that can potentially allow attackers to subvert the program's control flow. These schemes have been optimized for performance, allowing developers to enable them in release builds.

To enable the available CFI schemes, use the flag `-fsanitize=cfi`. You can also enable a subset of available *schemes*. As currently implemented, all schemes rely on link-time optimization (LTO); so it is required to specify `-flto`.

The `-fsanitize=cfi-{vcall,nvcall,derived-cast,unrelated-cast}` flags require that a `-fvisibility=` flag also be specified. This is because the default visibility setting is `-fvisibility=default`, which would disable CFI checks for classes without visibility attributes. Most users will want to specify `-fvisibility=hidden`, which enables CFI checks for such classes.

Available Schemes

Available schemes are:

- `-fsanitize=cfi-cast-strict`: Enables *strict cast checks*.

- `-fsanitize=cfi-derived-cast`: Base-to-derived cast to the wrong dynamic type.

- `-fsanitize=cfi-unrelated-cast`: Cast from `void*` or another unrelated type to the wrong dynamic type.

- `-fsanitize=cfi-nvcall`: Non-virtual call via an object whose vptr is of the wrong dynamic type.

- `-fsanitize=cfi-vcall`: Virtual call via an object whose vptr is of the wrong dynamic type.

- `-fsanitize=cfi-icall`: Indirect call of a function with wrong dynamic type.

- `-fsanitize=cfi-mfcall`: Indirect call via a member function pointer with wrong dynamic type.

You can use `-fsanitize=cfi` to enable all the schemes and use `-fno-sanitize` flag to narrow down the set of schemes as desired. For example, you can build your program with `-fsanitize=cfi -fno-sanitize=cfi-nvcall,cfi-icall` to use all schemes except for non-virtual member function call and indirect call checking.

Remember that you have to provide `-flto` if at least one CFI scheme is enabled.

Forward-Edge CFI for Virtual Calls

This scheme checks that virtual calls take place using a vptr of the correct dynamic type; that is, the dynamic type of the called object must be a derived class of the static type of the object used to make the call. This CFI scheme can be enabled on its own using `-fsanitize=cfi-vcall`.

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not), other than members of *ignored* types or types with public visibility, must be compiled with `-flto` enabled and be statically linked into the program.

Bad Cast Checking

This scheme checks that pointer casts are made to an object of the correct dynamic type; that is, the dynamic type of the object must be a derived class of the pointee type of the cast. The checks are currently only introduced where the class being casted to is a polymorphic class.

Bad casts are not in themselves control flow integrity violations, but they can also create security vulnerabilities, and the implementation uses many of the same mechanisms.

There are two types of bad cast that may be forbidden: bad casts from a base class to a derived class (which can be checked with `-fsanitize=cfi-derived-cast`), and bad casts from a pointer of type `void*` or another unrelated type (which can be checked with `-fsanitize=cfi-unrelated-cast`).

The difference between these two types of casts is that the first is defined by the C++ standard to produce an undefined value, while the second is not in itself undefined behavior (it is well defined to cast the pointer back to its original type) unless the object is uninitialized and the cast is a `static_cast` (see C++14 [basic.life]p5).

If a program as a matter of policy forbids the second type of cast, that restriction can normally be enforced. However it may in some cases be necessary for a function to perform a forbidden cast to conform with an external API (e.g. the `allocate` member function of a standard library allocator). Such functions may be *ignored*.

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not), other than members of *ignored* types or types with public visibility, must be compiled with `-flto` enabled and be statically linked into the program.

Non-Virtual Member Function Call Checking

This scheme checks that non-virtual calls take place using an object of the correct dynamic type; that is, the dynamic type of the called object must be a derived class of the static type of the object used to make the call. The checks are currently only introduced where the object is of a polymorphic class type. This CFI scheme can be enabled on its own using `-fsanitize=cfi-nvcall`.

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not), other than members of *ignored* types or types with public visibility, must be compiled with `-flto` enabled and be statically linked into the program.

Strictness

If a class has a single non-virtual base and does not introduce or override virtual member functions or fields other than an implicitly defined virtual destructor, it will have the same layout and virtual function semantics as its base. By default, casts to such classes are checked as if they were made to the least derived such class.

Casting an instance of a base class to such a derived class is technically undefined behavior, but it is a relatively common hack for introducing member functions on class instances with specific properties that works under most compilers and should not have security implications, so we allow it by default. It can be disabled with `-fsanitize=cfi-cast-strict`.

Indirect Function Call Checking

This scheme checks that function calls take place using a function of the correct dynamic type; that is, the dynamic type of the function must match the static type used at the call. This CFI scheme can be enabled on its own using `-fsanitize=cfi-icall`.

For this scheme to work, each indirect function call in the program, other than calls in *ignored* functions, must call a function which was either compiled with `-fsanitize=cfi-icall` enabled, or whose address was taken by a function in a translation unit compiled with `-fsanitize=cfi-icall`.

If a function in a translation unit compiled with `-fsanitize=cfi-icall` takes the address of a function not compiled with `-fsanitize=cfi-icall`, that address may differ from the address taken by a function in a translation unit not compiled with `-fsanitize=cfi-icall`. This is technically a violation of the C and C++ standards, but it should not affect most programs.

Each translation unit compiled with `-fsanitize=cfi-icall` must be statically linked into the program or shared library, and calls across shared library boundaries are handled as if the callee was not compiled with `-fsanitize=cfi-icall`.

`-fsanitize-cfi-icall-generalize-pointers`

Mismatched pointer types are a common cause of cfi-icall check failures. Translation units compiled with the `-fsanitize-cfi-icall-generalize-pointers` flag relax pointer type checking for call sites in that translation unit, applied across all functions compiled with `-fsanitize=cfi-icall`.

Specifically, pointers in return and argument types are treated as equivalent as long as the qualifiers for the type they point to match. For example, `char*`, `char**`, and `int*` are considered equivalent types. However, `char*` and `const char*` are considered separate types.

`-fsanitize-cfi-canonical-jump-tables`

The default behavior of the compiler's indirect function call checker will replace the address of each CFI-checked function in the output file's symbol table with the address of a jump table entry which will pass CFI checks. We refer to this as making the jump table *canonical*. This property allows code that was not compiled with `-fsanitize=cfi-icall` to take a CFI-valid address of a function, but it comes with a couple of caveats:

- There is a performance and code size overhead associated with each exported function, because each such function must have an associated jump table entry, which must be emitted even in the common case where the function is never address-taken anywhere in the program, and must be used.

- There is no good way to take a CFI-valid address of a function written in assembly or a language not supported by Clang. The reason is that the code generator would need to insert a jump table in order to form a CFI-valid address for assembly functions, but there is no way in general for the code generator to determine the language of the function.

For these reasons, we provide the option of making the jump table non-canonical with the flag `-fno-sanitize-cfi-canonical-jump-tables`. When the jump table is made non-canonical, symbol table entries point directly to the function body. Any instances of a function's address being taken in C will be replaced with a jump table address.

This scheme does have its own caveats, however. It does end up breaking function address equality more aggressively than the default behavior.

Furthermore, it is occasionally necessary for code not compiled with `-fsanitize=cfi-icall` to take a function address that is valid for CFI. For example, this is necessary when a function's address is taken by assembly code and then called by CFI-checking C code. The `__attribute__((cfi_canonical_jump_table))` attribute may be used to make the jump table entry of a specific function canonical so that the external code will end up taking an address for the function that will pass CFI checks.

Member Function Pointer Call Checking

This scheme checks that indirect calls via a member function pointer take place using an object of the correct dynamic type. Specifically, we check that the dynamic type of the member function referenced by the member function pointer matches the "function pointer" part of the member function pointer, and that the member function's class type is related to the base type of the member function. This CFI scheme can be enabled on its own using `-fsanitize=cfi-mfcall`.

The compiler will only emit a full CFI check if the member function pointer's base type is complete. This is because the complete definition of the base type contains information that is necessary to correctly compile the CFI check. To ensure that the compiler always emits a full CFI check, it is recommended to also pass the flag `-fcomplete-member-pointers`, which enables a non-conforming language extension that requires member pointer base types to be complete if they may be used for a call.

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not), other than members of *ignored* types or types with public visibility, must be compiled with `-flto` enabled and be statically linked into the program.

Ignorelist

An IgnoreList can be used to relax CFI checks for certain source files, functions and types using the `src`, `fun` and `type` entity types. Specific CFI modes can be be specified using `[section]` headers.

```
# Suppress all CFI checking for code in a file.
src:bad_file.cpp
src:bad_header.h
# Ignore all functions with names containing MyFooBar.
fun:*MyFooBar*
# Ignore all types in the standard library.
type:std::*
# Disable only unrelated cast checks for this function
[cfi-unrelated-cast]
fun:*UnrelatedCast*
# Disable CFI call checks for this function without affecting␣
↪cast checks
[cfi-vcall|cfi-nvcall|cfi-icall]
fun:*BadCall*
```

Publications

Control-Flow Integrity: Principles, Implementations, and Applications. Martin Abadi, Mihai Budiu, Úlfar Erlingsson, Jay Ligatti.

Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, Geoff Pike.

3.13.2 Stack Smashing Detection

- *Introduction*

- *Stack Smashing Detection Options*

    - *Enabling Stack Smashing Detection*

    - *Stack Smashing Detection Example*

Introduction

The TI Arm Clang Compiler Tools (tiarmclang) support options to instrument protection against stack smashing attacks like buffer overflows.

Stack Smashing Detection Options

–fstack–protector

Instructs the compiler to emit extra code to check for buffer overflows, such as stack-smashing attacks. This is done by adding a guard variable to vulnerable functions that contain certain types of objects. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error handling function is called. The error handling function can be made to indicate the error in some way and exit the program. Only variables that are actually allocated on the stack are considered, optimized away variables or variables allocated in registers are not considered.

Vulnerable functions for this setting are:

- Functions with buffers or arrays larger than 8 bytes

- Functions that call *alloca()* with parameters larger than 8 bytes

–fstack–protector–strong

Instructs the compiler to behave as if *-fstack-protector* were specified, except that the vulnerable functions for which the compiler emits stack buffer overflow checking code are:

- Functions that contain any array

- Functions with any local variable that has its address taken

- Functions that call to alloca()

–fstack–protector–all

Instructs the compiler to behave as if *-fstack-protector* were specified, except that the compiler emits stack buffer overflow checking code for *all* functions instead of limiting protection as *-fstack-protector* and *-fstack-protector-strong* do.

```
#include <string.h>

void foo(void);

int main() {
  foo();
  return 0;
}

void foo(void) {
  char buffer[3];
  strcpy(buffer, "Oi! I am smashing your stack");
}
```

The *stack_check.c* source can then be compiled to generate *stack_check.o*:

```
%> tiarmclang -mcpu=cortex-m4 -c stack_check.c
```

and the *stack_smash.c* source file is compiled and linked with stack smashing detection enabled via the use of the *-fstack-protector-all* option:

```
%> tiarmclang -mcpu=cortex-m4 -fstack-protector-all stack_smash.
 ↪c stack_check.o -o stack_smash.out -Wl,-llnk.cmd
```

When loaded and run, the following error message is emitted, and the program exits when the stack check fails before returning from *foo*:

```
ERROR: __stack_chk_guard has been corrupted
```

3.14 Name and C++ Name Demangler Utilities

The compiler tools include the name utility, **tiarmnm**, and the C++ name demangling utility, **tiarmdem**. The **tiarmnm** utility is useful for listing the symbols referenced and defined within a file, and the **tiarmdem** utility can translate mangled C++ symbol and type names into human-readable form.

Contents:

3.14.1 tiarmnm - Name Utility

The **tiarmnm** utility can be used to print a list of the names of symbols from tiarmclang compiler-generated bitcode files, object files, object file libraries, or static executables.

Usage

> **tiarmnm** [*options*] [*input files*]

- *options* - affect how the tiarmnm utility behaves in processing any specified *input files*.

- *input files* - a list of one or more tiarmclang compiler-generated bitcode files, object files, object file libraries, or static executables. If there is no *input files* list specified, **tiarmnm** attempts to read *a.out* as an input file. If **-** is specified in place of the *input files* argument, then **tiarmnm** expects a user-specified input file from *stdin*.

Output Format

The tiarmnm utility outputs a list of symbols including the symbol name along with some simple information about its provenance. The default output format from tiarmnm is the traditional BSD **nm** output format. Each such output record consists of an (optional) 8-digit hexadecimal address, followed by a type code character to indicate the symbol's kind, followed by a name, for each symbol. One record is printed per line; fields are separated by spaces. When the address field is omitted, it is replaced by 8 spaces.

Because compiler-generated bitcode files typically contain objects that are not considered to have addresses until they are linked into an executable image or dynamically compiled "just-in-time", tiarmnm does not print an address for any symbol in an bitcode file, even symbols which are defined in the bitcode file.

Symbol Kind Annotations

As mentioned above, each symbol output record is annotated with a single character that indicates a symbol's type or kind.

The supported symbol kind characters are listed in the table below. Both lower and upper case versions of the a given character are interpreted with the same meaning with respect to a given symbol's kind, but lower-case characters are used for local symbols and upper-case characters are used for global symbols.

Kind Characters	Meaning
a,A	Absolute symbol.
b,B	Uninitialized data (.bss) symbol.
C	Common symbol. Multiple definitions link together into one definition.
d,D	Writable data object.
n	Local symbol from unallocated section.
N	Debug symbol or global symbol from unallocated section.
r,R	Read-only data object.
t,T	Code (.text) object.
u	GNU unique symbol.
U	Named object is undefined in this file.
v	Undefined weak object. It is not a link failure if the object is not defined.
V	Defined weak object symbol. This definition will only be used if no regular definitions exist in a link. If multiple weak definitions and no regular definitions exist, then one of the weak definitions will be used.
?	Something unrecognizable.

Options

-B

Use BSD output format. This is an alias for the *--format=bsd* option.

--debug-syms, -a

Show all symbols, including those that are usually suppressed.

--defined-only, -U

Print only symbols defined in this file.

--demangle, -C

Demangle symbol names.

--dynamic, -D

Display dynamic symbols instead of normal symbols.

--extern-only, -g

Print only symbols whose definitions are externally accessible.

--format=<format>, **-f**=<format>

Select an output *<format>*. Supported values for the *<format>* argument include:

- *sysv*

- *posix*

- *darwin*

- *bsd* (default)

--help, -h

>   Print a summary of the command-line options and their meanings.

--help-list

>   Print an uncategorized summary of command-line options and their meanings.

--just-symbol-name, -j

>   Print only the symbol names.

-m

>   Use darwin output format. This is an alias for the *--format=darwin* option.

--no-demangle

>   Don't demangle symbol names. This option is enabled by default.

--no-llvm-bc

>   Disable the bitcode reader.

--no-sort, -p

>   Show symbols in the order encountered.

--no-weak, -W

>   Don't print weak symbols.

--numeric-sort, -n, -v

>   Sort symbols by address.

--portability, -P

>   Use POSIX.2 output format. This is an alias for the *--format=posix* option.

--print-armap, -M

>   Print the archive symbol table, in addition to the symbols.

--print-file-name, -A, -o

>   Precede each symbol record with the file that it came from.

--print-size, -S

>   Show symbol size as well as address.

--radix=<radix>, **-t**=<radix>

>   Specify the *<radix>* of the symbol addresses. Values accepted include:

>   - *d* - decimal

>   - *x* - hexadecimal

>   - *o* - octal

3.14. Name and C++ Name Demangler Utilities 689

--reverse-sort, -r

> Sort symbols in reverse order.

--size-sort

> Sort symbols by size.

--special-syms

> Do not filter special symbols from the output.

--undefined-only, -u

> Print only undefined symbols.

--version

> Display the version of the **tiarmnm** executable. This option causes tiarmnm to immediately return without reading any specified *input files* to print the symbols in those files.

--without-aliases

> Exclude aliases from the output.

@<file>

> Read command-line options from specified *<file>*.

Examples

- Simple C++ "Hello World" example:

  Consider the following source file (hello.cpp):

  ```cpp
  #include <iostream>
  using namespace std;

  int main ()
  {
    int i;
    cout << "Please enter an integer value: ";
    cin >> i;
    cout << "The value you entered is " << i;
    cout << " and its double is " << i*2 << ".\n";
    return 0;
  }
  ```

  If we then compile *hello.cpp* to an object file:

  ```
  %> tiarmclang -mcpu=cortex-m0 -c hello.cpp
  ```

  We can output the list of symbols in the symbol table for *hello.o*:

```
%> tiarmnm hello.o
00000000 W _ZNKSt3__112basic_stringIcNS_11char_traitsIcEENS_
↪9allocatorIcEEE13__get_pointerEv
00000000 W _ZNKSt3__112basic_stringIcNS_11char_traitsIcEENS_
↪9allocatorIcEEE18__get_long_pointerEv
...
00000000 W _ZNSt3__19use_facetINS_5ctypeIcEEEERKT_RKNS_
↪6localeE
00000000 W _ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_
↪ostreamIcT_EES6_PKc
00000000 T main
         U strlen
```

We could also filter the tiarmnm output to only include symbols that are not defined in *hello.o*:

```
%> tiarmnm -u hello.o
         U _ZNKSt3__16locale9use_facetERNS0_2idE
         U _ZNKSt3__18ios_base6getlocEv
         U _ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_
↪9allocatorIcEEE6__initEjc
         U _ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_
↪9allocatorIcEEED1Ev
         U _ZNSt3__113basic_istreamIcNS_11char_
↪traitsIcEEErsERi
         U _ZNSt3__113basic_ostreamIcNS_11char_
↪traitsIcEEE6sentryC1ERS3_
         U _ZNSt3__113basic_ostreamIcNS_11char_
↪traitsIcEEE6sentryD1Ev
         U _ZNSt3__113basic_ostreamIcNS_11char_
↪traitsIcEEElsEi
         U _ZNSt3__13cinE
         U _ZNSt3__14coutE
         U _ZNSt3__15ctypeIcE2idE
         U _ZNSt3__16localeD1Ev
         U _ZNSt3__18ios_base5clearEj
         U strlen
```

Now if we build a static executable for *hello.cpp*:

```
%> tiarmclang -mcpu=cortex-m0 hello.cpp -o hello.out -Wl,-
↪llnk.cmd,-mhello.map
```

We can now see addresses assigned for some of the symbols that were referenced, but not defined in *hello.o*:

3.14. Name and C++ Name Demangler Utilities 691

```
%> tiarmnm hello.out
...
00026e54 T _ZNKSt3__16locale9use_facetERNS0_2idE
...
00028e1c T _ZNKSt3__18ios_base6getlocEv
...
00027198 W _ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_
 ↪9allocatorIcEEE6__initEjc
...
2000b024 B _ZNSt3__13cinE
...
2000b17c B _ZNSt3__14coutE
...
00028b5a T strlen
```

- Piping output of **tiarmnm** as input to **tiarmdem**:

  Consider the following source file (test.cpp):

```
int g_my_num;
namespace NS { int ns_my_num = 2; }
int f() { return g_my_num + NS::ns_my_num; }
int main() { return f(); }
```

  If the above test.cpp is compiled:

```
tiarmclang -mcpu=cortex-m4 -c test.cpp
```

  We can then use the **tiarmnm** utility to write out the symbol names in test.o:

```
%> tiarmnm test.o
00000000 T _Z1fv
00000000 D _ZN2NS9ns_my_numE
00000000 B g_my_num
00000000 T main
```

  and we could pass the output of **tiarmnm** to **tiarmdem** to demangle the mangled names that are present in the **tiarmnm** output:

```
%> tiarmnm test.o | tiarmdem
00000000 T f()
00000000 D NS::ns_my_num
00000000 B g_my_num
00000000 T main
```

  Incidentally, we could get the same result without using **tiarmdem** by just using the --

demangle option or the *-C* option with **tiarmnm** to instruct **tiarmnm** to demangle the symbol names that it prints out:

```
%> tiarmnm --demangle test.o
00000000 T f()
00000000 D NS::ns_my_num
00000000 B g_my_num
00000000 T main
```

Exit Status

The **tiarmnm** utility should exit with a return code of zero.

3.14.2 tiarmdem - C++ Name Demangler Utility

The **tiarmdem** utility can read a series of C++ mangled symbol names and print their demangled form to *stdout*. If a name cannot be demangled, it is simply printed as is.

Usage

tiarmdem [*options*] [*mangled names*]

- *options* - affect how the tiarmdem utility behaves in processing any specified *mangled names*.

- *mangled names* - a list of one or more symbol names that are presumed to be potentially C++ symbol names. If no *mangled names* are specified as input to the tiarmdem utility, then tiarmdem reads any input symbol names from *stdin*. When reading symbol names from standard input, each input line is split on characters that are not part of valid Itanium name manglings, i.e. characters that are not alphanumeric, '.', '$', or '_'. Separators between names are copied to the output as is. A common use case for feeding symbol names as input to the tiarmdem utility is to pipe the output of the tiarmnm name utility to a tiarmdem command invocation.

Options

--format=<scheme>, **-S**=<scheme>

Select a mangled *<scheme>* to assume. Supported values for the *<scheme>* argument include:

- *auto* (default)

- *gnu*

--help, -h

> Print a summary of the command-line options and their meanings.

--help-list

> Print an uncategorized summary of command-line options and their meanings.

--no-strip-underscore, -n

> Do not strip leading underscore. This option is enabled by default.

--strip-underscore, -_

> Strip a leading underscore, if present, from each input symbol name before demangling.

--types, -t

> Attempt to demangle symbol names as type names as well as function names.

--version

> Display the version of the **tiarmdem** executable.

@<file>

> Read command-line options from specified *<file>*.

Examples

- Specifying mangled names on the command line:

  Symbol names can be specified as input to the tiarmdem C++ name demangler utility on the command line as in the following:

  ```
  %> tiarmdem _Z3foov _Z3bari not_mangled
  foo()
  bar(int)
  not_mangled
  ```

- Specifying mangled names in a text file:

  Symbol names can be specified on separate lines in a text file:

  ```
  %> cat sym_names.txt
  _Z3foov
  _Z3bari
  not_mangled
  ```

  The text file can then be specified as input to tiarmdem as follows:

  ```
  %> tiarmdem < sym_names.txt
  foo()
  ```

```
bar(int)
not_mangled
```

- Piping output of tiarmnm as input to tiarmdem:

  Consider the following source file (test.cpp):

  ```
  int g_my_num;
  namespace NS { int ns_my_num = 2; }
  int f() { return g_my_num + NS::ns_my_num; }
  int main() { return f(); }
  ```

  If the above test.cpp is compiled:

  ```
  tiarmclang -mcpu=cortex-m4 -c test.cpp
  ```

  We can then use the tiarmnm utility to write out the symbol names in test.o:

  ```
  %> tiarmnm test.o
  00000000 T _Z1fv
  00000000 D _ZN2NS9ns_my_numE
  00000000 B g_my_num
  00000000 T main
  ```

  and we could pass the output of **tiarmnm** to **tiarmdem** to demangle the mangled names that are present in the **tiarmnm** output:

  ```
  %> tiarmnm test.o | tiarmdem
  00000000 T f()
  00000000 D NS::ns_my_num
  00000000 B g_my_num
  00000000 T main
  ```

Exit Status

The tiarmdem utility returns 0 unless it encounters a user error, in which case a non-zero exit code is returned.

3.15 Object File Utilities

Several object file utilities are included with the tiarmclang compiler toolchain installation. Some of these, specifically **tiarmobjcopy** and **tiarmstrip**, can be used to edit the content of an ELF object file. Others, like **tiarmobjdump** and **tiarmofd**, are useful for displaying or inspecting the content of an ELF object file.

More information about each of the object file utilities that are provided with the tiarmclang installation can be found in the sections listed below.

Contents:

3.15.1 tiarmdis - Standalone Disassembler Tool

The **tiarmdis** standalone disassembler tool is provided as part of the tiarmclang compiler toolchain installation. This tool processes an ELF object file or executable file as input and writes the disassembled output to *stdout* or a specified output file.

Usage

> **tiarmdis** [*options*] *input_file* [*output_file*]

- **tiarmdis** - is the command that invokes the disassembler.

- *options* - affect the behavior of tiarmdis while processing the *input_file*.

- *input_file* - identifies an input object file or executable file. A file extension for the *input_file* is optional. If a file extension is not provided, then tiarmdis searches for the specified *input_file* in the following order:

    1) *input_file*

    2) *input_file*.out

    3) *input_file*.obj

- *output_file* - is the name of the optional output file where the disassembly output will be written. If no *output_file* is specifiedF, then the disassembly output will be written to *stdout*.

Options

--all, -l

> Disassemble all sections, process .cinit sections.

-be8

> Disassemble in BE-8 mode.

--bytes, -b

> Display data as bytes instead of words.

--copy_tables, -y

> Display copy tables and the sections copied. The table information is dumped first, then each copy record followed by its load data and run data.

--data_as_text, -i

> Disassemble data sections as text.

--diag_wrap=[*on*,*off*]

> Wrap diagnostic messages. This option is on by default.

--help, -h

> Display the tiarmdis help screen.

--hex, -e

> Display integer values in hexadecimal format.

--loadtime_addr, -L

> Display both load and run addresses, if they are different.

--noaddr, -a

> Disable printing of address with label names within instructions.

--nodata, -d

> Disable printing of data sections.

--notext, -t

> Disable printing of text sections.

--quiet, -q

> Suppress the printing of the banner and progress information.

--raw_registers, -r

> Display registers using raw register format (R0, R1, etc.).

--realquiet, -qq

> Suppress all headers.

--suppress, -s

> Suppress printing of address and data words.

--text_as_data, -I

> Disassemble text sections as data.

--ual[=on|off]

> Display assembly in Unified Assembly Language (UAL). This option is *on* by default.

3.15.2 tiarmobjcopy - Object Copying and Editing Tool

The **tiarmobjcopy** tool can be used to copy and manipulate object files. In basic usage, it makes a semantic copy of the input to the output. If any options are specified, the output may be modified along the way (e.g. by removing sections).

Usage

> **tiarmobjcopy** [*options*] *input file* [*output file*]

- **tiarmobjcopy** - is the command that invokes the object copying and editing tool.

- *options* - affect the behavior of tiarmobjcopy.

- *input file* - identifies an object file or archive of object files. If **-** is specified as the *input file* argument, then tiarmobjcopy takes its input from *stdin*. If the *input file* is an archive, then any requested operations are applied to each archive member individually.

- *output file* - specifies where the tiarmobjcopy output is written. This is typically a file name, where the named file is created or overwritten with the tiarmobjcopy output. If no *output file* argument is specified, then the input file is modified in-place. If **-** is specified for the *output file* argument, then the tiarmobjcopy output is written to *stdout*.

Options

The following tiarmobjcopy options are either agnostic of the object file format, or apply to multiple file formats:

--add-section <section>=<file>

> Add a section named *<section>* with the contents of *<file>* to the output. If the specified *<section>* name starts with ".note", then the type of the *<section>* (indicated in the ELF section header) is *SHT_NOTE*. Otherwise, the *<section>* type is <*SHT_PROGBITS*.
>
> The *--add_section* option can be specified multiple times on the **tiarmobjcopy** command line to add multiple sections.

--discard-all, -x

Remove most local symbols from the output. File and section symbols are not discarded from ELF object files.

--dump-section <section>=<file>

Dump the contents of the specified *<section>* into the specified *<file>*. This option can be specified multiple times to dump multiple sections to different files. The indicated *<file>* argument is unrelated to the input and output files provided to tiarmobjcopy and as such the normal copying and editing operations are still performed. No operations are performed on the specified sections prior to dumping them.

--help, -h

Print a summary of command line options.

--only-keep-debug

Produce a debug file as the output that only preserves contents of sections useful for debugging purposes.

For ELF object files, this removes the contents of *SHF_ALLOC* sections that are not *SHT_NOTE* type by making them *SHT_NOBITS* type and shrinking the program headers where possible.

--only-section <section>, **-j** <section>

Remove all sections from the output, except for specified *<sections>*. This option can be specified multiple times to keep multiple sections.

--redefine-sym <old symbol>=<new symbol>

Rename symbols called *<old symbol>* to *<new symbol>* in the output. This option can be specified multiple times to rename multiple symbols.

--redefine-syms <file>

Rename symbols in the output as described in the specified *<file>*. Each line in the *<file>* represents a single symbol to rename, with the old symbol name and new symbol name separated by whitespace. Leading and trailing whitespace is ignored, as is anything following a #. This option can be specified multiple times to read names from multiple files.

--remove-section <section>, **-R** <section>

Remove the specified *<section>* from the output. Can be specified multiple times to remove multiple sections simultaneously.

--set-section-alignment <section>=<align>

Set the alignment of specified *<section>* to *<align>*. This option can be specified multiple times to update multiple sections.

--set-section-flags <section>=<flag>[,<flag>,...]

Set section properties in the output of a *<section>* based on the specified *<flag>* values. This option can be specified multiple times to update multiple sections.

Supported *<flag>* names include:

- *alloc* - Add the *SHF_ALLOC* flag.

- *load* - If the section has *SHT_NOBITS* type, mark it as a *SHT_PROGBITS* section.

- *readonly* - If this flag is not specified, add the *SHF_WRITE* flag.

- *exclude* - Add the *SHF_EXCLUDE* flag.

- *code* - Add the *SHF_EXECINSTR* flag.

- *merge* - Add the *SHF_MERGE* flag.

- *strings* - Add the *SHF_STRINGS* flag.

- *contents* - If the section has *SHT_NOBITS* type, mark it as a *SHT_PROGBITS* section.

--strip-all, -S

Remove from the output all symbols and non-alloc sections not within segments, except for .ARM.attribute section and the section name table.

--strip-debug, -g

Remove all debug sections from the output.

--strip-symbol <symbol>, **-N** <symbol>

Remove specified <*symbol*> from the output. This option can be specified multiple times to remove multiple symbols.

--strip-symbols <file>

Remove all symbols whose names appear in the specified <*file*> from the output. Each line in the <*file*> represents a single symbol name, with leading and trailing whitespace ignored, as is anything following a **#**. This option can be specified multiple times to read names from multiple files.

--strip-unneeded-symbol <symbol>

Remove from the output all symbols named <*symbol*> that are local or undefined and are not required by any relocation.

--strip-unneeded-symbols <file>

Remove all symbols whose names appear in the specified <*file*> from the output, if they are local or undefined and are not required by any relocation. Each line in the <*file*> represents a single symbol name, with leading and trailing whitespace ignored, as is anything following a **#**. This option can be specified multiple times to read names from multiple files.

--strip-unneeded

Remove from the output all local or undefined symbols that are not required by relocations. Also remove all debug sections.

--version, -V

Display the version of the **tiarmobjcopy** executable.

--wildcard, -w

Allow wildcards (like "*" and "?") for symbol-related option arguments. Wildcards are available for section-related option arguments by default.

@<file>

Read command-line options and commands from specified *<file>*.

--add-symbol <name>=[<section>:]<value>[,<flags>]

Add a new symbol called *<name>* to the output symbol table in the specified *<section>*, defined with the given *<value>*. If *<section>* is not specified, the symbol is added as an absolute symbol. The *<flags>* affect the symbol properties.

Accepted values for *<flags>* include:

- *global* - The symbol will have global binding.

- *local* - The symbol will have local binding.

- *weak* - The symbol will have weak binding.

- *default* - The symbol will have default visibility.

- *hidden* - The symbol will have hidden visibility.

- *protected* - The symbol will have protected visibility.

- *file* - The symbol will be an *STT_FILE* symbol.

- *section* - The symbol will be an *STT_SECTION* symbol.

- *object* - The symbol will be an *STT_OBJECT* symbol.

- *function* - The symbol will be an *STT_FUNC* symbol.

This option can be specified multiple times to add multiple symbols.

--allow-broken-links

Allow tiarmobjcopy to remove sections even if it would leave invalid section references. Any invalid *sh_link* fields in the section header table are set to zero.

--change-start <incr>, **--adjust-start** <incr>

Add *<incr>* to the program's start address. This option can be specified multiple times, in which case the *<incr>* values are applied cumulatively.

--compress-debug-sections [<style>]

Compress DWARF debug sections in the output, using the specified *<style>*. Supported styles are *zlib-gnu* and *zlib*. If *<style>* is not specified, *zlib* is assumed by default.

--decompress-debug-sections

Decompress any compressed DWARF debug sections in the output.

--discard-locals, -X

> Remove local symbols starting with ".L" from the output.

--extract-dwo

> Remove all sections that are not DWARF *.dwo* sections from the output.

--extract-main-partition

> Extract the main partition from the output.

--extract-partition <name>

> Extract the *<name>* partition from the output.

--globalize-symbol <symbol>

> Mark any defined symbols named *<symbol>* as global symbols in the output. This option can be specified multiple times to mark multiple symbols.

--globalize-symbols <file>

> Read a list of names from the specified *<file>* and mark defined symbols with those names as global in the output. Each line in the *<file>* represents a single symbol, with leading and trailing whitespace ignored, as is anything following a **#**. This option can be specified multiple times to read names from multiple files.

--input-target <format>, **-I**

> Read the input as the specified *<format>*. See the *Supported Target Formats* section below for a list of valid *<format>* values. If unspecified, tiarmobjcopy attempts to determine the format automatically.

--keep-file-symbols

> Keep symbols of type *STT_FILE*, even if they would otherwise be stripped.

--keep-global-symbol <symbol>

> Make all symbols local in the output, except for symbols with the name *<symbol>*. This option can be specified multiple times to ignore multiple symbols.

--keep-global-symbols <file>

> Make all symbols local in the output, except for symbols named in the specified *<file>*. Each line in the *<file>* represents a single symbol, with leading and trailing whitespace ignored, as is anything following a **#**. This option can be specified multiple times to read names from multiple files.

--keep-section <section>

> When removing sections from the output, do not remove sections named *<section>*. This option can be specified multiple times to keep multiple sections.

--keep-symbol <symbol>, **-K** <symbol>

> When removing symbols from the output, do not remove symbols named *<symbol>*. This option can be specified multiple times to keep multiple symbols.

--keep-symbols `<file>`

When removing symbols from the output do not remove symbols named in the specified *<file>*. Each line in the *<file>* represents a single symbol, with leading and trailing whitespace ignored, as is anything following a **#**. This option can be specified multiple times to read names from multiple files.

--localize-hidden

Make all symbols with *hidden* or *internal* visibility local in the output.

--localize-symbol `<symbol>`, **-L** `<symbol>`

Mark any defined non-common symbol named *<symbol>* as a local symbol in the output. This option can be specified multiple times to mark multiple symbols as local.

--localize-symbols `<file>`

Read a list of names from the specified *<file>* and mark defined non-common symbols with those names as local in the output. Each line in the *<file>* represents a single symbol, with leading and trailing whitespace ignored, as is anything following a **#**. This option can be specified multiple times to read names from multiple files.

--new-symbol-visibility `<visibility_kind>`

Specify the *<visibility_kind>* of the symbols automatically created when using binary input or the *--add-symbol* option. Valid *<visibility_kind>* argument values are:

- *default*

- *hidden*

- *protected*

An automatically created symbol gets *default* visibility unless otherwise specified with the *--new-symbol-visibility* option.

--output-target `<format>`, **-O** `<format>`

Write the output as the specified *<format>*. See the *Supported Target Formats* section below for a list of valid *<format>* values. If a *<format>* argument is not specified, the output format is assumed to be the same as the value specified for *--input-target* or the input file's format if that option is also unspecified.

--prefix-alloc-sections `<prefix>`

Add *<prefix>* to the front of the names of all allocatable sections in the output.

--prefix-symbols `<prefix>`

Add *<prefix>* to the front of every symbol name in the output.

--preserve-dates, -p

Preserve access and modification timestamps in the output.

--rename-section <old section>=<new section>[,<flag>,...]

> Rename sections called *<old section>* to *<new section>* in the output, and apply any specified *<flag>* values. See the *--set-section-flags* option description for a list of supported *<flag>* values. This option can be specified multiple times to rename multiple sections.

--set-start-addr <addr>

> Set the start address of the output to *<addr>*. This option overrides any previously specified *--change-start* or *--adjust-start* options.

--split-dwo <dwo-file>

> Equivalent to running **tiarmobjcopy** with the *--extract-dwo* option and *<dwo-file>* as the output file and no other options, and then with the *--strip-dwo* option on the input file.

--strip-dwo

> Remove all DWARF *.dwo* sections from the output.

--strip-non-alloc

> Remove from the output all non-allocatable sections that are not within segments.

--strip-sections

> Remove from the output all section headers and all section data not within segments. Note that many tools are not able to use an object without section headers.

--target <format>, **-F** <format>

> Equivalent to using the *--input-target* and *--output-target* for the specified *<format>*. See the *Supported Target Formats* for a list of valid *<format>* values.

--weaken-symbol <symbol>, **-W** <symbol>

> Mark any global symbol named *<symbol>* as a weak symbol in the output. This option can be specified multiple times to mark multiple symbols as weak.

--weaken-symbols <file>

> Read a list of names from the specified *<file>* and mark global symbols with those names as weak in the output. Each line in the *<file>* represents a single symbol, with leading and trailing whitespace ignored, as is anything following a **#**. This option can be specified multiple times to read names from multiple files.

--weaken

> Mark all defined global symbols as weak in the output.

Supported Target Formats

The following values are supported by tiarmobjcopy for the *<format>* argument to the *--input-target*, *--output-target*, and *--target* options.

- *binary*

- *elf32-littlearm*

- *elf32-littlearm-freebsd*

- *ihex*

- *ti-txt*

Note: There is currently a known issue with *--input-target* and *--target* causing only *binary* and *ihex <format>* values to have any effect. Other *<format>* values are ignored and tiarmobjcopy attempts to guess the input format.

Note: The *ti-txt* format is the same as the TI-TXT format supported by the TI hex conversion utility. See *TI-TXT Hex Format (--ti_txt Option)* for more details.

Binary Input and Output

If *binary* is used as the *<format>* value for the *--input-target* option, the input file is embedded as a data section in an ELF relocatable object, with symbols *_binary_<file_name>_start*, *_binary_<file_name>_end*, and *_binary_<file_name>_size* representing the start, end and size of the data, where *<file_name>* is the path of the input file as specified on the command line with non-alphanumeric characters converted to *_*.

If *binary* is used as the *<format>* value for *--the output-target*, the output is a raw binary file, containing the memory image of the input file. Symbols and relocation information are discarded. The image starts at the address of the first loadable section in the output.

Exit Status

The tiarmobjcopy utility exits with a non-zero exit code if there is an error. Otherwise, it exits with code 0.

3.15.3 tiarmobjdump - Object File Dumper

The **tiarmobjdump** utility can be used to print the contents of object files and linker output files that are named on the **tiarmobjdump** command line.

Usage

> **tiarmobjdump** [*commands*] [*options*] [*filenames* . . .]

- **tiarmobjdump** - is the command used to invoke the object file dumper utility.

- *commands* - are option-like commands that dictate the tiarmobjdump mode of operation.

- *options* - affect the behavior of tiarmobjdump in a particular mode of operation.

- *filenames* - identify one or more input object files. If no input file is specified, then tiarmobj-dump attempts to read from *a.out*. If **-** is used as an input file name, tiarmobjdump processes a file on its standard input stream.

Commands

At least one of the following commands are required, and some commands can be combined with other commands:

-a, --archive-headers

> Display the information contained within an archive's headers.

-d, --disassemble

> Disassemble all text sections found in the input files.

-D, --disassemble-all

> Disassemble all sections found in the input files.

--disassemble-symbols=<symbol1>[,<symbol2>,...]

> Disassemble only the specified *<symbolN>* arguments. This command will accept deman-gled C++ symbol names when the *--demangle* option is specified. Otherwise this command will accept mangled C++ symbol names. The *--disassemble-symbols* command implies the *--disassemble* command.

--dwarf=<value>

> Dump the specified DWARF debug sections. The supported *<value>* arguments are:
>
> - *frames* - .debug_frame

-f, --file-headers

> Display the contents of the overall file header.

--fault-map-section

Display the content of the fault map section.

-h, --headers, --section-headers

Display summaries of the headers for each section.

--help

Display usage information and exit. This command will prevent other commands from executing.

-p, --private-headers

Display format-specific file headers.

-r, --reloc

Display the relocation entries encoded in the input file.

-R, --dynamic-reloc

Display the dynamic relocation entries encoded in the input file.

--raw-clang-ast

Dump the raw binary contents of the clang AST section.

-s, --full-contents

Display the contents of each section.

-t, --syms

Display the symbol table.

-T, --dynamic-syms

Display the contents of the dynamic symbol table.

-u, --unwind-info

Display the unwind information associated with the input file(s).

--version

Display the version of the tiarmobjdump executable. This command will prevent other commands from executing.

-x, --all-headers

Display all available header information. Equivalent to specifying a combination of the following commands:

- *–archive-headers*
- *–file-headers*
- *–private-headers*
- *–reloc*
- *–section-headers*

3.15. Object File Utilities 707

- *–syms*

Options

The tiarmobjdump utility supports the following options:

--adjust-vma=<offset>

> Increase the displayed address in disassembly or section header printing by the specified *<offset>*.

--arch-name=<string>

> Specify the target architecture with a *<string>* argument when disassembling. Use the *--version* option for a list of available targets.

-C, --demangle

> Demangle C++ symbol names in the output.

--debug-vars=<format>

> Print the locations (in registers or memory) of source-level variables alongside disassembly. The *<format>* argument may be *unicode* or *ascii*, defaulting to *unicode* if omitted.

--debug-vars-indent=<width>

> The distance to indent the source-level variable display relative to the start of the disassembly is indicated by the value of the *<width>* argument. The default value for *<width>* is 40 characters.

-j, --section=<section1>[,<section2>,...]

> Perform commands on the specified sections only.

-l, --line-numbers

> When disassembling, display source line numbers. The use of this option implies the use of the *--disassemble* command.

-M, --disassembler-options=<opt1>[,<opt2>,...]

> Pass target-specific disassembler options. Available options are *reg-names-std* and *reg-names-raw*.

--mcpu=<cpu-name>

> Target a specific CPU with *<cpu-name>* argument for disassembly. Specify *--mcpu=help* to display available values for *<cpu-name>*.

--mattr=<attr1>,+<attr2>,-<attr3>,...

> Enable/disable target-specific attributes. Specify *--mattr=help* to display the available attributes.

--no-leading-addr

> When disassembling, do not print leading addresses.

--no-show-raw-insn

When disassembling, do not print the raw bytes of each instruction.

--prefix=<prefix>

When disassembling with the *--source* option, prepend *<prefix>* to absolute paths.

--print-imm-hex

Use hex format when printing immediate values in disassembly output.

-S, --source

When disassembling, display source interleaved with the disassembly. Use of this option implies the use of the *--disassemble* command.

--show-lma

Display the LMA (section load address) column when dumping ELF section headers. By default, this option is disabled unless a section has different VMA (virtual memory address) and LMAs.

--start-address=<address>

When disassembling, only disassemble from the specified *<address>*.

When printing relocations, only print the relocations patching offsets from at least *<address>*.

When printing symbols, only print symbols with a value of at least *<address>*.

--stop-address=<address>

When disassembling, only disassemble up to, but not including the specified *<address>*.

When printing relocations, only print the relocations patching offsets up to *<address>*.

When printing symbols, only print symbols with a value up to *<address>*.

--triple=<string>

Target triple to disassemble for, see description of *--version* option for a list of available target triple *<string>* values.

-z, --disassemble-zeroes

Do not skip blocks of zeroes when disassembling.

@<file>

Read command-line options and commands from specified *<file>*.

Exit Status

tiarmobjdump returns 1 if the command is omitted or is invalid, if it cannot read input files, or if there is a mismatch between their data.

Known Issues

- tiarmobjdump does not support processing of big-endian object files.

  As an alternative until the fix becomes available, we recommend that the tiarmdis standalone disassembler utility be used to disassemble big-endian object files. The command:

  ```
  tiarmdis file.obj file.dis
  ```

  Writes the disassembly output from the object file *file.o* to *file.dis*. If an output file is not specified, the disassembly output will be written to *stdout*.

3.15.4 tiarmofd - Object File Display Utility

The object file display utility, **tiarmofd**, can be used to print the contents of object files, executable files, and/or archive libraries in both text and XML formats.

Usage

> **tiarmofd** [*options*] *filename*

- **tiarmofd** - is the command used to invoke the object file display utility.
- *options* - affect the behavior of tiarmofd.
- *filename* - identifies an ELF input object file to be read by tiarmofd. If an archive file is specified as an input file, then tiarmofd processes each object file member of the archive as if it was passed on the command line. The object files are processed in the order in which they appear in the archive file.

If the *-o* option is not used to identify a file to write the tiarmofd output into, then the output is written to *stdout*.

Note: Object File Display Format

The object file display utility produces data in a text format by default. This data is not intended to be used as input to programs for further processing of the information. Use the *-x* option to generate output in XML format that is appropriate for mechanical processing.

Options

--call_graph, -cg

> Print function stack usage and callee information in XML format. While the XML output may be accessed by a developer, the function stack usage and callee information XML output was designed to be used by tools such as *Code Composer Studio* to display an application's worst case stack usage. See Stack Usage View in CCS for more information.

> **Note:** This feature requires that source code be built with *debug enabled*.

--diag_wrap[=on|off]

> Wrap diagnostic messages in the output display. This option is enabled by default.

--dwarf, -g

> Append DWARF debug information to the tiarmofd output.

--dwarf_display=<attr1>[,<attr2>, ...]

> The DWARF display settings can be controlled by specifying a comma-separated list of one or more *<attrN>* arguments. A list of the available *<attrN>* values that can be specified are displayed if you invoke **tiarmofd** with the *--dwarf_display=help* option.

> For the following *<attrN>* values, when prefixed with the word *no*, the specified attribute is disabled:
>
> - *dabbrev* - display .debug_abbrev section information (on by default)
> - *daranges* - display .debug_aranges section information (on by default)
> - *dframe* - display .debug_frame section information (on by default)
> - *dinfo* - display .debug_info section information (on by default)
> - *dline* - display .debug_line section information (on by default)
> - *dloc* - display .debug_loc section information (on by default)
> - *dmacinfo* - display .debug_macinfo section information (on by default)
> - *dpubnames* - display .debug_pubnames section information (on by default)
> - *dpubtypes* - display .debug_pubtypes section information (on by default)
> - *dranges* - display .debug_ranges section information (on by default)
> - *dstr* - display .debug_str section information (on by default)
> - *dtypes* - display .debug_types section information (on by default)
> - *regtable* - display register table information (on by default)
> - *types* - display type information (on by default)

The following attribute values can be used to help manage the overall state of the DWARF display attributes:

- *all* - enable all attributes

- *none* - disable all attributes

Examples

In this example, the *dabbrev* attribute is enabled and the *daranges* attribute is disabled:

```
--dwarf_display=dabbrev,nodaranges
```

In this example, all of the DWARF debug display attributes are enabled except for the *dabbrev* attribute:

```
--dwarf_display=all,nodabbrev
```

In this example, all DWARF debug attributes are disabled except for the *daranges* attribute:

```
--dwarf_display=none,daranges
```

--dynamic_info

Display dynamic linking information.

--emit_warnings_as_errors, -pdew

Treat warnings as errors.

--func_info

Display function information.

--help, -h

Display summary of tiarmofd usage and options information.

--obj_display=<attr1>[,<attr2>, ...]

The object file display settings can be controlled by specifying a comma-separated list of one or more *<attrN>* arguments. A list of the available *<attrN>* values that can be specified are displayed if you invoke **tiarmofd** with the *--obj_display=help* option.

For the following *<attrN>* values, when prefixed with the word *no*, the specified attribute is disabled:

- *battrs* - display information about build attributes (on by default)

- *dynamic* - display .dynamic section information (on by default)

- *groups* - display information about ELF groups (on by default)

- *header* - display file header information (on by default)

- *rawdata* - display section raw data (off by default)

- *relocs* - display information about relocation entries (on by default)

- *sections* - display section information (on by default)

- *segments* - display information about ELF segments (on by default)

- *strings* - display string table information (on by default)

- *symbols* - display symbol table information (on by default)

- *symhash* - display information about ELF symbol hash table (on by default)

- *symver* - display symbol version information (on by default)

The following attribute values can be used to help manage the overall state of the object file display attributes:

- *all* - enable all attributes

- *none* - disable all attributes

Examples

In this example, the *battrs* attribute is enabled and the *dynamic* attribute is disabled:

```
--obj_display=battrs,nodynamic
```

In this example, all of the object file display attributes are enabled except for the *battrs* attribute:

```
--obj_display=all,nobattrs
```

In this example, all object file display attributes are disabled except for the *symbols* attribute:

```
--obj_display=none,symbols
```

--output=<file>, **-o**=<file>

Write tiarmofd output to specified *<file>*.

--verbose, -v

Print verbose text output.

--xml, -x

Display output in XML format.

--xml_indent=<N>

Set the number of spaces, *<N>*, to indent nested XML tags.

Exit Status

tiarmofd returns 1 if the command is omitted or is invalid, if it cannot read input files, or if there is a mismatch between their data.

3.15.5 tiarmreadelf - Object File Reader

The **tiarmreadelf** utility displays low-level, format-specific information about one or more object files.

Usage

> **tiarmreadelf** [*options*] [*filenames* ...]

- **tiarmreadelf** - is the command used to invoke the object file reader.

- *options* - affect the behavior of tiarmreadelf.

- *filenames* - identifies one or more ELF object files as input to tiarmreadelf. If **-** is specified as the input file, tiarmreadelf reads its input from *stdin*.

Options

--all
> Equivalent to specifying all the main display options.

--addrsig
> Display the address-significance table.

--arch-specific, -A
> Display architecture-specific information.

--color
> Use colors in the output for warnings and errors.

--demangle, -C
> Display demangled C++ symbol names in the output.

--dyn-relocations
> Display the dynamic relocation entries.

--dyn-symbols, --dyn-syms
> Display the dynamic symbol table.

--dynamic-table, --dynamic, -d

    Display the dynamic table.

--cg-profile

    Display the callgraph profile section.

--elf-hash-histogram, --histogram, -I

    Display a bucket list histogram for dynamic symbol hash tables.

--elf-linker-options

    Display the linker options section.

--elf-section-groups, --section-groups, -g

    Display section groups.

--expand-relocs

    When used with the *--relocations* option, the *--expand-relocs* option causes tiarmreadelf to display each relocation in an expanded multi-line format.

--file-headers, -h

    Display file headers.

--hash-symbols

    Display the expanded hash table with dynamic symbol data.

--hash-table

    Display the hash table for dynamic symbols.

--headers, -e

    Equivalent to combining the *--file-headers*, *--program-headers*, and *--sections* options.

--help

    Display a summary of command line options.

--help-list

    Display an uncategorized summary of command line options.

--hex-dump=<section1>[,<section2>,...], **-x**

    Display the specified *<sectionN>* as hexadecimal bytes. A given *<sectionN>* argument may be a section index or section name.

--needed-libs

    Display the needed libraries.

--notes, -n

    Display all notes.

--program-headers, --segments, -l

    Display the program headers.

--raw-relr

Do not decode relocations in RELR relocation sections when displaying them.

--relocations, --relocs, -r

Display the relocation entries in the file.

--sections, --section-headers, -S

Display all sections.

--section-data

When used with the *--sections* option, this option causes tiarmreadelf to display section data for each section shown.

--section-details, -t

Display all section details. Used as an alternative to the *--sections* option.

--section-mapping

Display the section to segment mapping.

--section-relocations

When used with the *--sections* option, this option causes tiarmreadelf to display relocations for each section shown.

--section-symbols

When used with the *--sections* option, the *--section-symbols* option causes tiarmreadelf to display symbols for each section shown.

--stackmap

Display contents of the stackmap section.

--string-dump=<section1>[,<section2>,...], **-p**

Display the specified *<sectionN>* as a list of strings. *<sectionN>* may be a section index or section name.

--symbols, --syms, -s

Display the symbol table.

--unwind, -u

Display stack unwinding information.

--version

Display the version of the **tiarmreadelf** executable.

--version-info, -V

Display version sections.

@<file>

Read command-line options from specified *<file>*.

3.15. Object File Utilities 716

Exit Status

The **tiarmreadelf** utility returns 0 under normal operation. It returns a non-zero exit code if there were any errors.

3.15.6 tiarmsize - Print Size Information

The **tiarmsize** tool prints size information for binary files.

Usage

>   **tiarmsize** [*options*] [*input . . .*]

- **tiarmsize** - is the command used to invoke the tool.

- *options* - affect the behavior of the tiarmsize tool.

- *input* - identify one or more input object files, for which tiarmsize prints the size information for each *input* file specified. If no input is specified, tiarmsize attempts to print size information for *a.out*. If **-** is specified as an input file, tiarmsize reads a file from the standard input stream. If an *input* is an archive, size information is displayed for all its members.

The tiarmsize output is written to *stdout*.

Options

−A

>   Equivalent to the *--format=sysv* option.

−B

>   Equivalent to the *--format=berkeley* option.

−−common

>   Include ELF common symbol sizes in .bss section size for *berkeley* output format, or as a separate section entry for *sysv* output. If the *--common* option is not specified, these symbols are ignored.

−d

>   Equivalent to the *--radix=10* option.

−−format=<format>

>   Set the output format to the specified *<format>*. Available *<format>* argument values are:

>   - *berkeley* (the default)

>   - *sysv*

Berkeley output summarizes text, data and bss sizes in each file, as shown below for a typical pair of ELF files:

```
$ tiarmsize --format=berkeley test.out ref_global.o def_
↪global.o
   text    data     bss     dec     hex filename
   4968    4096   41885   50949    c705 test.out
     62       0       0      62      3e ref_global.o
      0       4       0       4       4 def_global.o
```

Sysv output displays size and address information for most sections, with each file being listed separately:

```
$ tiarmsize --format=sysv test.out ref_global.o def_global.o
test.out  :
section                   size          addr
.intvecs                     0             0
.bss                       460     536912372
.data                      465     536911904
.sysmem                   8192     536903712
.stack                   32768     536870944
.text                     4788            32
.cinit                     128          8968
.const                       0             0
.rodata                     52          4820
.ARM.exidx                   0             0
.init_array                  0             0
.TI.ramfunc                  0             0
.TI.noinit                   0             0
.TI.persistent               0             0
__llvm_prf_cnts              0     536870944
.binit                       0            32
Veneer$$CMSE                 0             0
.args                     4096          4872
.ARM.attributes             83             0
.TI.section.flags           26             0
.symtab_meta               173             0
Total                    51231

ref_global.o  :
section                   size    addr
.text                        0       0
.text.main                  36       0
.ARM.exidx.text.main         8       0
```

```
.rodata.str1.1               18        0
.comment                    121        0
.note.GNU-stack               0        0
.ARM.attributes              65        0
.symtab_meta                  5        0
.llvm_addrsig                 2        0
Total                       255


def_global.o  :
section       size     addr
.text            0        0
.data            4        0
Total            4
```

--help, -h

Display a summary of command line options.

--help-list

Display an uncategorized summary of command line options.

-o

Equivalent to the *--radix=8* option.

--radix=<value>

Display size information in the specified radix indicated by the *<value>* argument. Permitted values are *8*, *10* (the default) and *16* for octal, decimal and hexadecimal output, respectively.

Example:

```
$ tiarmsize --radix=8 ref_global.o
   text     data      bss      oct      hex filename
    076        0        0       76       3e ref_global.o

$ tiarmsize --radix=10 ref_global.o
   text     data      bss      dec      hex filename
     62        0        0       62       3e ref_global.o

$ tiarmsize --radix=16 ref_global.o
   text     data      bss      dec      hex filename
   0x3e        0        0       62       3e ref_global.o
```

--totals, -t

Applies only to *berkeley* output format. Display the totals for all listed fields, in addition to the individual file listings.

3.15. Object File Utilities   **719**

Example:

```
$ llvm-size --totals ref_global.o def_global.o
   text    data     bss     dec     hex filename
     62       0       0      62      3e ref_global.o
      0       4       0       4       4 def_global.o
     62       4       0      66      42 (TOTALS)
```

--version

Display the version of the **tiarmsize** executable.

-x

Equivalent to the *--radix=16* option.

@<file>

Read command-line options from specified *<file>*.

Exit Status

The tiarmsize utility exits with a non-zero exit code if there is an error. Otherwise, it exits with code 0.

3.15.7 tiarmstrip - Object File Stripping Tool

The **tiarmstrip** tool can be used to strip sections and symbols from object files.

Usage

>**tiarmstrip** [*options*] *inputs* …

- **tiarmstrip** - is the command that invokes the object file stripping tool.

- *options* - that affect the behavior of tiarmstrip. If no other stripping or remove options are specified on the command line, then tiarmstrip assumes the *--strip-all* option is enabled.

- *inputs* - identify one or more object files that are processed by tiarmstrip. By default, the *input* files are modified in-place. If **-** is specified for the input file, the input is read from the program's standard input stream. If the *input* is an archive, any requested operations are applied to each archive member individually.

Options

--disable-deterministic-archives, -U

Use real values for UIDs, GIDs and timestamps when updating archive member headers.

--discard-all, -x

Remove most local symbols from the output. File and section symbols in ELF objects are not discarded.

--enable-deterministic-archives, -D

Enable deterministic mode when stripping archives, i.e. use 0 for archive member header UIDs, GIDs and timestamp fields. This option is enabled by default.

--help, -h

Print a summary of command line options.

--no-strip-all

Disable the *--strip-all* option.

-o `<file>`

Write output to specified *<file>*. Multiple input files cannot be used in combination with the *-o* option.

--remove-section `<section>`, **-R** `<section>`

Remove the specified *<section>* from the output. This option can be specified multiple times to remove multiple sections simultaneously.

--strip-all, -S

For ELF objects, remove from the output all symbols and non-alloc sections not within segments, except for the .ARM.attribute section and the section name table.

--strip-debug, -g

Remove all debug sections from the output.

--strip-symbol `<symbol>`, **-N** `<symbol>`

Remove all symbols named *<symbol>* from the output. This option can be specified multiple times to remove multiple symbols.

--strip-unneeded

Remove from the output all local or undefined symbols that are not required by relocations. Also remove all debug sections.

--version, -V

Display the version of the **tiarmstrip** executable.

--wildcard, -w

Allow wildcard syntax (such as "*" and "?") for symbol-related options. Wildcards are enabled by default for section-related options.

@<file>

> Read command-line options and commands from specified *<file>*.

--allow-broken-links

> Allow tiarmstrip to remove sections even if it would leave invalid section references. Any invalid *sh_link* fields are set to zero.

--discard-locals, -X

> Remove local symbols starting with ".L" from the output.

--keep-file-symbols

> Keep symbols of type *STT_FILE*, even if they would otherwise be stripped.

--keep-section <section>

> When removing sections from the output, do not remove sections named *<section>*. This option can be specified multiple times to keep multiple sections.

--keep-symbol <symbol>, **-K** <symbol>

> When removing symbols from the output, do not remove symbols named *<symbol>*. This option can be specified multiple times to keep multiple symbols.

--preserve-dates, -p

> Preserve access and modification timestamps in the output.

--strip-sections

> Remove from the output all section headers and all section data not within segments. Note that many tools cannot use an object without section headers.

Exit Status

The tiarmstrip utility exits with a non-zero exit code if there is an error. Otherwise, it exits with code 0.

3.16 Hex Conversion Utility Description

The Arm assembler and linker create object files which are in binary formats that encourage modular programming and provide powerful and flexible methods for managing code segments and target system memory.

Most EPROM programmers do not accept object files as input. The hex conversion utility converts an object file into one of several standard ASCII hexadecimal formats, suitable for loading into an EPROM programmer. The utility is also useful in other applications requiring hexadecimal conversion of an object file (for example, when using debuggers and loaders).

The hex conversion utility can produce these output file formats:

- ASCII-Hex, supporting 16-bit addresses (see *ASCII-Hex Object Format (--ascii Option)*)

- Binary file in 8-bit format (see *Specifying the Memory Width*)

- Extended Tektronix (Tektronix) (see *Extended Tektronix Object Format (--tektronix Option)*)

- Intel MCS-86 (Intel) (see *Intel MCS-86 Object Format (--intel Option)*)

- Motorola Exorciser (Motorola-S), supporting 16-bit addresses (see *Motorola Exorciser Object Format (--motorola Option)*)

- Texas Instruments SDSMAC (TI-Tagged), supporting 16-bit addresses (see *Texas Instruments SDSMAC (TI-Tagged) Object Format (--ti_tagged Option)*)

- Texas Instruments TI-TXT format, supporting 16-bit addresses (see *TI-TXT Hex Format (--ti_txt Option)*)

- C arrays

3.16.1 The Hex Conversion Utility's Role in the Software Development Flow

The following figure highlights the role of the hex conversion utility in the software development process.

Figure: The Hex Conversion Utility Software Development Flow
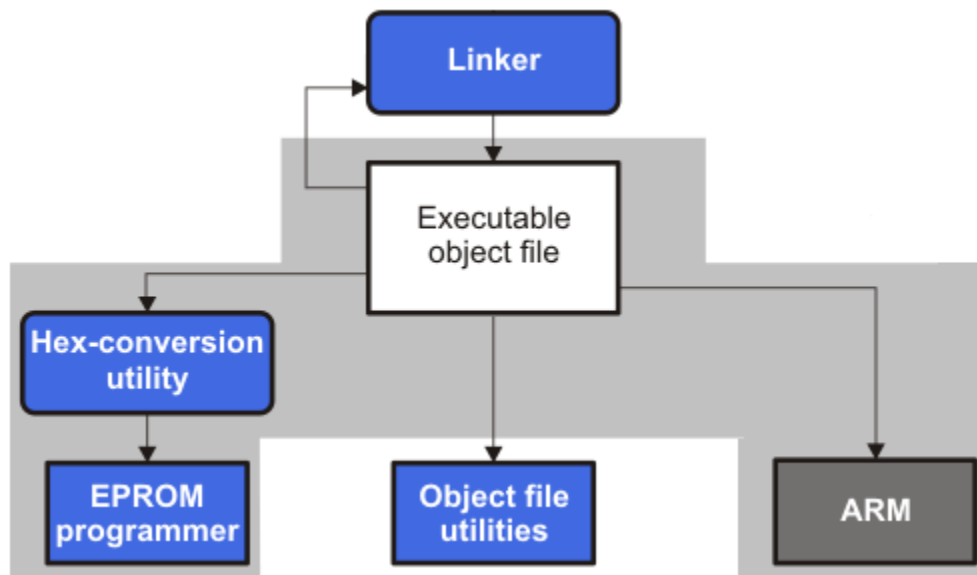
3.16.2 Invoking the Hex Conversion Utility

There are two basic methods for invoking the hex conversion utility:

- **Specify the options and filenames on the command line.** The following example converts the file firmware.out into TI-Tagged format, producing two output files, firm.lsb and firm.msb:

```
tiarmhex -t firmware -o firm.lsb -o firm.msb
```

- **Specify the options and filenames in a command file.** You can create a file that stores command line options and filenames for invoking the hex conversion utility. The following example invokes the utility using a command file called hexutil.cmd:

```
tiarmhex hexutil.cmd
```

In addition to regular command line information, you can use the hex conversion utility ROMS and SECTIONS directives in a command file.

Invoking the Hex Conversion Utility From the Command Line

To invoke the hex conversion utility, enter:

tiarmhex [*options*] *filename*

- **tiarmhex** is the command that invokes the hex conversion utility.

- *options* supplies additional information that controls the hex conversion process. You can use options on the command line or in a command file. *Hex Conversion Utility Options* lists the available options.

  - All options are preceded by a hyphen and are not case sensitive.

  - Several options have an additional parameter that must be separated from the option by at least one space.

  - Options with multi-character names must be spelled exactly as shown in this document; no abbreviations are allowed.

  - Options are not affected by the order in which they are used. The exception to this rule is the --quiet option, which must be used before any other options.

- *filename* names an object file or a command file. (For more information, see *Invoking the Hex Conversion Utility With a Command File*.)

Hex Conversion Utility Options

In the following list, the shortened alias (if any) is shown in parentheses after the option. Unless otherwise specified, the alias expects the same parameters as the main option syntax.

General Options

--byte (`-byte`)

> Syntax: --byte
>
> Number output locations by bytes rather than by target addressing. See *Controlling the ROM Device Address*.

--entrypoint (`-e`)

> Syntax: --entrypoint=*addr*
>
> Specify the entry point at which to begin execution after boot loading. See *How to Build the Boot Table*.

--exclude (`-exclude`)

> Syntax: --exclude={*fname*(*sname*) |*sname*}
>
> If the filename (*fname*) is omitted, all sections matching *sname* will be excluded. See *Excluding a Specified Section*.

--fill (`-fill`)

> Syntax: --fill=*value*
>
> Fill holes with *value*. See *Specifying a Fill Value*.

--help (`-options`, **-h**)

> Syntax: --help
>
> Display the syntax for invoking the utility and list available options. If the option is followed by another option or phrase, detailed information about that option or phrase is displayed. See *Invoking the Hex Conversion Utility With a Command File*.

--image (`-image`)

> Syntax: --image
>
> Select image mode. See *Generating a Memory Image*.

--linkerfill (`-linkerfill`)

> Syntax: --linkerfill
>
> Include linker fill sections in images. See *Filling Holes*.

--map (`-map`)

> Syntax: --map=*filename*
>
> Generate a map file. See *An Example of the ROMS Directive*.

--memwidth (-memwidth)

Syntax: --memwidth=*value*

Define the system memory word width (default 16 bits). See *Specifying the Memory Width*.

--outfile (-o)

Syntax: --outfile=*filename*

Specify an output filename. See *Assigning Output Filenames*.

--quiet (-q)

Syntax: --quiet

Run quietly. If used, this option must appear *before* other options. See *Invoking the Hex Conversion Utility With a Command File*.

--romwidth (-romwidth)

Syntax: --romwidth=*value*

Specify the ROM device width (default depends on format used). This option is ignored for the TI-TXT, binary, and TI-Tagged formats. See *Partitioning Data Into Output Files*.

--zero (-zero, **-z)**

Syntax: --zero

Reset the address origin to 0 in image mode. See *Steps to Follow in Using Image Mode*.

Diagnostic Options

The hex conversion utility uses certain C/C++ compiler options to control hex-converter-generated diagnostics. See *Control Hex Conversion Utility Diagnostics* for information about the diagnostic options.

--diag_error

Syntax: --diag_error=*id*

Categorizes the diagnostic identified by *id* as an error.

--diag_remark

Syntax: --diag_remark=*id*

Categorizes the diagnostic identified by *id* as a remark.

--diag_suppress

Syntax: --diag_suppress=*id*

Suppresses the diagnostic identified by *id*.

--diag_warning

Syntax: --diag_warning=*id*

Categorizes the diagnostic identified by *id* as a warning.

--display_error_number

>   Syntax: --display_error_number

>   Displays a diagnostic message's identifiers along with its text.

--issue_remarks

>   Syntax: --issue_remarks

>   Issues remarks (non-serious warnings).

--no_warnings

>   Syntax: --no_warnings

>   Suppresses all warning diagnostics. Errors are still issued.

--set_error_limit

>   Syntax: --set_error_limit=*count*

>   Sets the error limit to *count*. The linker abandons linking after the specified number of errors.
>   (The default is 100.)

Boot Options

--boot_align_sect

>   Syntax: --boot_align_sect

>   Causes boot table records to be aligned to the section alignment. See *Building a Table for*
>   *an On-Chip Boot Loader*.

--boot_block_size=size

>   Syntax: --boot_block_size=*size*

>   Set the desired block size for output from the hex conversion utility. The size may be spec-
>   ified as a hex or decimal value. The default size is 65535 (0xFFFF). For ARM targets, the
>   boot block size refers to 8-bit bytes. See *Building a Table for an On-Chip Boot Loader*.

--cmac=file

>   Syntax: --cmac=*file*

>   Specify a file containing the CMAC key for use with secure flash boot on TMS320F2838x
>   devices. See *Using Secure Flash Boot on TMS320F2838x Devices*.

Output Options

The hex conversion utility provides several options to specify the output format. These options are
described in *Description of the Object Formats* and its subsections.

--array

>   Syntax: --array

>   Select array output format. See *Array Output Format*.

--ascii (-a)

Syntax: --ascii

Select the ASCII-Hex output format. See *ASCII-Hex Object Format (--ascii Option)*.

--binary (-b)

Syntax: --binary

Select the binary output format. (The object file must have a memory width of 8 bits.) See *Binary Object Format (--binary Option)*.

--intel (-i)

Syntax: --intel

Select the Intel output format. See *Intel MCS-86 Object Format (--intel Option)*.

--motorola=1 (-m1)

Syntax: --motorola=1

Select the Motorola-S1 output format. See *Motorola Exorciser Object Format (--motorola Option)*.

--motorola=2 (-m2)

Syntax: --motorola=2

Select the Motorola-S2 output format. See *Motorola Exorciser Object Format (--motorola Option)*.

--motorola=3 (-m3)

Syntax: --motorola=3

Select the Motorola-S2 output format. See *Motorola Exorciser Object Format (--motorola Option)*.

--tektronix (-x)

Syntax: --tektronix

Select the Tektronix output format. This is the default when no output format is specified. See *Extended Tektronix Object Format (--tektronix Option)*.

--ti_tagged (-t)

Syntax: --ti_tagged

Select the TI-Tagged output format. (The object file must have a memory width of 16 bits.) See *Texas Instruments SDSMAC (TI-Tagged) Object Format (--ti_tagged Option)*.

--ti_txt

Syntax: --ti_txt

Select the TI-Txt output format. (The object file must have a memory width of 8 bits.) See *TI-TXT Hex Format (--ti_txt Option)*.

3.16. Hex Conversion Utility Description   **728**

Load Image Options

The hex conversion utility provides several options to specify the load image and its format. These options are described in *The Load Image Format (--load_image Option)* and its subsections.

--load_image

>   Syntax: --load_image

>   Select load image object file format. See *The Load Image Format (--load_image Option)*.

--load_image:combine_sections

>   Syntax: --load_image:combine_sections=[true|false]

>   Specify whether sections should be combined or not. The default is true.

--load_image:endian

>   Syntax: --load_image:endian=[big|little]

>   Specify the object file endianness. If this option is omitted, the endianness of the first file on the command line is used.

--load_image:file_type

>   Syntax: --load_image:file_type=[relocatable|executable]

>   Specify a file type other than object files. Object files can be linked with one another, but addresses are lost. Relocatable files contain the address in the sh_addr field of a section. Executable files maintain address bindings and can be directly loaded.

--load_image:format

>   Syntax: --load_image:format=[coff|elf]

>   Specify the ABI format of the object file. If this option is omitted, the format is determined from the first file on the command line.

--load_image:globalize

>   Syntax: --load_image:globalize=*string*

>   Do not localize the specified symbol. The default can be set with the –load_image:symbol_binding option.

--load_image:localize

>   Syntax: --load_image:localize=*string*

>   Make the specified symbol local. The default can be set with the –load_image:symbol_binding option.

--load_image:machine

>   Syntax: --load_image:machine=[ARM|C2000|C6000|C7X|MSP430|PRU]

>   Specify the object file machine type. If this option is omitted, the machine type from the first file on the command line is used.

--load_image:output_symbols

> Syntax: --load_image:output_symbols=[true|false]

> Specify whether symbols should be output to the file. The default is false.

--load_image:section_addresses

> Syntax: --load_image:section_addresses=[true|false]

> Specify whether the load address should be written in the output file. This option applies to relocatable files only. The default is true.

--load_image:section_prefix

> Syntax: --load_image:section_prefix=*string*

> Specify a prefix for section names. The default is `image_`.

--load_image:symbol_binding

> Syntax: --load_image:symbol_binding=[local|global]

> Specify the default binding of symbols in the load image.

Invoking the Hex Conversion Utility With a Command File

A command file is useful if you plan to invoke the utility more than once with the same input files and options. It is also useful if you want to use the ROMS and SECTIONS hex conversion utility directives to customize the conversion process.

Command files are ASCII files that contain one or more of the following:

- **Options and filenames.** These are specified in a command file in exactly the same manner as on the command line.

- **ROMS directive.** The ROMS directive defines the physical memory configuration of your system as a list of address-range parameters. (See *The ROMS Directive*.)

- **SECTIONS directive.** The hex conversion utility SECTIONS directive specifies which sections from the object file are selected. (See *The SECTIONS Directive*.)

- **Comments.** You can add comments to your command file by using the /* and */ delimiters. For example:

```
/* This is a comment. */
```

To invoke the utility and use the options you defined in a command file, use a command with syntax like this:

tiarmhex *command_filename*

You can also specify other options and files on the command line. For example, you could invoke the utility by using both a command file and command line options:

3.16. Hex Conversion Utility Description 730

```
tiarmhex firmware.cmd --map=firmware.mxp
```

The order in which these options and filenames appear is not important. The utility reads all input from the command line and all information from the command file before starting the conversion process. However, if you are using the -q option, *it must appear as the first option on the command line or in a command file.*

The **--help** option displays the syntax for invoking the compiler and lists available options. If the --help option is followed by another option or phrase, detailed information about the option or phrase is displayed. For example, to see information about options associated with generating a boot table use --help boot.

The **--quiet** option suppresses the hex conversion utility's normal banner and progress information.

Assume that a command file named `firmware.cmd` contains these lines:

```
firmware.out           /* input file */
--ti-tagged            /* TI-Tagged */
--outfile=firm.lsb     /* output file */
--outfile=firm.msb     /* output file */
```

You can invoke the hex conversion utility by entering:

```
tiarmhex firmware.cmd
```

This example shows how to convert a file called `appl.out` into eight hex files in Intel format. Each output file is one byte wide and 4K bytes long.

```
appl.out            /* input file   */
--intel             /* Intel format */
--map=appl.mxp      /* map file     */

ROMS
{
    ROW1: origin=0x00000000 len=0x4000 romwidth=8
          files={ appl.u0 appl.u1 app1.u2 appl.u3 }
    ROW2: origin=0x00004000 len=0x4000 romwidth=8
          files={ app1.u4 appl.u5 appl.u6 appl.u7 }
}

SECTIONS
{
    .text, .data, .cinit, .sect1, .vectors, .const, .rodata:
}
```

3.16.3 Understanding Memory Widths

The hex conversion utility makes your memory architecture more flexible by allowing you to specify memory and ROM widths. To use the hex conversion utility, you must understand how the utility treats word widths. Three widths are important in the conversion process:

- Target width
- Memory width
- ROM width

The terms target word, memory word, and ROM word refer to a word of such a width.

The following figure illustrates the separate and distinct phases of the hex conversion utility's process flow.

Figure: Hex Conversion Utility Process Flow

Raw data in object files is represented in the target's addressable units. For the ARM device, this is 32 bits.

Input file

Phase I

The raw data in the object file is grouped into words according to the size specified by the --memwidth option.

Phase II

The memwidth-sized words are broken up according to the size specified by the --romwidth option and are written to a file(s) according to the specified format (i.e., Intel, Tektronix, etc.).

Output file(s)

Target Width

Target width is the unit size (in bits) of the target processor's word. The width is fixed for each target and cannot be changed. The Arm targets have a width of 32 bits.

Specifying the Memory Width

Memory width is the physical width (in bits) of the memory system. Usually, the memory system is physically the same width as the target processor width: a 16-bit processor has a 32-bit memory architecture. However, some applications require target words to be broken into multiple, consecutive, and narrower memory words.

By default, the hex conversion utility sets memory width to the target width (in this case, 32 bits).

You can change the memory width (except for TI-TXT, binary, and TI-Tagged formats) by:

- Using the **--memwidth** option. This changes the memory width value for the entire file.

- Setting the **memwidth** parameter of the ROMS directive. This changes the memory width value for the address range specified in the ROMS directive and overrides the --memwidth option for that range. See *The ROMS Directive*.

For both methods, use a value that is a power of 2 greater than or equal to 8.

You should change the memory width default value of 16 only when you need to break single target words into consecutive, narrower memory words.

Note: Binary Format is 8 Bits Wide You cannot change the memory width of the Binary format. The Binary hex format supports an 8-bit memory width only. See *TI-TXT Hex Format (--ti_txt Option)* for more about using the ROMS directive with an 8-bit format.

Note: TI-TXT Format is 8 Bits Wide You cannot change the memory width of the TI-TXT format. The TI-TXT hex format supports an 8-bit memory width only. See *TI-TXT Hex Format (--ti_txt Option)* for more about using the ROMS directive with the TI-TXT hex format.

The following figure shows how memory width is related to object file data.

Figure: Object File Data and Memory Widths
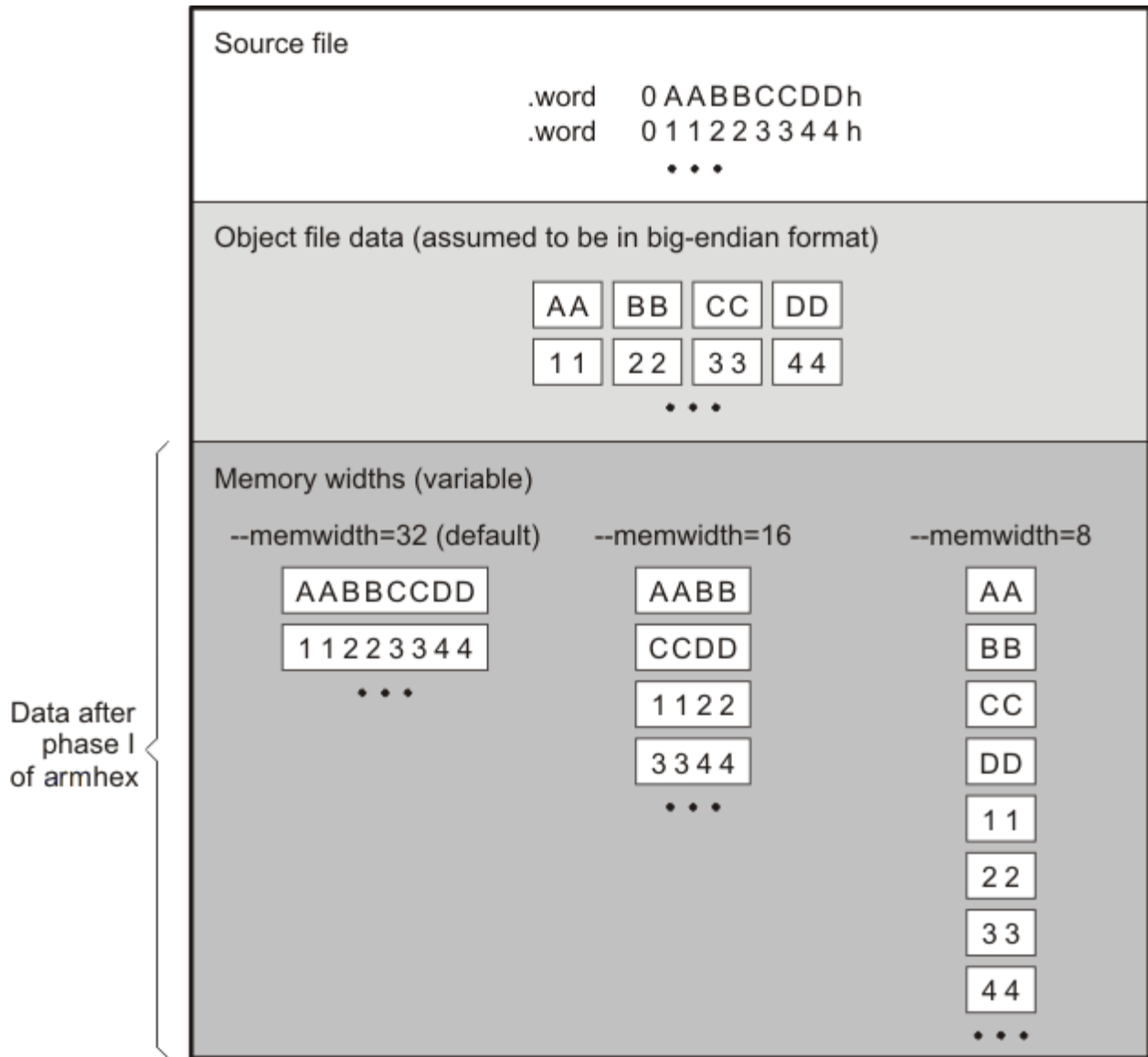
Partitioning Data Into Output Files

ROM width determines how the hex conversion utility partitions the data into output files. ROM width specifies the physical width (in bits) of each ROM device and corresponding output file (usually one byte or eight bits). After the object file data is mapped to the memory words, the memory words are broken into one or more output files. The number of output files is determined by the following formulas:

- If memory width ROM width:

```
number of files = memory width ÷ ROM width
```

- If memory width < ROM width:

```
number of files = 1
```

For example, for a memory width of 32, you could specify a ROM width value of 32 and get a single output file containing 32-bit words. Or you can use a ROM width value of 16 to get two files, each containing 16 bits of each word.

The default ROM width that the hex conversion utility uses depends on the output format:

- All hex formats except TI-Tagged are configured as lists of 8-bit bytes; the default ROM width for these formats is 8 bits.

- TI-Tagged is a 16-bit format; the default ROM width for TI-Tagged is 16 bits.

Note: The TI-Tagged Format is 16 Bits Wide You cannot change the ROM width of the TI-Tagged format. The TI-Tagged format supports a 16-bit ROM width only.

Note: TI-TXT Format is 8 Bits Wide You cannot change the ROM width of the TI-TXT format. The TI-TXT hex format supports only an 8-bit ROM width. See *TI-TXT Hex Format (--ti_txt Option)* for more about using the ROMS directive with the TI-TXT hex format.

You can change ROM width (except for TI-Tagged and TI-TXT formats) by:

- Using the --romwidth option. This option changes the ROM width value for the entire object file.

- Setting the **romwidth** parameter of the ROMS directive. This parameter changes the ROM width value for a specific ROM address range and overrides the --romwidth option for that range. See *The ROMS Directive*.

For both methods, use a value that is a power of 2 greater than or equal to 8.

If you select a ROM width that is wider than the natural size of the output format, the utility simply writes multibyte fields into the file. The --romwidth option is ignored for the TI-TXT and TI-Tagged formats.

Memory width and ROM width are used only for grouping the object file data; they do not represent values. Thus, the byte ordering of the object file data is maintained throughout the conversion process. To refer to the partitions within a memory word, the bits of the memory word are always numbered from right to left as follows:

```
--memwidth=32
A A B B C C D D 1 1 2 2 3 3 4 4
31                            0
```
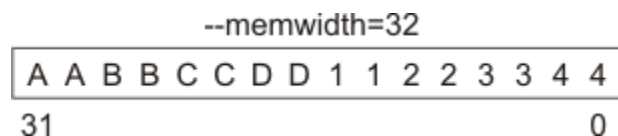
The following figure shows how the object file data, memory, and ROM widths are related to one another.

Figure: Data, Memory, and ROM Widths

Source file

.word   0 AABBCCDDh
.word   011223344h
• • •

Object file data (assumed to be in big-endian format)

AA	BB	CC	DD
11	22	33	44

• • •

Memory widths (variable)

--memwidth=32

AABBCCDD
11223344

• • •

--memwidth=16

AABB
CCDD
1122
3344

• • •

--memwidth=8

AA
BB
CC
DD
11
22
33
44

• • •

Data after phase I of armhex

Output files

--romwidth=8

--outfile=file.b0 | DD 44 | • • •

--outfile=file.b1 | CC 33 | • • •

--outfile=file.b2 | BB 22 | • • •

--outfile=file.b3 | AA 11 | • • •

--romwidth=16

--outfile=file.wrd | AABBCCDD11223344 | • • •

--romwidth=8

--outfile=file.b0 | BB DD 22 44 | • • •

--outfile=file.b1 | AA CC 11 33 | • • •

--romwidth=8

--outfile=file.byt | AABBCCDD11223344 | • • •

Data after phase II of armhex

3.16.4 The ROMS Directive

The ROMS directive specifies the physical memory configuration of your system as a list of address-range parameters.

Each address range produces one set of files containing the hex conversion utility output data that corresponds to that address range. Each file can be used to program one single ROM device.

The ROMS directive is similar to the MEMORY directive of the Arm linker: both define the memory map of the target address space. Each line entry in the ROMS directive defines a specific address range. The general syntax is:

```
ROMS
{
    romname : [origin=value,] [length=value,] [romwidth=value,]
              [memwidth=value,] [fill=value]
              [files={ filename, filename, ... }]

    romname : [origin=value,] [length=value,] [romwidth=value,]
              [memwidth=value,] [fill=value]
              [files={ filename, filename, ... }]

    ...
}
```

- **ROMS** begins the directive definition.

- *romname* identifies a memory range. The name of the memory range can be one to eight characters in length. The name has no significance to the program; it simply identifies the range, except when the output is for a load image in which case it denotes the section name. (Duplicate memory range names are allowed.)

- **origin** | specifies the starting address of a memory range. It can be entered as origin, org, or o. The associated value must be a decimal, octal, or hexadecimal constant. If you omit the origin value, the origin defaults to 0. The following table summarizes the notation you can use to specify a decimal, octal, or hexadecimal constant:

Constant	Notation	Example
Hexadecimal	0x prefix or h suffix	0x77 or 077h
Octal	0 prefix	077
Decimal	No prefix or suffix	77

- **length** | specifies the length of a memory range as the physical length of the ROM device. It can be entered as length, len, or l. The value must be a decimal, octal, or hexadecimal constant. If you omit the length, it defaults to the length of the entire address space.

- **romwidth** | specifies the physical ROM width of the range in bits (see *Partitioning Data Into*

Output Files). Any value you specify here overrides the --romwidth option. The value must be a decimal, octal, or hexadecimal constant that is a power of 2 greater than or equal to 8.

- **memwidth** specifies the memory width of the range in bits (see *Specifying the Memory Width*). Any value you specify here overrides the --memwidth option. The value must be a decimal, octal, or hexadecimal constant that is a power of 2 greater than or equal to 8. *When using the memwidth parameter, you must also specify the paddr parameter for each section in the SECTIONS directive.* (See *The SECTIONS Directive*.)

- **fill** specifies a fill value to use for the range. In image mode, the hex conversion utility uses this value to fill any holes between sections in a range. A hole is an area between the input sections that comprises an output section that contains no actual code or data. The fill value must be a decimal, octal, or hexadecimal constant with a width equal to the target width. Any value you specify here overrides the --fill option. When using fill, you must also use the --image command line option. (See *Specifying a Fill Value*.)

- **files** identifies the names of the output files that correspond to this range. Enclose the list of names in curly braces and order them from *least significant* to *most significant* output file, where the bits of the memory word are numbered from right to left. The number of file names must equal the number of output files that the range generates. To calculate the number of output files, see *Partitioning Data Into Output Files*. The utility warns you if you list too many or too few filenames.

Unless you are using the --image option, all of the parameters that define a range are optional; the commas and equal signs are also optional. A range with no origin or length defines the entire address space. In image mode, an origin and length are required for all ranges.

Ranges must not overlap and must be listed in order of ascending address.

When to Use the ROMS Directive

If you do not use a ROMS directive, the utility defines a single default range that includes the entire address space. This is equivalent to a ROMS directive with a single range without origin or length.

Use the ROMS directive when you want to:

- **Program large amounts of data into fixed-size ROMs**. When you specify memory ranges corresponding to the length of your ROMs, the utility automatically breaks the output into blocks that fit into the ROMs.

- **Restrict output to certain segments**. You can also use the ROMS directive to restrict the conversion to a certain segment or segments of the target address space. The utility does not convert the data that falls outside of the ranges defined by the ROMS directive. Sections can span range boundaries; the utility splits them at the boundary into multiple ranges. If a section falls completely outside any of the ranges you define, the utility does not convert that section and issues no messages or warnings. Thus, you can exclude sections without listing them by name with the SECTIONS directive. However, if a section falls partially in a range

---

and partially in unconfigured memory, the utility issues a warning and converts only the part within the range.

- **Use image mode.** When you use the --image option, you must use a ROMS directive. Each range is filled completely so that each output file in a range contains data for the whole range. Holes before, between, or after sections are filled with the fill value from the ROMS directive, with the value specified with the --fill option, or with the default value of 0.

An Example of the ROMS Directive

The ROMS directive in the following example shows how 16K bytes of 16-bit memory could be partitioned for two 8K-byte 8-bit EPROMs. The figure that follows the code illustrates the input and output files.

Example: A ROMS Directive Example

```
infile.out
--image
--memwidth 16

ROMS
{
    EPROM1: org = 0x00004000, len = 0x2000, romwidth = 8
            files = { rom4000.b0, rom4000.b1}
    EPROM2: org = 0x00006000, len = 0x2000, romwidth = 8,
            fill = 0xFF00FF00,
            files = { rom6000.b0, rom6000.b1}
}
```

Figure: The infile.out File Partitioned Into Four Output Files
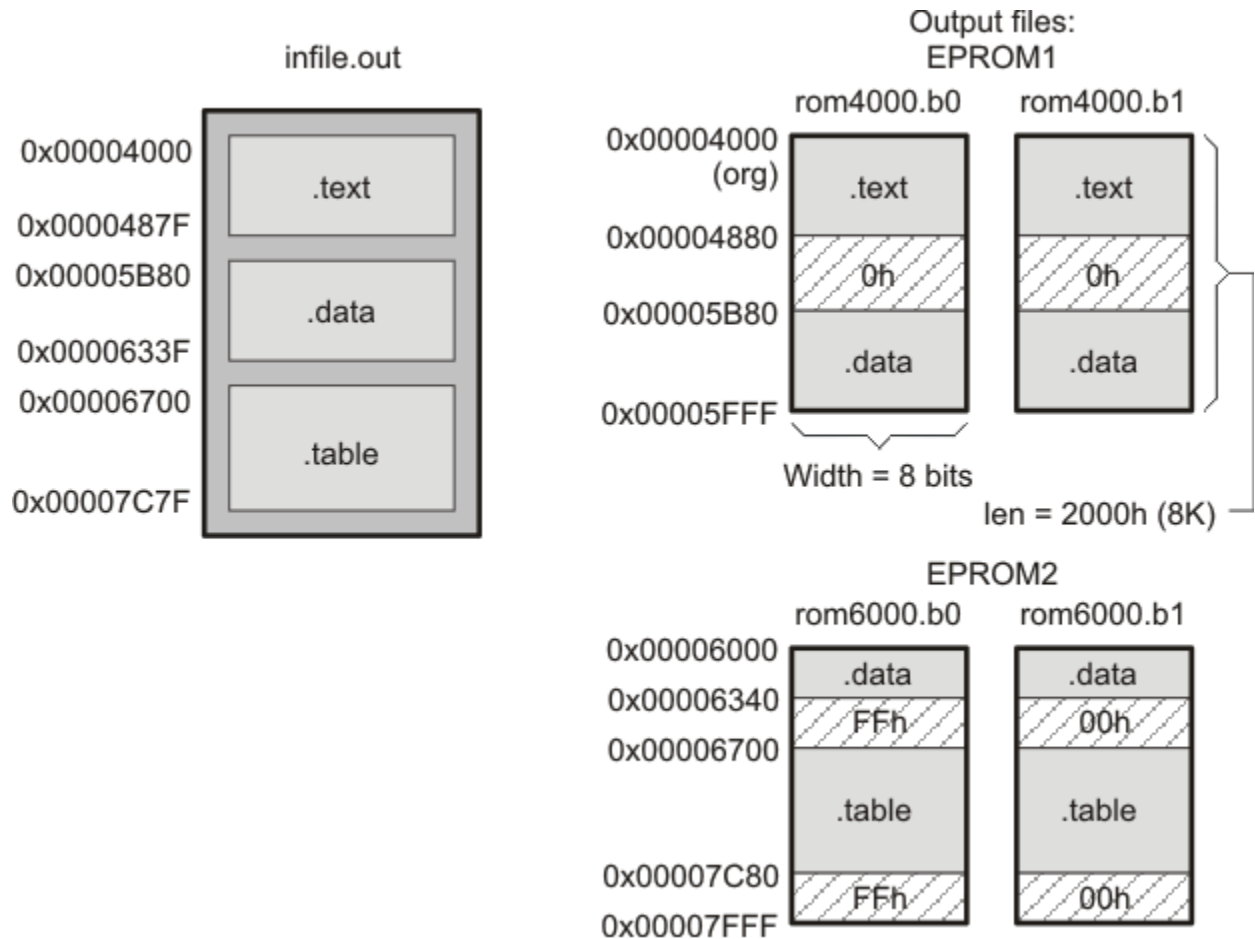
The map file (specified with the --map option) is advantageous when you use the ROMS directive with multiple ranges. The map file shows each range, its parameters, names of associated output files, and a list of contents (section names and fill values) broken down by address. The following example is a segment of the map file resulting from the previous ROMS directive.

Example: Map File Output Showing Memory Ranges From ROMS Example

```
-----------------------------------------------------------
00004000..00005fff Page=0 Width=8 "EPROM1"
-----------------------------------------------------------
   OUTPUT FILES:    rom4000.b0    [b0..b7]
                    rom4000.b1    [b8..b15]
   CONTENTS: 00004000..0000487f .text
             00004880..00005b7f FILL = 00000000
             00005b80..00005fff .data
-----------------------------------------------------------
00006000..00007fff Page=0 Width=8 "EPROM2"
-----------------------------------------------------------
   OUTPUT FILES:    rom6000.b0    [b0..b7]
                    rom6000.b1    [b8..b15]
```

```
CONTENTS:  00006000..0000633f .data
           00006340..000066ff FILL = ff00ff00
           00006700..00007c7f .table
           00007c80..00007fff FILL = ff00ff00
```

EPROM1 defines the address range from 0x00004000 through 0x00005FFF with the following sections:

This section …	Has this range …
.text	0x00004000 through 0x0000487F
.data	0x00005B80 through 0x00005FFF

The rest of the range is filled with 0h (the default fill value), converted into two output files:

- rom4000.b0 contains bits 0 through 7
- rom4000.b1 contains bits 8 through 15

EPROM2 defines the address range from 0x00006000 through 0x00007FFF with the following sections:

This section …	Has this range …
.data	0x00006000 through 0x0000633F
.table	0x00006700 through 0x00007C7F

The rest of the range is filled with 0xFF00FF00 (from the specified fill value). The data from this range is converted into two output files:

- rom6000.b0 contains bits 0 through 7
- rom6000.b1 contains bits 8 through 15

3.16.5 The SECTIONS Directive

You can convert specific sections of the object file by name with the hex conversion utility SECTIONS directive. You can also specify those sections that you want to locate in ROM at a different address than the *load* address specified in the linker command file. If you:

- Use a SECTIONS directive, the utility converts only the sections that you list in the directive and ignores all other sections in the object file.

- Do not use a SECTIONS directive, the utility converts all initialized sections that fall within the configured memory.

Uninitialized sections are *never* converted, whether or not you specify them in a SECTIONS directive.

Note: **Sections Generated by the C/C++ Compiler** The Arm C/C++ compiler automatically generates these sections:

- **Initialized sections:** .text, .const, .rodata, .cinit, and .switch

- **Uninitialized sections:** .bss, .stack, and .sysmem

Use the SECTIONS directive in a command file. (See *Invoking the Hex Conversion Utility With a Command File*.) The general syntax is:

```
SECTIONS
{

    oname(sname)[:]  [paddr=value]
    oname(sname)[:]  [paddr=boot]
    oname(sname)[:]  [boot]
    ...

}
```

- **SECTIONS** begins the directive definition.

- *oname* identifies the object filename the section is located within. The filename is optional when only a single input file is given, but required otherwise.

- *sname* identifies a section in the input file. If you specify a section that does not exist, the utility issues a warning and ignores the name.

- **paddr=***value* specifies the physical ROM address at which this section should be located. This value overrides the section load address given by the linker. This value must be a decimal, octal, or hexadecimal constant. It can also be the word **boot** (to indicate a boot table section for use with a boot loader). *If your file contains multiple sections, and if one section uses a paddr parameter, then all sections must use a paddr parameter.*

- **boot** configures a section for loading by a boot loader. This is equivalent to using **paddr=boot**. Boot sections have a physical address determined by the location of the boot table. The origin of the boot table is specified with the --bootorg option.

For more similarity with the linker's SECTIONS directive, you can use colons after the section names (in place of the equal sign on the boot keyboard). For example, the following statements are equivalent:

```
SECTIONS { .text: .data: boot }
SECTIONS { .text: .data = boot }
```

In the example below, the object file contains six initialized sections: .text, .data, .const, .rodata, .vectors, .coeff, and .tables. Suppose you want only .text and .data to be converted. Use a SECTIONS directive to specify this:

```
SECTIONS { .text: .data: }
```

To configure both of these sections for boot loading, add the boot keyword:

```
SECTIONS { .text = boot .data = boot }
```

3.16.6 The Load Image Format (--load_image Option)

A load image is an object file which contains the load addresses and initialized sections of one or more executable files. The load image object file can be used for ROM masking or can be relinked in a subsequent link step.

Several command-line options allow you to control the format of the file produced when --load_image is used. Options allow you to do the following:

- Create a relocatable or executable output file with the --load_image:file_type option.

- Specify the ABI, machine type, and endianness with the --load_image:format, --load_image:machine, and --load_image:endian options, respectively.

- Combine sections, add a prefix to section names, or include load addresses in the output file with the --load_image:combine_sections, --load_image:section_prefix, and --load_image:section_addresses options.

- Choose whether to output symbols and specify their binding in the output with the --load_image:output_symbols and --load_image:symbol_binding options.

- Control whether individual symbols are local or global with the -load_image:localize and --load_image:globalize options.

These command-line options are described in *Hex Conversion Utility Options*.

Load Image Section Formation

The load image sections are formed by collecting the initialized sections from the input executables. There are two ways the load image sections are formed:

- **Using the ROMS Directive**. Each memory range that is given in the ROMS directive denotes a load image section. The romname is the section name. The origin and length parameters are required. The memwidth, romwidth, and files parameters are invalid and are ignored. When using the ROMS directive and the load_image option, the --image option is required.

- **Default Load Image Section Formation**. If no ROMS directive is given, the load image sections are formed by combining contiguous initialized sections in the input executables.

Sections with gaps smaller than the target word size are considered contiguous. The default section names are image_1, image_2, and so on. If another prefix is desired, the --load_image:section_prefix=prefix option can be used.

Load Image Characteristics

All load image sections are initialized data sections. In the absence of a ROMS directive, the load/run address of the load image section is the load address of the first input section in the load image section. If the SECTIONS directive was used and a different load address was given using the paddr parameter, this address will be used.

The load image format always creates a single load image object file. The format of the load image object file is determined based on the input files. The file is not marked executable and does not contain an entry point. The default load image object file name is `ti_load_image.o`. This can be changed using the --outfile option. Only one --outfile option is valid when creating a load image, all other occurrences are ignored.

Note: Concerning Load Image Format These options are invalid when creating a load image:

- --memwidth

- --romwidth

- --zero

- --byte

If a boot table is being created, either using the SECTIONS directive or the --boot option, the ROMS directive must be used.

3.16.7 Excluding a Specified Section

The --exclude *section_name* option can be used to inform the hex utility to ignore the specified section. If a SECTIONS directive is used, it overrides the --exclude option.

For example, if a SECTIONS directive containing the section name *mysect* is used and an --exclude *mysect* is specified, the SECTIONS directive takes precedence and *mysect* is not excluded.

The --exclude option has a limited wildcard capability. The * character can be placed at the beginning or end of the name specifier to indicate a suffix or prefix, respectively. For example, --exclude sect* disqualifies all sections that begin with the characters sect.

If you specify the --exclude option on the command line with the * wildcard, use quotes around the section name and wildcard. For example, --exclude"sect*". Using quotes prevents the * from being interpreted by the hex conversion utility. If --exclude is in a command file, do not use quotes.

If multiple object files are given, the object file in which the section to be excluded can be given in the form oname(sname). If the object filename is not provided, all sections matching the section name are excluded. Wildcards cannot be used for the filename, but can appear within the parentheses.

3.16.8 Assigning Output Filenames

When the hex conversion utility translates your object file into a data format, it partitions the data into one or more output files. When multiple files are formed by splitting memory words into ROM words, *filenames are always assigned in order from least to most significant*, where bits in the memory words are numbered from right to left. This is true, regardless of target or endian ordering.

The hex conversion utility follows this sequence when assigning output filenames:

1. **It looks for the ROMS directive.** If a file is associated with a range in the ROMS directive and you have included a list of files (files = { . . . }) on that range, the utility takes the filename from the list. For example, assume that the target data is 32-bit words being converted to four files, each eight bits wide. To name the output files using the ROMS directive, you could specify the following. The utility will create the output files by writing the least significant bits to xyz.b0 and the most significant bits to xyz.b3:

```
ROMS
{
    RANGE1: romwidth=8, files={ xyz.b0 xyz.b1 xyz.b2 xyz.b3␣
↪}
}
```

2. **It looks for the --outfile options.** You can specify names for the output files by using the --outfile option. If no filenames are listed in the ROMS directive and you use --outfile options, the utility takes the filename from the list of --outfile options. The following line has the same effect as the example ROMS directive above:

```
--outfile=xyz.b0 --outfile=xyz.b1 --outfile=xyz.b2 --
↪outfile=xyz.b3
```

3. **It assigns a default filename.** If you specify no filenames or fewer names than output files, the utility assigns a default filename. A default filename consists of the base name from the input file plus a 2- to 3-character extension. The extension has three parts:

   1. A format character, based on the output format (see *Description of the Object Formats*):

      - **a** for ASCII-Hex

      - **i** for Intel

      - **m** for Motorola-S

- **t** for TI-Tagged

- **x** for Tektronix

2. The range number in the ROMS directive. Ranges are numbered starting with 0. If there is no ROMS directive, or only one range, the utility omits this character.

3. The file number in the set of files for the range, starting with 0 for the least significant file.

For example, assume a.out is for a 32-bit target processor and you are creating Intel format output. With no output filenames specified, the utility produces four output files named a.i0, a.i1, a.i2, a.i3.

If you include the following ROMS directive when you invoke the hex conversion utility, you would have eight output files:

```
ROMS
{
    range1: o = 0x00001000 l = 0x1000
    range2: o = 0x00002000 l = 0x1000
}
```

These output files . . .	Contain data in these locations . . .
a.i00, a.i01, a.i02, a.i03	0x00001000 through 0x00001FFF
a.i10, a.i11, a.i12, a.i13	0x00002000 through 0x00002FFF

If both the ROMS directive and --outfile options are used together, the ROMS directive overrides the --outfile options.

3.16.9 Image Mode and the --fill Option

This section points out the advantages of operating in image mode and describes how to produce output files with a precise, continuous image of a target memory range.

Generating a Memory Image

With the --image option, the utility generates a memory image by completely filling all of the mapped ranges specified in the ROMS directive.

An object file consists of blocks of memory (sections) with assigned memory locations. Typically, all sections are not adjacent: there are holes between sections in the address space for which there is no data. When such a file is converted *without* the use of image mode, the hex conversion utility bridges these holes by using the address records in the output file to skip ahead to the start of

the next section. In other words, there may be discontinuities in the output file addresses. Some EPROM programmers do not support address discontinuities.

In image mode, there are no discontinuities. Each output file contains a continuous stream of data that corresponds exactly to an address range in target memory. Any holes before, between, or after sections are filled with a fill value that you supply.

An output file converted by using image mode still has address records, because many of the hexadecimal formats require an address on each line. However, in image mode, these addresses are always contiguous.

Note: Defining the Ranges of Target Memory If you use image mode, you must also use a ROMS directive. In image mode, each output file corresponds directly to a range of target memory. You must define the ranges. If you do not supply the ranges of target memory, the utility tries to build a memory image of the entire target processor address space. This is potentially a huge amount of output data. To prevent this situation, the utility requires you to explicitly restrict the address space with the ROMS directive.

Specifying a Fill Value

The --fill option specifies a value for filling the holes between sections. The fill value must be specified as an integer constant following the --fill option. The width of the constant is assumed to be that of a word on the target processor. For example, specifying --fill=0xFFFF results in a fill pattern of 0x0000FFFF. The constant value is not sign extended.

The hex conversion utility uses a default fill value of 0 if you do not specify a value with the fill option. *The --fill option is valid only when you use --image*; otherwise, it is ignored.

Steps to Follow in Using Image Mode

- **Step 1:** Define the ranges of target memory with a ROMS directive. See *The ROMS Directive*.

- **Step 2:** Invoke the hex conversion utility with the --image option. You can optionally use the --zero option to reset the address origin to 0 for each output file. If you do not specify a fill value with the ROMS directive and you want a value other than the default of 0, use the --fill option.

3.16.10 Array Output Format

The --array option causes the output to be generated in C array format. In this format, data contained in initialized sections of an executable file are defined as C arrays. Output arrays may be compiled along with a host program and used to initialize the target at runtime.

Arrays are formed by collecting the initialized sections from the input executable. There are two ways arrays are formed:

- *With the ROMS directive.* Each memory range that is given in the ROMS directive denotes an array. The romname is used as the array name. The origin and length parameters of the ROM directive are required. The memwidth, romwidth, and files parameters are invalid and are ignored.

- *No ROMS directive (default).* If no ROMS directive is given, arrays are formed by combining initialized sections within each page, beginning with the first initialized section. Arrays will reflect any gaps that exist between sections.

The --array:name_prefix option can be used to override the default prefix for array names. For example, use --array:name_prefix=myarray to cause the prefix of myarray to be used.

The data type for array elements is uint8_t.

3.16.11 Building a Table for an On-Chip Boot Loader

The Arm hex utility provides the ability to create a boot table for use with an on-chip boot loader. The supported boot formats are intended for use on C28x devices with Arm cores. The boot table is stored in memory or loaded from a device peripheral to initialize code or data.

See *Bootstrap Loading* for a general discussion of bootstrap loading.

Description of the Boot Table

The input for a boot loader is the boot table. The boot table contains records that instruct the on-chip loader to copy blocks of data contained in the table to specified destination addresses. The table can be stored in memory (such as EPROM) or read in through a device peripheral (such as a serial or communications port).

The hex conversion utility automatically builds the boot table for the boot loader. Using the utility, you specify the sections you want the boot loader to initialize and the table location. The hex conversion utility builds a complete image of the table according to the format specified and converts it into hexadecimal in the output files. Then, you can burn the table into ROM or load it by other means.

The Boot Table Format

The boot table format is simple. Typically, there is a header record containing a key value that indicates memory width, entry point, and values for control registers. Each subsequent block has a header containing the size and destination address of the block followed by data for the block. Multiple blocks can be entered. The table ends with a header containing size zero.

How to Build the Boot Table

The following list summarizes the hex conversion utility options available for the boot loader.

--boot

Syntax: --boot

Converts all sections into boot table form (use instead of a SECTIONS directive).

--bootorg

Syntax: --bootorg=*value*

Specifies the source address of the boot-loader table.

--boot_align_sect

Syntax: --boot_align_sect

Causes boot table records to be aligned to the section alignment.

--boot_block_size=size

Syntax: --boot_block_size=*size*

Set the desired block size for output from the hex conversion utility. The size may be specified as a hex or decimal value. The default size is 65535 (0xFFFF). For ARM targets, the boot block size refers to 8-bit bytes.

--entrypoint

Syntax: --entrypoint=*value*

Specifies the entry point at which to begin execution after boot loading. The *value* can be an address or a global symbol.

--gpio8

Syntax: --gpio8

Specifies the source of the boot-loader table as the GP I/O port, 8-bit mode.

--gpio16

Syntax: --gpio16

Specifies the source of the boot-loader table as the GP I/O port, 16-bit mode.

--lospcp

Syntax: --lospcp=*value*

Specifies the initial value for the LOSPCP register. The value is used only for the spi8 boot table format and is ignored for all other formats. A value greater than 0x7F is truncated to 0x7F.

--spi8

Syntax: --spi8

Specifies the source of the boot-loader table as the SPI-A port, 8-bit mode.

--spibrr

Syntax: --spibrr=*value*

Specifies the initial value for the SPIBRR register. The value is used only for the spi8 boot table format and is ignored for all other formats. A value greater than 0x7F is truncated to 0x7F.

Building the Boot Table

To build the boot table, follow these steps:

- **Step 1: Link the file.** Each block of the boot table data corresponds to an initialized section in the object file. Uninitialized sections are not converted by the hex conversion utility (see *The SECTIONS Directive*).

  When you select a section for placement in a boot-loader table, the hex conversion utility places the section's *load address* in the destination address field for the block in the boot table. The section content is then treated as raw data for that block. *The hex conversion utility does not use the section run address.* When linking, you need not worry about the ROM address or the construction of the boot table; the hex conversion utility handles this.

- **Step 2: Identify the bootable sections.** You can use the --boot option to tell the hex conversion utility to configure all sections for boot loading. Or, you can use a SECTIONS directive to select specific sections to be configured (see *The SECTIONS Directive*). If you use a SECTIONS directive, the --boot option is ignored.

- **Step 3: Set the boot table format.** Specify the --gpio8, --gpio16, or --spi8 options to set the source format of the boot table. You do not need to specify the memwidth and romwidth as the utility will set these formats automatically. If --memwidth and --romwidth are used after a format option, they override the default for the format.

- **Step 4: Set the ROM address of the boot table.** Use the --bootorg option to set the source address of the complete table.

- **Step 5: Set boot-loader-specific options.** Set entry point and control register values as needed.

- **Step 6: Describe your system memory configuration.** See *Understanding Memory Widths* and *The ROMS Directive*.

Leaving Room for the Boot Table

The complete boot table is similar to a single section containing all of the header records and data for the boot loader. The address of this section is the boot table origin. As part of the normal conversion process, the hex conversion utility converts the boot table to hexadecimal format and maps it into the output files like any other section.

Be sure to leave room in your system memory for the boot table, especially when you are using the ROMS directive. The boot table cannot overlap other nonboot sections or unconfigured memory. Usually, this is not a problem; typically, a portion of memory in your system is reserved for the boot table. Simply configure this memory as one or more ranges in the ROMS directive, and use the --bootorg option to specify the starting address.

Booting From a Device Peripheral

You can choose the port to boot from by using the --gpio8, --gpio16, or --spi8 boot table format option.

The initial value for the LOSPCP register can be specified with the --lospcp option. The initial value for the SPIBRR register can be specified with the --spibrr option. Only the --spi8 format uses these control register values in the boot table.

If the register values are not specified for the --spi8 format, the hex conversion utility uses the default values 0x02 for LOSPCP and 0x7F for SPIBRR. When the boot table format options are specified and the ROMS directive is not specified, the ASCII format hex utility output does not produce the address record.

Setting the Entry Point for the Boot Table

After completing the boot load process, execution starts at the default entry point specified by the linker and contained in the object file. By using the --entrypoint option with the hex conversion utility, you can set the entry point to a different address.

For example, if you want your program to start running at address 0x0123 after loading, specify --entrypoint=0x0123 on the command line or in a command file. You can determine the --entrypoint address by looking at the map file that the linker generates.

Note: Valid Entry Points The value can be a constant, or it can be a symbol that is externally defined (for example, with a .global) in the assembly source.

Using the Arm Boot Loader

This subsection explains how to use the hex conversion utility with the boot loader for C28x devices with Arm cores. The boot loader accepts the formats listed in the following table.

Boot Table Source Formats

Format	Option
Parallel boot GP I/O 8 bit	--gpio8
Parallel boot GP I/O 16 bit	--gpio16
8-bit SPI boot	--spi8

The Arm on C28x devices with Arm cores can boot through the SPI-A 8-bit, GP I/O 8-bit, or GP I/I 16-bit interface. The format of the boot table is shown in the following table.

Boot Table Format

Description	Bytes	Content
Boot table header	1-2	Key value (0x10AA or 0x08AA)
	3-18	Register initialization value or reserved for future use
	19-22	Entry point
Block header	23-24	Block size in number of bytes (nl)
	25-28	Destination address of the block
Block data	29-30	Raw data for the block (nl bytes)
Block header	31 + nl	Block size in number of bytes
	.	Destination address of the block
Block data	.	Raw data for the block
Additional block headers and data, as required	...	Content as appropriate
Block header with size 0		0x0000; indicates the end of the boot table.

The Arm on C28x devices with Arm cores can boot through either the serial 8-bit or parallel interface with either 8- or 16-bit data. The format is the same for any combination: the boot table consists of a field containing the destination address, a field containing the length, and a block containing the data. You can boot only one section. If you are booting from an 8-bit channel, 8-bit bytes are stored in the table with MSBs first; the hex conversion utility automatically builds the table in the correct format. Use the following options to specify the boot table source:

- To boot from a SPI-A port, specify --spi8 when invoking the utility. Do not specify --memwidth or --romwidth. Use --lospcp to set the initial value for the LOSPCP register and --spibrr to set the initial value for the SPIBRR register. If the register values are not specified for the --spi8 format, the hex conversion utility uses the default value 0x02 for LOSPCP and 0x7F for SPIBRR.

- To load from a general-purpose parallel I/O port, invoke the utility with --gpio8 or --gpio16. Do not specify --memwidth or --romwidth.

The command file in the following example allows you to boot the .text and .cinit sections of test.out from a 16-bit-wide EPROM at location 0x3FFC00. The map file test.map is also generated.

Example: Sample Command File for Booting From 8-Bit SPI Boot

```
/*-------------------------------------------------------------
↪------------*/
/* Hex converter command file.                                 ␣
 ↪            */
/*-------------------------------------------------------------
↪------------*/
test.out                 /* Input file */
--ascii                  /* Select ASCII format */
--map=test.map           /* Specify the map file */
--outfile=test_spi8.hex  /* Hex utility out file */
--boot                   /* Consider all the input sections as␣
↪boot sections */
--spi8                   /* Specify the SPI 8-bit boot format */
--lospcp=0x3F            /* Set the initial value for the LOSPCP␣
↪as 0x3F */
                         /* The -spibrr option is not specified␣
↪to show that */
                         /* the hex utility uses the default␣
↪value (0x7F) */
--entrypoint=0x3F0000    /* Set the entry point */
```

The command file in the example above generates the out file in the following figure. The control register values are coded in the boot table header and that header has the address that is specified with the --entrypoint option.

Figure: Sample Hex Converter Out File for Booting From 8-Bit SPI Boot
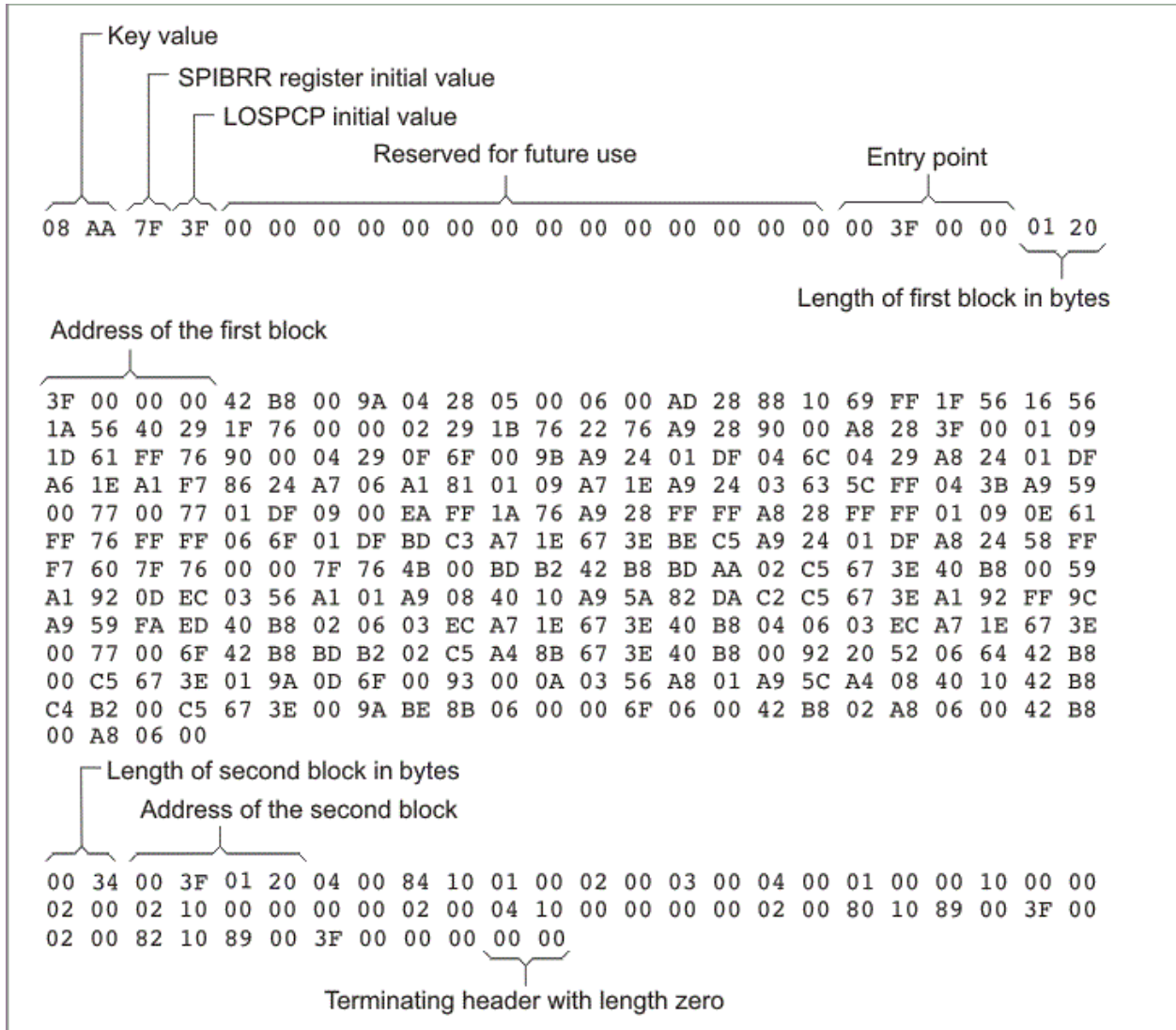
The command file in the following example allows you to boot the .text and .cinit sections of test.out from the 16-bit parallel GP I/O port. The map file test.map is also generated.

Example: Sample Command File for Arm 16-Bit Parallel Boot GP I/O

```
/*----------------------------------------------------------------
↪------*/
/* Hex converter command file.                                    ␣
↪      */
/*----------------------------------------------------------------
↪------*/
test.out                    /* Input file */
--ascii                     /* Select ASCII format */
--map=test.map              /* Specify the map file */
--outfile=test_gpio16.hex   /* Hex utility out file */
```

```
--gpio16                         /* Specify the 16-bit GP I/O boot␣
↪format */

SECTIONS
{
    .text: paddr=BOOT
    .cinit: paddr=BOOT
}
```

The command file above generates the out file shown below:

Figure: Sample Hex Converter Out File for Arm 16-Bit Parallel Boot GP I/O

3.16.12 Using Secure Flash Boot on TMS320F2838x Devices

The hex conversion utility supports the secure flash boot capability provided by TMS320F2838x devices, which have both C28 and Arm cores. The secure flash boot applies the Cipher-based Message Authentication Protocol (CMAC) algorithm to verify CMAC tags for regions of allocated memory.

Secure flash boot is similar to the regular flash boot mode in that the boot flow branches to the configured memory address in flash. The difference is that this branch occurs only after the flash memory contents have been authenticated. The flash authentication uses CMAC to authenticate 16 KB of flash. The CMAC calculation requires a 128-bit key that you define. Additionally, you must calculate a golden CMAC tag based on the 16 KB flash memory range and store it along with the application code at a hardcoded address in flash. During secure flash boot, the calculated CMAC tag is compared to the golden CMAC tag in flash to determine the pass/fail status of the CMAC authentication. If authentication passes, the boot flow continues and branches to flash to begin executing the application. See the *TMS320F2838x Microcontrollers Technical Reference Manual* (SPRUII0) for further details about secure flash boot and the CMAC algorithm.

In order to apply the CMAC algorithm to the appropriate regions in allocated memory, use the hex conversion utility as follows:

- Use the --cmac=file option. The file should contain a 128-bit hex CMAC key.

  The CMAC key in the file specified by the --cmac command-line option must use the format 0xkey0key1key2key3 in order to access the device registers for CMACKEY0-3. For example, the following file contents represent CMACKEY registers containing key0=0x7c0b7db9, key1=`0x811f10d0`, key2=`0x0e476c7a`, and key3=`0x0d92f6e0`:

  ```
  0x7c0b7db9811f10d00e476c7a0d92f6e0
  ```

- Use either the --image option or the --load_image option when using the --cmac option. If you use the --image option, set both --memwidth and --romwidth to the same value.

- If you use the --boot option (and other boot table options described in *Building a Table for an On-Chip Boot Loader*) with the --cmac option, the CMAC algorithm assumes that a fill value of 1 is used for gaps between boot table regions. Because of this assumption, you should also set --fill=0xFFFFFFFF when using the --boot and --cmac options together.

- Specify a HEX directive with one entry that represents all the allocated flash memory. Use a 128-bit aligned length and specify the optional fill value. (The default fill is set to 0's.)

- Define the global CMAC tags in C code.

The CMAC feature uses four secure flash boot memory regions that are hardcoded for start/end/tag addresses, and one flexible CMAC region. The flexible region can encompass the entire allocated region as input in the HEX directive or user-specified start/end addresses defined in C code.

C code definitions like the following are required to reserve space for the CMAC tag symbols.

```c
struct   CMAC_TAG
{  uint8_t  tag[16];
   uint32_t start;
   uint32_t end;
};

#pragma RETAIN(cmac_sb_1)
#pragma LOCATION(cmac_sb_1, 0x00200004)
const uint8_t cmac_sb_1[16] =  { 0 };

#pragma RETAIN(cmac_sb_2)
#pragma LOCATION(cmac_sb_2, 0x00210004)
const uint8_t cmac_sb_2[16] =  { 0 };

#pragma RETAIN(cmac_sb_3)
#pragma LOCATION(cmac_sb_3, 0x00250004)
const uint8_t cmac_sb_3[16] =  { 0 };

#pragma RETAIN(cmac_sb_4)
#pragma LOCATION(cmac_sb_4, 0x0027C004)
const uint8_t cmac_sb_4[16] =  { 0 };

#pragma RETAIN(cmac_all)
#pragma LOCATION(cmac_all, 0x00204004)
const struct CMAC_TAG cmac_all =  { { 0 }, 0x0, 0x0};
```

The four secure flash boot region CMAC tags are stored in the *cmac_sb_1* through *cmac_sb_4*
symbols. The *cmac_all* symbol stores the CMAC tag for the flexible user-specified region. For
cmac_all:

- If the *start* and *end* CMAC_TAG struct members are zero, then the CMAC algorithm runs
  over entire memory region specified in the HEX directive. The hex conversion utility pop-
  ulates the start and end memory locations with the addresses input from the HEX directive
  entry.

- If the *start* and *end* members are non-zero, then the CMAC algorithm is instead applied
  between the specified addresses.

RETAIN pragmas are required in the C code if these symbols are not accessed in the application
code.

LOCATION pragmas are required to place symbols at the required memory locations. The LOCA-
TION entries for *cmac_sb_1* through *cmac_sb_4* are at fixed addresses. The LOCATION address
for *cmac_all* can be user-specified. However, it must not be located within any secure flash boot
regions, because the ROM CMAC implementation on the devices does not support this.

The CMAC algorithm is applied prior to the hex conversion. No changes are made to the original input ELF executable.

The hex conversion utility applies the CMAC algorithm only to CMAC regions that have global symbols defined. So if an ELF executable defines only *cmac_sb_1* and *cmac_all*, then only those two CMAC tags will be generated and populated in the generated hex output file.

3.16.13 Controlling the ROM Device Address

The hex conversion utility output address field corresponds to the ROM device address. The EPROM programmer burns the data into the location specified by the hex conversion utility output file address field. The hex conversion utility offers some mechanisms to control the starting address in ROM of each section. However, many EPROM programmers offer direct control of the location in ROM in which the data is burned.

The address field of the hex-conversion utility output file is controlled by the following items, which are listed from low to high priority:

1. **The linker command file.** By default, the address field of the hex conversion utility output file is the load address (as given in the linker command file).

2. **The paddr parameter of the SECTIONS directive.** When the paddr parameter is specified for a section, the hex conversion utility bypasses the section load address and places the section in the address specified by paddr.

3. **The --zero option.** When you use the --zero option, the utility resets the address origin to 0 for each output file. Since each file starts at 0 and counts upward, any address records represent offsets from the beginning of the file (the address within the ROM) rather than actual target addresses of the data.

   You must use the --zero option in conjunction with the --image option to force the starting address in each output file to be zero. If you specify the --zero option without the --image option, the utility issues a warning and ignores the --zero option.

4. **The --byte option.** Some EPROM programmers may require the output file address field to contain a byte count rather than a word count. If you use the byte option, the output file address increments once for each byte. For example, if the starting address is 0h, the first line contains eight words, and you use no byte option, the second line would start at address 8 (8h). If the starting address is 0h, the first line contains eight words, and you use the byte option, the second line would start at address 16 (010h). The data in both examples are the same; byte affects only the calculation of the output file address field, not the actual target processor address of the converted data.

   The --byte option causes the address records in an output file to refer to byte locations within the file, whether the target processor is byte-addressable or not.

3.16.14 Control Hex Conversion Utility Diagnostics

The hex conversion utility uses certain C/C++ compiler options to control hex-converter-generated diagnostics.

--diag_error

> Syntax: --diag_error=*id*
>
> Categorizes the diagnostic identified by *id* as an error. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_error=*id* to recategorize the diagnostic as an error. You can only alter the severity of discretionary diagnostics.

--diag_remark

> Syntax: --diag_remark=*id*
>
> Categorizes the diagnostic identified by *id* as a remark. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_remark=*id* to recategorize the diagnostic as a remark. You can only alter the severity of discretionary diagnostics.

--diag_suppress

> Syntax: --diag_suppress=*id*
>
> Suppresses the diagnostic identified by *id*. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_suppress=*id* to suppress the diagnostic. You can only suppress discretionary diagnostics.

--diag_warning

> Syntax: --diag_warning=*id*
>
> Categorizes the diagnostic identified by *id* as a warning. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_warning=*id* to recategorize the diagnostic as a warning. You can only alter the severity of discretionary diagnostics.

--display_error_number

> Syntax: --display_error_number
>
> Displays a diagnostic message's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (--diag_suppress, --diag_error, --diag_remark, and --diag_warning). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix -D; otherwise, no suffix is present. See *Diagnostic Options* for more information on diagnostic messages.

--issue_remarks

> Syntax: --issue_remarks

Issues remarks (nonserious warnings), which are suppressed by default.

--no_warnings

Syntax: --no_warnings

Suppresses warning diagnostics (errors are still issued).

--set_error_limit

Syntax: --set_error_limit=*count*

Sets the error limit to *count*, which can be any decimal value. The linker abandons linking after this number of errors. (The default is 100.)

--verbose_diagnostics

Syntax: --verbose_diagnostics

Provides verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line.

3.16.15 Description of the Object Formats

The hex conversion utility has options that identify each format. The following table specifies the format options. They are described in the subsections that follow.

- You should use only one of these options on the command line. If you use more than one option, the last one you list overrides the others.

- The default format is Tektronix (--tektronix option).

Table: Options for Specifying Hex Conversion Formats

Option	Alias	Format	Address Bits	Default Width
--array		Array	8	8
--ascii	-a	ASCII-Hex	16	8
--binary	-b	Binary	8	8
--intel	-i	Intel	32	8
--motorola=1	-m1	Motorola-S1	16	8
--motorola=2	-m2	Motorola-S2	24	8
--motorola=3	-m3	Motorola-S3	32	8
--tektronix	-x	Tektronix	32	8
--ti-tagged	-t	TI-Tagged	16	16
--ti_txt		TI_TXT	8	8

Address bits determine how many bits of the address information the format supports. Formats with 16-bit addresses support addresses up to 64K only. The utility truncates target addresses to fit in the number of available bits.

The **default width** determines the default output width of the format. You can change the default width by using the --romwidth option or by using the romwidth parameter in the ROMS directive. You cannot change the default width of the TI-Tagged format, which supports a 16-bit width only.

Array Object Format (--array Option)

See *Array Output Format*.

ASCII-Hex Object Format (--ascii Option)

The ASCII-Hex object format supports 16-bit addresses. The format consists of a byte stream with bytes separated by spaces. The following figure illustrates the ASCII-Hex format.

Figure: ASCII-Hex Object Format
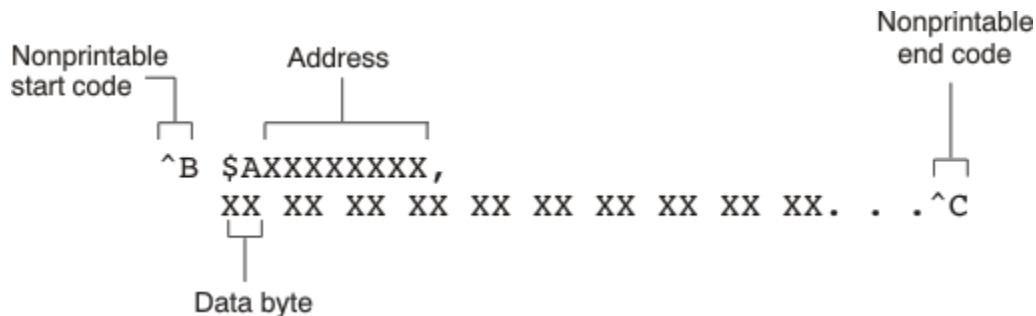
The file begins with an ASCII STX character (ctrl-B, 02h) and ends with an ASCII ETX character (ctrl-C, 03h). Address records are indicated with $AXXXXXXX, in which XXXXXXXX is a 8-digit (16-bit) hexadecimal address. The address records are present only in the following situations:

- When discontinuities occur
- When the byte stream does not begin at address 0

You can avoid all discontinuities and any address records by using the --image and --zero options. This creates output that is simply a list of byte values.

Binary Object Format (--binary Option)

The output is a raw binary file, containing a memory image of the input file. Symbols and relocation information are discarded. The image starts at the address of the first loadable section in the output.

Because the binary format supports only an 8-bit physical memory width and an 8-bit ROM width, the ROMS directive needs to have the origin and length specifications doubled when moving from a 16-bit format to an 8-bit format. If you receive a warning, check the ROMS directive.

Use a ROMS directive to work around the width limitation of the binary format. First, create a file called roms_directive.txt with the following contents:

```
ROMS {
    all_mem: o = 0x100, l = 0x28
}
```

Then, use the --image option to use this ROMS directive.

```
$ tiarmhex --romwidth=32 --binary --image roms_directive.txt␣
→file.out -o binaryFile.bin
```

Using a ROMS directive also prevents the hex utility from skipping holes in the input. Alternately, you can use the **tiarmobjcopy** utility, which does not skip holes in binary output.

Intel MCS-86 Object Format (--intel Option)

The Intel object format supports 16-bit addresses and 32-bit extended addresses. Intel format consists of a 9-character (4-field) prefix (which defines the start of record, byte count, load address, and record type), the data, and a 2-character checksum suffix.

The 9-character prefix represents three record types:

Record Type	Description
00	Data record
01	End-of-file record
04	Extended linear address record

Record type00, the data record, begins with a colon (:) and is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. The address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 04 (extended linear address) record to create a full 32-bit address. The checksum is the 2s complement (in binary form) of the preceding bytes in the record, including byte count, address, and data bytes.

Record type 01, the end-of-file record, also begins with a colon (:), followed by the byte count, the address, the record type (01), and the checksum.

Record type 04, the extended linear address record, specifies the upper 16 address bits. It begins with a colon (:), followed by the byte count, a dummy address of 0h, the record type (04), the most significant 16 bits of the address, and the checksum. The subsequent address fields in the data records contain the least significant bytes of the address.

The following figure illustrates the Intel hexadecimal object format.

Figure: Intel Hexadecimal Object Format

Motorola Exorciser Object Format (--motorola Option)

The Motorola S1, S2, and S3 formats support 16-bit, 24-bit, and 32-bit addresses, respectively. The formats consist of a start-of-file (header) record, data records, and an end-of-file (termination) record. Each record consists of five fields: record type, byte count, address, data, and checksum. The three record types are:

Record Type	Description
S0	Header record
S1	Code/data record for 16-bit addresses (S1 format)
S2	Code/data record for 24-bit addresses (S2 format)
S3	Code/data record for 32-bit addresses (S3 format)
S7	Termination record for 32-bit addresses (S3 format)
S8	Termination record for 24-bit addresses (S2 format)
S9	Termination record for 16-bit addresses (S1 format)

The byte count is the character pair count in the record, excluding the type and byte count itself.

The checksum is the least significant byte of the 1s complement of the sum of the values represented by the pairs of characters making up the byte count, address, and the code/data fields.

The following figure illustrates the Motorola-S object format.

Figure: Motorola-S Format

Extended Tektronix Object Format (--tektronix Option)

The Tektronix object format supports 32-bit addresses and has two types of records:

- **Data records** contain the header field, the load address, and the object code.
- **Termination records** signify the end of a module.

The header field in the data record contains the following information:

Item	# of ASCII Characters	Description
%	1	Data type is Tektronix format
Block length	2	Number of characters in the record, minus the %
Block type	1	6 = data record 8 = termination record
Check-sum	2	A 2-digit hex sum modulo 256 of all values in the record except the % and the checksum itself.

The load address in the data record specifies where the object code will be located. The first digit specifies the address length; this is always 8. The remaining characters of the data record contain the object code, two characters per byte.

The following figure illustrates the Tektronix object format.

Figure: Extended Tektronix Object Format

Texas Instruments SDSMAC (TI-Tagged) Object Format (--ti_tagged Option)

The Texas Instruments SDSMAC (TI-Tagged) object format supports 16-bit addresses, including start-of-file record, data records, and end-of-file record. Each data records consists of a series of small fields and is signified by a tag character:

Tag Character	Description
K	Followed by the program identifier
7	Followed by a checksum
8	Followed by a dummy checksum (ignored)
9	Followed by a 16-bit load address
B	Followed by a data word (four characters)
F	Identifies the end of a data record
*	Followed by a data byte (two characters)

The following figure illustrates the tag characters and fields in TI-Tagged object format.

Figure 3.35: TI-Tagged Object Format

If any data fields appear before the first address, the first field is assigned address 0000h. Address fields may be expressed but not required for any data byte. The checksum field, preceded by the tag character 7, is the 2s complement of the sum of the 8-bit ASCII values of characters, beginning with the first tag character and ending with the checksum tag character (7 or 8). The end-of-file record is a colon (:).

TI-TXT Hex Format (--ti_txt Option)

The TI-TXT hex format supports 8-bit hexadecimal data. It consists of section start addresses, data byte, and an end-of-file character. These restrictions apply:

- The number of sections is unlimited.

- Each hexadecimal start address must be even.

- Each line must have 8 data bytes, except the last line of a section.

- Data bytes are separated by a single space.

- The end-of-file termination tag q is mandatory.

Because the TI-TXT format (along with the binary format) supports only an 8-bit physical memory width and an 8-bit ROM width, the ROMS directive needs to have the origin and length specifications doubled when moving from a 16-bit format to an 8-bit format. If you receive a warning like the following, check the ROMS directive.

```
warning: section file.out(.data) at 07e000000h falls in␣
↪unconfigured memory
```

For example, suppose the ROMS directive for a format that uses 16-bit ROM widths, such as ASCII-Hex with the --romwidth=16 option used, is as follows:

```
/* Memory counted as 16-bit words */
ROMS
{
    FLASH: origin=0x3f000000, length=0x1000
}
```

You would double the address and length in the ROMS directive when using an 8-bit ROM width:

```
/* Memory counted as 8-bit bytes */
ROMS
{
    FLASH: origin=0x7e000000, length=0x2000
}
```

The data record contains the following information:

Item	Description
@ADDR	Hexadecimal start address of a section
DATAn	Hexadecimal data byte
q	End-of-file termination character

Figure: TI-TXT Object Format
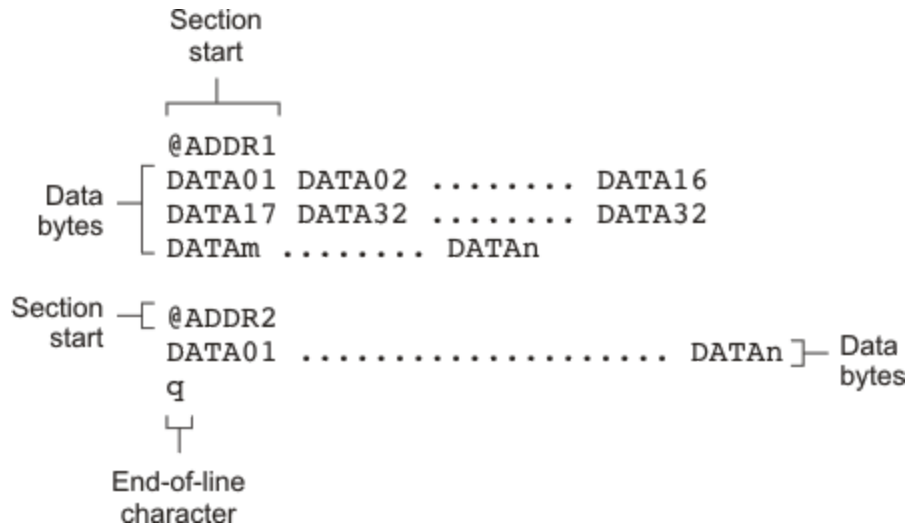
Example: TI-TXT Object Format

```
@F000
31 40 00 03 B2 40 80 5A 20 01 D2 D3 22 00 D2 E3
21 00 3F 40 E8 FD 1F 83 FE 23 F9 3F
@FFFE
00 F0
Q
```

3.16.16 Hex Conversion Utility Examples

The flexible hex conversion utility offers many options and capabilities. Once you understand the proper ways to configure your EPROM system and the requirements of the EPROM programmer, you will find that converting a file for a specific application is easy.

The three scenarios in this appendix show how to develop a hex conversion command file for avoiding holes, using 16-BIS (16-bit instruction set) code, and using multiple-EPROM systems. The scenarios use this assembly code:

```
*************************************************
* Assemble two words into section "secA"        *
*************************************************

    .sect "secA"
    .word 012345678h
    .word 0abcd1234h


*************************************************
* Assemble two words into section "secB"        *
```

```
*************************************************

    .sect "secB"
    .word 087654321h
    .word 04321dcbah
```

Scenario 1 – Hex Conversion Command File for a Single 8-Bit EPROM

Scenario 1 shows how to build the hex conversion command file for converting an object file for the memory system shown in the following figure . In this system, there is one external 128K × 8-bit EPROM interfacing with a TMS470 target processor.

Figure: EPROM Memory System for Scenario 1
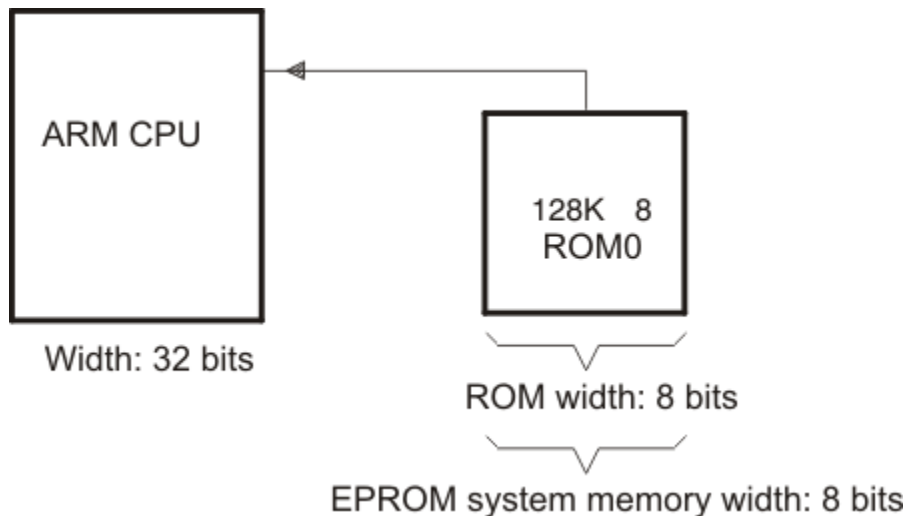
A object file consists of blocks of memory (sections) with assigned memory locations. Typically, all sections are not adjacent: there are holes between sections in the address space for which there is no data. Scenario 1 shows how you can use the hex conversion utility's image mode to fill any holes before, between, or after sections with a fill value.

For this scenario, the application code resides in the program memory (ROM) on the TMS470 CPU, but the data tables used by this code reside in an off-chip EPROM.

The circuitry of the target board handles the access to the data; the native TMS470 address of 0x1000 accesses location 0x0 on the EPROM.

To satisfy the address requirements for the code, this scenario requires a linker command file that allocates sections and memory as follows:

- The program/application code (represented in this scenario by the secA section shown in the following linker command file must be linked so that its address space resides in the program memory (ROM) on the TMS470 CPU.

TI Arm Clang Compiler Tools User Guide, Release v4.0.0.LTS

- To satisfy the condition that the data be loaded on the EPROM at address 0x0 but be referenced by the application code at address 0x1000, secB (the section that contains the data for this application) must be assigned a linker load address of 0x1000 so that all references to data in this section will be resolved with respect to the TMS470 CPU address. In the hex conversion utility command file, the paddr option must be used to burn the section of data at EPROM address 0x0. This value overrides the section load address given by the linker.

The following linker command file shows the MEMORY and SECTIONS directives that resolve the addresses needed in the stated specifications.

Example: Linker Command File and Link Map for Scenario 1

```
/
 ↪********************************************************************
 ↪
/* Scenario 1  Link Command                                         ␣
 ↪          */
/*                                                                  ␣
 ↪          */
/*   Usage: armlnk <obj files...>    -o <out file> -m <map file>␣
 ↪lnk32.cmd  */
/*          tiarmclang <src files...> -Wl,-o=out_file,-m=map_
 ↪file lnk32.cmd */
/*                                                                  ␣
 ↪          */
/*   Description: This file is a sample command file that can be␣
 ↪used       */
/*               for linking programs built with the TMS470 C  ␣
 ↪          */
/*               compiler.  Use it as a guideline; you may␣
 ↪want to change */
/*               the allocation scheme according to the size of␣
 ↪your       */
/*               program and the memory layout of your target␣
 ↪system.      */
/*                                                                  ␣
 ↪          */
/*   Notes: (1)   You must specify the directory in which rts32.
 ↪lib is       */
/*               located.  Either add a "-i<directory>" line to␣
 ↪this        */
/*               file, or use the system environment variable C_
 ↪DIR to      */
/*               specify a search path for libraries.         ␣
 ↪          */
```

(continues on next page)

3.16. Hex Conversion Utility Description 769

```
/*                                                                          ␣
↪             */
/*         (2)   If the runtime-support library you are using␣
↪is not        */
/*               named rts32.lib, be sure to use the correct␣
↪name here.     */
/
↪******************************************************************************
↪
-m example1.map

/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    I_MEM    : org = 0x00000000   len = 0x00000020  /*␣
↪INTERRUPTS         */
    D_MEM    : org = 0x00000020   len = 0x00010000  /* DATA␣
↪MEMORY    (RAM) */
    P_MEM    : org = 0x00010020   len = 0x00100000  /* PROGRAM␣
↪MEMORY (ROM) */
}
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
SECTIONS
{
    secA: load = P_MEM
    secB: load = 0x1000
}
```

You must create a hex conversion command file to generate a hex output with the correct addresses and format for the EPROM programmer.

In the memory system outlined in the figure above, only the application data is stored on the EPROM; the data resides in secB of the object file created by the linker. By default, the hex conversion utility converts all initialized sections that appear in the object file. To prevent the conversion of the application code in secA, a SECTIONS directive must be defined in the hex conversion command file to list explicitly the section(s) to be converted. In this case, secB must be listed explicitly as the section to be converted.

The EPROM programmer in this scenario has the following system requirements:

- The EPROM programmer loads only a complete ROM image. A complete ROM image is one in which there is a contiguous address space (there are no holes in the addresses in the converted file), and each address in the range contains a known value. Creating a complete ROM image requires the use of the image option and the ROMS directive.

- – Using the image option causes the hex conversion utility to create an output file that has contiguous addresses over the specified memory range and forces the utility to fill address spaces that are not previously filled by raw data from sections defined in the input object file. By default, the value used to fill the unused portions of the memory range is 0.

- – Because the -image option operates over a known range of memory addresses, a ROMS directive is needed to specify the origin and length of the memory for the EPROM.

- To burn the section of data at EPROM address 0x0, the paddr option must be used. This value overrides the section load address given by the linker.

- In this scenario, the EPROM is 128K × 8 bits. Therefore, the memory addresses for the EPROM must range from 0x0 to 0x20000.

- Because the EPROM memory width is eight bits, the memwidth value must be set to 8.

- Because the physical width of the ROM device is eight bits, the romwidth value must be set to 8.

- Intel format must be used.

Since memwidth and romwidth have the same value, only one output file is generated (the number of output files is determined by the ratio of memwidth to romwidth). The output file is named with the -o option.

The hex conversion command file for Scenario 1 is shown below. This command file uses the following options to select the requirements of the system:

Option	Description
-i	Create Intel format
-image	Generate a memory image
-map example1.mxp	Generate example1.mxp as the map file of the conversion
-o example1.hex	Name example1.hex as the output file
-memwidth 8	Set EPROM system memory width to 8
-romwidth 8	Set physical ROM width to 8

Example: Hex Conversion Command File for Scenario 1

```
/* Hex Conversion Command file for Scenario 1            */
a.out                    /* linked object file, input    */
-I                       /* Intel format                 */
-image
-map example1.mxp        /* Generate a map of the conversion */
-o example1.hex          /* Resulting hex output file     */
-memwidth 8              /* EPROM memory system width     */
-romwidth 8              /* Physical width of ROM         */
```

```
ROMS
{
    EPROM: origin = 0x0, length = 0x20000
}

SECTIONS
{
    secB: paddr = 0x0   /* Select only section, secB, for␣
 ↪conversion */
}
```

The following example shows the contents of the resulting map file (example1.mxp).

Example: Contents of Hex Map File example1.mxp

```
*********************************************************
TMS470 Hex Converter               Version x.xx
*********************************************************

Mon Sep 18 15:57:00 1995

INPUT FILE NAME: <a.out>
OUTPUT FORMAT:   Intel

PHYSICAL MEMORY PARAMETERS
    Default data width:     8
    Default memory width:   8
    Default output width:   8

OUTPUT TRANSLATION MAP
---------------------------------------------------------
00000000..0001ffff  Page=0  ROM Width=8  Memory Width=8  "EPROM"
---------------------------------------------------------
    OUTPUT FILES: example1.hex [b0..b7]

    CONTENTS: 00000000..00000007  Data Width=1  secB
              00000007..0001ffff   FILL = 00000000
```

The following figure shows the contents of the resulting hex output file (example1.hex). The hex conversion utility places the data tables, secB, at address 0 and then fills the remainder of the address space with the default fill value of 0. For more information about the Intel MCS-86 object format, see *Intel MCS-86 Object Format (--intel Option)*.

Figure: Contents of Hex Output File

Start character
Address   secB data tables

```
:200000008765432143210CBA0000000000000000000000000000000000000000000000000096
:20002000000000000000000000000000000000000000000000000000000000000000000000C0
:2000400000000000000000000000000000000000000000000000000000000000000000000A0
.
.
.
:20FFE0000000000000000000000000000000000000000000000000000000000000000000001
:020000040001F9          Extended linear address record
:200000000000000000000000000000000000000000000000000000000000000000000000E0
:20002000000000000000000000000000000000000000000000000000000000000000000000C0
.
.
.
:20FFC00000000000000000000000000000000000000000000000000000000000000000000021
:20FFE0000000000000000000000000000000000000000000000000000000000000000000001
:00000001FF
```

End-of-file record
Record type
Byte count
Checksum

Scenario 2 – Hex Conversion Command File for 16-BIS Code

Scenario 2 shows how to build the hex conversion command file to generate the correct converted file for the application code and data that will reside on a single 16-bit EPROM. The EPROM memory system for this scenario is shown in the following figure. For this scenario, the TMS470 CPU operates with the T control bit set, so the processor executes instructions in 16-BIS mode.

Figure: EPROM Memory System for Scenario 2

ARM CPU

128K 16
ROM0

Width: 32 bits

ROM width: 16 bits

EPROM system memory width: 16 bits
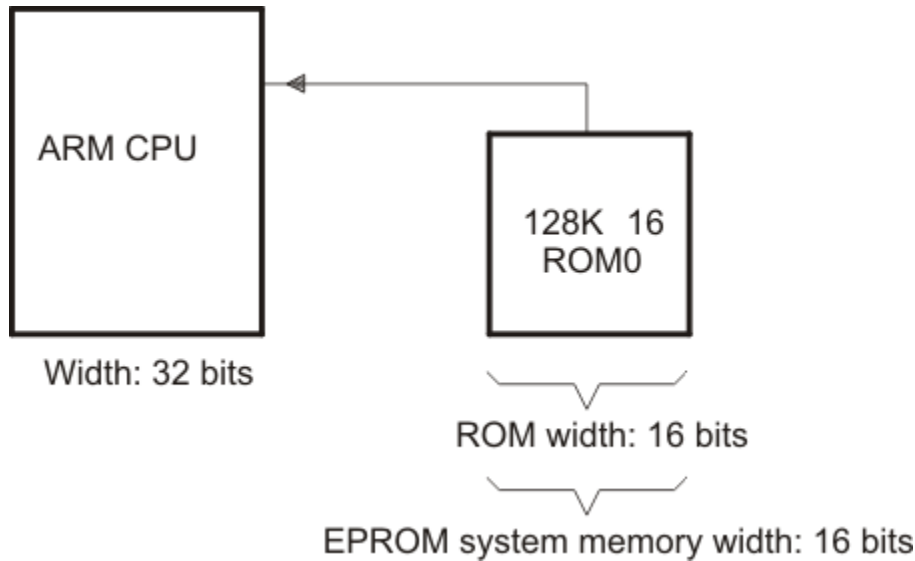
For this scenario, the application code and data reside on the EPROM: the lower 64K words of EPROM memory are dedicated to application code space and the upper 64K words are dedicated

to the data tables. The application code is loaded starting at address 0x0 on the EPROM but maps to the TMS470 CPU at address 0x3000. The data tables are loaded starting at address 0x1000 on the EPROM and map to the TMS470 CPU address 0x20.

The following linker command file contains MEMORY and SECTIONS directives that resolve the addresses needed for the load on EPROM and the TMS470 CPU access.

Example: Linker Command File for Scenario 2

```
/
↪******************************************************************************
↪
/* Scenario 2  Link Command                                              ␣
↪             */
/*                                                                       ␣
↪             */
/*   Usage: armlnk <obj files...>    -o <out file> -m <map file>␣
↪lnk16.cmd   */
/*          tiarmclang <src files...> -Wl,-o=out_file,-m=map_
↪file lnk32.cmd */
/*                                                                       ␣
↪             */
/*   Description: This file is a sample command file that can be␣
↪used       */
/*               for linking programs built with the TMS470 C  ␣
↪             */
/*               compiler.  Use it as a guideline; you may␣
↪want to change */
/*               the allocation scheme according to the size of␣
↪your       */
/*               program and the memory layout of your target␣
↪system.     */
/*                                                                       ␣
↪             */
/*   Notes: (1)  You must specify the directory in which rts16.
↪lib is      */
/*               located.  Either add a "-i<directory>" line to␣
↪this       */
/*               file, or use the system environment variable C_
↪DIR to      */
/*               specify a search path for libraries.         ␣
↪             */
/*                                                                       ␣
↪             */
/*          (2)  If the runtime-support library you are using␣
↪is not       */
```

3.16. Hex Conversion Utility Description 774

```
/*                   named rts16.lib, be sure to use the correct␣
↪name here.    */
/
 ↪********************************************************************␣
 ↪

-m example2.map

/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    I_MEM     : org = 0x00000000   len = 0x00000020   /*␣
↪INTERRUPTS          */
    D_MEM     : org = 0x00000020   len = 0x00010000   /* DATA␣
↪MEMORY     (RAM) */
    P_MEM     : org = 0x00010020   len = 0x00100000   /* PROGRAM␣
↪MEMORY (ROM) */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
SECTIONS
{
    secA: load = 0x3000
    secB: load = 0x20
}
```

You must create a hex conversion command file to generate a hex output with the correct addresses and format for the EPROM programmer. The EPROM programmer in this scenario has the following system requirements:

- Because the EPROM memory width is 16 bits, the memwidth value must be set to 16.

- Because the physical width of the ROM device is 16 bits, the romwidth value must be set to 16.

- Intel format must be used.

The EPROM programmer does not require a ROM image, so the addresses in the input hex output file do not need to be contiguous.

Because memwidth and romwidth have the same value, only one output file is generated (the number of output files is determined by the ratio of memwidth to romwidth). The output file is named with the -o option.

A ROMS directive is used in this scenario since the paddr option is used to relocate both secA and secB.

3.16. Hex Conversion Utility Description   **775**

The hex conversion command file for Scenario 2 is shown in the example below. This command file uses the following options to select the requirements of the system:

Option	Description
-i	Create Intel format
-map example2.mxp	Generate example2.mxp as the map file of the conversion
-o example2.hex	Name example2.hex as the output file
-memwidth 8	Set EPROM system memory width to 8
-romwidth 8	Set physical ROM width to 8

Example: Hex Conversion Command File for Scenario 2

```
/* Hex Conversion Command file for Scenario 2            */
a.out                    /* linked object file, input       */
-I                       /* Intel format                    */

/* The following two options are optional                  */
-map example2.mxp        /* Generate a map of the conversion */
-o example2.hex          /* Resulting Hex Output file        */

/* Specify EPROM system Memory Width and Physical ROM width */
-memwidth 16             /* EPROM memory system width        */
-romwidth 16             /* Physical width of ROM            */

ROMS
{
    EPROM: origin = 0x0, length = 0x20000
}
SECTIONS
{
    secA: paddr = 0x0
    secB: paddr = 0x1000
}
```

The following example shows the contents of the resulting map file (example2.mxp).

Example: Contents of Hex Map File example2.mxp

```
******************************************************
TMS470 Hex Converter              Version x.xx
******************************************************
Mon Sep 18 19:34:47 1995

INPUT FILE NAME: <a.out>
```

```
OUTPUT FORMAT:    Intel

PHYSICAL MEMORY PARAMETERS
     Default data width:      8
     Default memory width:   16
     Default output width:   16

OUTPUT TRANSLATION MAP
------------------------------------------------------------
00000000..0001ffff   Page=0   ROM Width=16   Memory Width=16 "EPROM"
------------------------------------------------------------
     OUTPUT FILES: example2.hex [b0..b15]

     CONTENTS: 00000000..00000003   Data Width=1   secA
               00001000..00001003   Data Width=1   secB
```
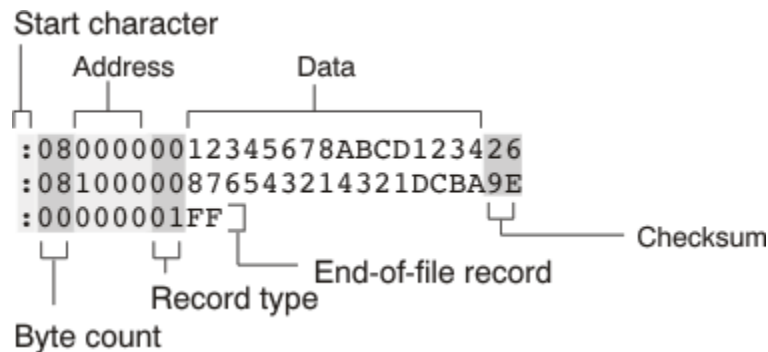
The following figure shows the contents of the resulting hex output file (example2.hex).

Figure: Contents of Hex Output File example2.hex

Scenario 3 – Hex Conversion Command File for Two 8-Bit EPROMs

Scenario 3 shows how to build the hex conversion command file for converting an object file for the memory system shown in the following figure. In this system, there are two external 64K × 16-bit EPROMs interfacing with the TMS470 target processor. The application code and data will be burned on the EPROM starting at address 0x20. The application code will be burned first, followed by the data tables.

Figure: EPROM Memory System for Scenario 3
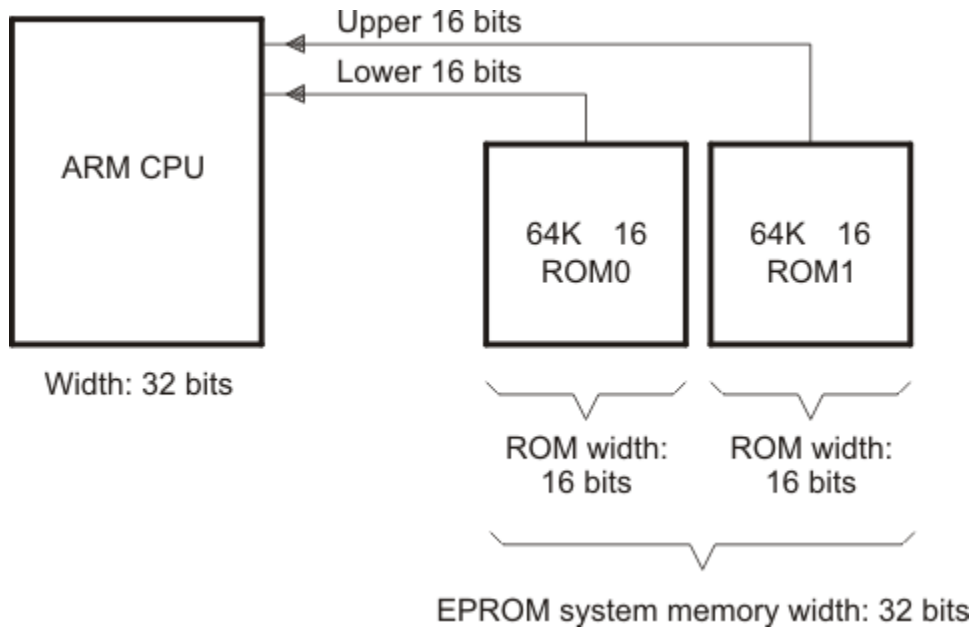
In this scenario, the EPROM load address for the application code and for the data also corresponds to the TMS470 CPU address that accesses the code and data. Therefore, only a load address needs to be specified.

The following linker command files contains the MEMORY and SECTIONS directives for this scenario:

Example: Linker Command File for Scenario 3

```
/
 ↪************************************************************************
 ↪
/* Scenario 3  Link Command                                          ⎵
 ↪           */
/*                                                                   ⎵
 ↪           */
/*   Usage: armlnk <obj files...>    -o <out file> -m <map file>⎵
 ↪lnk32.cmd   */
/*         tiarmclang <src files...> -Wl,-o=out_file,-m=map_
 ↪file lnk32.cmd */
/*                                                                   ⎵
 ↪           */
/*   Description: This file is a sample command file that can be⎵
 ↪used       */
/*              for linking programs built with the TMS470 C  ⎵
 ↪           */
/*              compiler.   Use it as a guideline; you may⎵
 ↪want to change */
```

(continues on next page)

```
/*                the allocation scheme according to the size of␣
→your       */
/*                program and the memory layout of your target␣
→system.      */
/*                                                               ␣
→            */
/*  Notes: (1)  You must specify the directory in which rts32.␣
→lib is       */
/*                located.  Either add a "-i<directory>" line to␣
→this        */
/*                file, or use the system environment variable C_␣
→DIR to       */
/*                specify a search path for libraries.         ␣
→            */
/*                                                               ␣
→            */
/*          (2)  If the runtime-support library you are using␣
→is not       */
/*                named rts32.lib, be sure to use the correct␣
→name here.    */
/
→******************************************************************************
→


-m example3.map

/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    I_MEM    : org = 0x00000000   len = 0x00000020  /*␣
→INTERRUPTS        */
    D_MEM    : org = 0x00000020   len = 0x00010000  /* DATA␣
→MEMORY   (RAM) */
    P_MEM    : org = 0x00010020   len = 0x00100000  /* PROGRAM␣
→MEMORY (ROM) */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
SECTIONS
{
    secA: load = 0x20
    secB: load = D_MEM
}
```

3.16. Hex Conversion Utility Description

You must create a hex conversion command file to generate a hex output with the correct addresses and format for the EPROM programmer.

The EPROM programmer in this scenario has the following system requirements:

- In the memory system outlined in the previous figure the EPROM system memory width is 32 bits because each of the physical ROMs provides 16 bits of a 32-bit word. Because the EPROM system memory width is 32 bits, the memwidth value must be set to 32.

- Because the width of each of the physical ROMs is 16 bits, the romwidth value must be set to 16.

- Intel format must be used.

With a memwidth of 32 and a romwidth of 16, two output files are generated by the hex conversion utility (the number of files is determined by the ratio of memwidth to romwidth). In previous scenarios, the output filename was specified with the -o option. Another way to specify the output filename is to use the files keyword within a ROMS directive. When you use -o or the files keyword, the first output filename always contains the low-order bytes of the word.

The hex conversion command file for Scenario 3 is shown in the example that follows. This command file uses the following options to select the requirements of the system:

Option	Description
-i	Create Intel format
-map example3.mxp	Generate example3.mxp as the map file of the conversion
-memwidth 32	Set EPROM system memory width to 32
-romwidth 16	Set physical ROM width to 16

The files keyword is used within the ROMS directive to specify the output filenames.

Example: Hex Conversion Command File for Scenario 3

```
/* Hex Conversion Command file for Scenario 3          */
a.out                   /* linked object file, input        */
-I                      /* Intel format */

/* Optional Commands */
-map example3.mxp       /* Generate a map of the conversion  */

/* Specify EPROM system memory width and physical ROM width */
-memwidth 32            /* EPROM memory system width         */
-romwidth 16            /* Physical width of ROM             */

ROMS
{
    EPROM: org = 0x0, length = 0x20000
```

3.16. Hex Conversion Utility Description 780

```
        files={ lower16.bit, upper16.bit }
}
```

The resulting map file (example3.mxp) is as follows:

Example: Contents of Hex Map File example3.mxp

```
************************************************************
TMS470 Hex Converter                 Version x.xx
************************************************************
Tue Sep 19 07:41:28 1995

INPUT FILE NAME: <a.out>
OUTPUT FORMAT:   Intel

PHYSICAL MEMORY PARAMETERS
    Default data width:     8
    Default memory width:  32
    Default output width:  16

OUTPUT TRANSLATION MAP
------------------------------------------------------------
00000000..0001ffff  Page=0  ROM Width=16  Memory Width=32  "EPROM
 ↪"
------------------------------------------------------------
    OUTPUT FILES: lower16.bit [b0..b15]
                  upper16.bit [b16..b31]

    CONTENTS: 00000020..00000021  Data Width=1  secA
              00000028..00000029  Data Width=1  secB
```

The contents of the output files lower16.bit and upper16.bit are shown in the following two figures, respectively. The low-order 16 bits of the 32-bit output word are stored in the lower16.bit file, while the upper 16 bits are stored in the upper16.bit file.

Figure: Contents of Hex Output File lower16.bit

Figure: Contents of Hex Output File upper16.bit

3.17 Smart Function and Data Placement

The tiarmclang compiler tools support a method to more easily place heavily accessed functions and data objects in faster memory using annotations either at the C/C++ source file level (using attributes) or at link-time using assembly directives. Heavily accessed or critical functions and data can be pre-determined manually by the user or by profiling infrastructure built on top of this. This method prevents the users from having to know function or data subsection information for explicitly placement using a linker command file, which may vary across projects and applications.

The annotations developed correspond to a memory hierarchy: LOCAL memory (Tightly-Coupled Memory/TCM), ONCHIP memory (RAM), or OFFCHIP memory (external flash). A priority can also be assigned to sort functions and data object placement within the same memory at link-time.

The memory regions in which these placements are allocated is controlled using directives in the linker command file according to documented output sections described below.

3.17.1 C/C++ Source-level annotations

The following source-level annotations can be used to explicitly place functions and data objects in corresponding locations. If priority is omitted, it is assumed to be priority '1' (i.e. "high priority").:

```
__attribute__(({local,onchip,offchip}(priority)))
```

Example:

```
__attribute__((local(1))) void func0(void) { .. } // Place in
↪TCM with priority 1
__attribute__((local(2))) void func1(void) { .. } // Place in
↪TCM with priority 2
__attribute__((onchip))   void func2(void) { .. } // Place in
↪SRAM with implied priority 1
```

The attributes can be added to a function definition or a function declaration (as long as that function is called/referenced in the same compilation unit).

3.17.2 Assembly metainfo directives

Functions can also be annotated by adding an assembly metainfo directive in an assembly file that is compiled and linked with the project using the following format. This would allow users to avoid having to compile other source code:

```
.global <global function symbol>
.sym_meta_info <global function symbol>, "of_placement", {"local
↪","onchip","offchip"}, <priority>
```

Example:

```
.global strcmp
.sym_meta_info strcmp, "of_placement", "local", 1
.global memcpy
.sym_meta_info memcpy, "of_placement", "memcpy", 1
```

A simple node.js script called *generate_syms.js* is included in the toolchain that generates an assembly file based on a CSV text file of the following format:

```
strcmp,local,1
main,onchip,3
memcpy,fast_local_copy,1
```

3.17.3 Smart Placement Linker Aggregation

With the placement described above, the TI link-step will aggregate function and data input sections into documented output sections while also sorting the input sections. For Smart Placement, the input sections are sorted based on the designated priority. The documented output sections for Smart Placement are:

- **.TI.local**: Code and initialized data designated for local memory (TCMs)

- **.TI.bss.local**: Uninitialized data designated for local memory

- **.TI.onchip**: Code and initialized data designated for onchip memory (RAM)

- **.TI.bss.onchip**: Uninitialized data designated for onchip memory

- **.TI.offchip**: Code designated for offchip memory (FLASH)

Note: that data objects placed in **.TI.local** or **.TI.onchip** are always directly initialized according to RAM-model initialization. This means that whatever is responsible for loading that code and data into RAM or TCM will also initialize the data, even if ROM-model auto-initialization is used. This means that in ROM-model, CINIT records are not created for this data.

When ROM-model auto-initialization is enabled, zero-initialization CINIT records will be created for the uninitialized memory regions **.TI.bss.local** and **.TI.bss.onchip**. When RAM-model initialization is used, it is up to the user to zero-initialized these sections. The linker will export symbols that an initialization routine can link against designated the start and end of these sections:

- **.TI.bss.local**: *__start___TI_bss_local* and *__stop___TI_bss_local*

- **.TI.bss.onchip**: *__start___TI_bss_onchip* and *__stop___TI_bss_onchip*

Note that because symbols are defined for these sections, they cannot be split between multiple memory regions.

A default linker command file needs to place the documented output sections in the corresponding memory regions in both development and deployment flows. This could be autogenerated by sysconfig based on the memory partition or linked using generic macros. For a development flow, this is pretty straightforward, as in the following example.

```
/* Partitioned memory map */
MEMORY
{
    R5F_VECS : ORIGIN = 0x00000000 , LENGTH = 0x00000040
    R5F_TCMA : ORIGIN = 0x00000040 , LENGTH = 0x00007FC0
    R5F_TCMB : ORIGIN = 0x41010000 , LENGTH = 0x00008000
    MSRAM    : ORIGIN = 0x70080000 , LENGTH = 0x40000
    FLASH    : ORIGIN = 0x60100000 , LENGTH = 0x80000
}
```

```
SECTIONS
{
    /* "local"   --> split between TCMs and RAM                          ␣
↪  */
    /* "onchip"  --> split between RAM and FLASH                         ␣
↪  */
    /* "offchip" --> FLASH                                              ␣
↪  */

    .TI.local   : {} >> R5F_TCMA | R5F_TCMB | MSRAM
    .TI.onchip  : {} >> MSRAM | FLASH
    .TI.offchip : {} > FLASH
    .TI.local.bss : {} > R5F_TCMB; /* Exports symbols __start___
↪TI_bss_local, __stop___TI_bss_local */
    .TI.onchip.bss: {} > MSRAM;    /* Exports symbols __start___
↪TI_bss_onchip, __stop___TI_bss_onchip */
}
```

By default, section splitting should be used as shown above between memory regions to get the full effect of function prioritization.

3.17.4 Enable Smart Data Collection for Smart Placement

When the **--smart_data_collect** linker option is enabled, the TI link-step will not only include explicitly annotated function and data objects into the appropriate documented output sections, it will also pull in referenced initialized data sections and assign them the same priority as the object that references them. For objects placed in .TI.local, referenced read-write and read-only (constant) data sections are pulled in. For objects placed in .TI.onchip, only referenced read-only (constant) data sections are pulled in. Nothing happens for objects placed in .TI.offchip.

GNU-SYNTAX ARM ASSEMBLY LANGUAGE REFERENCE GUIDE

This part of the **tiarmclang** user guide provides information about GNU-syntax Arm assembly language that is recognized and processed by the **tiarmclang** integrated GNU-syntax Arm assembler.

- **Command Line:** For an overview of assembler command-line options, see *GNU-Syntax Arm Assembler Command Line*.

- **Assembly Syntax:** For an overview of GNU Arm assembly syntax, see *GNU-Syntax Arm Assembly Source Anatomy*.

- **Instructions:** For links to documentation of assembly instructions for your Arm device, see *GNU-Syntax Arm Assembly Instructions*.

- **Directives:** For documentation of the assembly directives supported with **tiarmclang**, see *GNU-Syntax Arm Assembly Directives*.

Additional documentation for using assembly with **tiarmclang** is provided elsewhere:

- For information about migrating legacy TI-syntax Arm assembly language instructions and directives to GNU-syntax Arm assembly, see *Migrating Assembly Language Source Code*.

- For information about the **tiarmclang** command line and about the **tiarmasm** standalone TI-syntax assembler, see *About the tiarmclang Arm Assemblers*.

- For information about the Application Binary Interface (ABI) for the Arm architecture, see the ABI specifications page and the Addenda.

- Information about GNU-syntax Arm assembly language is available in GNU and Arm documentation online. The majority of the documented syntax is recognized and processed by **tiarmclang**'s integrated GNU-syntax Arm assembler. Here are some suggested sources:

  - Arm Developer's Instruction Set Architecture

  - GNU Assembler - Using as

  - GNU Assembler (as) - ARM Dependent Features

Contents:

4.1 GNU-Syntax Arm Assembler Command Line

4.1.1 Usage

To invoke the tiarmclang assembler, use the following syntax:

tiarmclang [*options*] <*gnu asm file*>.*S* [. . .]
tiarmclang [*options*] <*gnu asm file*>.*s* [. . .]

- **tiarmclang** - This command runs the assembler and other tools (compiler and linker, for example). For information about running the compiler, see *Invoking the Compiler*. For information about running the linker, see *Invoking the Linker*.

- *options* - Options that affect the way that the compiler tools process input files. For the assembler, these may include:

  - *tiarmclang options* - affect the behavior of the C/C++ compiler or the integrated GNU-syntax Arm assembler. See *Command-Line Options*.

  - *Legacy TI-syntax Arm assembler options* - are prefixed with either the *-Wti-a,* or *-Xti-assembler* option indicating that the option that follows should be passed directly to the legacy TI-syntax Arm assembler. This legacy assembler can also be invoked using the **tiarmasm** command. Legacy TI-syntax Arm assembler options are described in more detail in the *TI-Syntax Arm Assembler* section. See *Passing Options to Other Tools from tiarmclang* for more information about passing options to the legacy TI-Syntax Arm Assembler.

- *filenames* - One or more input files. For the GNU-syntax Arm assembler, by default, an input file with either a *.s* or *.S* file extension is interpreted as a GNU-syntax Arm assembly source file. See *Effects of the C Preprocessor* for details about the difference between how .S and .s files are handled.

You may use the *-x assembler* option to instruct **tiarmclang** to interpret input files that follow the -x assembler option on the tiarmclang command-line as GNU-syntax Arm assembly source files. Similarly, you may use the *-x assembler-with-cpp* option to instruct **tiarmclang** to run the input files that follow the option through the C pre-processor before passing the output of the C pre-processor to the GNU-syntax Arm assembler. For more information, see *Using -x Option to Control Input File Interpretation*.

Additional information about the assembler command line can be found in *Integrated GNU-Syntax Arm Assembler*.

4.1.2 Example

The following simple example uses the **tiarmclang** command to run the assembler:

```
tiarmclang -mcpu=cortex-m0 print_global.S def_global.s -c
```

The `print_global.S` file is run through the C preprocessor and then passed to the GNU-syntax Arm assembler. The `def_global.s` file is passed directly to the assembler. The -c option instructs **tiarmclang** to generate an object file without invoking the linker.

4.1.3 Effects of the C Preprocessor

Input files with an uppercase `.S` file extension are first run through the C pre-processor, and the output is passed to the integrated GNU-syntax Arm assembler. Input files with a lowercase `.s` file extension are passed directly to the integrated GNU-syntax Arm assembler without using the C pre-processor.

Preprocessing directives in the assembly source file are useful for configuring assembly source content based on predefined macro symbols that are available to the C preprocessor. These macro symbols comply with the Arm C Language Extensions (ACLE) by predefining the appropriate ACLE macro symbols based on the processor options specified on the command line.

For example, the following assembly code has different results depending on whether it is contained in a `.S` or `.s` file:

```
.section        .data, "aw", %progbits

.if     __ARM_ARCH == 7
.long   7
.elseif __ARM_ARCH == 6
.long   6
.else
.long   5
.endif
```

If this example code is contained in a file with a `.S` extension, the __ARM_ARCH macro symbol is defined by the C preprocessor and is available to the assembler. If this code is contained in a file with a `.s` extension, the directives that reference the __ARM_ARCH macro cause an error to be generated. See the *Runtime Support* and *Pre-Defined Macro Symbols* topics for more about the defined macro symbols.

4.1.4 Command-Line Options

When specifying an assembler-only option from the **tiarmclang** command line, you must precede the assembler option with the -Xasssembler option. For example:

```
%> tiarmclang -mcpu=cortex-m0 -Xassembler -I"./asm_inc_dir"␣
↪hello.s -c
```

Or, you can use the -Wa option. For example:

```
%> tiarmclang -mcpu=cortex-m0 -Wa,-I"./asm_inc_dir" hello.s -c
```

Either option tells the integrated assembler to add the relative path `./asm_inc_dir` to the directory search path in which the assembler looks for files that are incorporated into the assembly source file via a directive listed in *Directives that Include Other Assembly Source Files*.

Standard command-line options available with the GNU assembler are described in the GNU Assembler - Using as document. See the Command-Line Options and Arm-Dependent Options for details. Please note that not all GNU assembler options documented in these online resources are supported by the **tiarmclang** compiler tools.

4.2 GNU-Syntax Arm Assembly Source Anatomy

4.2.1 Fields of an Assembly Source Line

GNU-syntax Arm assembly language source statements follow the following general form:

label field: *<mnemonic field>* *<operand list field>*

For example, the following Arm instruction is legal in GNU-syntax Arm assembly language:

```
add_me:    add    r0, r1
```

where:

- **add_me** occupies the label field,
- **add** occupies the mnemonic field, and
- the operand list field consists of registers **r0** and **r1** with operands separated by a comma.

The following sections provide more details about the rules that govern the content of the *label field*, *mnemonic field*, and *operand list field* of a given GNU-syntax Arm assembly language source statement.

4.2.2 Labels

An optional label field can be used to associate a value with a symbol. Label symbol names are case sensitive, and a label must begin in the leftmost column of the assembly source line. For GNU-syntax Arm assembly source:

- Label symbols must begin with a letter, an underscore, or a period (".").

- Label symbols can consist of alphanumeric characters, the dollar sign ("$"), an underscore ("_"), or a period (".").

- Label symbol definitions *must* be delimited with a terminating colon (":"), otherwise the **tiarmclang** assembler tries to interpret the symbol as a mnemonic identifier.

- A label may be specified on an assembly source line by itself in which case the label symbol's value will be dependent on the assembly source lines above and below the label specification.

The value assigned to a symbol defined in a label field varies depending on whether the label occurs within the context of an instruction or a directive.

If a label is specified on the same line as or immediately precedes a specification of a GNU-syntax Arm assembly instruction, then the address of the first byte in the object encoding of the instruction is assigned to the label symbol's value. For example, if the following assembly source:

```
        nop
        nop
        nop

add_me:
        add    r0,r1
        nop
        nop
```

is assembled with the following command:

```
%> tiarmclang -mcpu=cortex-m0 -c add_me.s
```

Then the output from %>tiarmdis add_me.o would look like this:

```
Disassembly of add_me.o:

TEXT Section .text, 0xe bytes at 0x00000000
000000:                   .thumb
000000:              :
000000: 00BF              NOP
000002: 00BF              NOP
000004: 00BF              NOP
```

```
000006:                    add_me:
000006: 0844              ADD                    R0, R1
000008: 00BF              NOP
00000a: 00BF              NOP
00000c: 00BF              NOP
```

Notice that the value assigned to the `add_me` label symbol matches the address of the encoding of the first instruction that follows the label.

The value of a label symbol can also be influenced by a directive that appears before it in the assembly source file. Consider the following assembly source:

```
        .text
        nop
        nop
        nop

        .p2align 2
add_me:
        add     r0,r1
        nop
        nop
        nop
```

Using the same tiarmclang command to assemble the source file, the disassembly output would look like this:

```
Disassembly of add_me.o:

TEXT Section .text, 0x10 bytes at 0x00000000
000000:                    .thumb
000000:                    :
000000: 00BF              NOP
000002: 00BF              NOP
000004: 00BF              NOP
000006: C046              MOV                    R8, R8
000008:                    add_me:
000008: 0844              ADD                    R0, R1
00000a: 00BF              NOP
00000c: 00BF              NOP
00000e: 00BF              NOP
```

In this example, the `.p2align 2` directive instructs the assembler to advance the current section counter to the next 4-byte boundary before emitting the object encoding for the next instruction.

4.2. GNU-Syntax Arm Assembly Source Anatomy 791

The label symbol is then given the address of the object encoding for the aligned add instruction as its value. The assembler automatically inserts executable padding between the third nop instruction and the encoding of the add instruction.

Local Labels

The GNU-syntax Arm assembler that is integrated into the tiarmclang compiler supports the notion of local labels whose scope and effect are temporary. Local labels cannot be declared with global linkage. The syntax for defining and referring to GNU-syntax local labels is as follows:

- Local label definitions use the form **N:** in the label field of a line of GNU-syntax assembly code, where *N* is an integer in the range [0,9].

- References to the most recently defined local label use the form **Nb**, where *N* is the ID of the local label (an integer in [0,9]) and *b* indicates a backward reference.

- References to the nest definition of a local label use the form **Nf**, where *N* is the ID of the local label (an integer in [0,9] and *f* indicates a forward reference.

GNU-syntax local labels can be redefined in the same compilation unit. The GNU-syntax assembler associates a unique ordinal ID for every local label definition so that it is able to distinguish one instance of a local label definition from another that was defined with the same value *N*.

Simple Local Label Example

Here is an example of local labels being used in the context of a loop:

```
// assume external global int "sum_tot"
    .global    sum_tot

// assume incoming r0 has loop limit
    .global    foo
    .section   .text
    .thumb
foo:
    ...
    MOVS    r1,#0
    CMP     r1, r0
    BLE     1f
0:
    LDR     r2, C_CON1
    LDR     r3, [r2]
    ADDS    r3, r3, r1
    STR     r3, [r2]
    ADDS    r1, r1, #1
    CMP     r1, r0
```

```
      BGT     0b
1:
      ...
      BX      LR

      .align 4
C_CON1:   .int    sum_tot
```

where:

- *0:* and *1:* are local label definitions,

- a forward reference to *1:* is specified as *1f* in the above **BLE** instruction, and

- there is a backward reference to *0:* specified as *0b* in the **BGT** instruction.

Macro Example Use of Local Labels

The following example shows the use of a local label in the context of a macro definition:

```
// GNU-syntax implementation of trace_pc macro using local labels
      .macro    trace_pc
\@:
      .section  .trace_scn,"aw",%progbits
      .int      \@b
      .previous
      .endm

      .section  .text
foo:
      nop
      trace_pc
      nop
      trace_pc
      nop
      trace_pc
      nop
```

In this case, the special \@ syntax will be replaced by an automatically-generated integer when the macro is invoked and expanded. The effect is that each invocation of the macro will contain a local label definition and a backwards reference to that local label:

```
// GNU-syntax implementation of trace_pc macro using local labels
      .macro    trace_pc
\@:
      .section  .trace_scn,"aw",%progbits
```

```
        .int        \@b
        .previous
        .endm

        .section    .text
foo:
        nop
0:
        .section    .trace_scn,"aw",%progbits
        .int        0b
        .previous
        nop
1:
        .section    .trace_scn,"aw",%progbits
        .int        1b
        .previous
        nop
2:
        .section    .trace_scn,"aw",%progbits
        .int        2b
        .previous
        nop
```

The disassembled object code for the above example looks like this:

```
Disassembly of try.o:

TEXT Section .text, 0x8 bytes at 0x00000000
000000:                 :
000000:                 foo:
000000:                   .thumb
000000: 00BF              NOP
000002: 00BF              NOP
000004: 00BF              NOP
000006: 00BF              NOP


DATA Section .trace_scn, 0xc bytes at 0x00000000
000000: 00000002        .word 0x00000002
000004: 00000004        .word 0x00000004
000008: 00000006        .word 0x00000006
```

The *.trace_scn* contains the addresses of the last three NOP instructions in the *.text* section.

If we were to use a normal label like *xyz_\@* for the GNU-syntax implementation of the macro, the

assembler would report a duplicate label definition for *xyz_0*. When the GNU-syntax assembler invokes the *trace_pc* macro using a local label definition, then the local label *0* is auto-generated for each invocation. Since the GNU-syntax assembler assigns a unique ordinal ID to each instance of a local label, it is able to avoid a duplicate label definition when local labels are used in the macro definition.

4.2.3 Mnemonics

The mnemonic field of a legal line of assembly code contains a pre-defined textual identifier that indicates whether the source line represents an *instruction* or a *directive*.

For example, the **push** mnemonic in the following line of assembly code is recognized as a valid Arm instruction:

```
// Simple example
    .text
    .thumb
    .global    simple_function

simple_function:
    push {r7,lr}
    ...
```

The **.text**, **.thumb**, and **.global** mnemonics are recognized as Arm assembly directives.

For GNU-syntax Arm assembly source, the *mnemonic field* may begin anywhere on a line of assembly source (including the left-most column 0) as long as it precedes the *operand list field* (if one is required) and any comments on the line. As mentioned earlier, an identifier that begins in the leftmost column is interpreted as a *mnemonic* unless it is delimited with a colon (':') suffix.

The *GNU-Syntax Arm Assembly Instructions* and *GNU-Syntax Arm Assembly Directives* sections contain further information about specific mnemonics that represent Arm instructions and directives and are recognized by the **tiarmclang** integrated GNU-syntax assembler.

4.2.4 Operand List

The syntax rules governing the *operand list field* is dependent on the identifier specified in the *mnemonic field*. For example, in the **push** instruction shown earlier in this section, the *operand list field* contains a list of one or more registers enclosed in braces, whereas the *operand list field* of a .global directive expects a legal symbol identifier.

Some operands must be *absolute*, which means they may not refer to any external symbols or any registers or memory references. The value of the expression must be knowable at assembly time.

Some operands must be *well-defined*, which means they must use only symbols or constants that have been declared or defined before the expression in which they appear is encountered by the assembler.

More information about GNU-syntax Arm assembler instructions and directives can be found in the *GNU-Syntax Arm Assembly Instructions* and *GNU-Syntax Arm Assembly Directives* sections of this user guide.

4.2.5 Comments

You can insert comments into your assembly source code to enhance readability of your code. In GNU-syntax Arm assembly source, comments can be delimited using:

- C-style comments; text enclosed between "/*" and "*/" which may span multiple lines.

- C++-style comments; text appearing after "//" on a line.

- Text appearing after an at-sign, '@', is interpreted as a comment unless that '@' character appears in a macro definition preceded by a backslash '\'.

Now consider the following snippet of GNU-syntax Arm assembly code, and note the use of C-style and C++-style comments:

```
/*
 * Loop entry - comment can span multiple lines
 */
loop_entry:
        bl      ef1                 // call ext func 1, ef1
        bl      ef2                 // call ext func 1, ef1
        ldr     r0, [sp]
        adds    r0, #1              // I++ (r0)
        str     r0, [sp]
        movw    r1, :lower16:evar
        movt    r1, :upper16:evar
        ldr     r1, [r1]            // load evar (r1)
        cmp     r0, r1              // I > evar?
        blt     loop_entry          // I < evar, go to loop_entry

/* Loop exit */
loop_exit:
        movs    r0, #0
        pop     {r7, PC}
```

4.3 GNU-Syntax Arm Assembly Instructions

A description of the GNU-syntax for Arm assembly instructions is provided in the *GNU-Syntax Arm Assembly Source Anatomy* section.

For descriptions of the instructions available on each of the Arm processor variants that are supported in the **tiarmclang** toolchain, see the following reference documentation:

- **Cortex-M0:** The Cortex-M0 Instruction Set (Cortex-M0 Devices Generic User Guide)

- **Cortex-M0+:** The Cortex-M0+ Instruction Set (Cortex-M0+ Devices Generic User Guide)

- **Cortex-M3:** The Cortex-M3 Instruction Set (Cortex-M3 Devices Generic User Guide)

- **Cortex-M4:** The Cortex-M4 Instruction Set (Cortex-M4 Devices Generic User Guide: PDF)

- **Cortex-M33:** The Cortex-M33 Instruction Set (Cortex-M33 Devices Generic User Guide)

- **Cortex-R4** and **Cortex-R5**: Both A32 (Arm) and T32 (Thumb) instructions are available on Cortex-R4 and R5 processors.

    - The ARM instruction sets (ARM Cortex-R Series Programmer's Guide)

    - Instruction Set Assembly Guide for Armv7 and Earlier Arm Architectures Reference Guide

    - The ARMv7-M Instruction Set (ARMv7-M Architecture Reference Manual)

    - Instruction set summary (ARM7TDMI Technical Reference Manual r4p1)

For Cortex-R4 and R5 (–mcpu=cortex-[r4|r5]), the compiler generates A32 instructions by default. If the --mthumb option is used, the compiler generates T32 instructions. In an assembly source file, the .arm and .thumb directives can be used to override this setting for individual functions. An application can contain both functions compiled in Arm mode and functions compiled in Thumb mode. Likewise, object modules compiled in Arm mode can be linked with modules compiled in Thumb mode. When generating object code for a call between any two functions, the linker detects the mode for both the caller and callee functions and generates the appropriate BL or BLX instruction depending on whether a code state transition is necessary.

4.4 GNU-Syntax Arm Assembly Directives

All assembler directives have names that begin with a period (.). The rest of the name is made up of letters (and occasionally numbers), almost always in lower case. A description of the GNU-syntax for Arm assembly, including use of directives, is provide in the *GNU-Syntax Arm Assembly Source Anatomy* section.

This section provides a description of each of the directives available on Arm processor variants supported by the **tiarmclang** toolchain (this includes Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M33, Cortex-R4, and Cortex-R5).

A full, alphabetic list of the available directives is provided in *List of GNU-Syntax Arm Assembly Directives*. Detailed descriptions of each directive are provided in the sections that follow, which organize directives by functional category.

GNU-syntax directives that are available only for Arm targets are indicated as "Arm only".

Contents:

4.4.1 List of GNU-Syntax Arm Assembly Directives

#

.2byte	*Directives that Initialize Values and Strings*
.4byte	*Directives that Initialize Values and Strings*
.8byte	*Directives that Initialize Values and Strings*

A

.abort	*Directives for Conditional Assembly and Control Flow*
.align	*Directives that Perform Alignment and Create Space*
.arch	*Directives that Change the Instruction Type*
.arm	*Directives that Change the Instruction Type*
.ascii	*Directives that Initialize Values and Strings*
.asciz	*Directives that Initialize Values and Strings*

B

.balign	*Directives that Perform Alignment and Create Space*
.bss	*Directives that Control Section Use*
.byte	*Directives that Initialize Values and Strings*

C

.code	*Directives that Change the Instruction Type*
.comm	*Directives that Affect Symbols*
.cpu	*Directives that Change the Instruction Type*

D

.data	*Directives that Control Section Use*
.def	*Directives that Can Aid Debugging*
.dim	*Directives that Can Aid Debugging*
.double	*Directives that Initialize Values and Strings*

E

.eabi_attribute	*Directives that Encode Metadata*
.else	*Directives for Conditional Assembly and Control Flow*
.elseif	*Directives for Conditional Assembly and Control Flow*
.end	*Directives for Conditional Assembly and Control Flow*
.endef	*Directives that Can Aid Debugging*
.endfunc	*Directives that Can Aid Debugging*
.endif	*Directives for Conditional Assembly and Control Flow*
.endm	*Directives that Affect Macros*
.endr	*Directives for Conditional Assembly and Control Flow*
.equ	*Directives that Affect Symbols*
.equiv	*Directives that Affect Symbols*
.eqv	*Directives that Affect Symbols*
.err	*Directives that Can Aid Debugging*
.error	*Directives that Can Aid Debugging*
.even	*Directives that Perform Alignment and Create Space*
.exitm	*Directives that Affect Macros*

F

.fail	*Directives that Can Aid Debugging*
.file	*Directives that Can Aid Debugging*
.fill	*Directives that Perform Alignment and Create Space*
.float	*Directives that Initialize Values and Strings*
.fpu	*Directives that Change the Instruction Type*
.func	*Directives that Can Aid Debugging*

G

.global	*Directives that Affect Symbols*
.globl	*Directives that Affect Symbols*

H

.hidden	*Directives that Affect Symbols*
.hword	*Directives that Initialize Values and Strings*

I

.ident	*Directives that Can Aid Debugging*
.if	*Directives for Conditional Assembly and Control Flow*
.ifb	*Directives for Conditional Assembly and Control Flow*
.ifc	*Directives for Conditional Assembly and Control Flow*
.ifdef	*Directives for Conditional Assembly and Control Flow*
.ifeq	*Directives for Conditional Assembly and Control Flow*
.ifeqs	*Directives for Conditional Assembly and Control Flow*
.ifge	*Directives for Conditional Assembly and Control Flow*
.ifgt	*Directives for Conditional Assembly and Control Flow*
.ifle	*Directives for Conditional Assembly and Control Flow*
.iflt	*Directives for Conditional Assembly and Control Flow*
.ifnb	*Directives for Conditional Assembly and Control Flow*
.ifnc	*Directives for Conditional Assembly and Control Flow*
.ifndef	*Directives for Conditional Assembly and Control Flow*
.ifne	*Directives for Conditional Assembly and Control Flow*
.ifnes	*Directives for Conditional Assembly and Control Flow*
.ifnotdef	*Directives for Conditional Assembly and Control Flow*
.incbin	*Directives that Include Other Assembly Source Files*
.include	*Directives that Include Other Assembly Source Files*
.int	*Directives that Initialize Values and Strings*
.internal	*Directives that Affect Symbols*
.irp	*Directives for Conditional Assembly and Control Flow*
.irpc	*Directives for Conditional Assembly and Control Flow*

L

.loc	*Directives that Can Aid Debugging*
.loc_mark_labels	*Directives that Can Aid Debugging*
.local	*Directives that Affect Symbols*
.long	*Directives that Initialize Values and Strings*

M

.macro	*Directives that Affect Macros*

N

.no_dead_strip	*Directives that Encode Metadata*

O

.octa	*Directives that Initialize Values and Strings*
.org	*Directives that Control Section Use*

P

.p2align	*Directives that Perform Alignment and Create Space*
.popsection	*Directives that Control Section Use*
.previous	*Directives that Control Section Use*
.protected	*Directives that Affect Symbols*
.purgem	*Directives that Affect Macros*
.pushsection	*Directives that Control Section Use*

Q

.quad	*Directives that Initialize Values and Strings*

R

.rept	*Directives for Conditional Assembly and Control Flow*

S

.section	*Directives that Control Section Use*
.set	*Directives that Affect Symbols*
.short	*Directives that Initialize Values and Strings*
.single	*Directives that Initialize Values and Strings*
.size	*Directives that Affect Symbols*
.skip	*Directives that Perform Alignment and Create Space*
.sleb128	*Directives that Change the Instruction Type*
.space	*Directives that Perform Alignment and Create Space*
.string	*Directives that Initialize Values and Strings*
.string8	*Directives that Initialize Values and Strings*
.string16	*Directives that Initialize Values and Strings*
.string32	*Directives that Initialize Values and Strings*
.string64	*Directives that Initialize Values and Strings*
.sym_meta_info	*Directives that Encode Metadata*
.syntax	*Directives that Change the Instruction Type*

T

.tag	*Directives that Can Aid Debugging*
.text	*Directives that Control Section Use*
.thumb	*Directives that Change the Instruction Type*
.thumb_func	*Directives that Change the Instruction Type*
.thumb_set	*Directives that Affect Symbols*
.title	*Directives that Can Aid Debugging*
.type	*Directives that Affect Symbols*

U

.uleb128	*Directives that Change the Instruction Type*

V

.val	*Directives that Affect Symbols*
.version	*Directives that Can Aid Debugging*

W

.warning	*Directives that Can Aid Debugging*
.weak	*Directives that Affect Symbols*
.weakref	*Directives that Affect Symbols*
.word	*Directives that Initialize Values and Strings*

Z

.zero	*Directives that Perform Alignment and Create Space*

4.4.2 Directives that Change the Instruction Type

By default, the assembler begins assembling all instructions in a file as 32-bit instructions. You can change the default action by using command line options or the following directives.

.arch *name*

.arm

.code [16|32]

.cpu *name*

.fpu *name*

.syntax [unified | divided]

.thumb

.thumb_func

See *Processor Options* for compiler command line options related to these assembler directives.

4.4.3 Directives that Perform Alignment and Create Space

The following alignment directives perform actions that adjust the current position in relation to alignment boundaries relative to the beginning of the section.

.align [*exponent* [, *fill_value*, [, *max_skip_count*]]]
.balign[w|l] [*alignment* [, *fill_value*, [, *max_skip_count*]]]
.even
.p2align[w|l] *alignment* [, *fill_value*]

The following directives reserve or create space in memory.

.fill *repeat*, *size*, *value*
.skip *size*, *fill*
.space *size*, *fill*
.zero *size*

4.4.4 Directives that Initialize Values and Strings

The following data-defining directives assemble numeric values in the current section. This has the effect of initializing space with one or more values as specified in the operand list field. The size of the space initialized for each value in the operand list is determined by the directive.

.byte *expr1*[, *expr2*,...] // 8-bit integer
.2byte *expr1*[, *expr2*,...] // 2 8-bit integers
.4byte *expr1*[, *expr2*,...] // 4 8-bit integers
.8byte *expr1*[, *expr2*,...] // 8 8-bit integers
.hword *expr1*[, *expr2*,...] // 16-bit integer
.short *expr1*[, *expr2*,...] // 16-bit integer
.int *expr1*[, *expr2*,...] // 32-bit integer
.word *expr1*[, *expr2*,...] // 32-bit integer
.long *expr1*[, *expr2*,...] // 32-bit integer
.quad *expr1*[, *expr2*,...] // 64-bit integer
.octa *expr1*[, *expr2*,...] // unsigned 128-bit integer

.sleb128 *expressions*
.uleb128 *expressions*

.float *expr1*[, *expr2,...*] // 32-bit floating-point numbers
.single *expr1*[, *expr2,...*] // 32-bit floating-point numbers
.double *expr1*[, *expr2,...*] // 64-bit floating-point numbers

The following directives assemble string values in the current section or output the contents of the literal pool to the current section.

.ascii "string"[,"string",...]
.asciz "string"[,"string",...]
.string[8|16|32|64] "string"[."string",...]

Operand List

The operand list for directives may contain a list of one or more *expr* operands separated by commas. Each *expr* operand is an arithmetic expression that can eventually be resolved to an integer constant by the **tiarmclang** assembler or the linker in the process of building a static executable.

An *expr* operand represented as an arithmetic expression may be in one of the following forms at assembly time:

- For an arithmetic expression that results in an absolute value at assembly time, the assembler encodes the result of the expression in the field defined by the directive.

- If an expression cannot be evaluated at assembly time, then the assembler generates relocations and the value will be computed at link time. For example, for an arithmetic expression that resolves to a single label symbol plus an optional integer constant, the assembler creates a relocation entry for the label symbol reference and includes the relocation entry in the object file that is written by the assembler.

If the operand list is empty, then the directive does nothing.

If the result of an expression will not fit in the size indicated by the directive, a warning message is provided and the least significant bytes of the expression's value that fit are used.

4.4.5 Directives for Conditional Assembly and Control Flow

Conditional assembly directives enable you to instruct the assembler to assemble certain sections of code according to a true or false evaluation of an expression.

.if *absolute expression*
.else
.elseif *absolute expression*
.endif
.ifdef *symbol*
.ifb *text*
.ifc *string1, string2*
.ifeq *absolute expression*
.ifeqs *string1, string2*
.ifge *absolute expression*
.ifgt *absolute expression*
.ifle *absolute expression*
.iflt *absolute expression*
.ifnb *text*
.ifnc *string1, string2*
.ifndef *symbol*
.ifnotdef *symbol*
.ifne *absolute expression*
.ifnes *string1, string2*

The following control flow directives provide ways to repeat a set of statements or to end assembly prior to the end of the source assembly file.

.abort
.end
.endr
.irp *symbol, values…*
.irpc *symbol, values…*
.rept *count*

4.4.6 Directives that Control Section Use

A "section" is a continuous range of addresses. Anything that requires space – be it code, read-only data, or read-write data – will be contained in the assembly language's notion of a section. All data within a section is handled the same. For example, a range of memory may be a "read only" section.

Some sections are manipulated by the linker; others are used by the assembler and may have no meaning except during assembly.

Object files written by the assembler have at least three sections, any of which may be empty. These are named the .text, .data and .bss sections. The assembler can generate additional named sections if you use the .section directive.

Within the object file, the .text section starts at address 0, the .data section follows, and the .bss section follows the .data section. If you do not use any directives that place output in the .text or .data sections, these sections still exist, but are empty.

Addresses used by the assembler are relative to the start of the section that contains them. The `{secname N}` or `(section) + (offset into section)` notation can be used to describe an address relative to the start of its section.

If the section for an address is unknown at assembly time, it is treated as part of the *undefined* section – `{undefined U}` – where `U` is filled in later. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol; its value is unknown at assembly time so it has section undefined.

The assembler keeps a stack of section/subsection combinations used. The *.previous*. *.popsection*, and *.pushsection* directives can be used to manipulate this stack.

The following directives control section use.

.bss
.data
.org *new-location-counter,fill*
.popsection
.previous
.pushsection *name[,subsection][, "flags"[,%type[,arguments]]]*
.section *name[,"flags"[,%type[,arguments]]]*
.text

4.4.7 Directives that Affect Symbols

The following directives control symbol use.

.comm *symbol,length*
.equ *symbol,expression*
.equiv *symbol,expression*
.eqv *symbol,expression*
.global *symbol*
.globl *symbol*
.hidden *names*
.internal *names*
.local *names*
.protected *names*
.set *symbol,expression*
.size *name,expression*
.thumb_set
.type *name,type-description*
.val *addr*
.weak *names*
.weakref *alias,target*

4.4.8 Directives that Include Other Assembly Source Files

The following directives provide a means of incorporating the content of other assembly language files into the source of the current assembly file.

.incbin *"file"[,skip[,count]]*
.include *"file"*

4.4.9 Directives that Can Aid Debugging

The following directives can be used to aid debugging:

.def *name*
.dim
.endef
.endfunc
.err
.error *"string"*
.fail *expression*
.file [*string*]
.func *name[,label]*
.ident
.loc *fileno lineno* [*column*] [*options*]
.loc_mark_labels *enable*
.print *"string"*
.tag *structname*
.version *"string"*
.title *"string"*
.warning *"string"*

4.4.10 Directives that Encode Metadata

The following directives provide a means of communicating information to downstream object file consumer tools like the linker by encoding information into the assembler generated object file.

.eabi_attribute
.no_dead_strip [*section*]
.sym_meta_info symbol, "[kind]", value

.eabi_attribute

> **.eabi_attribute** *tag*, *value*

The .eabi_attribute directive sets an EABI object file build attribute specified by *tag* to *value*. Build attributes are used to check for compatibility between object files and to select suitable versions of the runtime libraries.

The *tag* argument is a numeric attribute ID. For example, 6 is the ID for Tag_CPU_arch. For information regarding these attribute IDs, see the "Build Attributes" section of the ARM ABI addendum.

The *value* is the value assigned to the attribute in the compiler-generated object file. This is typically an integer, but may be a string, depending on the tag.

For example, suppose a simple `hello.c` source file is compiled with the following command line:

```
tiarmclang -mcpu=cortex-m0 -S hello.c
```

The resulting assembly file contains the following examples of the .eabi_attribute directive:

```
.text
.syntax unified
.eabi_attribute 67, "2.09"      @ Tag_conformance
.cpu     cortex-m0
.eabi_attribute 6, 12   @ Tag_CPU_arch
.eabi_attribute 7, 77   @ Tag_CPU_arch_profile
.eabi_attribute 8, 0    @ Tag_ARM_ISA_use
.eabi_attribute 9, 1    @ Tag_THUMB_ISA_use
.eabi_attribute 34, 0   @ Tag_CPU_unaligned_access
.eabi_attribute 17, 1   @ Tag_ABI_PCS_GOT_use
.eabi_attribute 20, 1   @ Tag_ABI_FP_denormal
.eabi_attribute 21, 0   @ Tag_ABI_FP_exceptions
.eabi_attribute 23, 3   @ Tag_ABI_FP_number_model
.eabi_attribute 24, 1   @ Tag_ABI_align_needed
.eabi_attribute 25, 1   @ Tag_ABI_align_preserved
.eabi_attribute 38, 1   @ Tag_ABI_FP_16bit_format
.eabi_attribute 18, 4   @ Tag_ABI_PCS_wchar_t
.eabi_attribute 26, 1   @ Tag_ABI_enum_size
.eabi_attribute 14, 0   @ Tag_ABI_PCS_R9_use
```

The .eabi_attribute directive does not affect that set of instructions the assembler accepts. Arm recommends that the *.arch*, *.cpu*, and *.fpu* directives be used instead where possible. These other directives are recommended because they also make sure that instructions for the selected architecture are valid while also setting the relevant build attributes. Therefore, the .eabi_attribute directive is only needed for build attributes not covered by these other directives.

Note: The following legacy values are also accepted for the *tag*: Tag_VFP_arch, Tag_ABI_align8_needed, Tag_ABI_align8_preserved, and Tag_VFP_HP_extension.

.no_dead_strip

> **.no_dead_strip** [*section*]

The .no_dead_strip directive instructs the linker to include the current or specified *section* in the linked output file, regardless of whether or not the section is referenced. That is, the *section* is not eligible for removal via conditional linking. The section name must be enclosed in double quotes. A section name can contain a subsection name in the form `section_name:subsection_name`.

You can also override conditional linking for a given section with the --retain linker option. You can disable conditional linking entirely with the --unused_section_elimination=off linker option.

By default, the linker assumes that all sections are eligible for removal via conditional linking. (However, the linker automatically retains the .reset section.) The .no_dead_strip directive is useful for overriding this default behavior for sections you want to retain in the link, even if the section is not referenced by any other section in the link. For example, you could apply a .no_dead_strip directive to an interrupt function that is written in assembly language, but is not referenced from any normal entry point in the application.

This directive is a TI extension that is not portable to other GNU assemblers.

.sym_meta_info

> **.sym_meta_info** *symbol*, *"[kind]"*, *value*

The .sym_meta_info directive provides ways to annotate the symbol table entry information about a symbol. This information can then be used by the linker to influence the linker's processing of the symbol or the section in which the symbol is defined.

This directive is a TI extension; it is not portable to other GNU assemblers.

The *symbol* argument specifies a symbol to representing a function or data object definition.

The *kind* argument is a quoted string that defines the type of information that the directive is providing for use by the linker. Available *kinds* include "location", "noinit", "persistent", "printf_formatstrinfo", and "prinf_forward".

- "location" indicates that the linker should place the symbol definition at the address in target memory specified by *value*.

- "noinit" with a *value* of 1 indicates that the symbol is an uninitialized symbol that should not be set to 0 during a reset.

- "persistent" with a *value* of 1 indicates that the symbol is an initialized symbol that should not be re-initialized during a reset.

- "printf_formatstrinfo" indicates that the *value* is a string representation of the collection of format specifiers that appear in the const char string argument to a printf-like function call. For example, `printf("hello:  %d\n", my_int);` will generate a `.sym_meta_info printf,"printf_formatstrinfo", "d"` directive when compiled.

- "prinf_forward" indicates that the *value* is the name of a function whose definition calls a printf-like function with a const char `*` to a string that it received as an argument.

The *value* can be an integer, a string constant, or a symbol name depending on the specified *kind*.

Example 1: The following example uses a *kind* of "location" to specify the location for the mysym symbol:

```
.sym_meta_info mysym, "location", 0x00ff
```

Example 2: The following example shows several .sym_meta_info directives. Suppose the following C code is compiled:

```
#include <stdio.h>

__attribute__((persistent)) int my_init_var = 10;

__attribute__((location(0x2000)))
int main() {
    printf("my_init_var is %d\n", my_init_var);
    return 0;
}
```

The resulting assembly code would contain the following (this has been somewhat simplified for readability):

```
    .section        .text.main,"ax",%progbits
    .hidden main                            @ -- Begin function␣
↪main
    .globl  main
    .p2align        4
    .type   main,%function
    .sym_meta_info  main, "location", 8192

    ...

    .section        .TI.persistent,"aw",%progbits
    .globl  my_init_var
```

4.4. GNU-Syntax Arm Assembly Directives

```
    .p2align       2
my_init_var:
    .long   10                                  @ 0xa
    .size   my_init_var, 4
    .sym_meta_info  my_init_var, "noinit", 1


    ...


    .sym_meta_info  printf, "printf_formatstrinfo", "d"
```

The `location` attribute in the C code indicates that the section where `main()` is defined should be placed at target address (0x2000). This information is communicated to the linker via a .sym_meta_info directive that encodes a type of "location" and a value of 0x2000 in a special .symtab_meta section. It associates this information with the `main` function symbol.

The `persistent` attribute in the C code indicates that the initial value for `my_init_var` when the program is loaded is 10, but the value will not be re-initialized if the processor is reset. In this case, the .sym_meta_info directive inserts symbol information into the .symtab_meta section telling the linker that the `my_init_var` variable can be directly initialized at load time, but that no .cinit record should be generated for it, since it is not to be re-initialized when the processor is reset and execution starts from the `_c_int00()` boot routine.

The printf() call in the C code results in a .sym_meta_info directive that passes information to the linker about how a call to the printf() function can be specialized. Each format specifier used in the format string passed to printf() is recorded in the .symtab_meta section. In this case, only the `%d` format specifier is used, so only one .sym_meta_info directive is generated. In this case, the linker will determine based on the directive that it can use a call to __TI_printf_nofloat instead of the larger __TI_printf function to implement printf() functionality at run-time. This can result in a significant code size savings for a function that does a lot of simple printf() calls.

4.4.11 Directives that Affect Macros

The following directives manage the creation of macros:

.endm
.exitm
.macro *name,parameters* . . .
.purgem *name*

NOTE ON LINUX INSTALLATIONS

5.1 tiarmclang Shared Library File Dependencies

It is useful to be aware of the shared library files that tiarmclang depends on when trying to figure
out which ones may be missing.

You can list the shared library file dependencies using the *ldd* command. For example:

```
%> ldd /path/to/installation/bin/tiarmclang
```

On the Ubuntu OS that was used to generate this example, the list of shared library files emitted
by the *ldd* command is as follows:

```
linux-vdso.so.1 =>  (0x00007ffd0cb3f000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
 ↪(0x00007f3565388000)
librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1
 ↪(0x00007f3565180000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2
 ↪(0x00007f3564f7c000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f3564c76000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1
 ↪(0x00007f3564835000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f356446c000)
/lib64/ld-linux-x86-64.so.2 (0x00007f35655a6000)
```

ADDITIONAL MATERIAL

See the following additional documentation for more about the TI Arm processors from Texas Instruments and the TI Arm code generation tools.

Introduction

- Introduction to the new TI Arm® Clang Compiler (2 min video)

- The future of compiler tools for TI Arm® Cortex®-based MCUs (TI E2E Article)

- Introduction to the new TI Arm® Clang Compiler (Electronic Design Webinar)

Enabling Functional Safety

- Simplify your functionally safe system development with the TI Arm® Clang compiler (TI E2E Article)

- How to apply the TI Compiler Qualification Kit for functional safety development

Code Coverage

- Uncover hidden bugs in 5 easy steps with TI Arm® Clang (TI E2E Article)

- Code Coverage with TI Arm Clang Compiler

Linker Command Files

- TI Linker Command File Primer

- Linking: Migrate from Arm GCC to tiarmclang

Video Series

- TI Arm Clang Compiler Short Video Series

Generic Clang and LLVM Documentation

- Clang Compiler User's Manual

- Clang Overview

- The LLVM Compiler Infrastructure

SUPPORT

Post compiler-related questions to the TI E2E™ design community forum and select the TI device being used.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DE-SIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DE-SIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MER-CHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, vali-dating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale (https://www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

Note: The TI Arm Clang C/C++ Compiler Tools replace the existing TI Arm C/C++ Compiler Tools. All new feature development will be done in the TI Arm Clang C/C++ Compiler Tools. The last feature release of the armcl compiler toolchain is v20.2.x.LTS, which will continue to be actively supported as long as necessary. However, only bug fixes will be provided in maintenance releases of v20.2.x.LTS of the armcl.

Both toolchains can be used to compile and link C/C++ and assembly source files to build static ex-ecutable applications. These applications can then be loaded and run on Arm Cortex-M and Cortex-

R series devices. Depending on the device family you are using a specific compiler toolchain will be recommended. Please refer to the SDK for the device family for information on which toolchain to use.

For offline use, a PDF version of the guide is available here: TI Arm Clang C/C++ Compiler Tools User's Guide