C29x Clang Compiler Tools User's Guide

v2.1

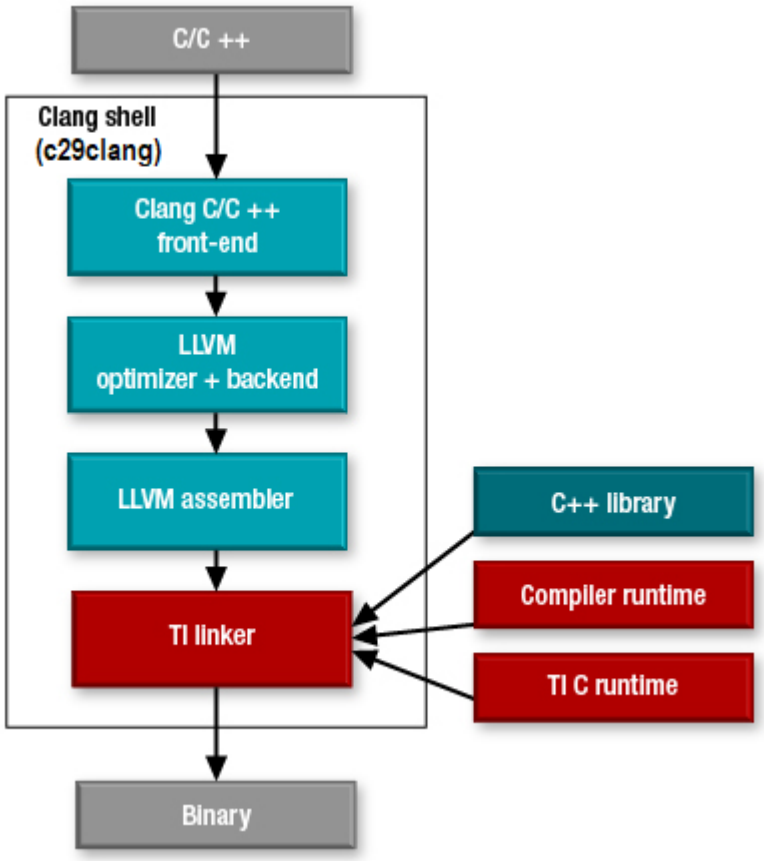
Copyright © 2024, Texas Instruments Incorporated

Online HTML version available here

CONTENTS

The C29x code generation tools support application development for next-generation TI C29x processors from Texas Instruments. This is the only compiler toolchain for C29x processors.

This user's guide documents the TI C29x Clang C/C++ Compiler Tools. The c29clang compiler is based on the open source LLVM compiler infrastructure and its Clang front-end. c29clang uses the TI Linker and C runtime, which provide additional benefits for stability and reduced code size.



Benefits of c29clang include:

- *Excellent C standards support (default C17)*
- *Excellent C++ standards support (default C++17)*
- *Source-based code coverage*
- *Support for migration from TI's C28x compiler (cl2000)*
- Compiler security support with *stack smashing detection*
- *C29x security support* to protect individual calls and frames
- 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

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
- Linker portability – the same *TI linker and linker command files* are used for C28x and C29x
- Function specialization, minimizing code size on common functions, including *printf*, *memcpy, and memset*
- *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* and *CRC*
- Segmented memory spaces, allowing section placement into *multiple ranges* as well as *split placement*

CHAPTER

ONE

C29CLANG GETTING STARTED GUIDE

This Getting Started Guide provides an introduction to the TI C29x Compiler toolchain, along with examples that demonstrate how to use the c29clang compiler to compile and link source files to create a simple application that can run on a C29x processor.

1.1 The c29clang Compiler Toolchain

The TI C29x Compiler Toolchain (c29clang) is Texas Instruments' compiler for the next-generation C29x processors. You can use the c29clang compiler toolchain to build applications from C or C++ source files to be loaded and run on the C29x processors supported by the toolchain.

1.1.1 Toolchain Components

The c29clang compiler toolchain consists of many components. A brief description of the major components is provided in the subsections below.

Essential Tools

- **c29clang**

The C/C++ compiler, c29clang, is used to compile C and C++ source files. By default, it automatically invokes the TI linker, c29lnk, which combines object files generated by the compiler with object libraries to create an executable program that can be loaded and run on a C29x processor.

The c29clang 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.

- **c29lnk**

The linker, c29lnk, 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 a C29x processor. It is typically invoked from the c29clang command line so that c29clang can implicitly set up the object library search path and implicitly include runtime libraries in the link.

This is the same linker used by the TI C28x code generation tools, so linker command files used for C28x applications are easy to migrate to use for C29x processors.

- **c29ar**

The archiver, c29ar, 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 c29clang.

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 C29x applications.

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 c29clang compiler as well as providing an implementation of low-level target-specific functions that can be used in compiler generated code.

Code Coverage Utilities

- **c29cov**

The c29cov tool shows code coverage information for programs that have been instrumented to emit profile data.

- **c29profdata**

The profile data tool, c29profdata, 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

- **c29dem**

The C++ name demangler, c29dem, 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 c29dem tool can be used on a linker-generated map file that contains instances of C++ mangled names.

- **c29libinfo**

The c29libinfo 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.

- **c29nm**

The name utility, c29nm, 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.

- **c29objcopy**

The object copying and editing tool, c29objcopy, 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.

- **c29objdump**

The object file dumper utility, c29objdump, 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 c29objdump's --help option.

- **c29ofd**

Like c29objdump, the object file display utility, c29ofd, 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 c29ofd options available to alter how text output is displayed and whether DWARF debug information is to be included in the output.

- **c29readelf**

The GNU-style ELF object reader, c29readelf, can be used to display low-level format-specific information about one or more object files. Like c29objdump and c29ofd, c29readelf provides

command-line options to allow you to display certain pieces of information from an object file like relocation entries or section headers.

- **c29size**

The GNU-style size information utility, c29size, 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.

- **c29strip**

The c29strip 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 source files (.c extension)
- C++ source files (.C or .cpp extension)

The compile and link part of your development flow will look something like this:

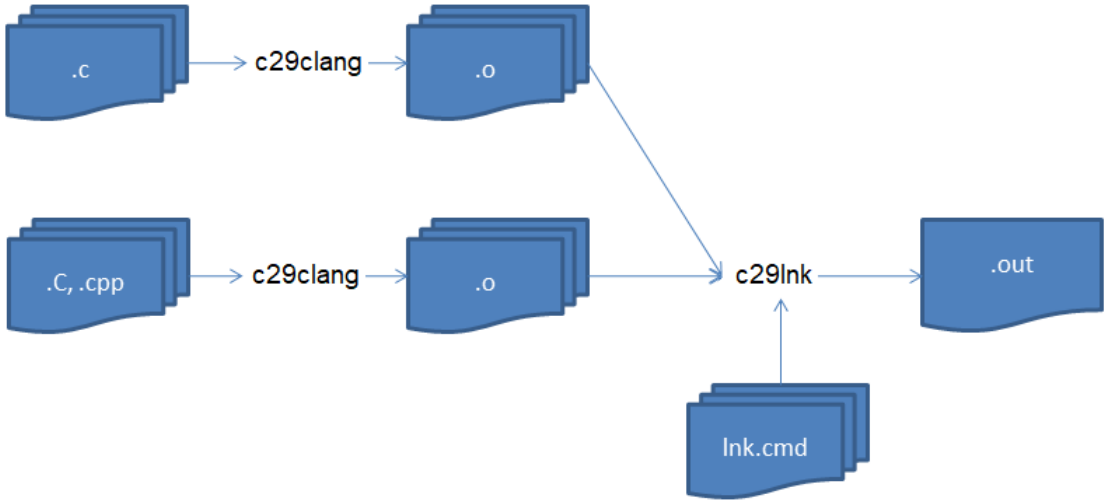


Figure 1.1: Software Development Flow

Note that:

- c29clang interprets files with a .c extension as C source, invoking the compiler
- c29clang interprets files with a .C or .cpp extension as C++ source, invoking the compiler

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 C29x processor.

1.2 Using the c29clang Compiler and Linker

1.2.1 Using the Compiler and Linker

Both the TI C29x Clang (c29clang) Compiler Tools' compiler and linker can be invoked from the **c29clang** command-line. The following subsections describe the basics of how to manage the compile and link steps of building an application from the **c29clang** command-line.

Compiling and Linking

The default behavior of the c29clang compiler is to compile specified C and C++ 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.

```
c29clang [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 **c29clang** 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 c29clang runtime libraries to create an executable output file named **test.out**:

```
c29clang -mcpu=c29.c0 file1.c file2.o -o test.out -Wl,link_test.
 ↪cmd
```

Note that there is no mention of the c29clang runtime libraries on the **c29clang** command-line or inside the **link_test.cmd** linker command file. When the linker is invoked from the **c29clang** command-line, the c29clang compiler implicitly tells the linker where to find applicable runtime libraries, such as the C runtime library (libc.a).

In the above **c29clang** 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 **c29clang** command, you will see exactly how the linker (**c29lnk**) is invoked and with what options. For example, this command:

```
c29clang -mcpu=c29.c0 -v file1.c file2.o -o test.out -Wl,link_
 ↪test.cmd
```

shows the following with regards to how the **c29lnk** command is invoked by the c29clang compiler:

```
<install directory>/bin/c29lnk -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 c29clang 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 **c29clang** command-line.

```
c29clang -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:

```
c29clang -c -mcpu=c29.c0 file1.c file2.c
```

Link-Only Using c29clang

When only object files are specified as input to the c29clang compiler command, the compiler automatically passes those files to the linker along with any other specified options that are applicable to the link.

```
c29clang [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 **c29clang** command:

```
c29clang -mcpu=c29.c0 file1.o file2.o -o test.out -Wl,link_test.
 ↪cmd
```

invokes the linker as follows:

```
<install directory>/bin/c29lnk -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 c29clang 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 c29clang compiler command-line.

Processor Options

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

The c29clang compiler supports the following C29x processor variant:

- **-mcpu=c29.c0**

If an **-mcpu** variant is not specified on the **c29clang** command-line, the compiler assumes a default of **-mcpu=c29.c0**.

Endianness

C29x devices are little-endian.

Floating-Point Support Options

The C29x compiler toolchain emits native 32-bit floating point operations by default. Optionally, some C29x hardware supports native 64-bit floating point operations. Users may control the usage of native floating point hardware with the **-mfpu** option.

Valid forms of the **-mfpu** option include:

- **-mfpu=none** - Emulate all floating point operations in software without the use of floating-point registers or hardware support.
- **-mfpu=f32** - Use native 32-bit floating-point hardware operations, but emulate 64-bit floating-point operations in software. This is the default setting for the compiler toolchain.
- **-mfpu=f64** - Use native 32-bit and 64-bit floating-point hardware operations.

Include Options

The c29clang 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 c29clang 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 c29clang compiler package and should be used in combination with linker and the runtime libraries that are included in the c29clang 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 c29clang 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.

```
c29clang -mcpu=c29.c0 -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 c29clang compiler, select a level of optimization from among the following **-O[0|1|2|3|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 increase code size are disabled.
- **-O3** - All optimizations available at **-O2** plus others that are likely to increase code size.
- **-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 c29clang 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 c29clang 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 **c29clang** command. If c29clang 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 c29clang 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:

```
c29clang -mcpu=c29.c0 -c task_42.c -o task.o
```

Source File Interpretation Option

The c29clang 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

The c29clang 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

---

Note: The **-x<language>** option is position-dependent. A given **-x** option on the **c29clang** 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 c29clang 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 c29clang 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 c29clang tool's linker is nearly identical to the linker in the proprietary 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 c29clang Command-Line

As noted in a few of the above examples, when invoking the linker from the **c29clang** 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 c29clang compiler passes the **link_test.cmd** linker command file directly to the linker:

```
c29clang -mcpu=c29.c0 file1.c file2.o -o test.out -Wl,link_test.
↪cmd
```

The **c29clang** 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:

```
c29clang -mcpu=c29.c0 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:

```
c29clang -mcpu=c29.c0 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 */
```

1.2.4 Runtime Support

Predefined Macro Symbols

The c29clang compiler pre-defines compile-time macro symbols for use in source to help distinguish code written particularly for C29x, for a specific C29x processor variant, or to be compiled by the c29clang compiler (as opposed to other C29x compilers) from other source code.

The c29clang compiler pre-defines several TI-specific and C29-specific pre-defined macro symbols that can be used to distinguish the use of the c29clang compiler from other C29x compilers. These include:

```
__ti__              1     - identify compiler vendor as TI
__ti_major__        1     - identify major version number
__ti_minor__        0     - identify minor version number
__ti_patchlevel__   0     - identify patch version number
__ti_version__      10000 - (__ti_major__*10000)+(__ti_minor__
 ↪*100)+__ti_patchlevel
__C29_ARCH__        0
__C29_C0__          1
__C29_OPTF64__      1
__C29_OPTF32__      1
__C29__             1
__c29__             1
```

See *TI-Specific Pre-Defined Macro Symbols* for more about these macro symbols.

For a complete list of pre-defined macro symbols that are defined by the c29clang 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 c29clang 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. The c29clang 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 c29clang compiler, the appropriate c29clang 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 **c29clang** 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 c29clang 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 c29clang compiler toolchain. An application built using the c29clang compiler tools must use a combination of target options that is compatible with one of the following configurations:

1.3 Creating a Simple Application with the c29clang Compiler Tools

This section of the Getting Started Guide provides an example of how to build a simple application using the c29clang 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 c29clang compiler.

1.3.1 Source Files

The subsections below describe how to build a simple "Hello World!" program using either the c29clang 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:

```
#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:

```
/
 ↪*************************************************************************
 ↪
/* lnk.cmd - V1.00  Command file for linking C29 programs                 ␣
 ↪           */
/
 ↪*************************************************************************
 ↪
/* This linker command file assumes C/C++ model                           ␣
 ↪           */
/
 ↪*************************************************************************
 ↪
-c
-stack  0x8000                              /* Software stack␣
 ↪size           */
-heap   0x2000                              /* Heap area size             ␣
 ↪           */

/* Specify the system memory map */
MEMORY
{
  ROM     : org = 0x00000020   len = 0x2FFFE0     /* 1.25 GB */
  FLASH   : org = 0x10000000   len = 0x300000     /* 1.25 GB */
  RAM     : org = 0x18000000   len = 0x300000     /* 1.25 GB */
```

(continues on next page)

(continued from previous page)

```
}
#define RO_CODE FLASH
#define RO_DATA FLASH
#define RW_DATA RAM

/* Specify the sections allocation into memory */

SECTIONS
{
    .text     : {} > RO_CODE             /* Code                          ␣
 ↪            */
    .cinit    : {} > RO_DATA             /* Initialization tables         ␣
 ↪            */
    .const    : {} > RO_DATA             /* Constant data                 ␣
 ↪            */
    .pinit    : {} > RO_DATA             /* C++ Constructor tables        ␣
 ↪            */

    .data     : {} > RW_DATA             /* Initialized variables         ␣
 ↪            */
    .bss      : {} > RW_DATA             /* Uninitialized                 ␣
 ↪variables            */
    .stack    : {} > RW_DATA             /* Software system stack         ␣
 ↪            */
    .sysmem   : {} > RW_DATA             /* Dynamic memory                ␣
 ↪allocation area    */
}
```

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 c29clang 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 c29clang -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:

```
%> c29clang -mcpu=c29.c0 hello.c -o hello_world.out -Xlinker -
 ↪llnkme.cmd -Xlinker -mhello_world.map
```

When the c29clang 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 c29clang, 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 C29x 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 c29clang application:

```
cmake_minimum_required(VERSION 3.29) # Minimum version for␣
 ↪c29clang support

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

# find c29clang in execution path
find_program(CMAKE_C_COMPILER c29clang)

# or set path to c29clang explicity
#set(CMAKE_C_COMPILER  /Users/ti/ti-cgt-c29_1.2.0.LTS/bin/
 ↪c29clang)
#-------------------------------------------------------------------
 ↪--------------
project(DDREyeFirmware C)

add_compile_options(-mcpu=c29.c0)
add_compile_options(-Oz -g)

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

(continues on next page)

(continued from previous page)

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

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

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 C29x processor, then you can select the processor from the **Target** drop-down menus. For the purposes of this tutorial, the "C29 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 c29clang 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 **C29 Compiler > Processor Options** category, select the appropriate options from each of the drop-down menus in the **Processor Options** window.

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 C29x processor. The resulting .out file can be loaded and run on the appropriate TI C29x processor.

CHAPTER

TWO

TI C28X TO C29CLANG MIGRATION GUIDE

This *Migration Guide* addresses tasks required and issues encountered when porting your existing TI C28x (cl2000) application to c29clang.

The following components of a TI C28x application need modification when migrating to c29clang:

- CCS project (see *Migrating cl2000 CCS Projects to c29clang*)
- Build options (see *Migrating Command-Line Options*)
- C/C++ source code for cl2000 compiler (see *Migrating C and C++ Source Code*)
- C/C++ source code for CLA compiler (see *Migrating CLA Code*)
- Linker command file (see *Migrating Linker Command Files for Use With c29clang*)
- TI C28x assembly code (see *Migrating Assembly Language Source Code*)

Note: COFF to EABI Migration: Only EABI output is supported for TI C29x. If your TI C28x application has COFF output files, you should first migrate from TI C28x COFF output to TI C28x EABI output before migrating to TI C29x. For information, see C2000 Migration from COFF to EABI.

The following migration aids can help you address issues when converting an existing cl2000 project to use the c29clang compiler:

- **Clang-Tidy Tool:** The c29clang-tidy tool diagnoses changes that need to be made to application code in order to migrate it from TI C28x to c29clang. The c29clang-tidy tool is a clang-based utility for diagnosing and fixing issues in a C/C++ application. It can be run from the command line or from within CCS. For more about using c29clang-tidy, see *Migrating Source Code with Clang-Tidy*.
- **C/C++ Source Code Diagnostics:** The c29clang compiler diagnostics can help you find proprietary TI pre-defined macro symbols, pragmas, and intrinsics that need to be converted. For more information, see *C/C++ Source Migration Aid Diagnostics*.

- **CCS Project Migration:** For help converting the build options for an existing cl2000 CCS project to use the c29clang compiler, see *Migrating cl2000 CCS Projects to c29clang*.

Contents:

2.1 Main Differences Between cl2000 and c29clang

This section of the *Migration Guide* describes the primary differences between the cl2000 compiler tools and the TI C29x Clang (c29clang) 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 C29x Clang (c29clang) Compiler Tools do not support proprietary TI-specific C/C++ mechanisms that the TI C28x compiler (cl2000) did. This section provides further details about the c29clang compiler's behavior regarding proprietary TI-specific C/C++ mechanisms.

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

The c29clang compiler does not support many of the proprietary TI pre-defined macro symbols, pragmas, or intrinsics that are supported in the TI C28x compiler. Use of such proprietary pre-defined macro symbols and intrinsics should be replaced by a functionally equivalent alternative.

Rather than using proprietary 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 proprietary TI mechanisms can be found and converted into a portable form that the c29clang compiler does support.

CLA Source Code

The TI C29x Clang (c29clang) Compiler Tools do not have a separate CLA compiler. Convert code intended for the TI C28x CLA compiler to regular C code. See *Migrating CLA Code* for details.

Assembly Source Code

Use of assembly language is discouraged for c29clang, except for assembly code embedded in C/C++ source files via *asm()* statements, which are processed inline by the c29clang integrated GNU-syntax assembler.

For more information on migrating assembly source code from proprietary applications, 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 C28x compiler to the c29clang compiler. These include the following:

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

  The c29clang compiler invokes the linker automatically by default, whereas the TI C28x compiler must be told to invoke the linker via the cl2000's **--run_linker** (**-z**) option. Further details about how to manage the linker invocation from the **c29clang** command-line can be found in the *Using the c29clang Compiler and Linker* section of the *c29clang Getting Started Guide*.

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

  Unlike the TI C28x compiler, which provides **-s**, **-ss**, and **-os** options to instruct the compiler to generate an interlisted assembly source file, the c29clang does *not* support an interlisting option on the compiler command-line. Instead, when a C/C++ source file is compiled with debug enabled, the **c29objdump** 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 c29clang compiler does not support options to alter the file extension of compiler-generated files. For more details about which TI C28x options do not have analogous c29clang 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 c29clang compiler supports a **-S** option that allows you to keep the compiler-generated assembly file, but unlike the cl2000's **-k** option, the c29clang'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 c29clang compiler allows a function containing an asm() statement to be considered for inlining. The TI C28x 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 C28x compiler does not allow any function that contains an asm() statement to be inlined. Thus, in the above example, the TI C28x compiler would not attempt to inline *func_a* in any other function that references *func_a*.

The c29clang compiler behavior with respect to functions that contain asm() statements is different from the TI C28x compiler. The c29clang 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 c29clang 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();
```

(continues on next page)

(continued from previous page)

```
  ...
}
```

If the c29clang 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 c29clang compiler emits an error diagnostic.

You can prevent the c29clang 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 (c29dem)

The TI C29x Clang (c29clang) Compiler Tools include an LLVM-based version of the C++ Name Demangler Utility (c29dem). While the LLVM-based version of this utility is functionally equivalent to the TI C28x compiler tools' version, the command-line interface for the new version is different from the TI C28x version.

The C++ name demangler (c29dem) is a debugging aid that translates C++ mangled names to their original name found in the relevant C++ source code. The c29dem 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 c29clang is:

```
c29dem [options] <mangled names ...>
```

- **options** - affect how the name demangler behaves. The c29dem utility is derived from the LLVM project's **llvm-cxxfilt** tool. To display a list of available options, use the help option (**-h**), **c29dem -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 C28x version of the C++ name demangler and the c29clang version of the C++ name demangler are as follows.

Processing Text Input

Unlike the TI C28x version of the C++ name demangler, the c29clang version of c29dem does not process a text file specified as an argument to c29dem. Assuming that **test.s** is a compiler generated assembly file, you cannot specify **test.s** as an argument to c29dem. Instead, you can pipe the text file as input to the c29dem utility as follows:

```
c29dem < test.s
```

or

```
cat test.s | c29dem
```

Saving Output to a File

The TI C28x version of the C++ name demangler supported an "--output-*file*" option that allowed you to write the output of the c29dem utility to a file. The c29clang version of c29dem does not support a --output option. Instead, the output can be redirected to a file like so:

```
cat test.s | c29dem > c29demout
```

No ABI Option Needed

The TI C28x 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 c29clang compiler. The c29clang version of c29dem assumes EABI and no ABI option is needed to process c29clang compiler generated C++ mangled names.

Updated Name Utility (c29nm)

The TI C29x Clang (c29clang) Compiler Tools include an LLVM-based version of the Name Utility (c29nm). While the LLVM-based version of this utility is functionally equivalent to the TI C28x compiler tools' version, the command-line interface for the new version is different from the TI C28x version.

The name utility (c29nm) 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:

```
c29nm [options] <input files>
```

- **options** - affect how the name utility behaves. The c29nm utility is derived from the LLVM project's **llvm-nm** tool. To display a list of available options, use the help option (**-h**), **c29nm -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 C28x version of the C++ name utility and the c29clang version of the C++ name utility are as follows.

Symbol Kind Annotations

In the output from the c29nm utility, symbol names are annotated with an indication of their kind. The c29clang version of the c29nm 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

Debug Symbol Names

The TI C28x version of the name utility would include debug symbol names in the output. However, to include debug symbols in the output of the c29clang version of c29nm, you must specify the **c29nm**'s **--debug-syms** option on the command-line.

Functionally Equivalent Option Mappings

Several of the options available in the TI C28x version of the name utility now have functionally equivalent options with different syntax in the c29clang version of the c29nm utility. Below is a list of option mappings where the TI C28x's nm2000 option syntax is specified first and the c29clang's **c29nm** option syntax is specified second:

KEY: TI C28x **nm2000** option syntax -> TI C29x **c29nm** 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 C28x version of the name utility supported several command-line options that are no longer supported in the c29clang version of the c29nm 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 C28x version of the name utility annotates some symbols with kind information differently than the c29clang version of the c29nm 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 c29nm uses 'N'. There may be other differences. Please consult the above list of symbol kind annotations for the c29nm utility for more information.

Saving Output to a File

As indicated above, the TI C28x version of the name utility supports a command-line option to write the output to a specified file, but the c29clang version of the c29nm utility does not support such a command-line option. Instead, you can elect to pipe the output of c29nm to a file:

```
c29nm test.o > c29nmout
```

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

```
c29nm test.o | c29dem > c29demout
```

An Example Using the Name Utility (c29nm) and the Name Demangler Utility (c29dem)

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:

```
c29clang -mcpu=c29.c0 -c test.cpp
```

We can then use the **c29nm** utility to write out the symbol names in test.o:

```
%> c29nm 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 **c29nm** to **c29dem** to demangle the mangled names that are present in the c29nm output:

```
%> c29nm test.o | c29dem
00000000 T f()
00000000 D NS::ns_my_num
00000000 B g_my_num
00000000 T main
```

Disassembling Object Files

The c29clang compiler toolchain provides a utility that can be used to disassemble TI C29x object files: c29objdump.

When invoked with the **--disassemble** (or **-d**) command, **c29objdump** emits disassembled output for a C29x object file. If a C29x 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.

> **c29objdump –disassemble** [*options*] *filename*

See the *c29objdump - Object File Dumper* section for more information about the c29objdump utility.

The **dis2000** utility that is provided with the proprietary TI C28x C/C++ compiler toolchain (cl2000) has no equivalent in the c29clang compiler toolchain. Use the c29objdump utility instead.

Language Support For C/C++ and Assembly Differences

- **C/C++:** There is no support for proprietary TI predefined macro symbols, pragmas, and intrinsics. Functionally equivalent alternatives should be used in their place.
- **CLA:** There is no separate CLA compiler. Convert code intended for the CLA compiler to regular C code.
- **Assembly:** Use of assembly language is discouraged for c29clang, except for assembly code that is embedded in C/C++ source files via *asm()* statements, which are processed inline by the c29clang integrated GNU-syntax assembler. Assembly language source files should be rewritten in C/C++.

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

Development Flow Related Differences

- The c29clang compiler invokes the linker by default, whereas the user must specify cl2000's **-run_linker** option to invoked the linker from the cl2000 command-line. However, the **--rom_model** (-c) linker option is not set by default by the c29clang compiler when running the linker. Therefore, the **--rom_model** (-c) or **--ram_model** (-cr) option must be passed to the linker on the c29clang command line or in the linker command file.
- The c29clang compiler does not support a C/C++ interlist option from the compiler command-line.

- The c29clang compiler ends compilation after emitting assembly output when using the **-S** option.
- The c29clang 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

- Only EABI output is supported for TI C29x. If your TI C28x application has COFF output files, you should first migrate from TI C28x COFF output to TI C28x EABI output before migrating to TI C29x. For information, see C2000 Migration from COFF to EABI.
- The cl2000 compiler does not attempt to inline a function that contains an asm() statement, but the c29clang compiler inlines a function containing an asm() statement if it is beneficial to do so.
- The command-line interface for the c29clang versions of the C++ Name Demangler Utility (**c29dem**) and the Name Utility (**c29nm**) behave differently than the cl2000 versions of dem2000 and nm2000.

For more information, see *Binary Utility Differences*.

Differences in Type Aliasing Assumptions

- When optimizing memory accesses, c29clang 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 cl2000, 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.

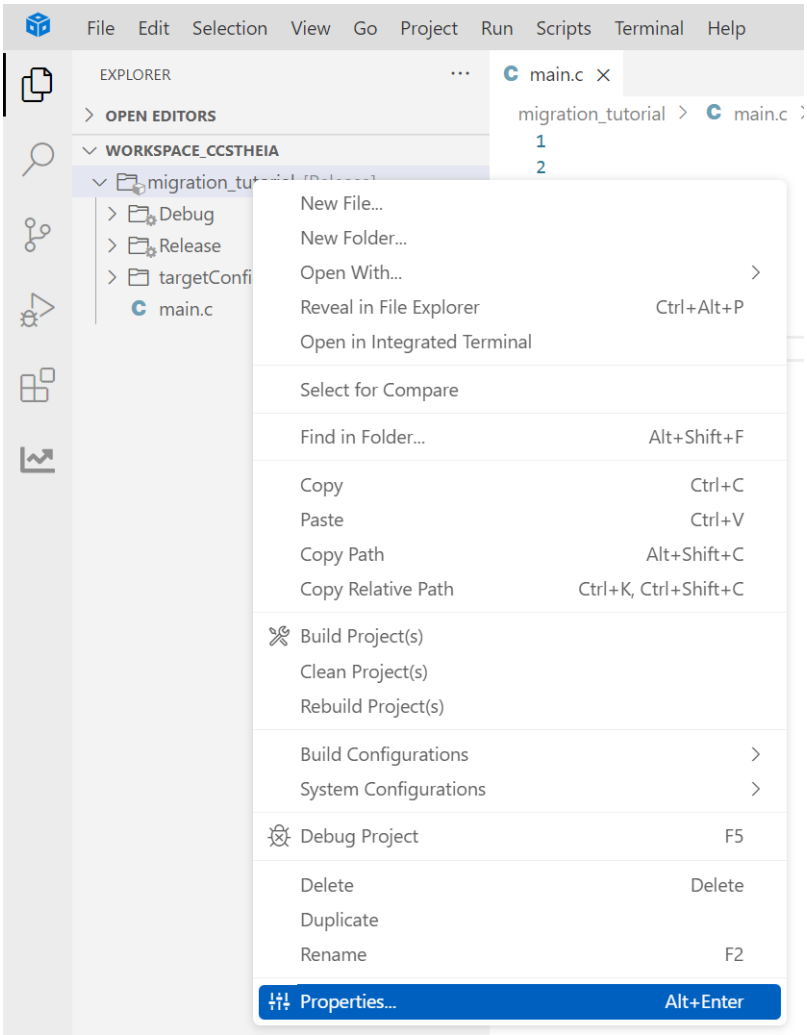
For more information, see *Controlling Optimization*.

2.2 Migrating cl2000 CCS Projects to c29clang

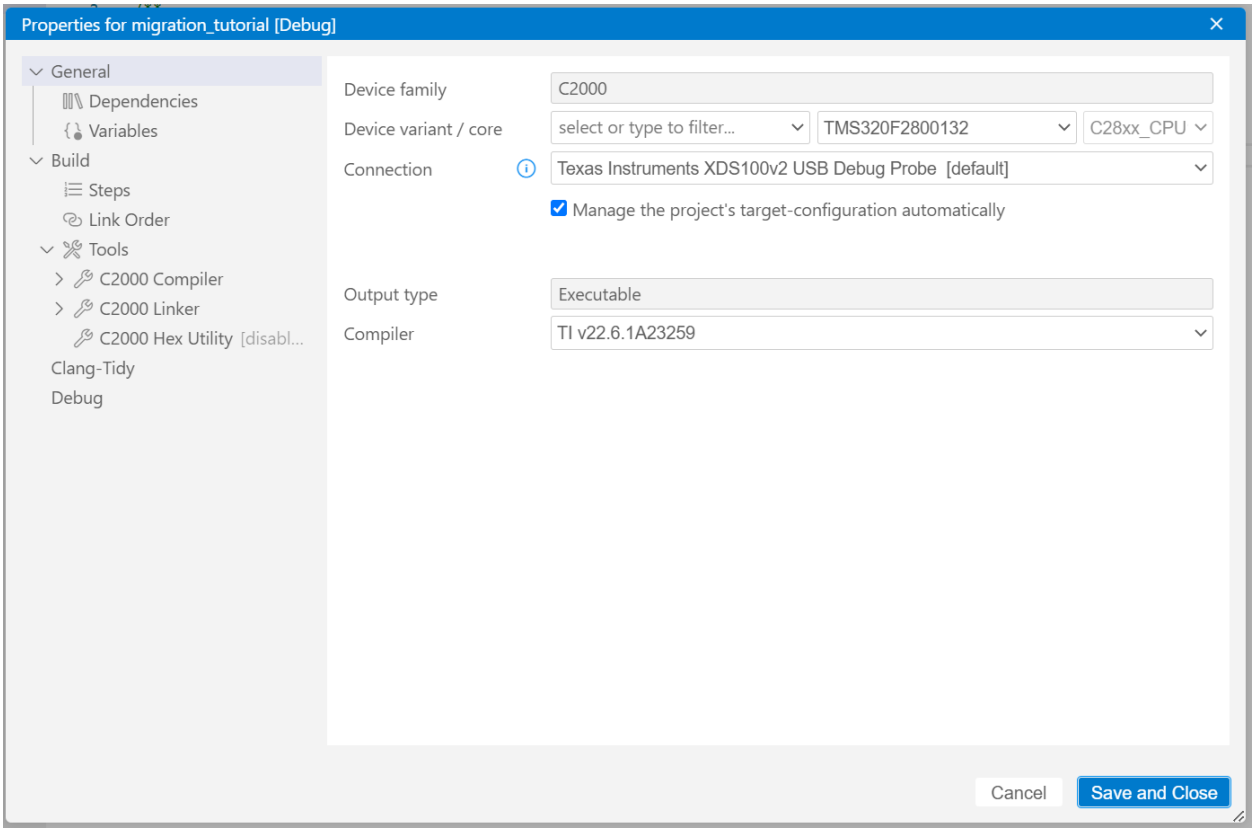
One of the challenges you may face when transitioning an existing TI C28x application in order to build the application with the c29clang compiler is in mapping cl2000 compiler options into their corresponding c29clang compiler options. If your TI C28x application exists as a CCS project, then CCS can help with the mapping of cl2000 compiler options to c29clang options.

The process of migrating an cl2000 CCS project to use the c29clang compiler is relatively straightforward:

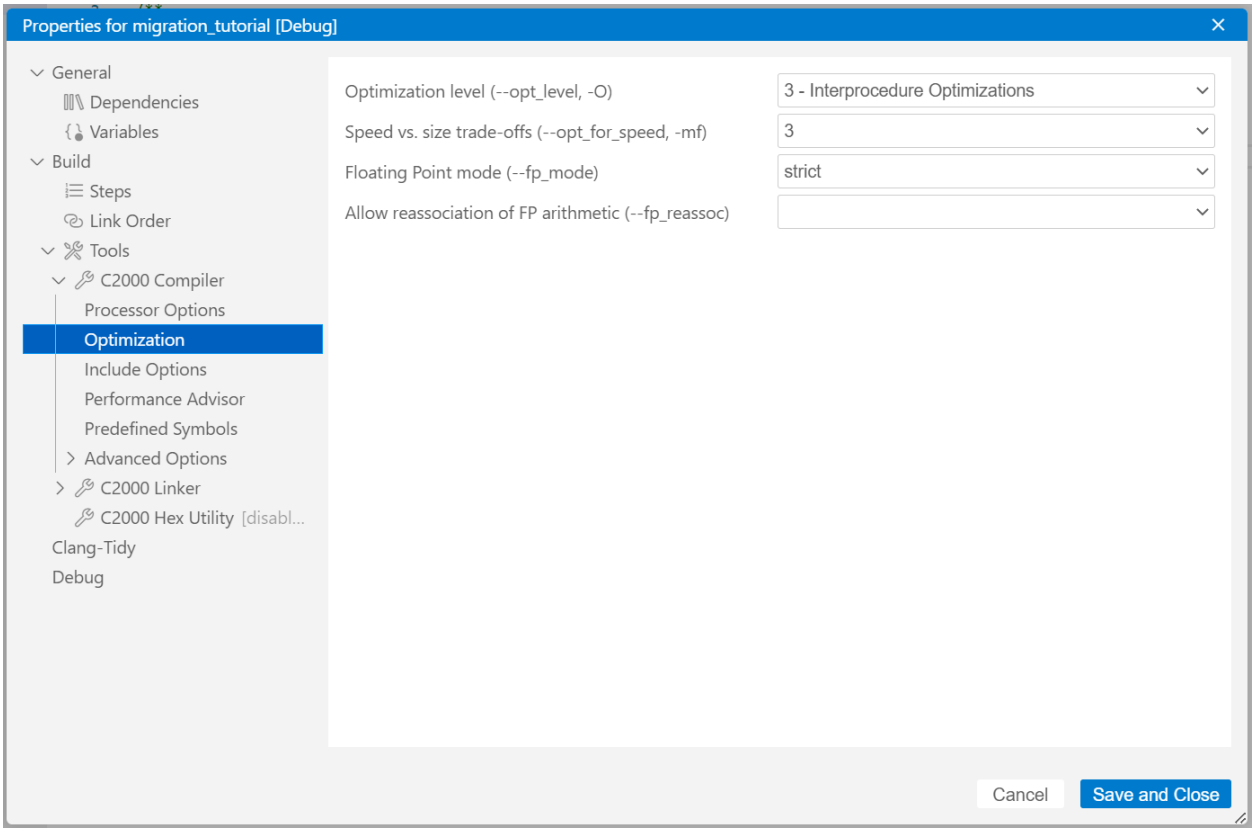
1. Import the cl2000 CCS project into a CCS workspace.
2. In the **Explorer** window, right-click on the cl2000 project name and select **Properties**.



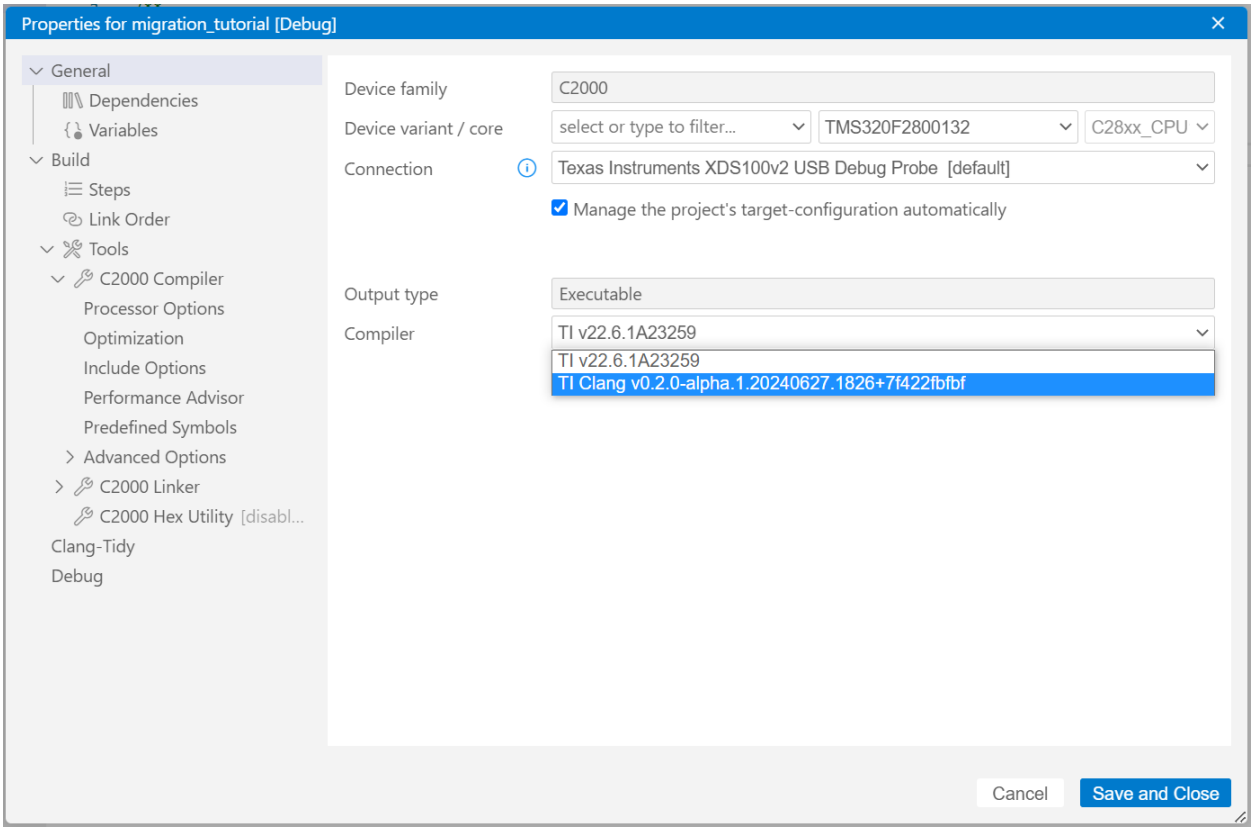
3. You will then see a **Properties** pop-up dialog box for the cl2000 project.



Before migrating the cl2000 project to use the c29clang compiler, you can check the cl2000 option settings. In this example, the **Optimization** options show that the *-O3* and *-mf3* options have been selected:

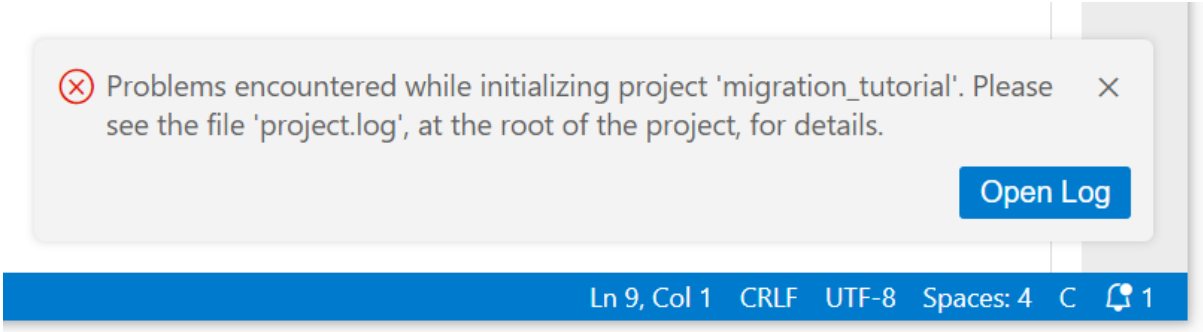


4. Click on the **General** category along the left-hand side of the dialog box, then change the **Compiler** from the current cl2000 compiler to the c29clang compiler (may be denoted as "TI Clang <version string>"):



5. Click on **Save and Close** at the bottom of the dialog box. CCS will create a new build configuration with the migrated compiler options.

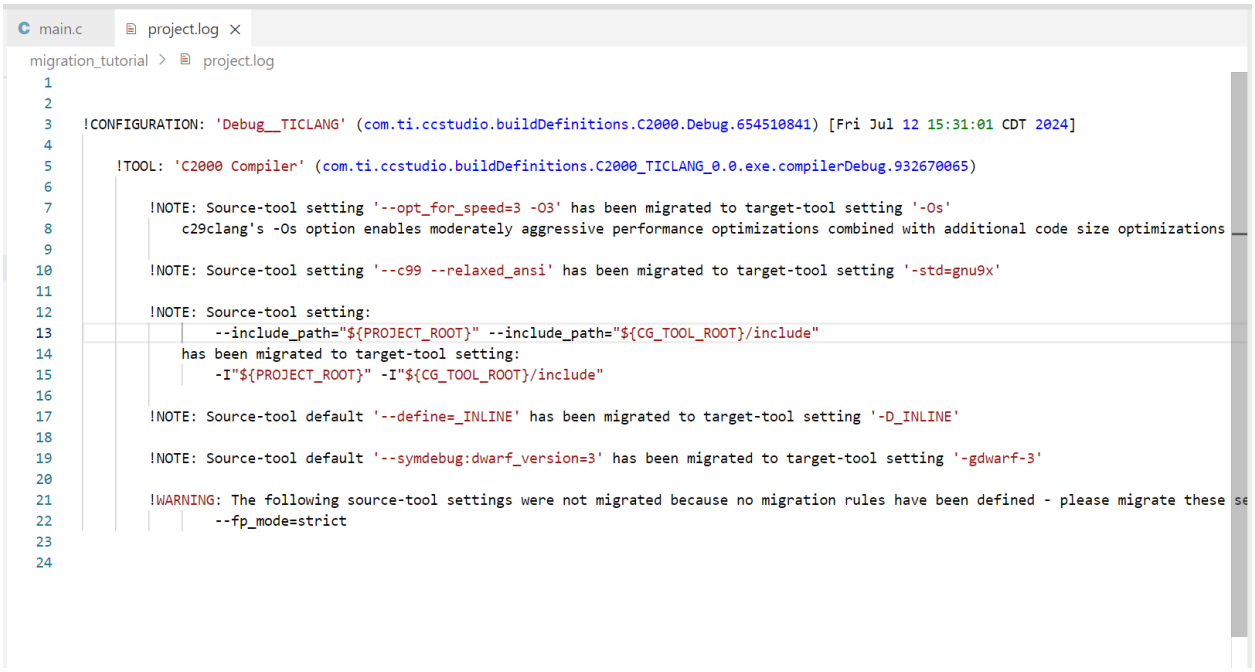
Unless all cl2000 compiler options were migrated flawlessly to their c29clang compiler counterparts, you will see a pop-up dialog box explaining that some issues were encountered when creating the new build configuration:



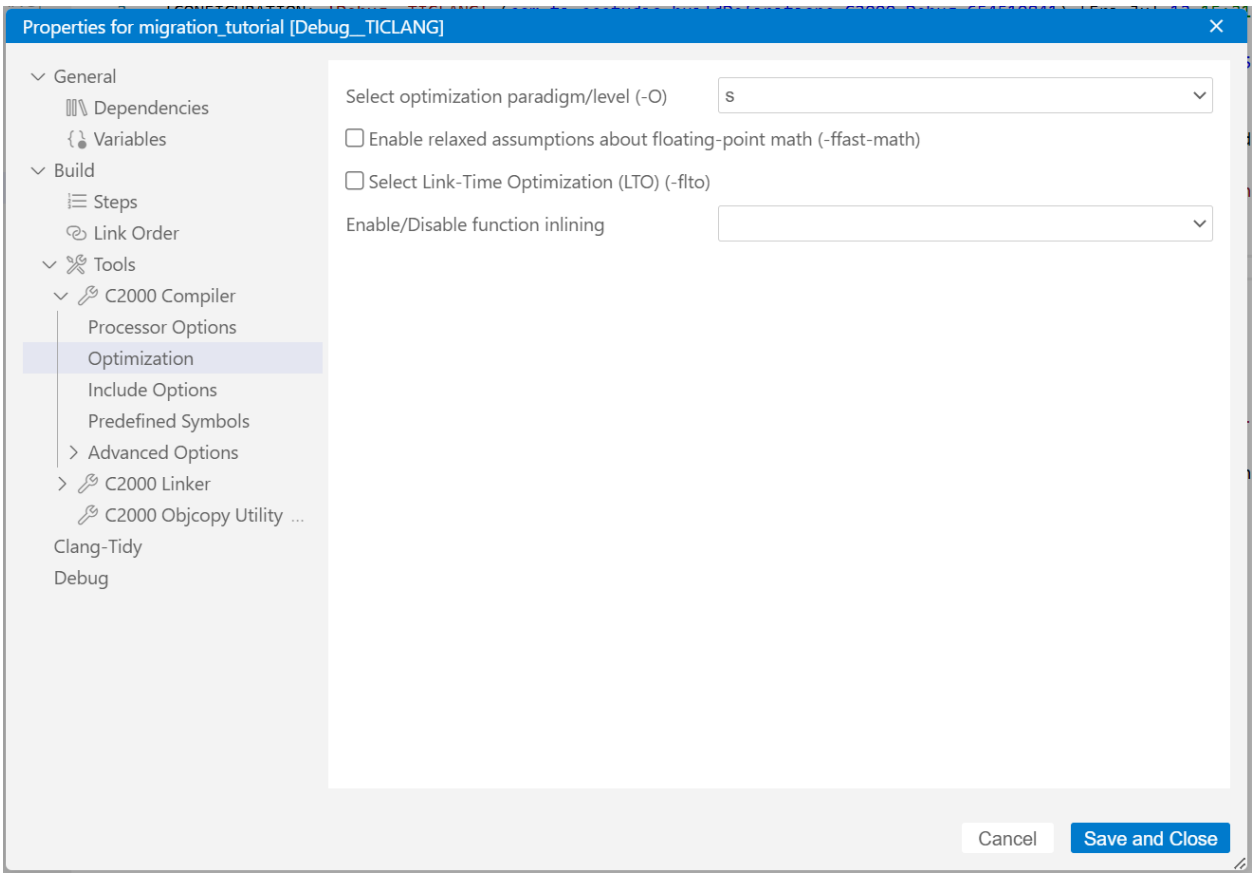
You can then click **Open Log** and proceed to view the project.log file in the CCS source file window.

The project.log file provides details about each of the cl2000 to c29clang option mappings that were enacted during the migration step. The cl2000 compiler options that CCS was not able to migrate into a functionally equivalent c29clang 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 cl2000 compiler option was mapped to a c29clang option as you expected.

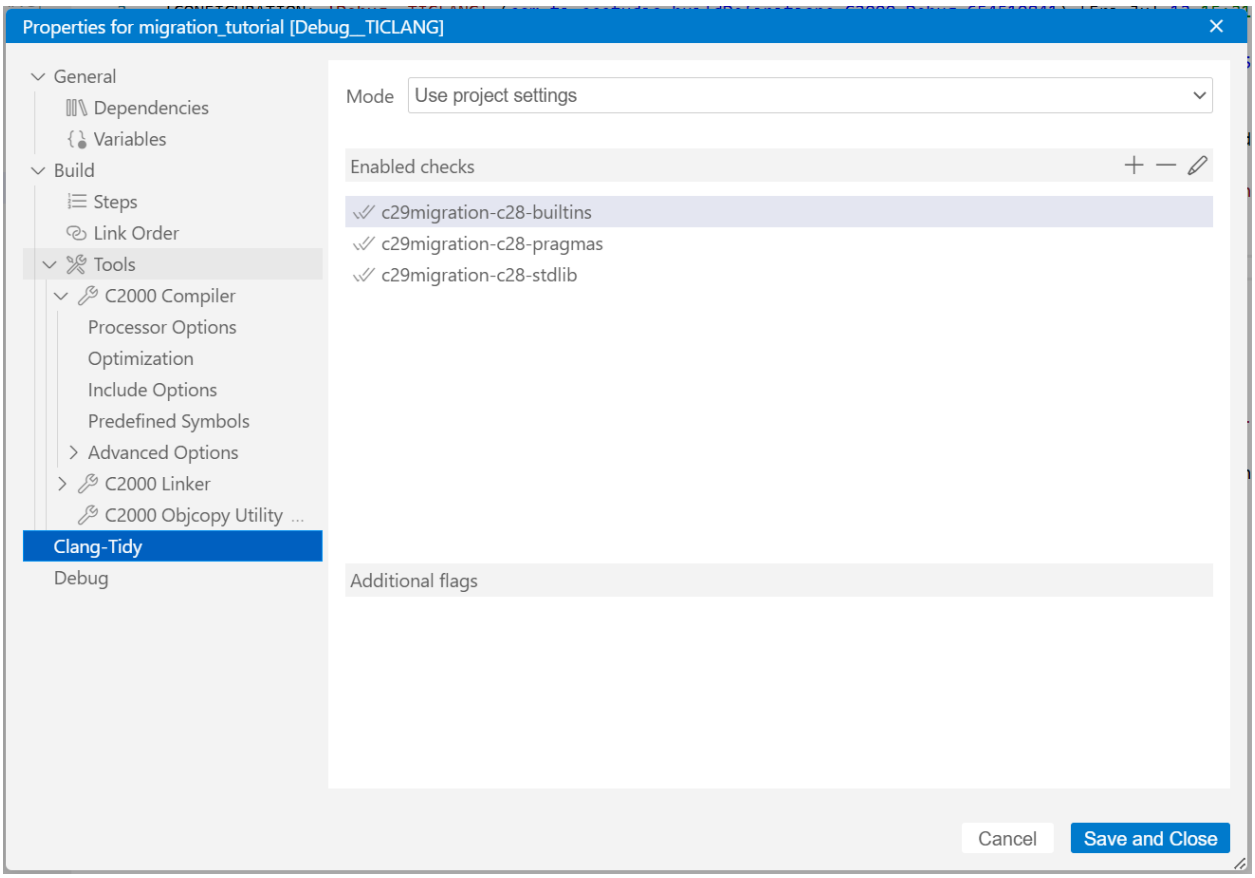
For this tutorial we can see that the cl2000 *-O3* and *-mf3* options were mapped to the c29clang *-Os* option:



To further ensure that the cl2000 options were properly converted to c29clang options, you can check the build settings for the newly created TICLANG configuration of your project. Right-click on the project name and select **Properties** to bring up the **Properties** dialog box associated with the *TICLANG* configuration of your project. For this tutorial, we can see the **Optimization** options for the c29clang compiler show that *-Os* option has been selected, as expected:



During migration, CCS will also enable a few Clang-Tidy checks that can help you update your TI C28x application source code so that it can be built using the c29clang compiler. To view or modify the enabled Clang-Tidy checks, click on **Clang-Tidy** in the **Properties** dialog:



For more information about Clang-Tidy and a list of the available checks, see *Migrating Source Code with Clang-Tidy*.

Further details about mapping cl2000 compiler options to c29clang 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 cl2000 command-line options to an appropriate c29clang command-line option, if a mapping is available. There are some cl2000 command-line options, like the -s interlisting options, that do not have a functional counterpart in the c29clang compiler. Such cases are clearly indicated in the tables in this chapter.

In several of this chapter's sub-sections, cl2000 options are shown side-by-side with one or more functionally relevant c29clang 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 cl2000 and c29clang compiler with regards to the options under consideration.

Please note that while this chapter tries to account for all the options provided by the cl2000 compiler, it does not list all the c29clang command-line options that are available. If you cannot find a c29clang 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 c29clang 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 c29clang.

cl2000 Option (and alias)	c29clang 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.

cl2000 Option (and alias)	c29clang Option
--preproc_only (-ppo)	-E

Run only the preprocessor. The cl2000 compiler saves the preprocessed output in a .pp file, but the preprocessed output from c29clang is streamed to stdout.

cl2000 Option (and alias)	c29clang Option
--skip_assembler (-n)	-S

Halt compilation after code generation. Both the cl2000 and c29clang 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 cl2000 compiler, the default file extension for a compiler generated assembly file is '.asm'. For the c29clang compiler, the file extension for a compiler generated assembly file is '.s'.

cl2000 Option (and alias)	c29clang Option
--run_linker (-z)	linker is invoked by default

The cl2000 compiler does not run the linker unless you use the --run_linker (-z) option. By default, the c29clang compiler automatically invokes the linker after compiling source files into object files. To see details about what command is used by the c29clang compiler to invoke the linker, you can specify the '-v' option on the c29clang command- line.

You can prevent c29clang 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.

cl2000 Option (and alias)	c29clang Option
-z <linker options>	-Xlinker <linker option> -Wl,<comma-separated list of linker options>

Any options specified after the -z option using cl2000 are passed to the linker.

To pass options to the linker from the c29clang command-line, use either -Xlinker or -Wl. The c29clang 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 cl2000, is not set by default by the c29clang 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 c29clang command line (or specified in the linker command file).

cl2000 Option (and alias)	c29clang 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.

cl2000 Option (and alias)	c29clang Option
--silicon_version=28 (-v28)	-mcpu=<processor variant>

The cl2000 compiler supports only the “28” value for the --silicon_version command-line option. For the c29clang compiler, use the -mcpu=c29.c0 option, which is also the default.

cl2000 Option	c29clang Option
--float_support=<float hardware ID>	-mfpu=<float hardware ID>

The cl2000 compiler uses the --float_support option to indicate whether floating-point hardware is available and if it is available, which floating-point hardware it is.

The C29x compiler toolchain emits native 32-bit floating point operations by default. Optionally, some C29x hardware supports native 64-bit floating point operations. Users may control the usage of native floating point hardware with the **-mfpu** option.

Valid forms of the **-mfpu** option include:

- **-mfpu=none** - Emulate all floating point operations in software without the use of floating-point registers or hardware support.
- **-mfpu=f32** - Use native 32-bit floating-point hardware operations, but emulate 64-bit floating-point operations in software. This is the default setting for the compiler toolchain.
- **-mfpu=f64** - Use native 32-bit and 64-bit floating-point hardware operations.

For additional information about options that can be used to manage compiler behavior with respect to floating-point support, please see *Managing Floating Point Support*.

cl2000 Option (and alias)	c29clang Option
--abi={coffabileabi}	only EABI is supported

The COFF ABI output format, which is supported by the proprietary TI C28x compiler, is not supported for TI C29x targets. The C29x Embedded Application Binary Interface (EABI) is always used.

cl2000 Option (and alias)	c29clang Option
--cla_support	Not supported

The TI C29x CPU provides improved processing performance, so separate hardware accelerators are not needed. See *Migrating CLA Code* for information about migrating C code that was intended for use with the CLA compiler.

cl2000 Option (and alias)	c29clang Option
--idiv_support	Not supported

The TI C29x CPU provides improved processing performance, so separate hardware accelerators are not needed.

cl2000 Option (and alias)	c29clang Option
--tmu_support	Not supported

The TI C29x CPU provides improved processing performance, so separate hardware accelerators are not needed.

cl2000 Option (and alias)	c29clang Option
--vcu_support	Not supported

The TI C29x CPU provides improved processing performance, so separate hardware accelerators are not needed.

cl2000 Option (and alias)	c29clang Option
--lfu_reference_elf=path	Not supported
-lfu_default[=none | preserve]	Not supported

Live Firmware Updates are not currently supported for TI C29x.

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.

cl2000 Option	c29clang Option
--c89 --c99 --c11	-std=<C standard identification>

The c29clang -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 c29clang 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 c29clang 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 c29clang 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.

cl2000 Option	c29clang Option
--c++03 --c++11 --c++14 --c++17	-std=<C++ standard identification>

The c29clang -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 c29clang 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 c29clang 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 c29clang 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.

Note: C++ support is based on a library that is focused on support for C++17. If you specify an earlier version of the C++ standard, it is not guaranteed that features that were not required by that standard will be unsupported.

cl2000 Option (and alias)	c29clang Option
--cpp_default (-fg)	-x c++ --language=c++

The cl2000 compiler provides a --cpp_default option that tells the compiler to process all source files with a '.c' extension as C++ source files.

The c29clang compiler's `-x c++` option provides similar support, but whereas cl2000's -fg option applies only to source files with '.c' extensions, the c29clang compiler's `-x c++` option applies to any source file that is specified after the `-x c++` option on the c29clang command-line up until another `-x` option is encountered.

cl2000 Option	c29clang Option
	-x <language type> --language=<language type>

The c29clang 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

For example, suppose you have two source files, t1.cpp and t2.c. Assuming t2.c is C++ compatible, one could then invoke the c29clang compiler with:

```
%> c29clang ... t1.cpp -x c++ t2.c 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.

cl2000 Option	c29clang Option
--cla_background_task	not supported
--cla_default	not supported
--cla_signed_compare_workaround	not supported

The cl2000 compiler supports these options to control the behavior of the CLA compiler.

The Control Law Accelerator (CLA) is not needed when using TI C29x devices because of their improved performance over TI C28x devices. For this reason, these

options are not supported by the c29clang compiler.

See *Migrating CLA Code* for further information.

cl2000 Option	c29clang Option
--exceptions	not supported
--extern_c_can_throw	not supported

The c29clang compiler currently does not support C++ exception handling and the -fexceptions compiler option.

cl2000 Option (and alias)	c29clang Option
--gen_cross_reference_listing (-px)	not supported

The cl2000 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 c29clang compiler does not support an analogous option.

cl2000 Option	c29clang Option
--gen_preprocessor_listing	not supported

The cl2000 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 c29clang compiler does not support an analogous option.

cl2000 Option	c29clang Option
--pending_instantiations=#	

The cl2000 compiler supports these options to specify the number of template instantiations that may be in progress at any given time.

cl2000 Option (and alias)	c29clang Option
--relaxed_ansi (-pr)	-std=gnu<90|99|11|17>

The cl2000 compiler supports GNU extensions to the C language if its -pr option is selected. To enable support of GNU extensions in the c29clang compiler, use one of the GNU settings (gnu90, gnu99, gnu11, gnu17) as the argument to c29clang's -std option.

cl2000 Option	c29clang Option
--rtti	-fno-rtti (default) -frtti

The cl2000 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 c29clang compiler has RTTI support disabled by default. Use the -frtti option to allow C++ RTTI to be generated.

cl2000 Option (and alias)	c29clang Option
--strict_ansi (-ps)	-std=c<90|99|11|17>

The cl2000 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 c29clang compiler and approximate the behavior of cl2000's -ps option, use one of the non-GNU language settings (c90, c99, c11, c17) as the argument to c29clang's -std option.

All c29clang "-std=c<XX>" C language variants define the __STRICT_ANSI__ predefined macro symbol.

You can combine "-std=c<XX>" with c29clang's -pedantic option, which causes warnings to be issued for any conflicts with ISO C and ISO C++. The c29clang compiler's -pedantic-errors option causes errors instead of warnings to be issued for such conflicts.

2.3.4 Controlling Optimization

The following command-line options control optimization behavior.

cl2000 Option (and alias)	c29clang Option
--opt_level=<off|0|1|2|3> (-O<off|0|1|2|3>	-O<0|1|2|3|g|s|z>
--opt_level=4 (-O4)	-flto -O<1|2|3|s|z>

The cl2000 compiler supports several levels of optimization beginning with *--opt_level=off*, or "no optimization," up to *-O4*, which enables cl2000's link time program optimization capability.

The c29clang compiler supports a variety of different optimization options, including:

- **-O0** - no optimization; generates code that is debug-friendly.
- **-O1** or **-O** - restricted optimizations, providing a good trade-off between code size and debuggability.
- **-O2** - most optimizations enabled with an eye towards preserving a reasonable compile-time.
- **-O3** - in addition to optimizations available at *-O2*, *-O3* enables optimizations that take longer to perform, trading an increase in compile-time for potential performance improvements.
- **-Og** - enables most optimizations from *-O1*, but may disable some optimizations to improve debuggability.

- **-Os** - enables all optimizations from *-O2* plus some additional optimizations intended to reduce code size while mitigating negative effects on performance.
- **-Oz** - enables all optimizations from *-Os* plus additional optimizations to further reduce code size with the risk of sacrificing performance.

The cl2000 -O4 option maps to a combination of the c29clang *-flto* option and one of the c29clang optimization level options. The c29clang *-flto* option enables link-time optimization in the c29clang linker. Note that the c29clang *-O0* and *-Og* options are not allowed in combination with the c29clang -flto option.

It is reasonable to expect that the ability to debug a program becomes more challenging at higher levels of optimization. This is especially true when c29clang's link-time optimization is enabled via the *-flto* option.

cl2000 Option (and alias)	c29clang Option
--opt_for_speed=<0|1|2|3|4|5> (-mf=<0|1|2|3|4|5>)	-O<z|s|3>
--opt_level=4 --opt_for_speed=<0|1|2|3|4|5> (-mf=<0|1|2|3|4|5>)	-flto -O<z|s|3 >

The cl2000 compiler supports an *--opt_for_speed* option, which allows you to select a code size versus performance "trade-off" level, *n*, which informs the compiler about how aggressive it can be when optimizing for improved performance at the risk of increasing code size. The available values for *n* range from 0, which favors optimizations geared towards reducing code size with a high risk of degrading performance, to 5, which favors optimizations intended to improve performance with a high risk of increasing code size.

Some of the c29clang compiler's optimization options roughly correspond to the intended code size vs. performance trade-off that is embodied in the use of cl2000's *--opt_for_speed* and *--opt_level* options. The following is an approximate mapping:

- **-Oz** - resembles using cl2000's *--opt_for_speed=0-1* in combination with *--opt_level=2-3* since it favors code size reducing optimizations even if performance is degraded.
- **-Os** - resembles using cl2000's *--opt_for_speed=2-3* in combination with *--opt_level=2-3* since it favors code size reducing optimizations, but tries to preserve performance while doing so.
- **-O3** - resembles using cl2000's *--opt_for_speed=3-4* in combination with *--opt_level=2-3* since it favors optimizations intended for improving performance, but tries to avoid increases in code size while doing so.

If link-time optimization is enabled via cl2000's -O4 optimization level, this will map to the c29clang *-flto* option to enable link- time optimization in the c29clang linker.

The cl2000 *--opt_for_speed* option will then map to one of four c29clang option combinations as follows:

- **-flto -Oz** - resembles using cl2000's *--opt_for_speed=0-1* in combination with *--opt_level=4*, favoring code size reducing inter-module optimizations even if performance is degraded.
- **-flto -Os** - resembles using cl2000's *--opt_for_speed=2-3* in combination with *--opt_level=4*, favoring code size reducing inter-module optimizations while trying to preserve performance.
- **-flto -O3** - resembles using cl2000's *--opt_for_speed=3-4* in combination with *--opt_level=4*, favoring performance improving inter-module optimizations while trying to avoid significant increases in code size.

cl2000 Option (and alias)	c29clang Option
--opt_for_space=<0|1|2|3>	-O<1|s|z>

The cl2000 compiler supports an --opt_for_space option, which is an older option for controlling code space.

The c29clang compiler reduces code size to different degrees when you use the -O, -Os, and -Oz options.

cl2000 Option	c29clang Option
--sat_reassoc=off (default) --sat_reassoc=on	not supported

The cl2000 compiler provides a --sat_reassoc option to enable or disable reassociation of saturating arithmetic. It is off by default.

The c29clang compiler does not support an analogous option.

cl2000 Option (and alias)	c29clang Option
--auto_inline=<size> (-oi<size>	-finline-limit=<size>

The cl2000 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 c29clang 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 c29clang 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 c29clang compiler's -fno-inline-functions option can be used to disable all inlining.

cl2000 Option (and alias)	c29clang Option
--disable_inlining	-fno-inline-functions

The cl2000 compiler provides the --disable_inlining option, which allows you prevent any inlining from being performed.

To prevent inlining with the c29clang compiler, use the -fno-inline-functions option.

cl2000 Option (and alias)	c29clang Option
--call_assumptions=<n> (-op<n>)	not supported

The cl2000 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.

cl2000 Option (and alias)	c29clang 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 cl2000 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 c29clang compiler does not provide an option that matches the exact behavior of cl2000's --gen_opt_info, but c29clang 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

cl2000 Option (and alias)	c29clang Option
--optimizer_interlist (-os)	not supported

The cl2000 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 c29clang compiler does not provide an analogous option. However, you can use c29clang'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.

cl2000 Option (and alias)	c29clang Option
--program_level_compile (-pm)	-flto

The cl2000 compiler's --program_level_compile option combines source files into a single compilation unit to enable the compiler's program-level optimizations.

The c29clang -flto option enables inter-module optimizations via link-time optimization. The c29clang -flto option can be combined with the c29clang -O<1|2|3|s|z> optimization level option to instruct the compiler whether to prioritize improving performance over reducing code size or vice versa.

cl2000 Option (and alias)	c29clang Option
--aliased_variables (-ma)	not supported

The cl2000 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 c29clang compiler does not provide an analogous option. However, c29clang'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 c29clang's optimization options.

cl2000 Option (and alias)	c29clang Option
--isr_save_vcu_regs={on|off}	not supported

The cl2000 compiler provides the --isr_save_vcu_regs compiler option, which generates instructions to save and restore VCU registers using the stack when interrupt service routines occur. This allows VCU code to be re-entrant. If an ISR interrupts a VCU computation, it will not impact results if this option is used.

The c29clang compiler does not provide an analogous option, because C29x devices do not contain VCU registers.

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.

cl2000 Option	c29clang Option
--float_operations_allowed=<all|none|32|64>	not supported

The cl2000 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 cl2000 compiler to flag an error if an attempt is made to use a floating-point type whose size is greater than 32-bits.

The c29clang compiler does not provide a mechanism to restrict the use of floating-point types by type size. The c29clang compiler assumes that all legal floating-point types are supported. This matches the cl2000 compiler's default behavior (e.g. --float_operations_allowed=all).

cl2000 Option	c29clang 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>

By default, the cl2000 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 instruc-

tions, allow 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 c29clang 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 c29clang command-line, then the compiler will assume the 'precise' floating-point model by default.

  '-ffp-model=precise' is the recommended c29clang compiler option to be used in place of cl2000'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 C29x processor's floating-point unit (FPU) supports a fused multiply and add (FMA) instruction, then the compiler will assume that any floating-point contraction optimizations are safe (*-ffp-contract=on*).

- **strict** - Specifying 'strict' as the argument to the -ffp-model option enables floating-point rounding (*-frounding-math*). 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 c29clang '-ffp-model=fast' option is the most functionally similar to the cl2000 '--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.

cl2000 Option	c29clang Option
--fp_reassoc	not supported

The c29clang compiler does not provide an option that controls whether reassociation is allowed in floating-point operations. Instead, the c29clang's -ffp-mode=std option can be used to disallow reassociation (the default), and c29clang's -ffp-mode=fast option can be used to allow reassociation.

cl2000 Option	c29clang Option
--fp_single_precision_constant	-cl-single-precision-constant

The cl2000 compiler supports a --fp_single_precision_constant option, which causes all unsuffixed floating-point constants to be treated as single-precision values instead of as double-precision constants.

The -cl-single-precision-constant option can be used with the c29clang compiler to treat double precision floating-point constants as single precision constants.

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.

cl2000 Option	c29clang Option
--gen_func_subsections=on (default) --gen_func_subsections=off	-ffunction-sections (default) -fno-function-sections

The cl2000 compiler's --gen_func_subsections option, which is on by default, places each function definition into a separate subsection.

The analogous c29clang option is -ffunction-sections, also on by default, which generates each function definition into its own section. You can further control section 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.

cl2000 Option	c29clang Option
--gen_data_subsections=on (default) --gen_data_subsections=off	-fdata-sections (default) -fno-data-sections

The cl2000 compiler's --gen_data_subsections option, which is on by default, places aggregate data (arrays, structs, and unions) into separate subsections.

The analogous c29clang 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.

cl2000 Option (and alias)	c29clang Option
--no_rpt (-mi)	not supported

The cl2000 provides this option to prevent the compiler from generating repeat (RPT) instructions. By default, repeat instructions are generated for certain memcpy, division, and multiply-accumulate operations. However, repeat instructions are not interruptible.

The c29clang compiler does not provide an analogous option.

cl2000 Option	c29clang Option
--printf_support=<nofloat|full|minimal>	automatic

The cl2000'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 c29clang 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.

cl2000 Option (and alias)	c29clang Option
--protect_volatile (-mv)	not supported

The cl2000 provides this option to enable volatile reference protection. Pipeline conflicts may occur between non-local variables that have been declared volatile. A conflict can occur between a write to one volatile variable that is followed by a read from a different volatile variable. The –protect_volatile option allows instructions to be placed between volatile references to ensure the write occurs before the read.

The c29clang compiler does not provide an analogous option.

cl2000 Option	c29clang Option
--ramfunc=off (default) --ramfunc=on	not supported

The cl2000 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 c29clang 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.

cl2000 Option (and alias)	c29clang Option
--rpt_threshold=k	not supported

The cl2000 provides this option to set a maximum value on the number of times generated repeat (RPT) loops, which are not interruptible, may iterate.

The c29clang compiler does not provide an analogous option.

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.

cl2000 Include File Directory Search Path

The cl2000 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 cl2000 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 C2000_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 C2000_C_DIR environment variable definition

By default, the cl2000 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 C2000_C_DIR, but the cl2000 compiler also honors the cl2000 environment variable C_DIR. In addition, the cl2000 compiler allows the C2000_C_OPTION environment variable to define a set of compiler options to be used as if they were on the command line.

c29clang Include File Directory Search Path

The c29clang 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 c29clang'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 c29clang 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 c29clang 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 c29clang compiler tools are installed. For example, the following include file directories are installed with the c29clang 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 c29clang 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 C2000_C_DIR, C_DIR, and C2000_C_OPTION environment variables are not supported by the c29clang 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.

cl2000 Option (and alias)	c29clang Option
--include_path=<dir> (-I=<dir>) (-i=<dir>)	-I

When using the cl2000 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 "cl2000 Include File Directory Search Path" section.

Likewise, using the c29clang 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 "c29clang 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 c29clang 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 c29clang invocation.
4. Directories listed in the C2000_C_DIR environment variable definition.

By default, the cl2000 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 C2000_C_DIR, but the cl2000 compiler also honors the cl2000 environment variable C_DIR. In addition, the cl2000 compiler allows the C2000_C_OPTION environment variable to define a set of compiler options to be used as if they were on the command line.

Unlike the cl2000 compiler, the c29clang 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 c29clang compiler tools are installed.

You may also control the object/library file directory search path using the C2000_C_DIR environment variable when the linker is invoked with either the cl2000 or c29clang compiler. The linker also honors the cl2000 environment variable C_DIR. Use of the C2000_C_OPTION environment variable should be migrated to command line options and environment variables that are supported by the c29clang compiler.

Please note that while the c29lnk linker that is provided with the c29clang compiler tools is identical to the lnk linker that is provided with the cl2000 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 cl2000 and c29clang* 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.

cl2000 Option (and alias)	c29clang Option
--include_path=<dir list> (-I=<dir list>) (-i=<dir list>)	-L <dir>

When using the cl2000 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 cl2000 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 cl2000 compiler command-line or within a linker command file using a '-l' prefix.

The c29clang compiler's -L option is functionally equivalent to the cl2000 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 c29clang command-line. However, you can specify more than one -L option on the c29clang 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.

cl2000 Option	c29clang Option
-z ... --search_path=<dir list>	-Xlinker --search_path=<dir> -Wl,–search_path,<dir>

When the linker is invoked from the cl2000 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 c29clang 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 c29clang compiler's -L option, only one directory path may be specified to the --search_path option when passed to the linker from the c29clang compiler command-line.

2.3.9 Specifying Temp Directories

The cl2000 compiler provides several options that allow you to control where a temporary file is written during a given compilation. However, the c29clang 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 c29clang compiler places temporary files in the current working directory (that is, whichever directory the c29clang executable was invoked from). Ordinarily, temporary files are removed when they are no longer needed by the c29clang compiler. However, like the cl2000 compiler, the c29clang compiler does support command-line options that keep one or more of the temporary files that are generated during a given compilation.

cl2000 Option (and alias)	c29clang Option
--abs_directory=<dir> (-fb)	not supported

The cl2000 compiler supports an absolute listing capability (-abs), which is not provided in the c29clang toolset. Thus the c29clang compiler does not provide an option to control where an absolute listing file would be written.

cl2000 Option (and alias)	c29clang Option
--asm_directory=<dir> (-fs)	-S -save-temps

The cl2000 compiler allows you to indicate where a temporary compiler-generated assembly file should be written, but the c29clang compiler does not provide this capability. The c29clang 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 c29clang 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 c29clang assembler (.o extension).

Note that the use of assembly language is discouraged for c29clang, except for assembly code that is embedded in C/C++ source files via *asm()* statements, which are processed inline by the c29clang integrated GNU-syntax assembler. Assembly language source files should be rewritten in C/C++. See *Migrating Assembly Language Source Code* for more information.

cl2000 Option (and alias)	c29clang Option
--list_directory=<dir> (-ff)	not supported

The cl2000 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 cl2000's --list_directory option allows you to indicate where to write the assembly listing file during the compilation.

The c29clang 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.

cl2000 Option (and alias)	c29clang Option
--obj_directory=<dir> (-fr)	-c -save-temps

The cl2000 compiler allows you to indicate where a temporary assembler-generated object file should be written, but the c29clang compiler does not provide this capability. The c29clang 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 c29clang 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 c29clang assembler (.o extension).

cl2000 Option (and alias)	c29clang Option
--output_file=<file> (-o)	-o <file>

Both the cl2000 and c29clang compilers support a -o option, which allows you to specify the name and location of the linked output file.

cl2000 Option	c29clang Option
--pp_directory=<dir>	-E

The cl2000 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 cl2000's --pp_directory option allows you to specify where to write the pre-processor file (.pp extension) during a given compilation.

Whereas the cl2000 compiler can be made to generate a pre-processor file with a .pp extension, the c29clang compiler supports the -E option, which writes the pre-processor output to stdout. You can direct c29clang's pre-processor output to a file using the appropriate UNIX or MS-DOS "pipe" command notation.

cl2000 Option (and alias)	c29clang Option
--temp_directory=<dir> (-ft)	not supported

The cl2000 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 c29clang compiler writes temporary files to the current working directory (where c29clang 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 c29clang's -save-temps option. The c29clang 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.

cl2000 Option (and alias)	c29clang Option
--asm_file=<file> (-fa=<file>)	-x assembler
	-x assembler-with-cpp

Note: Use of assembly language is discouraged for c29clang, except for assembly code that is embedded in C/C++ source files via *asm()* statements, which are processed inline by the c29clang integrated GNU-syntax assembler. Assembly language source files should be rewritten in C/C++. See *Migrating Assembly Language Source Code* for more information.

The cl2000 compiler provides the --asm_file option to identify a specific file as an assembly source file regardless of its extension.

The c29clang 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.

Note that like other instances of the c29clang 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.

cl2000 Option (and alias)	c29clang Option
--c_file=<file> (-fc=<file>)	-x c

The cl2000 compiler provides the --c_file option to identify a specific file as a C source file regardless of its extension.

The c29clang compiler processes source files specified on the command line after the '-x c' option as C source files. Like other instances of the c29clang 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.

cl2000 Option (and alias)	c29clang Option
--cpp_default (-fg)	-x c++
--cpp_file=<file> (-fp=<file>)	

The cl2000 compiler provides the --cpp_default file option to indicate that C files (with '.c' file extension) should be interpreted as C++ source files. The cl2000 compiler also provides the --cpp_file option to identify a specific file as a C++ source file regardless of its extension.

The c29clang compiler processes source files specified on the command line after the '-x c++' option as C source files. Like other instances of the c29clang 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.

cl2000 Option (and alias)	c29clang Option
--obj_file=<file> (-fo=<file>)	not supported

The cl2000 compiler provides the --obj_file option to identify a specific file as an object file regardless of its extension.

The c29clang 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 c29clang command- line, the c29clang 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 c29clang 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.

cl2000 Option	c29clang Option
--preinclude=<file>	-include <file>

The cl2000 compiler provides the --preinclude option to include a source file at the beginning of compilation.

The c29clang compiler's -include option provides the same functionality.

cl2000 Option (and alias)	c29clang Option
--asm_extension=<ext> (-ea=<ext>)	not supported

Note: Use of assembly language is discouraged for c29clang, except for assembly code that is embedded in C/C++ source files via *asm()* statements, which are processed inline by the c29clang integrated GNU-syntax assembler. Assembly language source

files should be rewritten in C/C++. See *Migrating Assembly Language Source Code* for more information.

The cl2000 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 c29clang 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.

cl2000 Option (and alias)	c29clang Option
--c_extension=<ext> (-ec=<ext>)	not supported

The cl2000 compiler provides the --c_extension option to indicate that files with the specified extension (<ext>) should be interpreted as C source files.

The c29clang compiler does not provide support for changing default file extensions. Files with the .c extension are compiled as C.

cl2000 Option (and alias)	c29clang Option
--cpp_extension=<ext> (-ep=<ext>)	not supported

The cl2000 compiler provides the --cpp_extension option to indicate that files with the specified extension (<ext>) should be interpreted as C++ source files.

The c29clang 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 c29clang command-line).

cl2000 Option (and alias)	c29clang Option
--listing_extension=<ext> (-es=<ext>)	not supported

The cl2000 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 c29clang 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.

cl2000 Option (and alias)	c29clang Option
--obj_extension=<ext> (-eo=<ext>)	not supported

The cl2000 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 c29clang compiler does not provide support for changing default file extensions. The c29clang 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 c29clang 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.

cl2000 Option (and alias)	c29clang Option
--define=<name>[=<value>] (-D=<name>[=<value>])	-D<name>[=<value>]

The cl2000 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 c29clang compiler's -D option provides the same functionality. The c29clang 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.

cl2000 Option (and alias)	c29clang Option
--undefine=<name> (-U=<name>)	-U<name>

The cl2000 compiler provides the --undefine option to undefine a preprocessor macro.

The c29clang compiler's -U option provides the same functionality.

cl2000 Option (and alias)	c29clang Option
--preproc_dependency[=<file>] (-ppd[=<file>])	-M

The cl2000 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 c29clang compiler's -M option provides the same functionality. However, the preprocessed output is sent to stdout by default.

cl2000 Option (and alias)	c29clang Option
--preproc_includes[=<file>] (-ppi[=<file>])	-MM

The cl2000 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 c29clang compiler's -MM option provides the same functionality. However, the preprocessed output is sent to stdout by default.

cl2000 Option (and alias)	c29clang Option
--preproc_only (-ppo)	-E

The cl2000 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 c29clang compiler's -E option provides the same functionality. However, the preprocessed output is sent to stdout by default.

cl2000 Option (and alias)	c29clang Option
--preproc_macros{=<file>] (-ppm{=<file>])	-E -dM

The cl2000 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 c29clang compiler's -E and -dM options used together provide the same functionality. However, the preprocessed output is sent to stdout by default.

cl2000 Option (and alias)	c29clang Option
--preproc_with_comment (-ppc)	-E -C

The cl2000 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 c29clang compiler's -E and -C options used together provide the same functionality. However, output is sent to stdout by default.

cl2000 Option (and alias)	c29clang Option
--preproc_with_compile (-ppa)	not supported

The cl2000 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 c29clang compiler does not support continuing compilation after generating preprocessor output with the -E option.

cl2000 Option (and alias)	c29clang Option
--preproc_with_line (-ppl)	-E

The cl2000 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 c29clang compiler's -E option stops the compiler after the preprocessing stage. The preprocessed source code is emitted to stdout containing line-control information. The c29clang 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 c29clang compiler tools do not support entry/exit hooks in the same way as the cl2000 compiler. However, c29clang 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.

cl2000 Option	c29clang Option
--entry_hook=<func>	-finstrument_functions

The cl2000 compiler's *--entry_hook* option allows you to specify the name of a function to be called on entry.

The c29clang 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 c29clang 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 c29clang 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 c29clang's *-finstrument-functions* option.

cl2000 Option	c29clang Option
--entry_parm=<none|name|address>	not supported

When using entry hook functions with the cl2000 compiler, you can pass the name or the address of the calling function as an argument to the entry hook function.

The c29clang compiler does not support an analogous capability.

cl2000 Option	c29clang Option
--exit_hook=<func>	-finstrument_functions

The cl2000 compiler's *--exit_hook* option allows you to specify the name of a function to be called before exiting.

The c29clang 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 c29clang 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 c29clang 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 c29clang's *-finstrument-functions* option.

cl2000 Option	c29clang Option
--exit_parm=<none|name|address>	not supported

When using exit hook functions with the cl2000 compiler, you can pass the name or the address of the calling function as an argument to the exit hook function.

The c29clang compiler does not support an analogous capability.

cl2000 Option	c29clang Option
--remove_hooks_when_inlining	not supported

If the cl2000 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 c29clang 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.

cl2000 Option (and alias)	c29clang Option
--symdebug:dwarf (-g)	-g

The -g option causes both the cl2000 and c29clang compilers generate debug information for a compilation unit in accordance with the DWARF standard. When the -g option is specified, the cl2000 compiler generates DWARF version 3 debug information by default. The c29clang 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 c29clang 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.

cl2000 Option	c29clang Option
--symdebug:dwarf_version=<version>	-gdwarf-<version> -gdwarf-3

The cl2000 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 c29clang 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 c29clang compiler tools.

The c29clang compiler provides the analogous -gdwarf-<version> option, allowing you to select between DWARF versions 2, 3 (the c29clang default), or 4.

cl2000 Option (and alias)	c29clang Option
--symdebug:none	(default)

Even if the -g option is not specified on the command-line, the cl2000 compiler still generates DWARF version 3 debug information by default. The cl2000 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 c29clang compiler is to not generate any DWARF debug information unless the -g or the -gdwarf-<version> option is specified on the c29clang command-line.

2.3.14 Diagnostic Message Options

Whereas the cl2000 compiler identifies diagnostics by number, the c29clang compiler identifies diagnostics by name. The following table explains how cl2000 diagnostics are managed via the cl2000 compiler and how certain diagnostic-related functionality in the cl2000 compiler might translate into a relevant c29clang option.

See *Diagnostic Options* for further information.

cl2000 Option	c29clang Option
--compiler_revision	--version-string

The cl2000 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 c29clang 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.

cl2000 Option	c29clang Option
--tool_version (-version)	--version

Both the cl2000 and c29clang compilers support an option to print out version information about the compiler to stdout. The cl2000 compiler also supports a -version option, which lists the version information associated with each of the executable components in the cl2000 compiler tools package.

The c29clang 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.

cl2000 Option (and alias)	c29clang Option
--diag_error=<number> (-pdse=<number>)	-Werror=<category>
	-W<category>
--diag_remark=<number> (-pdsr=<number>)	
	-Wno-<category>
--diag_suppress=<number> (-pds=<number>)	
--diag_warning=<number> (-pdsw=<number>)	

The cl2000 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 c29clang compiler provides several options that are similar to the cl2000 --diag_[error|remark|warning|suppress] options, but there are subtle differences in functionality:

- -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

cl2000 Option	c29clang Option
--diag_wrap=<on|off>	not supported

The cl2000's --diag_wrap option, which is on by default, tells the compiler to wrap diagnostic messages at 79 columns.

The c29clang compiler does not provide this capability.

cl2000 Option (and alias)	c29clang Option
--display_error_number (-pden)	-fdiagnostics-show-option (default)
	-fno-diagnostics-show-option

In order to determine the identity of a particular diagnostic, the cl2000 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 cl2000's diagnostic control options such as --diag_suppress, for example.

Similarly, the c29clang 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 c29clang's diagnostic control options (like -Werror=<category>, for example, which treats warnings that are flagged by the specified <category> as errors).

cl2000 Option (and alias)	c29clang Option
--emit_warnings_as_errors (-pdew)	-Werror[=<category>]
	-Wno-error=<category>

The cl2000 compiler's option --emit_warnings_as_errors functionally maps to c29clang'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 c29clang 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.

cl2000 Option	c29clang Option
--flash_prefetch_warn	not supported

The cl2000 compiler provides the --flash_prefetch_warn to display warnings about a specific sequence of instructions that may cause a prefetch buffer overflow. This case does not apply to C29x devices.

The c29clang compiler does not provide an analogous option.

cl2000 Option (and alias)	c29clang Option
--issue_remarks (-pdr)	not supported

The cl2000 compiler can be made to emit remark diagnostics (non-serious warnings) during a compilation when the --issue_remarks option is specified.

While the c29clang 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).

cl2000 Option (and alias)	c29clang Option
--no_warnings (-pdw)	-w

Both the cl2000 and c29clang compilers support an option to disable the reporting of all warning diagnostics. On the cl2000 compiler, this option is --no_warnings (or -pdw). On c29clang, it is simply -w. Lower case 'w' is essentially the opposite of upper case 'W', which enables all diagnostic warnings.)

cl2000 Option (and alias)	c29clang Option
--quiet (-q)	(default)

The cl2000 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 cl2000's -q option.

The c29clang compiler does not generate progress or status information even while compiling more than one file, so there is no need for a -q option.

cl2000 Option (and alias)	c29clang Option
--set_error_limit=<number> (-pdel=<number>)	-ferror-limit=<number>

Both the cl2000 and c29clang compilers provide an option that allows you to indicate the number of errors to be detected / reported before a compilation attempt is aborted. The cl2000 compiler uses the --set_error_limit option for this purpose. The c29clang compiler's -ferror-limit serves the same purpose.

By default, the cl2000 compiler abandons compilation after 100 errors are detected / reported. The default error limit for c29clang is 20. You can disable the error limit by specifying a <number> of 0 as the option argument.

cl2000 Option	c29clang Option
--super_quiet (-q)	No supported exactly

The cl2000 –super_quiet option disables non-diagnostic output, but allows remarks, errors, and warnings.

You may decide to migrate this option to the clang c29clang -w option, which disables non-diagnostic output but also suppresses all warnings. Note that the two options are not equivalent for this reason.

cl2000 Option	c29clang Option
--verbose	-v

Both cl2000 and c29clang support an option to display verbose progress and status information during the compilation of one or more source files. The c29clang 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.

cl2000 Option (and alias)	c29clang Option
--verbose_diagnostics (-pdv)	-fdiagnostics-...

If the --verbose_diagnostics option is specified on the cl2000 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 c29clang compiler can be made to annotate diagnostics with extra information that is gathered by the compiler during a given compilation. For example, c29clang'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.

cl2000 Option (and alias)	c29clang Option
--write_diagnostics_file (-pdf)	not supported

You can redirect the diagnostics reported by the cl2000 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 c29clang 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 cl2000 compiler. The c29clang compiler does not currently support profile guided optimizations, although the c29clang compiler does provide some support for generating code coverage information. Support for profile guided optimizations may be considered in future version of the c29clang compiler tools.

cl2000 Option	c29clang Option
--analyze=codecov	-fprofile-instr-generate
	-fcoverage-mapping

The cl2000 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 c29clang 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 c29clang* section in the *c29clang Compiler User Manual* for more information on what code coverage capabilities are available in the c29clang compiler tools.

cl2000 Option	c29clang Option
--analyze_only	-fprofile-instr-generate
	-fcoverage-mapping

The cl2000 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 com-

piler 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 c29clang* section in the *c29clang Compiler User Manual* for more information on what code coverage capabilities are available in the c29clang compiler tools.

cl2000 Option	c29clang Option
--gen_profile_info	-fprofile-instr-generate
	-fcoverage-mapping

The cl2000 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 c29clang* section in the *c29clang Compiler User Manual* for more information on what code coverage capabilities are available in the c29clang compiler tools.

cl2000 Option	c29clang Option
--use_profile_info=<file1>[,<file2>,...]	not supported

The cl2000 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 c29clang compiler tools include executables such as c29profdata and c29cov 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 c29clang* section in the *c29clang Compiler User Manual* for more information on what code coverage capabilities are available in the c29clang compiler tools.

2.3.16 Assembler Options

Applications developed with the cl2000 compiler tools may include some source code written in assembly language.

Use of assembly language is discouraged for c29clang, except for assembly code embedded in C/C++ source files via *asm()* statements, which are processed inline by the c29clang integrated GNU-syntax assembler. For this reason, the TI C29x assembly language syntax is not documented.

When migrating a TI C28x application that contains assembly language source files to a TI C29x application, it is recommended that you convert assembly language code to C/C++ code.

cl2000 Option (and alias)	c29clang Option
--keep_asm (-k)	-S

The cl2000 compiler's --keep_asm option causes the assembly language output from the compiler or the assembly optimizer to be kept.

The c29clang compiler allows you to keep the assembly language output from the compiler, but only if you use the -S option to stop the compiler after the assembly files are emitted, preventing the compiler from generating object files or invoking the linker.

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.

cl2000 Option (and alias)	c29clang Option
--cmd_file=<file> (-@=<file>)	@<file>

Both the cl2000 and c29clang 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 c29clang version of the option should not be preceded with a hyphen.

2.3.18 ULP Advisor Options

The c29clang compiler does not support the ULP Advisor options.

cl2000 Option	c29clang Option
--advice:performance[=all|none]	not supported

2.4 Migrating C and C++ Source Code

The process of converting the C/C++ source code for an existing TI C28x application that is built with the cl2000 compiler so that it can be built using the c29clang compiler can be thought of as the process of making your C/C++ source code portable. The cl2000 compiler supports the use of proprietary TI-specific versions of many predefined macro symbols, intrinsics, and pragmas that are not supported by other compilers. The use of such proprietary TI mechanisms will render a program unbuildable by other compilers, including the c29clang compiler.

2.4.1 Migrating Source Code with Clang-Tidy

To aid the migration process from the cl2000 compiler to the c29clang compiler, you can use the c29clang-tidy utility either from the command line or within Code Composer Studio (CCS) Theia. This tool performs various checks of your source code and identifies code that uses cl2000-specific syntax that should be modified.

The general procedure for using the c29clang-tidy utility is as follows:

1. Make initial code changes to reduce the number of compiler errors and false positives that will occur when running c29clang-tidy. These code changes should include:
   - Remove any definitions of the fixed width types `int8_t` and `uint8_t`.
   - Remove any `extern` declarations of C28x builtin functions.
   - Change any use of C28-only macros such as `__TMS320C28XX__` to C29-macros such as `__C29__`.
2. Run the c29clang-tidy utility using either the command line (see *Running the c29clang-tidy Utility from the Command Line*) or CCS Theia (see *Running the c29clang-tidy Utility from CCS Theia*).
3. Make changes to your code based on the diagnostics provided.
4. Run the c29clang-tidy utility iteratively as needed.

Running the c29clang-tidy Utility from the Command Line

The c29clang-tidy command line is similar to the c29clang command line.

For example, suppose your c29clang command line is as follows:

```
c29clang file.c -I./include -mcpu=c29.c0
```

The corresponding c29clang-tidy command line would be as follows:

```
c29clang-tidy --checks=-*,c29migration* -header-filter=.* file.c␣
↪-- -I./include -mcpu=c29.c0
```

- Everything on the command line after `--` is treated as a normal compiler argument. Those options can be moved and duplicated as-is.
- The `--checks` determines which checks are disabled and/or enabled. In this example, the initial `-*` turns off all checks. This is followed by `c29migration*`, which enables all checks that begin with "c29migration", which includes the C28x-C29x migration checks.
- The `-header-filter` option tells clang-tidy which header files to check. By default, it checks none of them. In this example, `.*` says to check all header files using regex.

You can use the `-list-checks` option to list all the enabled checks. For example:

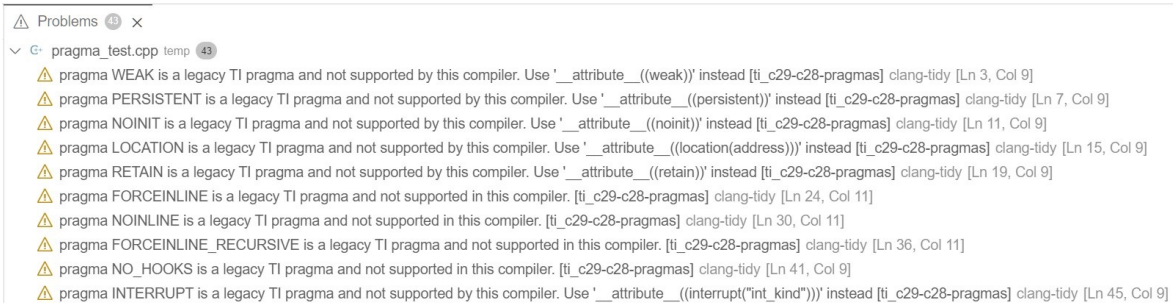
```
c29clang-tidy -list-checks                          \\ list default␣
↪checks
c29clang-tidy -list-checks --checks=-*,c29*  \\ list all C29x␣
↪checks
c29clang-tidy -list-checks --checks=*        \\ list all␣
↪available checks
```

See the Clang-Tidy documentation for additional command line options that may be supported by c29clang-tidy. The `-fix` flag is not supported. You may use a .clang-tidy file as described in that documentation to configure the actions performed by c29clang-tidy.

Running the c29clang-tidy Utility from CCS Theia

To run the c29clang-tidy utility within CCS Theia, follow these steps:

1. Make initial code changes to reduce the number of compiler errors and false positives that will occur when running c29clang-tidy. These code changes should include:
   - Remove any definitions of the fixed width types `int8_t` and `uint8_t`.
   - Remove any `extern` declarations of C28x builtin functions.
   - Change any use of C28-only macros such as `__TMS320C28XX__` to C29-macros such as `__C29__`.
2. The c29clang-tidy utility runs in the background for actions such as project creation, project import, and file modification. By default, the c29migration-c28-builtins, c29migration-c28-pragmas, and c29migration-c28-stdlib checks are run.
3. A list of diagnostics found by the c29clang-tidy utility is shown in the Problems view. For example:



4. Make changes to your code based on the diagnostics provided.

Limitations

- The c29clang-tidy utility does not analyze assembly code (.asm or inline asm() directives)
- As with all static analyzers, false positives are possible and expected. You will need to inspect the code that results in a diagnostic message to determine whether it is a real issue.

Checks Performed

By default, the following checks are performed:

- *c29migration-c28-builtins*
- *c29migration-c28-pragmas*
- *c29migration-c28-stdlib*

The other c29migration checkers require knowledge of the intent of the code. They may suggest updates that change the behavior of the program if not applied carefully. Therefore, the following checkers are treated as advanced migration aids and are not run by default:

- *c29migration-c28-char-range*
- *c29migration-c28-int-decls*
- *c29migration-c28-suspicious-dereference*
- *c29migration-c28-types*
- *c29-fp-promotion*
- *c29-nosave-fp*

c29migration-c28-builtins

This check looks for use of C28x intrinsics such as __fmax and __byte. It suggests alternatives if they are available. For example:

```
void test___add(int * m, int b) {
    __add(m, b);
}
```

The diagnostic warning states that the call to `__add` is a C28x intrinsic, which is not supported by this compiler. It suggests that you refer to the C29x intrinsic documentation to find an equivalent. In this case, it suggests using `__builtin_c29_i32_add32_rm_d(b, m)`.

In order to use this check, you must pre-include the `c28_builtins.h` file on the command line.

Builtin functions are declared internally. Since they are only declared for C28x, these builtin functions are treated as undefined identifiers by the C29x compiler. This is not an issue using the C89

standard, but causes errors with C99 and later and with all C++ standards. To resolve this issue, the compiler tools supply a header file that declares all C28x builtins. This file, `c28-builtins.h`, can be pre-included by clang-tidy. Using the command line above, add the `--include` option as follows:

```
c29clang-tidy --checks=-*,c29migration* -header-filter=.* file.c␣
↪-- -I./include -mcpu=c29.c0 --include c28-builtins.h
```

This check is performed by default by CCS Theia when migrating projects from C28 to C29.

c29migration-c28-char-range

This check detects operations on `char` typed expressions that would be out of range for an 8-bit type. Because the `char` and `unsigned char` types are 16 bits on C28x and 8 bits on C29x, overflows may occur. For example:

```
int ret_pos_cast(char x) {
    return (char)(x + 10000);
}
```

The diagnostic warning states that the cast to 'char' receives an out-of-range value from code that may assume a 16-bit byte type. It notes that the result of the expression is calculated as 10000.

To correct this issue, either: * Choose the C28x behavior by using a fixed-width 16-bit type instead of `char` (int16_t) * Accept the new C29x behavior, adjust the calculation to fit within an 8-bit type, and optionally use a fixed-width 8-bit type (int8_t).

This check is not performed by default by CCS Theia when migrating projects from C28 to C29 because it requires understanding the intent of the program that is not evident in the code itself.

c29migration-c28-int-decls

This check suggests replacing the `int` and `unsigned int` types with `int16_t` and `uint16_t` types. For example:

```
void foo(int x) { int y; }
```

The diagnostic warning states that you should consider using a fixed-width 16-bit type to avoid issues with the increased width of int on C29x devices.

A fix-it is available that replaces int/unsigned int types with int16_t/uint16_t types and includes <stdint.h> if it is not already included.

This check is not performed by default by CCS Theia when migrating projects from C28 to C29 because it suggests a major change to the application.

c29migration-c28-pragmas

This check looks for use of C28x pragmas such as #pragma DATA_SECTION. It suggests alternatives if they are available. For example:

```
#pragma WEAK(weakx)
int weakx;
```

The diagnostic warning states that the WEAK pragma is a legacy TI pragma that is not supported by this compiler. It suggests that you use `__attribute__((weak))` instead.

This check is performed by default by CCS Theia when migrating projects from C28 to C29.

c29migration-c28-stdlib

This check looks for calls to library functions that take a number of bytes argument where the argument is not scaled by a `sizeof` expression. For example:

```
void malloc_c_pos() {
    malloc((1+23));
}
```

The diagnostic warning states that the call to malloc has a byte-size argument without a `sizeof` expression. It suggests that you scale the size of the argument by a factor of 'sizeof' to avoid issues with changing byte sizes between C28x and C29x.

Calls to the following library functions are checked:

- aligned_alloc
- calloc
- malloc
- realloc
- memccpy
- memchr
- memcmp
- memcpy
- memmove
- memset
- strncmp
- strncpy

- fread
- fwrite

This check is performed by default by CCS Theia when migrating projects from C28 to C29. Be aware that sufficiently complex `sizeof` expressions with multiple terms may not be correctly diagnosed.

c29migration-c28-suspicious-dereference

This check detects dereferences (*) of pointers, where the source expression is integer-typed and contains an additive operation, which is commonly used to access memory-mapped values. For example:

```
int main() {
    *(int*)(x + 10) = 12;
}
```

The diagnostic warning states that the pointer cast has a suspicious integer expression incremented by bytes, which are 16 bits on C28x and 8 bits on C29x. It suggests that you inspect the intent of the address and scale it accordingly if needed.

This check is not performed by default by CCS Theia when migrating projects from C28 to C29. Complex expressions with multiple casts to and from pointer types may cause this checker to fail to diagnose an issue.

c29migration-c28-types

This check detects pointer arithmetic on char/int-based pointers, whose bit stride changes between C28x to C29x. For example:

```
extern void foo(char *);
void test_plus(char *x) {
    foo(x + 2);
}
```

The diagnostic warning states that pointer arithmetic is performed on an expression with type 'char'. Since 'char' is 16 bits on C28x and 8 bits on C29x, you should choose either C28x behavior by using a fixed-width 16-bit type (here, int16_t), or accept C29x behavior and use a fixed-width 8-bit type (here, int8_t).

This check is not performed by default by CCS Theia when migrating projects from C28 to C29. All matches should be individually vetted for intent and behavior:

- Code that accesses characters of a string should remain char/unsigned char.

- Code that accesses any other type through a char* pointer should be updated in one of two ways:
  1. The char* pointer should be changed to the type of the object being accessed.
  2. The offset should be scaled by the size of the object being accessed.

c29-fp-promotion

This check examines all floating point type usages. It also examines implicit promotion, in which a float is converted automatically to double or long double. Warnings about implicit promotion help you debug performance issues or accidental precision loss.

c29-nosave-fp

This check reports any floating point usage within an interrupt marked with the *interrupt_nosave_fp* function attribute.

2.4.2 C/C++ Source Migration Aid Diagnostics

When migrating a C29x C/C++ application project from using the TI C28x compiler to using the c29clang compiler you may have instances of TI-specific pragmas, pre-defined macro symbols, or intrinsics that are supported by the cl2000 compiler, but not the c29clang 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 supported counterparts.

To help with this process, the c29clang compiler emits a diagnostic when it encounters the use of a proprietary 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 proprietary TI pre-defined macro symbol, pragma, or intrinsic, the c29clang compiler emits a diagnostic to inform you about the presence of that proprietary TI mechanism.

Let's consider a couple of examples . . .

Proprietary TI Pragmas

The proprietary TI pragma **FUNC_CANNOT_INLINE** has a valid alternative, so if the c29clang compiler encounters the following line of code:

```
#pragma FUNC_CANNOT_INLINE
```

The c29clang 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 proprietary TI pragmas can be converted into attribute form, please see the *Pragmas and Attributes* section.

Proprietary TI Pre-Defined Macro Symbols

The proprietary TI pre-defined macro symbol __TMS320C28XX__ is an example of a pre-defined macro symbol that can be used to configure C29x-specific code in an application, but this pre-defined macro symbol is not supported by c29clang and needs to be replaced by a functionally equivalent expression. Specifically, when the following line of code is encountered by the c29clang compiler:

```
#if defined(__TMS320C28XX__)
...
#endif
```

the c29clang compiler will emit the following diagnostic:

```
warning: __TMS320C28XX__ is a legacy TI macro that is not defined
in clang compilers and will evaluate to 0, use '(__C29_ARCH == 0)
instead [-Wti-macros]
```

The warning can then be averted by replacing the __TMS320C28XX__ symbol reference with the following:

```
#if (__C29_ARCH == 1)
```

However, there are other proprietary TI pre-defined macro symbols, like __TMS320C28XX_VCRC__ , that do not have a viable alternative, so the following code:

```
#if defined(__TMS320C28XX_VCRC__ )
```

yields the following diagnostic when encountered by the c29clang compiler:

```
warning: '__TMS320C28XX_VCRC__ ' is a legacy TI macro and not␣
 ↪supported in clang
compilers
```

For more information about how many of the proprietary TI pre-defined macro symbols can be converted into their functionally equivalent form, please refer to the *Pre-Defined Macro Symbols* section.

Proprietary TI Intrinsics

The proprietary TI intrinsic "__lmin" is an example of an cl2000 compiler intrinsic that has a viable alternative form, so when the c29clang compiler encounters the following function:

```
long __lmin(long dst, long src)
```

The c29clang compiler does not provide diagnostics. However, the diagnostic checks provided by the c29clang-tidy utility suggest converting this intrinsic to the following:

```
int __builtin_c29_i32_min32_d(int, int)

int c29_min(int a, int b) {
  return __builtin_c29_i32_min32_d(a, b);
}
```

Not all cl2000 intrinsics are as easy as "__lmin" to migrate to a functionally equivalent c29clang form. For more details on how specific cl2000 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 proprietary TI macro symbols, pragmas, and intrinsics are enabled by default in the c29clang compiler. The following c29clang compiler options can be specified to selectively turn off the migration aid diagnostic categories:

- **-Wno-ti-pragmas** : to suppress migration aid diagnostics for proprietary TI pragmas
- **-Wno-ti-macros** : to suppress migration aid diagnostics for proprietary TI pre-defined macro symbols
- **-Wno-ti-intrinsics** : to suppress migration aid diagnostics for proprietary 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.

While several pre-defined macro symbols supported by the cl2000 compiler are also supported by the c29clang compiler, many are not. For a given cl2000 pre-defined macro symbol that is not supported by the c29clang compiler, there are ways one can successfully transition the use of such a pre-defined symbol to be compatible with the c29clang compiler.

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 cl2000 compiler and, if conversion is needed, explains how to modify the C/C++ source to make it compatible with the c29clang compiler.

For details about macro symbols that are pre-defined by the c29clang compiler, see *Generic Compiler Pre-Defined Macro Symbols*.

Pre-Defined Macro Symbols that are Available in Both cl2000 and c29clang

Some pre-defined macro symbols supported by the cl2000 compiler are also supported by the c29clang 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 cl2000 to c29clang:

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 cl2000 or c29clang 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 cl2000 -fg option or the c29clang -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 cl2000 and c29clang 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 cl2000 compiler allows C/C++ source code to undefine the _INLINE symbol to disable some optimization while processing C/C++ source. The c29clang compiler does not support turning off inlining by undefining macros. The c29clang 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 cl2000 and c29clang compilers define the __STDC__ macro symbol to indicate compliance with the ISO C standard. Please refer to the TI C28x Optimizing C/C++ Compiler User's Guide for exceptions to ISO C compliance that apply to the cl2000 compiler. Exceptions to ISO C compliance in the c29clang compiler can be found in the TI C29x 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 not defined. The compiler does not support C11 threads and does not provide

Converting cl2000 Pre-Defined Macro Symbols to c29clang Compatible Form

Several cl2000 pre-defined macro symbols are not supported by c29clang, but these macro symbols can often be re-written in terms of GCC pre-defined macro symbols that c29clang does support. The following table lists cl2000 pre-defined macro symbols that are not supported by the c29clang compiler, and how they may be converted to a form that is supported by c29clang:

cl2000 Macro Symbol	c29clang Equivalent
__little_endian__	defined(_LITTLE_ENDIAN_)

The cl2000 compiler defines the __little_endian__ macro symbol to indicate that the compiler uses little-endian mode. References to __little_endian__ can safely be replaced with a test that evaluates to True, since the C29x architecture does not support big-endian mode.

The c29clang compiler defines the _LITTLE_ENDIAN_ macro symbol in all cases, because only little-endian mode is supported.

cl2000 Macro Symbol	c29clang Equivalent
__PTRDIFF_T_TYPE__	__PTRDIFF_TYPE__

The cl2000 compiler defines the __PTRDIFF_T_TYPE__ macro symbol to indicate the equivalent base type associated with the ptrdiff_t type.

The c29clang compiler defines __PTRDIFF_TYPE__ to reflect the underlying type for the ptrdiff_t typedef.

cl2000 Macro Symbol	c29clang Equivalent
__SIZE_T_TYPE__	__SIZE_TYPE__

The cl2000 compiler defines the __SIZE_T_TYPE__ macro symbol to indicate the equivalent base type associated with the size_t type.

The c29clang compiler defines __SIZE_TYPE__ to reflect the underlying type for the size_t typedef.

cl2000 Macro Symbol	c29clang Equivalent
__TI_COMPILER_VERSION__	__ti_version__

The cl2000 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 c29clang 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.

cl2000 Macro Symbol	c29clang Equivalent
__TI_EABI__	defined(__ELF__)

Both the cl2000 and c29clang compilers support the generation of ELF object format code only. Consequently, the cl2000 __TI_EABI__ macro symbol is always defined when the cl2000 compiler is invoked.

c29clang does not support the __TI_EABI__ macro symbol, but it does support the __ELF__ macro symbol which is also supported by cl2000. Therefore, references to __TI_EABI__ in the C/C++ source can be safely replaced by __ELF__.

cl2000 Macro Symbol	c29clang Equivalent
__TI_GNU_ATTRIBUTE_SUPPORT__	defined(__clang__)

The cl2000 compiler defines the __TI_GNU_ATTRIBUTE_SUPPORT__ macro symbol to indicate that a C/C++ dialect mode where generic attributes is supported. c29clang does not support an analogous macro symbol, but generic attributes are supported by c29clang, nonetheless.

cl2000 Macro Symbol	c29clang Equivalent
__TI_STRICT_ANSI_MODE__	__STRICT_ANSI__

The cl2000 compiler defines the __TI_STRICT_ANSI_MODE__ macro symbol to 1 if the cl2000 compiler is invoked with the --strict_ansi option. The cl2000 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 c29clang compiler defines the __STRICT_ANSI__ macro symbol if any of the --std=<spec> options are specified on the c29clang command-line (where spec indicates the identity of a C or C++ language standard).

cl2000 Macro Symbol	c29clang Equivalent
__TMS320C2000__	__C29__

The cl2000 compiler defines the __TMS320C2000__ macro symbol to true to indicated the application is compiled for the TI C28x architecture.

The c29clang compiler defines __C29__ to 1 if the application is compiled for a C29x target. See *TI C29x-Specific Pre-Defined Macro Symbols* for related macro symbols.

cl2000 Macro Symbol	c29clang Equivalent
__WCHAR_T_TYPE__	__WCHAR_TYPE__

The cl2000 compiler defines the __WCHAR_T_TYPE__ macro symbol to indicate the equivalent base type associated with the wchar_t type.

The c29clang compiler supports the analogous GCC __WCHAR_TYPE__ macro symbol to indicate the underlying type for the wchar_t typedef.

Pre-Defined Macro Symbols in cl2000 that are Not Applicable in c29clang

There are several pre-defined macro symbols that are supported by the cl2000 compiler that are either not applicable for c29clang or are simply not supported.

For example, the __TMS320C28XX__ pre-defined macro symbol indicates support for TI C28x, which does not apply for C29x applications.

cl2000 Macro Symbol	Description / Comments
__TMS320C28XX__	The cl2000 __TMS320C28XX__ macro symbol is not applicable since c29clang does not support the TI C28x architecture.
__TMS320C28XX_CLA0__ __TMS320C28XX_CLA1__ __TMS320C28XX_CLA2__	The cl2000 __TMS320C28XX_CLAn__ macro symbols are not applicable since c29clang does not support the Control Law Accelerator (CLA) or the CLA compiler.
__TMS320C28XX_FPU32__ __TMS320C28XX_FPU64__	The cl2000 __TMS320C28XX_FPUnn__ macro symbols are not applicable since c29clang does not support the Floating Point Unit (FPU).
__TMS320C28XX_IDIV__	The cl2000 __TMS320C28XX_IDIV__ macro symbol are not applicable since c29clang does not support the IDIV intrinsics.
__TMS320C28XX_TMU0__ __TMS320C28XX_TMU1__	The cl2000 __TMS320C28XX_TMUn__ macro symbols are not applicable since c29clang does not support the Trigonometric Math Unit (TMU).
__TMS320C28XX_VCU0__ __TMS320C28XX_VCU2__	The cl2000 __TMS320C28XX_VCUn__ macro symbols are not applicable since c29clang does not support the Viterbi, Complex Math and CRC Unit (VCU).
__TI_STRICT_FP_MODE__	The cl2000 __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.

Additional Pre-Defined Macro Symbols Supported in c29clang

The following pre-defined macro symbols are provided by the c29clang but not by the cl2000 compiler.

The c29clang 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.

c29clang Macro Symbol	Description / Comments
__clang__	An invocation of c29clang will always define the __clang__ macro symbol.
__EXCEPTIONS	The c29clang compiler would define the __EXCEPTIONS macro symbol if -fexceptions were specified when compiling a C++ source file. However, exceptions are not currently supported by the c29clang 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 c29clang compiler's C pre-processor is compatible with version 3 of the GNU C pre-processor.
__GNUC_GNU_INLINE__	The c29clang 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 c29clang 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 c29clang 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 c29clang compiler defines __INTMAX_TYPE__ to reflect the underlying type for the intmax_t typedef.
__NO_INLINE__	If an optimization level is specified on the c29clang command-line, then c29clang defines the __NO_INLINE__ macro symbol to indicate that compilation mode.
__OPTIMIZE__	If an optimization level is specified via the -O option on the c29clang command-line, then c29clang defines the __OPTIMIZE__ macro symbol to indicate that optimization is enabled for the current compilation.

2.4.4 Intrinsics and Built-in Functions

The compiler intrinsics supported by the cl2000 compiler are fully detailed in the TMS320C28x Optimizing C/C++ Compiler User's Guide in Section 7.6.

For a list of builtin intrinsics supported by the cl2000 compiler, see the *C2000 C29x CPU and Instruction Set User's Guide* (SPRUIY2), which is available through your TI Field Application Engineer.

2.4.5 Pragmas and Attributes

Pragmas

While the c29clang compiler does support some of the same pragma directives that the cl2000 compiler supports, there are several pragma directives supported by cl2000 that are not supported by c29clang. 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 cl2000 compiler, providing guidance on how to transition each pragma directive for a project to be built with the c29clang compiler.

In general, if there is a c29clang functionally equivalent attribute or pragma form for an cl2000 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 cl2000 and c29clang.

cl2000 Pragmas to be Converted to Attribute or Pragma Form

Listed below are several commonly occurring cl2000 pragmas that, when converted to attribute form, are supported by the c29clang compiler.

- **CODE_ALIGN** pragma -> **aligned** attribute

  cl2000 pragma:

  ```
  #pragma CODE_ALIGN(func_name, n)
  ```

  c29clang functionally equivalent attribute:

  ```
  __attribute__((aligned(n)))
  ```

  The CODE_ALIGN pragma aligns the function along the specified alignment boundary. The alignment constant must be a power of 2. The CODE_ALIGN pragma is useful if you have functions that you want to start at a certain boundary.

- **CODE_SECTION** pragma -> **section** attribute

cl2000 pragma:

```
#pragma CODE_SECTION(func_name, "scn_name")
```

c29clang 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

  cl2000 pragma:

  ```
  #pragma DATA_ALIGN("sym_name", alignment)
  ```

  c29clang 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

  cl2000 pragma:

  ```
  #pragma DATA_SECTION(sym_name, "scn_name")
  ```

  c29clang 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

  cl2000 pragma:

  ```
  #pragma FORCEINLINE
  ```

  c29clang 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

  cl2000 pragma:

  ```
  #pragma FUNC_ALWAYS_INLINE(func_name)
  ```

  c29clang 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

  cl2000 pragma:

  ```
  #pragma FUNC_CANNOT_INLINE(func_name)
  ```

  c29clang 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

  cl2000 pragma:

```
#pragma LOCATION(address)
```

c29clang 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

  cl2000 pragma:

  ```
  #pragma NOINIT(sym_name)
  ```

  c29clang 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

  cl2000 pragma:

  ```
  #pragma NOINLINE
  ```

  c29clang 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

cl2000 pragma:

```
#pragma PERSISTENT(sym_name)
```

c29clang 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

  cl2000 pragma:

  ```
  #pragma RETAIN(sym_name)
  ```

  c29clang 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

  cl2000 pragma:

  ```
  #pragma SET_CODE_SECTION("scn_name")
  ```

  c29clang 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

  cl2000 pragma:

  ```
  #pragma SET_DATA_SECTION("scn_name")
  ```

  c29clang 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

  cl2000 pragma:

  ```
  #pragma UNROLL(n)
  ```

  c29clang functionally equivalent pragmas:

  ```
  #pragma clang loop unroll_count(n)
  ```

  The c29clang 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

  cl2000 pragma:

  ```
  #pragma #pragma WEAK(sym_name)
  ```

  c29clang 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.

cl2000 Pragmas That Are Not Available in c29clang

The following cl2000 pragma directives are not supported by the c29clang compiler and don't have a functionally equivalent attribute form:

- **CLINK**

```
#pragma CLINK(sym_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)
```

For more information about these pragmas and how they function, please refer to the TMS320C28x Optimizing C/C++ Compiler User's Guide (Section 6.9).

Attributes

Both the cl2000 and c29clang compilers support the notion of attributes that can be applied to functions, variables, or types. The attributes supported in cl2000 and c29clang 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 cl2000 and c29clang compilers. This section also provides details about which attributes are supported in the cl2000 compiler, but not the c29clang 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 c29clang compiler. Finally, this section provides information on additional attributes that are supported in the c29clang compiler, but not the cl2000 compiler.

Function Attributes

The function attributes listed below are supported in both the cl2000 and c29clang 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 cl2000, the available "int_kind"s are: DABT, FIQ, IRQ, PABT, RESET, or UNDEF. In c29clang, "int_kind" can be: IRQ, FIQ, SWI, ABORT, or UNDEF.

- **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 c29clang'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 cl2000 function attributes are not supported in the c29clang compiler:

- **aligned**
- **calls**
- **deprecated**
- **ramfunc**

Variable Attributes

The variable attributes listed below are supported in both the cl2000 and c29clang 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 c29clang compiler the used attribute acts as a synonym for the c29clang'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 cl2000 variable attributes are not supported in the c29clang compiler:

- **aligned**
- **blocked**
- **mode**
- **noblocked**
- **preserve**
- **update**

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 cl2000 type attributes are not supported in the c29clang compiler:

- **byte_peripheral**

For Loop Attributes

- **TI::unroll** for loop attribute -> **clang loop unroll** pragma

cl2000 attribute:

```
[[TI::unroll(4)]]
for (...)
{
    ...
}
```

c29clang functionally equivalent pragma:

```
#pragma unroll 4
 for (...)
 {
     ...
 }
```

The following cl2000 for loop attributes are not supported in the c29clang compiler:

- **TI::must_iterate**

2.4.6 Migrating CLA Code

The TI C28x (cl2000) compiler also compiles for the Control Law Accelerator (CLA). This accelerator is not needed when using TI C29x devices because of their improved performance over TI C28x devices.

In TI C28x applications, files with a .cla file extension and files compiled using the --cla_default command-line option use the CLA compiler. The CLA compiler can be used only with C source files, not with C++ source files.

When migrating C code that was compiled with CLA compiler, examine your code to see if changes are needed. In general, the CLA compiler is more restrictive than the general cl2000 compiler. For this reason, few changes are needed to migrate code that was compiled with the CLA compiler.

Look for the following CLA-specific code and make changes as needed:

- **CLA intrinsics:** See Table 10-2 in the TMS320C28x Optimizing C/C++ Compiler User's Guide (Section 10.2.2) for a list of intrinsics that were recognized only by the CLA compiler. Find an equivalent c29clang intrisic or write C/C++ code to accomplish the desired behavior.
- **Int sizes:** The int size for CLA is 32 bits, which is the same size used for c29clang, but different from the TI C28x int size of 16 bits.
- **Pointer sizes:** Pointer sizes for CLA are 16 bits, but for c29clang are 32 bits.
- **Section names:** Sections called .bss_cla and .const_cla are specific to the CLA compiler and should be changed to .bss and .const. The .scratchpad section was used in place of a system stack.

You may also want to simplify code that was written in a more compilcated way to work around CLA limitations. Some of the CLA compiler limitations, which are not present for the c29clang compiler, were as follows:

- Global and static data were not supported.
- CLA code could not call C28x functions.
- Recursive function calls were not supported.
- Function pointers were not supported.
- C standard libraries were not supported.
- There was no C system heap, because there was no support for the malloc() function.
- Data shared between C28x and CLA compilers had to be defined in the C28x code.
- CLA0 and CLA1 did not support background tasks.

2.5 Migrating Linker Command Files for Use With c29clang

To a large extent, linker command files for C29x applications that manage the placement of code and data generated by the cl2000 compiler will also work with object files that are generated by the c29clang compiler. However, a few adjustments may be needed to make your linker command file c29clang-friendly.

2.5.1 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 c29clang compiler creates temporary names for compiler-generated object files. For example, given a C source file named xyz.c, the c29clang 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 c29clang compiler generates object files with a '.o' file extension, whereas the default file extension for an cl2000-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.

2.5.2 Compiler-Generated Section Names

Both the c29clang and cl2000 compilers generate code and data into object file sections. However, there are some differences to be aware of:

Compiler-Generated Section Names:

Section Description	cl2000 Generated Section	c29clang 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 c29clang 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 a C29x 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	cl2000 Generated Section	c29clang Generated Section
arguments (argc/argv)	.args	.args
stack space	.stack	.stack
heap space	.sysmem	.sysmem

2.5.3 Linker Options

The --rom_model (-c) linker option, which is the default for cl2000, is not set by default by the c29clang 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 c29clang command line or must be specified in the linker command file.

2.6 Migrating Assembly Language Source Code

Applications developed with the cl2000 compiler tools may include some source code written in assembly language.

Use of assembly language is discouraged for c29clang, except for assembly code embedded in C/C++ source files via *asm()* statements, which are processed inline by the c29clang integrated GNU-syntax assembler. For this reason, the TI C29x assembly language syntax is not documented.

When migrating a TI C28x application that contains assembly language source files to a TI C29x application, it is recommended that you convert assembly language to C/C++ code.

2.6.1 Converting Proprietary TI C29x asm() Statements Embedded in C/C++ Source

Embedded in the C/C++ source code for your TI C29x 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 cl2000 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 cl2000's implementation of asm() statements does not support the notion of C expression operands, for example.

The c29clang 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 c29clang 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 c29clang compiler can cause a "symbol multiply defined" error.

Please see *Inlining Functions that Contain asm() Statements* for more details.

CHAPTER

THREE

C29CLANG COMPILER USER MANUAL

The TI C29x Clang Compiler Tools, commonly referred to in this user guide as c29clang, support the development of software applications intended to run on a C29x processor.

This section of the documentation provides a detailed description of each of the parts of the c29clang compiler toolchain. It provides guidance on how these tools can be used to develop C29x applications.

Contents:

3.1 Using the C/C++ Compiler

This section of the *c29clang Compiler User Manual* describes the c29clang compiler, the compiler options that can be specified on the **c29clang** command-line, and how the compiler works with the linker to produce static executables that can be loaded and run on a C29x processor.

Contents:

3.1.1 About the Compiler

The TI C29x Clang Compiler (c29clang) lets you compile, optimize, and link an application in one step. The compiler performs the following steps on one or more source modules:

- The c29clang compiler compiles 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)

Note: The **-x** option can be used on the **c29clang** 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*.

- Internally, the c29clang compiler generates assembly code and assembles the assembly files to create object modules. But, you do not need to be concerned with assembly code when using the C29x code generation tools.
- By default, the c29clang 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 **c29lnk** command line.)

Note: The link step can be disabled using the **-c** option on the **c29clang** command line. See *Stop Compiler After Object File Output (Omit Linking)*

3.1.2 Invoking the Compiler

Usage

To invoke the c29clang compiler, enter:

c29clang [*options*] [*filenames*]

- **c29clang** - 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:
  - c29clang options - affect the behavior of the C/C++ compiler. These are described in more detail in the *Compiler Options* section.
  - 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 **c29clang** command line, but you can disable the link step by specifying the *-c* option on the **c29clang** command line. You can invoke the linker by itself using the **c29lnk** 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 **c29clang** 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 **c29clang** 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.

Note: For more information about the **c29clang** *-x* option and controlling how **c29clang** interprets input files, see the *Using -x Option to Control Input File Interpretation* section.

Example

The following simple example shows how the **c29clang** command can be used to build an ELF format static executable file that can be loaded and run on a C29x processor.

Source Files There are two input files to specify on the **c29clang** command line.

The C file *print_global.c* references a global variable that is defined in a second C file *def_global.c* and prints out the value of that global variable.

Contents of *print_global.c*:

```
#include <stdio.h>

extern int a_global;

int main() {
  printf("a_global: %d\n", a_global);
  return 0;
}
```

Contents of *def_global.c*:

```
#ifdef __ti_version__
#define a_global 12345
#else
#error "a_global is not defined"
#endif /* __ti_version__ */
```

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*:

```
/
 ↪**************************************************************************
 ↪
```

(continues on next page)

(continued from previous page)

```
/* lnk.cmd - V1.00  Command file for linking C29 programs                 */
/*****************************************************************************/
/* This linker command file assumes C/C++ model                              */
/*****************************************************************************/
-c
-stack  0x8000                              /* Software stack size           */
-heap   0x2000                              /* Heap area size                */

/* Specify the system memory map */
MEMORY
{
  ROM     : org = 0x00000020   len = 0x2FFFE0    /* 1.25 GB */
  FLASH   : org = 0x10000000   len = 0x300000    /* 1.25 GB */
  RAM     : org = 0x18000000   len = 0x300000    /* 1.25 GB */
}
#define RO_CODE FLASH
#define RO_DATA FLASH
#define RW_DATA RAM

/* Specify the sections allocation into memory */

SECTIONS
{
    .text     : {} > RO_CODE            /* Code                            */
    .cinit    : {} > RO_DATA            /* Initialization tables           */
    .const    : {} > RO_DATA            /* Constant data                   */
    .pinit    : {} > RO_DATA            /* C++ Constructor tables          */

    .data     : {} > RW_DATA            /* Initialized variables           */
```

(continues on next page)

(continued from previous page)

```
    .bss      : {} > RW_DATA              /* Uninitialized␣
↪variables                 */
    .stack    : {} > RW_DATA              /* Software system stack ␣
↪                 */
    .sysmem   : {} > RW_DATA              /* Dynamic memory␣
↪allocation area     */
}
```

Compile and Link Steps Explained The following **c29clang** command compiles and links the input files to create a static executable file called *a.out*:

```
%> c29clang -mcpu=c29.c0 print_global.c def_global.c -o a.out -
↪Xlinker lnkme.cmd
```

The above **c29clang** command performs the following actions during the process of building the static executable file *a.out*:

Compile and Link (Default Operation)

The default behavior of the c29clang 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 **c29clang** 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 c29clang runtime libraries to create an executable output file named **test.out**:

```
c29clang -mcpu=c29.c0 file1.c file2.o -o test.out -Wl,link_test.
↪cmd
```

More About Invoking the Linker With c29clang

Note that there is no mention of the c29clang runtime libraries on the **c29clang** command line or inside of the **link_test.cmd** linker command file. When the linker is invoked from the **c29clang** command line, the c29clang 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 **c29clang** 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 c29clang runtime libraries
- **--start-group/--end-group** - specifies exactly which runtime libraries are incorporated into the link

In the above **c29clang** 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 send the preprocessed output to the output location.

c29clang -E [*options*] [*filenames*]

If no other options on the command line specify an output location, the preprocessed output is sent to *stdout*. If, for example, the **-E** option is used in combination with the **-o** option, the preprocessed output is sent to the file specified with the **-o** option. In this case, the file that would normally be a binary object file instead contains text.

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:

```
c29clang -mcpu=c29.c0 -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 **c29clang** to run the C preprocessor, parse the C/C++ input file to check for syntax errors, and perform type checking before halting compilation.

c29clang -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

Note: Internally, the c29clang compiler normally generates assembly code and assembles the assembly code to create object modules. You do not need to be concerned with assembly code when using the C29x code generation tools. The assembly syntax for C29x is undocumented; it may change without notice in future versions.

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.

c29clang -S [*options*] [*filenames*]

The following example generates assembly files, **file1.s** and **file2.s**, each containing compiler-generated GNU-syntax C29x assembly language directives and instructions:

```
c29clang -S -mcpu=c29.c0 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 **c29clang** command line.

c29clang -c [*options*] [*filenames*]

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

```
c29clang -c -mcpu=c29.c0 file1.c file2.c
```

3.1.3 Compiler Options

This section of the *c29clang Compiler User Manual* serves as a reference guide for the available command-line options that affect the behavior of the **c29clang** executable.

Contents:

Commonly Used Options

The commonly used options listed in the subsections below are available on the **c29clang** compiler command line.

- *Processor Options*
- *Include Options*
- *Predefined Symbol Options*
- *Optimization Options*
- *Debug Options*
- *Control Options*
- *Compiler Output Option*

Processor Options

Select a Target C29x Processor

-mcpu=<processor>
: Select the target <*processor*> version.

The c29clang compiler currently supports only the following C29x processor variant:

- **-mcpu=c29.c0** - C29x instructions

If the **-mcpu** option is not specified on the **c29clang** command-line, the compiler assumes a default of **-mcpu=c29.c0**.

Endianness

C29x devices are always little-endian. No command-line options are provide to specify or change this.

Floating-Point Support Options

The C29x CPU can perform native 32-bit and 64-bit floating-point hardware operations on the CPU, without sending data to a separate FPU for processing.

-mfpu=<arg>

The C29x compiler toolchain emits native 32-bit floating point operations by default. Optionally, some C29x hardware supports native 64-bit floating point operations. Users may control the usage of native floating point hardware with the **-mfpu** option.

Valid forms of the **-mfpu** option include:

- **-mfpu=none** - Emulate all floating point operations in software without the use of floating-point registers or hardware support.
- **-mfpu=f32** - Use native 32-bit floating-point hardware operations, but emulate 64-bit floating-point operations in software. This is the default setting for the compiler toolchain.
- **-mfpu=f64** - Use native 32-bit and 64-bit floating-point hardware operations.

Include Options

The c29clang 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 c29clang 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 c29clang compiler package and should be used in combination with linker and the runtime libraries that are included in the c29clang 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 c29clang 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.

```
c29clang -mcpu=c29.c0 -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 c29clang compiler, select a level of optimization from among the following **-O[0|1|2|3|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 are likely to increase code size are disabled.

-O3
All optimizations available at **-O2** plus others that may increase code size.

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

By default, the **-Og** option enables the *-fextend-variable-liveness=all* option, which improves debuggability of user variables.

-Os
All optimizations available at **-O2** plus additional optimizations designed to reduce code size. In comparison to the **-Oz** option, the **-Os** option avoids code size optimizations that are likely to harm performance.

-Oz
All optimizations available at **-O2** plus additional optimizations designed to reduce code size. In comparison to the **-Os** option, the **-Oz** option performs additional code size optimizations that may harm performance. This optimization setting is recommended if small code size is a priority.

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 c29clang 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 c29clang 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 c29clang 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 **c29clang** command. If **c29clang** 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 **c29clang**, 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:

```
c29clang -mcpu=c29.c0 -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 C29x processor variant indicated by <*arg*>, where <*arg*> can be:

- **c29.c0**

Since only one C29x option is currently recognized, this setting is the default. The **-mcpu** option is not required.

Select Floating-Point Code Generated

The C29x CPU can perform native 32-bit and 64-bit floating-point hardware operations on the CPU, without sending data to a separate FPU for processing.

-mfpu=<arg>

The C29x compiler toolchain emits native 32-bit floating point operations by default. Optionally, some C29x hardware supports native 64-bit floating point operations. Users may control the usage of native floating point hardware with the **-mfpu** option.

Valid forms of the **-mfpu** option include:

- **-mfpu=none** - Emulate all floating point operations in software without the use of floating-point registers or hardware support.
- **-mfpu=f32** - Use native 32-bit floating-point hardware operations, but emulate 64-bit floating-point operations in software. This is the default setting for the compiler toolchain.
- **-mfpu=f64** - Use native 32-bit and 64-bit floating-point hardware operations.

Select Endianness

C29x devices are always little-endian. No command-line options are provide to specify or change this.

C/C++ Language Options

The **c29clang** 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 **c29clang** 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

Note: C++ support is based on a library that is focused on support for C++17. If you specify an earlier version of the C++ standard, it is not guaranteed that features that were not required by that standard will be unsupported.

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 C29x 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 **c29clang** 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.

Runtime Model Options

-fcommon, -fno-common
The *-fcommon* option is disabled by default.

For C source code, enabling 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.

-fdata-sections, -fno-data-sections
The *-fdata-sections* option is enabled by default and instructs the c29clang 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 **c29clang** 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.

-ffunction-sections, -fno-function-sections
The *-ffunction-sections* option is enabled by default and instructs the c29clang 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 **c29clang** 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 c29clang 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 c29clang 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 **c29clang** 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*.

Symbol Management Options

Define/Undefine Symbols

You can define or undefine a symbol on the **c29clang** 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 c29clang compiler to stop after the preprocessor phase of compilation:

-E, --preprocess
Halt compilation after running the C preprocessor.

Preprocessor Options

c29clang 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

c29clang 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

c29clang 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 c29clang compiler, select a level of optimization from among the following **-O[0|1|2|3|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.

Optimization Level Options

Note: Optimization Option Recommendations

- The **-O0** option is not recommended.
- The **-O1** option is recommended for maximum debuggability.
- 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.

This optimization level is *not* recommended for C29x devices; when no optimization is performed, operations are inefficient and runtime behavior is significantly degraded. Use at least the -O1 optimization level.

-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 disables some optimizations that may result in larger compiler-generated code size.

-O3

Enables all optimizations available at **-O2** plus others that may increase compiler-generated code size. This option is recommended for optimizing performance, but it is likely to increase code size.

This optimization level enables software pipelining.

-Og

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

By default, the **-Og** option enables the *-fextend-variable-liveness=all* option, which improves debuggability of user variables.

-Os
Enables all optimizations available at **-O2** plus additional optimizations designed to reduce code size. In comparison to the **-Oz** option, the **-Os** option avoids code size optimizations that are likely to harm performance.

-Oz
Enables all optimizations available at **-O2** plus additional optimizations designed to reduce code size. In comparison to the **-Os** option, the **-Oz** option performs additional code size optimizations that may harm performance. This optimization setting is recommended if small code size is a priority.

-Ofast
This option is deprecated. It enables all optimizations available at **-O3** plus additional aggressive optimizations that are not guaranteed to be in strict compliance with language standards. If you want this behavior, it is recommended that you use the **-O3** option in combination with **-ffast-math**.

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.

The deprecated -Ofast optimization option automatically enabled the -ffast-math option. If you want the behavior provided by the -Ofast option, it is recommended that you use the -O3 and -ffast-math options in combination.

-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 c29clang 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*).
- **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.

-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

Function inlining is supported by the C29x compiler; however, function outlining is not currently supported.

-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.

Loop Unrolling

-funroll-loops, -fno-unroll-loops
Enable optimizer to unroll loops. The *-fno-unroll-loops* option disables this optimization.

Variable Liveness

-fextend-variable-liveness=[all|this|none]
This option causes user variables (and the *this* pointer in C++ non-static member functions) to remain "live" during compilation so that they are not optimized out. The result is easier debugging and profiling, but less efficient generated code. This behavior is useful when debugging applications that require some level of optimization in order to run.

The possible settings are as follows:

- **all** This setting causes all user variables and the "this" pointer to remain live. This is the default if the -fextend-variable-liveness option is used with no setting specified. The **-Og** option enables this setting by default.
- **this** Only the "this" pointer in C++ non-static member functions remains live.
- **none** Variables may be optimized out by the compiler. This is the same as not specifying the -fextend-variable-liveness option at all.

Using -x Option to Control Input File Interpretation

The c29clang 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
- **.o** - object file to be forwarded on to the linker

The c29clang 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.

Note:

The **-x<language>** option is position-dependent. A given **-x** option on the **c29clang** command line is in effect until the end of the command line *or* until a subsequent **-x** option is encountered on the command line.

The following example uses input files with missing or non-standard file extensions. In this case, the *-x* options serve to inform c29clang how each input file is to be interpreted:

```
c29clang -mcpu=c29.c0 -c -x c file1 -x c++ file2.cxx
```

Passing Options to Other Tools from c29clang

This section of the *c29clang Compiler User Manual* describes how the c29clang's **-W<x>,** and **-X<y>** options can be used to pass options from c29clang 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 c29clang 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 **c29clang** command line.

Using the *-Wl,* Option

The c29clang *-Wl,* option can be used to identify one or more linker command line options to be forwarded from c29clang to the linker when the linker is invoked from the **c29clang** command line.

c29clang [*options*] [*filenames*] **-Wl,*<opt-list>***

- **-Wl,** - is the c29clang 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:

```
c29clang -mcpu=c29.c0 hello.c -o a.out -Wl,--rom_model,lnkme.cmd
```

Using *-Xlinker* Options

Alternatively, you can use the c29clang *-Xlinker* option to identify a single linker command line option to be forwarded from c29clang to the linker when the linker is invoked from the **c29clang** command line.

c29clang [*options*] [*filenames*] **-Xlinker** <*option*>

- **-Xlinker** - is the c29clang 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:

```
c29clang -mcpu=c29.c0 hello.c -o a.out -Xlinker --rom_model -
↪Xlinker lnkme.cmd
```

(continues on next page)

(continued from previous page)

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.

Diagnostic Options

Controlling Error, Warning, and Remark Diagnostics

The c29clang 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.

-Wextra
Enable the diagnostics that are enabled by **-Wall** plus a number of additional warning types.

-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.

-w
Disable all warnings.

In addition to enabling and disabling diagnostics on the command line, pragmas can be used control specific diagnostics in a section of code. See *clang diagnostic* for details and examples.

Optimization Feedback Options

The following options can be used to instruct the c29clang 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 c29clang 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

-gdwarf-4
: The c29clang 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 c29clang 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:
```

```
#include "func_timer.h"

extern "C" {

// Entry Hook Function
__attribute__((no_instrument_function))
```

(continues on next page)

(continued from previous page)

```
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:
```

```
#include "func_timer.h"

extern "C" {

// Function Exit Hook
```

(continues on next page)

(continued from previous page)

```
__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 c29clang 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;
   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;
```

(continues on next page)

(continued from previous page)

```
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:

```
%> c29clang -mcpu=c29.c0 -finstrument-functions <app source␣
↪files> \
 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.

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 c29clang 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 c29clang runtime libraries do not support a 16-bit *wchar_t* type. Attempts to use the **c29clang** *-fshort-wchar* option may cause issues when linked with the c29clang 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.
- Data and registers on C29x devices are always stored in little-endian format.

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,
```

(continues on next page)

(continued from previous page)

```
  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 c29clang 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 **c29clang** *-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,
```

(continues on next page)

(continued from previous page)

```
  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 C29x 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.

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 *C29x C Implementation-Defined Behavior*.

The following C library features are *not* currently supported for TI C29x:

- 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.
- C11 atomic operations are not supported.
- Threads and threads.h, which are optional in the C11 specification. The __STDC_NO_THREADS__ macro is not defined.

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 c29clang compiler supports a number of extended C language features that are commonly provided by other compilers.

Extensions supported in c29clang 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
Built-In Functions and Intrinsics

3.2.4 C29x 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 C29x 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 not 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 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 nearest". (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 an unsigned 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)* in function: *function*". (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 C29x C++

The c29clang 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
- Run-time type information (RTTI), which can be enabled with the *-frtti* compiler option.

The following features are *not* implemented or fully supported:

- Exception handling and the -fexceptions compiler option

- 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 c29clang compiler *does not* currently support the C++ exception handling features defined by the ANSI/ISO 14882 C++ Standard. The *-fexceptions* command-line option is not supported.

3.2.6 Pre-Defined Macro Symbols

The c29clang 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 **c29clang** option combination, you can compile using the *-E* and *-dM* options. For example,

```
%> c29clang -mcpu=c29.c0 -E -dM test.c
```

emits to *stdout* the list of pre-defined macro symbols that are defined when compiling with the *-mcpu=c29.c0* option.

The following sub-sections contain tables listing the various pre-defined macro symbols that are created by the c29clang compiler:

TI-Specific Pre-Defined Macro Symbols

The c29clang 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 c29clang 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 C29x-Specific Pre-Defined Macro Symbols

The c29clang compiler pre-defined macro symbols to identify the target processor. The following table summarizes such pre-defined macros:

Symbol	Value Kind	Value / Description
__C29__	<constant>	Defined to 1 if target is a C29x
__c29__	<constant>	Defined to 1 if target is a C29x
__C29_ARCH	<version>	Identifies the C29x architecture version being compiled for. Currently, the only value used is 0.
__C29_C0__	<constant>	Defined to 1 if the -mcpu=c29.c0 option is defined either on the command line or by default.
__C29_OPTF64	<constant>	Defined to 1 if the -mfpu option is set to f64 on the command line.
__C29_OPTF32	<constant>	Defined to 1 if the -mfpu option is set to f32 either on the command line or by default.

Generic Compiler Pre-Defined Macro Symbols

Version-Related Predefined Macro Symbols

Symbol	Value Kind	Value / Description
__clang__	<constant>	Defined to 1 if compiler uses Clang- based front-end.
__clang_major__	<version>	Identifies major version number of Clang front-end.
__clang_minor__	<version>	Identifies minor version number of Clang front-end.
__clang_patchlevel__	<version>	Identifies patch number of Clang front-end.
__clang_version__	<string>	String representation of Clang front-end version identification.
__GNUC_MINOR__	<version>	2
__GNUC_PATCHLEVEL__	<version>	1
__GNUC__	<version>	4
__GXX_ABI_VERSION__	<version>	1002
__llvm__	<constant>	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_UTF_16__	<constant>	Defined to 1 if char16_t type values are UTF-16 encoded.
__STDC_UTF_32__	<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, c29clang 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>	Never defined for C29x.
__LITTLE_ENDIAN__	<constant>	Always defined to 1 for C29x.
__BYTE_ORDER__	<constant>	Always matches the value of __ORDER_LITTLE_ENDIAN__ (1234) for C29x.
__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 deprecated -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[123szg]).
__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 data type on the C29x processor. This value is 8.
__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” or “unsigned” keyword) are interpreted by the compiler to be unsigned.
__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

continues on next page

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
__INT_FAST8_FMTd__	<string>	“hhd”
__INT_FAST8_FMTi__	<string>	“hhi”
__INT_FAST8_MAX__	<constant>	127
__INT_FAST8_TYPE__	<type>	signed char
__INT_FAST8_WIDTH__	<constant>	8
__UINT_FAST8_FMTX__	<string>	“hhX”
__UINT_FAST8_FMTo__	<string>	“hho”
__UINT_FAST8_FMTu__	<string>	“hhu”
__UINT_FAST8_FMTx__	<string>	“hhx”
__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
__INT_LEAST8_WIDTH__	<constant>	8
__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
__SHRT_WIDTH__	<constant>	16
__INT_FAST16_FMTd__	<string>	“hd”
__INT_FAST16_FMTi__	<string>	“hi”
__INT_FAST16_MAX__	<constant>	32767
__INT_FAST16_TYPE__	<type>	short
__INT_FAST16_WIDTH__	<constant>	16

continues on next page

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
__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
__INT_LEAST16_WIDTH__	<constant>	16
__UINT_LEAST16_FMTX__	<string>	“hX”
__UINT_LEAST16_FMTo__	<string>	“ho”
__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
__INT_WIDTH__	<constant>	32
__LONG_MAX__	<constant>	2147483647
__SIZEOF_LONG__	<bytes>	4
__LONG_WIDTH__	<constant>	32
__INT_FAST32_FMTd__	<string>	“d”
__INT_FAST32_FMTi__	<string>	“i”
__INT_FAST32_MAX__	<constant>	2147483647
__INT_FAST32_TYPE__	<type>	int
__INT_FAST32_WIDTH__	<constant>	32
__UINT_FAST32_FMTX__	<string>	“X”

continues on next page

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
__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
__INT_LEAST32_WIDTH__	<constant>	32
__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
__INT64_FMTd__	<string>	“lld”
__INT64_FMTi__	<string>	“lli”
__INT64_MAX__	<constant>	92233720368547758O7LL
__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>	92233720368547758O7LL
__SIZEOF_LONG_LONG__	<bytes>	8
__INT_FAST64_FMTd__	<string>	“lld”
__INT_FAST64_FMTi__	<string>	“lli”
__INT_FAST64_MAX__	<constant>	92233720368547758O7LL
__INT_FAST64_TYPE__	<type>	long long int
__INT_FAST64_WIDTH__	<constant>	64
__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

continues on next page

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
__UINT_FAST64_TYPE__	<type>	long long unsigned int
__INT_LEAST64_FMTd__	<string>	“lld”
__INT_LEAST64_FMTi__	<string>	“lli”
__INT_LEAST64_MAX__	<constant>	922337203685477580 7LL
__INT_LEAST64_TYPE__	<type>	long long int
__INT_LEAST64_WIDTH__	<constant>	64
__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>	922337203685477580 7LL
__INTMAX_TYPE__	<type>	long long int
__INTMAX_WIDTH__	<bits>	64
__UINTMAX_C_SUFFIX__	<text>	ULL
__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
__UINTMAX_WIDTH__	<constant>	64
__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

continues on next page

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
_ILP32	<constant>	Defined to 1 if pointer type width and long type width is 32-bits.
__ILP32__	<constant>	Defined to 1 if pointer type width and long type width is 32-bits.
__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_MIN__	<constant>	(-2147483647-1)
__WCHAR_TYPE__	<constant>	int
__WCHAR_WIDTH__	<bits>	32
__SIZEOF_WCHAR_T__	<bytes>	4
__WINT_MAX__	<constant>	4294967295
__WINT_TYPE__	<constant>	unsigned int
__WINT_WIDTH__	<bits>	32
__WINT_UNSIGNED__	<constant>	1
__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

continues on next page

Table 3.2 – continued from previous page

Symbol	Value Kind	Value / Description
__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-324L
__LDBL_DIG__	<digits>	15
__LDBL_EPSILON__	<constant>	2.2204460492503131e-16L
__LDBL_MANT_DIG__	<bits>	53
__LDBL_MAX_10_EXP__	<constant>	308
__LDBL_MAX_EXP__	<constant>	1024
__LDBL_MAX__	<constant>	1.7976931348623157e+308L
__LDBL_MIN_10_EXP__	<constant>	-307
__LDBL_MIN_EXP__	<constant>	-1021
__LDBL_MIN__	<constant>	2.2250738585072014e-308L

3.2.7 Attributes

Contents:

Attribute Syntax

There are three different kinds of attributes supported by the c29clang 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*
- *no_profile_instrument_function*
- *noinline*
- *nomerge*
- *nonnull*
- *noreturn*
- *optnone*
- *pure*
- *section*
- *used/retain*
- *visibility*
- *weak*
- *weakref*

The following additional function attribute is specific to C29x.

- *interrupt_nosave_fp*

See *C29x Security Model* for information about protected calls and use of the `c29_protected_call` function attribute.

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);
}
...
```

```
%> c29clang -mcpu=c29.c0 -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:

```
// alias_func_attr.c
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
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*:

```
%> c29clang -mcpu=c29.c0 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*:

```
#include <stdio.h>

void event_handler() {
  printf("This is the event handler implementation\n");
}
```

```
%> c29clang -mcpu=c29.c0 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:

```
#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):

```
%> c29clang -mcpu=c29.c0 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 **c29clang** 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*:

```
#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 inlines the body of *emit_msg*, even though no optimization was specified on the command line:

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));

Example

The following C code is a simple example of the use of the *const* attribute:

```
#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:

```
%> c29clang -mcpu=c29.c0 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:

```
#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, c29clang will emit a warning diagnostic to indicate a reference to the *deprecated* function *dep_func* has been encountered:

```
%> c29clang -mcpu=c29.c0 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 c29clang 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:

```
#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, c29clang 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.

```
%> c29clang -mcpu=c29.c0 -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:

```
#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, c29clang 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*.

```
%> c29clang -mcpu=c29.c0 -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.

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 c29clang compiler supports the following *interrupt-type* identifiers:
  - `__attribute__((interrupt("RTINT")))`
  - `__attribute__((interrupt("INT")))`

You can generate a Realtime Interrupt (RTINT) or maskable interrupt (INT) from C using the __attribute__((interrupt())) function attribute. For example:

```
void my_high_priority interrupt() __attribute__((interrupt("RTINT
↪")));
void my_low_priority interrupt() __attribute__((interrupt("INT
↪")));

__attribute__((interrupt("RTINT"))) void int2() {
  do_something2();
}
__attribute__((interrupt("INT"))) void int3() {
  do_something3();
}
```

An interrupt is responsible for saving any and all registers that may be used, so as to preserve the state of the registers as they were before the interrupt was called. An interrupt containing a call will save a large number of registers, which degrades code size and speed.

See *C29x Security Model* for information about protected calls and use of the `c29_protected_call` function attribute.

In addition, built-in C29x interrupt intrinsics can be used to disable/enable INT interrupts and block RTINT/INT interrupts. See *Built-In Functions and Intrinsics* for details.

location

The *location* function attribute can be used to specify a function's run-time address from within the C/C++ source. The c29clang 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.

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.

```
#include <stdio.h>

__attribute__((location(0x2001))) int main() {
```

(continues on next page)

(continued from previous page)

```
  printf("Good morning, Dave.\n");
  return 0;
}
```

```
%> c29clang -mcpu=c29.c0 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:

```
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 C29x 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:

```
__attribute__((naked)) int sub(int arg1, int arg2) {
  __asm(" SUB D0, D0, D1");
}
```

Note: *naked* attribute implies *no_profile_instrument_function* attribute

If a *naked* attribute is applied to a function, then the *no_profile_instrument_function* attribute will also be applied to the function. This is to prevent the use of code coverage instrumentation from introducing code into the body of the *naked* function definition, which would violate the purpose of the *naked* attribute.

---

no_profile_instrument_function

The *no_profile_instrument_function* function attribute instructs the compiler not to instrument a function's definition with code that will record coverage data when code coverage is enabled during a compilation.

Syntax

__attribute__((no_profile_instrument_function)) <return type> *symbol* (<arguments>);

Example

Given a function definition without the *no_profile_instrument_function* attribute applied to it:

```
extern int func2(int n);

int func1(int n) {
  return func2(n);
}
```

If compiled with function coverage enabled:

```
%> c29clang -mcpu=c29.c0 -fprofile-instr-generate \
   -fcoverage-mapping -ffunction-coverage-only -S func.c
```

The compiler-generated assembly for *func1()* looks like this:

```
...
    .file   "func.c"
    .section        .text.func1,"ax",@progbits
    .globl  func1                           ; -- Begin function␣
↪func1
    .type   func1,@function
    .c29_arg_rmask  func1,4096
    .c29_ret_rmask  func1,4096
func1:                                  ; @func1
    ADD.U16 A15,#0x8
    MV      A4,D0
    ST.32   *(ADDR1)(A15-#0x8),A4
    LD.64   XD0,@.L__profc_func1    @ <-- coverage␣
↪instrumentation load
    ONE     XD2                     @ <-- XD2 = 1
    ADD     XD0,XD0,XD2             @ <-- increment
    ST.64   @.L__profc_func1,XD0    @ <-- store
    LD.32   D0,*(ADDR1)(A15-#0x8)
    CALL    @func2
    RET     *(ADDR1)(A15-=#0x10)
```

As you can see, the above generated assembly code includes a sequence of four instructions that loads the code coverage function counter associated with *func1()*, increments the function counter, and finally stores the function counter

If a *no_profile_instrument_function* attribute is applied to the definition of *func1()*:

```
extern int func2(int n);

__attribute__((no_profile_instrument_function))
```

(continues on next page)

(continued from previous page)

```
int func1(int n) {
  return func2(n);
}
```

And code coverage options are again specified for the compilation of the source file containing the updated definition of *func1()*:

```
%> c29clang -mcpu=c29.c0 -fprofile-instr-generate \
   -fcoverage-mapping -ffunction-coverage-only -S func.c
```

The compiler-generated assembly code emitted for the definition of *func1()* looks like this:

```
...
    .file   "func.c"
    .section        .text.func1,"ax",@progbits
    .globl  func1                           ; -- Begin function␣
↪func1
    .type   func1,@function
    .c29_arg_rmask  func1,4096
    .c29_ret_rmask  func1,4096
func1:                                  ; @func1
    ADD.U16 A15,#0x8
    MV      A4,D0
    ST.32   *(ADDR1)(A15-#0x8),A4
    LD.32   D0,*(ADDR1)(A15-#0x8)
    CALL    @func2
    RET     *(ADDR1)(A15-=#0x10)
```

Thus, the application of the *no_profile_instrument_function* attribute prevents the compiler from inserting coverage isntrumentation code into the definition of *func1()* even if code coverage enabling options are specified on the tiarmclang command-line.

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:

```
#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 generates assembly instructions for *main* that do not inline *incr_counter*. However, the loop is unrolled to eliminate the loop control flow overhead.

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:

```
void callee_func(const char *str) __attribute__((nomerge));

int caller_func(int n) {

  if (n < 10) {
    callee_func("string for return 1");
    return 1;
  }
```

(continues on next page)

(continued from previous page)

```
  else {
    callee_func("string for return 2");
    return 2;
  }

  return 0;
}
```

The compiler generates assembly instructions for the definition of *caller_func* that do not merge calls to *callee_func*.

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*:

```
void callee_func(int n, int *p) __attribute__((nonnull));

void caller_func(int n) {
  callee_func(n, &n);
  callee_func(n, (int *)0);
}
```

```
%> c29clang -mcpu=c29.c0 -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; }
```

```
%> c29clang -mcpu=c29.c0 -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 to apply no optimizations in the generation of code for the function. This has the same effect on a single function as the -O0 command-line option has on the entire application.

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*:

```
void check_peripheral();
__attribute__((optnone)) void park_and_wait();

void run_a_check_on_peripheral(void) {
  check_peripheral();
```

(continues on next page)

(continued from previous page)

```
}

void check_peripheral() {
  if (*((unsigned long *)(268612608)) != 286529877) {
    park_and_wait();
  }
}

__attribute__((optnone)) void park_and_wait() {
  while(1) {
    ;
  }
}
```

The compiler generates assembly instructions that reference *park_and_wait* even though the *-O2* optimization option was specified. If the *optnone* attribute had 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*.

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*:

```
#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*:

```
%> c29clang -mcpu=c29.c0 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 c29clang 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:

```
#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:

```
%> c29clang -mcpu=c29.c0 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*:

```
#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:

```
%> c29clang -mcpu=c29.c0 -c visibility_function_attr.c
%> c29ofd -v visibility_function_attr.o
...
 Symbol Table ".symtab"
...
```

(continues on next page)

(continued from previous page)

```
    <6> "my_func"
       Value:    0x00000000  Kind:        undefined
       Binding:  global      Type:        none
       Size:     0x0         Visibility:  STV_PROTECTED
...
```

weak

The *weak* function attribute causes the c29clang 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*:

```
#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*:

```
__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*:

```
%> c29clang -mcpu=c29.c0 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*:

```
%> c29clang -mcpu=c29.c0 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 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*:

```
#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*:

```
__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 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:

```
%> c29clang -mcpu=c29.c0 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*:

```
%> c29clang -mcpu=c29.c0 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
```

interrupt_nosave_fp

Like the *interrupt* attribute, the *interrupt_nosave_fp* function attribute, when applied, identifies a function as an interrupt function and instructs the compiler to generate code on entry and exit to preserve the system state.

In addition, the *interrupt_nosave_fp* function attribute instructs the compiler to avoid generating floating point register save/restore. By default the *interrupt* attribute causes floating point registers to be saved/restored for the c29clang compiler.

This attribute can improve code size and performance if you know an application does not use floating point operations or registers.

Syntax

__attribute__((interrupt_nosave_fp)) <return type> *symbol* (<arguments>) { ... }

__attribute__((interrupt_nosave_fp[("*interrupt-type*")])) void *symbol* () { ... }

- *interrupt-type* - is the optional string literal argument that indicates the type of interrupt being defined. The c29clang compiler supports the following *interrupt-type* identifiers:
  - `__attribute__((interrupt("RTINT")))`
  - `__attribute__((interrupt("INT")))`

Example

See the *interrupt* attribute for examples that can also be used with the *interrupt_nosave_fp* attribute.

Variable Attributes

The following variable attributes are supported by the c29clang compiler:

- *alias*
- *aligned*
- *deprecated*
- *location*
- *noinit*
- *packed*
- *persistent*
- *section*
- *unused*
- *used/retain*

- *visibility*
- *weak*
- *weakref*

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 c29clang 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:

```
#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 generates assembly instructions for the above code that defines the global variable *red_fish* and 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):

```
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 generates assembly instructions for the above code that do the following:

- Align *var1* to an 8-byte boundary.
- Align *var2* to a 16-byte boundary.
- Align *var3*'s char type member, *m3*, by padding 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:

```
%> c29clang -mcpu=c29.c0 -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.

See *clang deprecated* for an example that uses the deprecated pragma to mark a macro as deprecated.

location

The *location* variable attribute can be used to specify a variable's run-time address from within the C/C++ source. The c29clang 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):

```
#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:

```
%> c29clang -mcpu=c29.c0 location_var_attr.c -o a.out -Wl,-llnk.
 ↪cmd,-ma.map
%> cat a.map
*******************************************************************************
C29 Clang Linker Unix v1.2.0
*******************************************************************************
>> Linked Tue Jan 19 18:49:47 2024
```

(continues on next page)

(continued from previous page)

```
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):

```
#include <stdio.h>

extern void usei(int *x);

__attribute__((noinit)) int noinit_global;
__attribute__((noinit,location(0x100))) int noinit_location_
 ↪global;
```

(continues on next page)

(continued from previous page)

```
int main() {
  usei(&noinit_global);
  usei(&noinit_location_global);
  return 0;
}
```

The compiler generates assembly instructions for the above code that do the following:

- Defines *noinit_global* in a special section, *.TI.noinit*, to keep such data objects apart from other variable definitions that will be initialized.
- Emits symbol metadata information along with the definition of *noinit_global* to indicate that the section where *noinit_global* is defined is not to be initialized.
- Defines *noinit_location_global* in a special section, *.TI.bound:noinit_location_global*, that the linker will consider for placement early in the link-step.
- Two pieces of symbol metadata information are 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 a C29x 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;

void main() {
  run_init();
  while (1) {
    run_actions(x);
    delay(1000000);
    x++;
  }
}
```

generates a definition of *x* that is directly initialized 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

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 generates assembly instructions that define *bufferA* in a common block, whereas *bufferB* is defined in the *my_sect* section.

unused

When the *unused-variable* category of warning diagnostics is enabled, the c29clang 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()
{
```

(continues on next page)

(continued from previous page)

```
  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), c29clang emits a warning about unused variable *b_var*, but not *a_var*.

```
%> c29clang -mcpu=c29.c0 -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 c29clang 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 c29clang 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:

```
#include <stdio.h>

int used_varX __attribute__((retain)) = 10;

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

After compiling and linking with the following command:

```
%> c29clang -mcpu=c29.c0 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)
                   ...
```

(continues on next page)

(continued from previous page)

```
                    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:

```
%> c29clang -mcpu=c29.c0 -c visibility_var_attr.c
%> c29ofd -v visibility_var_attr.o
...
 Symbol Table ".symtab"
```

(continues on next page)

(continued from previous page)

```
    <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 c29clang 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*:

```
#include <stdio.h>

extern int my_var;

int main() {
  printf("my_var is: %d\n", my_var);
  return 0;
}
```

weak_def.c:

```
int my_var __attribute__((weak)) = 5;
```

and *strong_def.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*:

```
%> c29clang -mcpu=c29.c0 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*:

```
%> c29clang -mcpu=c29.c0 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:

```
%> c29clang -mcpu=c29.c0 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:

```
%> c29clang -mcpu=c29.c0 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>
```

(continues on next page)

(continued from previous page)

```
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 c29clang reports an unresolved symbol reference:

```
%> c29clang -mcpu=c29.c0 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.
```

Type Attributes

The c29clang 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 c29clang 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*:

```
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:

```
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:

```
#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)));
} 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);
```

(continues on next page)

(continued from previous page)

```
  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*:

```
%> c29clang -mcpu=c29.c0 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 c29clang compiler:

- *clang attribute push/pop*
- *clang diagnostic*
- *clang section text*
- *clang section data*
- *clang section bss*
- *clang section rodata*
- *clang deprecated*

clang attribute push/pop

The *clang attribute push* and *clang attribute pop* pragmas apply an attribute to multiple declarations. An attribute can be pushed to or popped from the scope of the specified declarations. The scopes can be nested using multiple push directives.

Syntax

```
#pragma clang attribute push (__attribute__((<attribute_name>)),␣
↪apply_to = <match_scope>)
#pragma clang attribute pop
```

Example

The following use of the *clang attribute push* and *clang attribute pop* pragmas causes the enclosed variables to have the noinit attribute.

```
#pragma clang attribute push([[used]], apply_to = any(record,␣
↪enum, function))

enum myEnum { ABC, XYZ }; // The enum has the used attribute
struct myRecord { };      // The struct has the used attribute
void myFunction();        // The function has the used attribute

#pragma clang attribute pop

enum otherEnum { LMN, OPQ }; // The enum does not have the used␣
↪attribute
```

See Specifying an attribute for multiple declarations in the Clang documentation for details and examples, including a list of supported match scopes and how to use a namespace to distinguish the push directive to which a pop directive applies.

clang diagnostic

The *clang diagnostic* pragma provides run-time control over which diagnostics are enabled and their severity levels. See *Diagnostic Options* for diagnostic-related command-line options that can be controlled with this pragma.

You can nest diagnostic behavior with the *clang diagnostic push* and *clang diagnostic pop* pragmas, but these pragmas must be used in pairs.

The *GCC diagnostic* pragma is functionally a synonym for the *clang diagnostic* pragma when using the c29clang compiler.

Syntax

```
#pragma clang diagnostic ignored "-W<option>"
#pragma clang diagnostic warning "-W<option>"
#pragma clang diagnostic error "-W<option>"
#pragma clang diagnostic fatal "-W<option>"
#pragma clang diagnostic push
#pragma clang diagnostic pop
```

Example

The following *clang diagnostic ignored* pragma causes warnings about extra tokens in a preprocessor directive to be ignored.

```
#pragma clang diagnostic ignored "-Wextra-tokens"
```

The following *clang diagnostic warning* pragma changes errors about implicit function declarations to warnings.

```
#pragma clang diagnostic warning "-Wimplicit-function-declaration
↪"
```

See Controlling Diagnostics via Pragmas in the Clang documentation for further details and examples.

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*

```
#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*

```
#include <stdio.h>

#pragma clang section bss=".bss.variables"

int var1 = 39;
```

(continues on next page)

(continued from previous page)

```
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*

```
#include <stdio.h>

#pragma clang section rodata="MyRodata"

const int var1 = 39;
const char *myString = "this is a test";

#pragma clang section rodata=""
```

(continues on next page)

(continued from previous page)

```
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,

```
#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:

```
#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.

---

clang deprecated

The *clang deprecated* pragma identifies a macro as deprecated and provides a message to be output when compiling code that uses the deprecated macro.

Syntax

#pragma clang deprecated(MACRO, "message")

Example

The following use of the *clang deprecated* pragma causes the specified error message to be output when compiling.

```
#define X 0
#define NEW_X 1
#pragma clang deprecated(X, "Macro X deprecated, use NEW_X␣
↪instead")

int foo() { return NEW_X; } // This use of NEW_X is fine
int main() { return X + foo(); } // This use of X is deprecated
```

See *deprecated* for an example that uses the deprecated attribute to mark a variable as deprecated.

3.2.9 Built-In Functions and Intrinsics

The C29x compiler has many intrinsic functions (also called "built-in functions") that provide built-in access from C/C++ to assembly instructions or sequences of instructions. All intrinsic functions begin with the prefix `__builtin_c29_`.

Note: This page describes only some of the intrinsics that are available. For a list of builtin intrinsics supported by the C29x compiler, see the *C2000 C29x CPU and Instruction Set User's Guide* (SPRUIY2), which is available through your TI Field Application Engineer. Search for `__builtin` in that guide.

Note that you can use the `__has_builtin` function-like preprocessor macro to test for the existence of a built-in function. This, and other preprocessor macros are extensions provided by Clang. It evaluates to 1 if the function is supported or 0 if not. It can be used like this:

```
#if __has_builtin(__builtin_c29_fast_strlen)
    mystrlen __builtin_c29_fast_strlen( mystr );
#else
    abort();
#endif
```

See the Clang Language Extensions section of the Clang documentation for details and additional macros and extensions.

Interrupt Control Intrinsics

Table 3.3: Interrupt Control Intrinsics

Intrinsic syntax	Description
unsigned int __builtin_c29_disable_INT()	Disable INT interrupts
unsigned int __builtin_c29_enable_INT()	Enable INT interrupts
void __builtin_c29_restore_INT(unsigned int ui0)	Restore previous INT interrupt enable/disable setting as stored in ui0
void __builtin_c29_atomic_enter()	Interrupts are blocked until ATOMIC counter reaches 0 or ATOMIC_END instruction is executed. NMI interrupts are not blocked.
void __builtin_c29_atomic_mem_enter()	Interrupts are blocked until ATOMIC counter reaches 0 or ATOMIC_END instruction is executed. NMI interrupts not blocked. Generate side band strobes to indicate that this is a window of mutually exclusive (MUTEX) memory operations.
void __builtin_c29_atomic_leave()	Clears ATOMIC counter set by __builtin_c29_atomic_enter() or __builtin_c29_atomic_mem_enter(.)

String and Memory Intrinsics

Table 3.4: String and Memory Intrinsics

Intrinsic syntax	Description
void __builtin_c29_fast_memcpy(void *dest, const void * src, size_t numBytes)	Optimized instruction sequence for memcpy() function. This has the same signature as its associated C library function.
int __builtin_c29_fast_strcmp(const char* str1, const char* str2)	Optimized instruction sequence for strcmp() function. This has the same signature as its associated C library function. It requires 4 bytes of post-padding.
size_t __builtin_c29_fast_strlen(const char* str)	Optimized instruction sequence for strlen() function. This has the same signature as its associated C library function. It requires up to 60 bytes of post-padding.

Arithmetic Intrinsics

The *C2000 C29x CPU and Instruction Set User's Guide* (SPRUIY2), which is available through your TI Field Application Engineer, describes the arithmetic instructions available for C29x processors. For each instruction that has a corresponding intrinsic function that can be called from C/C++, the syntax is provided. The arithmetic intrinsics include intrinsics to perform:

- Integer addition, subtraction, multiplication, and division
- Absolute and negative value operations
- Integer comparisons
- Increment/decrement operations
- Logarithmic operations
- Trigonometric operations

- Square root and inverse power operations
- Bit counting and searching
- Bitwise AND, ANDOR, OR, and XOR operations
- Left and right shifts
- Cyclic Redundancy Checks (CRC)

Separate versions of the intrinsics operate on various registers and datatypes and/or perform variants such as signed or unsigned integer saturation.

For division operations, the following intrinsics return the quotient and remainder in *quot* and *rem. These functions fall into the following variant categories:

- Unsigned division: div
- Traditional (truncated) signed division: tdiv
- Euclidean signed division: ediv
- Modulo (floored) signed division: mdiv

```
void __builtin_c29_div_u32_u32(unsigned *quot, unsigned *rem,␣
↪unsigned a, unsigned b)
void __builtin_c29_div_u64_u32(unsigned long long *quot,␣
↪unsigned *rem,
                               unsigned long long a, unsigned b)
void __builtin_c29_div_u64_u64(unsigned long long *quot,␣
↪unsigned long long *rem,
                               unsigned long long a, unsigned␣
↪long long b)

void __builtin_c29_tdiv_s32_u32(int *quot, int *rem, int a,␣
↪unsigned b)
void __builtin_c29_tdiv_s32_s32(int *quot, int *rem, int a, int␣
↪b)
void __builtin_c29_tdiv_s64_u32(long long *quot, int *rem, long␣
↪long a, unsigned b)
void __builtin_c29_tdiv_s64_s32(long long *quot, int *rem, long␣
↪long a, int b)
void __builtin_c29_tdiv_s64_s64(long long *quot, long long *rem,␣
↪long long a, long long b)

void __builtin_c29_ediv_s64_s32(long long *quot, int *rem, long␣
↪long a, int b)
void __builtin_c29_ediv_s32_s32(int *quot, int *rem, int a, int␣
↪b)
```

(continues on next page)

(continued from previous page)

```
void __builtin_c29_ediv_s64_s64(long long *quot, long long *rem,␣
↪long long a, long long b)

void __builtin_c29_mdiv_s64_s32(long long *quot, int *rem, long␣
↪long a, int b)
void __builtin_c29_mdiv_s32_s32(int *quot, int *rem, int a, int␣
↪b)
void __builtin_c29_mdiv_s64_s64(long long *quot, long long *rem,␣
↪long long a, long long b)
```

Floating-Point Arithmetic Intrinsics

The *C2000 C29x CPU and Instruction Set User's Guide* (SPRUIY2), which is available through your TI Field Application Engineer, describes the floating-point arithmetic instructions available for C29x processors. For each instruction that has a corresponding intrinsic function that can be called from C/C++, the syntax is provided. The floating-point arithmetic intrinsics provided include intrinsics to perform:

- Floating-point addition, subtraction, and multiplication
- Floating-point comparisons
- Absolute value and negation operations
- Fractional portion extraction

Note: The following intrinsics expand to a sequence of instructions. This is in contrast to other intrinsics listed on this page, which generally correspond to a single instruction. The purpose of the following two intrinsics is to perform more accurate floating point division than the provided floating point division instructions, such as DIVF32.

The following intrinsics return the quotient of the floating point division `a/b`.

```
float __builtin_c29_div_f32(float a, float b)

double __builtin_c29_div_f64(double a, double b)
```

Conversion Intrinsics

The *C2000 C29x CPU and Instruction Set User's Guide* (SPRUIY2), which is available through your TI Field Application Engineer, describes the conversion instructions available for C29x processors. For each instruction that has a corresponding intrinsic that can be called from C/C++, the syntax is provided. The conversion intrinsics provided include intrinsics to perform:

- Convert between 16-bit signed integer and 32-bit float formats
- Convert between 16-bit unsigned integer and 32-bit float formats

Co-Processor Interface (CPI) Intrinsics

The *C2000 C29x CPU and Instruction Set User's Guide* (SPRUIY2), which is available through your TI Field Application Engineer, describes the CPI instructions available for C29x processors. For each instruction that has a corresponding intrinsic function that can be called from C/C++, the syntax is provided. The CPI intrinsics provided include intrinsics to perform:

- Copy values between registers on the CPU to registers on the CPI Interface port (CIDy)
- Generate a tag value to be used for data logging.

Other Control Flow Intrinsics

The *C2000 C29x CPU and Instruction Set User's Guide* (SPRUIY2), which is available through your TI Field Application Engineer, describes the control flow instructions available for C29x processors. For each instruction that has a corresponding intrinsic function that can be called from C/C++, the syntax is provided. The control flow intrinsics provided include intrinsics to perform:

- Copy and conditional copy from register to register
- Test a bit position in a register
- Perform integer comparison; set flags based on results
- Perform floating-point comparison; set flags based on results

Load, Store, and Move Intrinsics

The *C2000 C29x CPU and Instruction Set User's Guide* (SPRUIY2), which is available through your TI Field Application Engineer, describes the load, store, and move instructions available for C29x processors. For each instruction that has a corresponding intrinsic function that can be called from C/C++, the syntax is provided. The load, store, and move intrinsics provided include intrinsics to perform:

- Load from memory location
- Store to memory location

- Copy from memory location to memory location/register
- Modify memory location
- Add to memory location and load
- Subtract from memory location and load
- Sign extension
- Zero extension
- Swap bit(s)
- Swap byte
- Split register
- Zero masking

3.2.10 Keywords

The c29clang 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 c29clang 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 c29clang 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 c29clang 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 may 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 c29clang 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 c29clang 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.

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().

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 A15 register to manage this stack. A15 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 **c29lnk** command (use *-Wl,* or *-Xlinker* prefix for the linker option if invoking the linker from **c29clang**, 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.

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 *c29ofd - 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 **c29lnk** command ((use *-Wl,* or *-Xlinker* prefix for the linker option if invoking the linker from **c29clang**, 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.

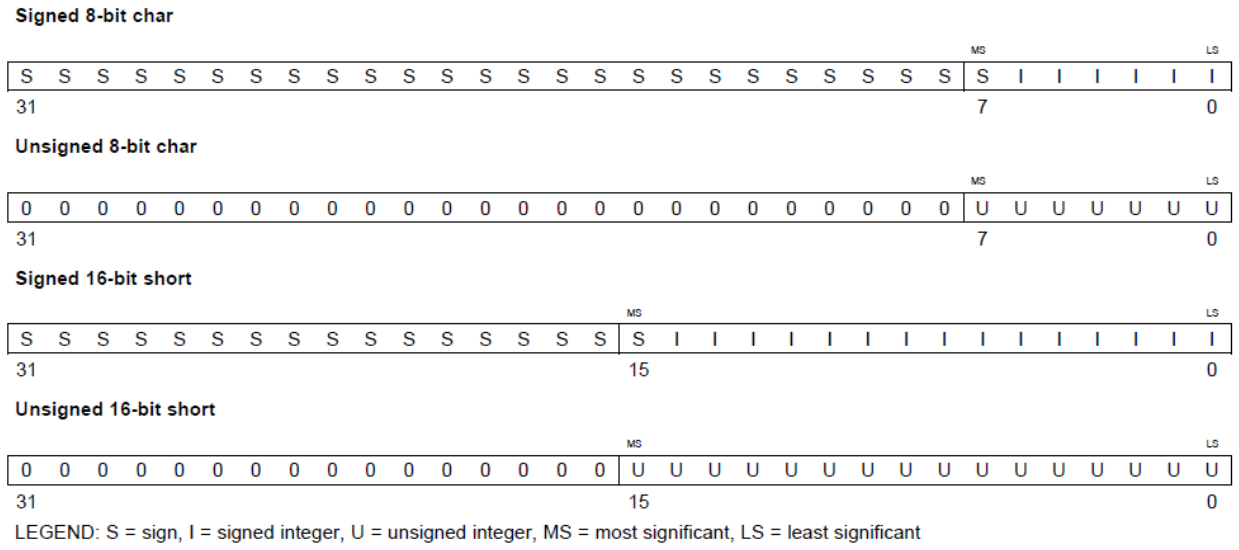


Figure 3.1: Char and Short Data Storage Format

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.

Because the C29x is always 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.

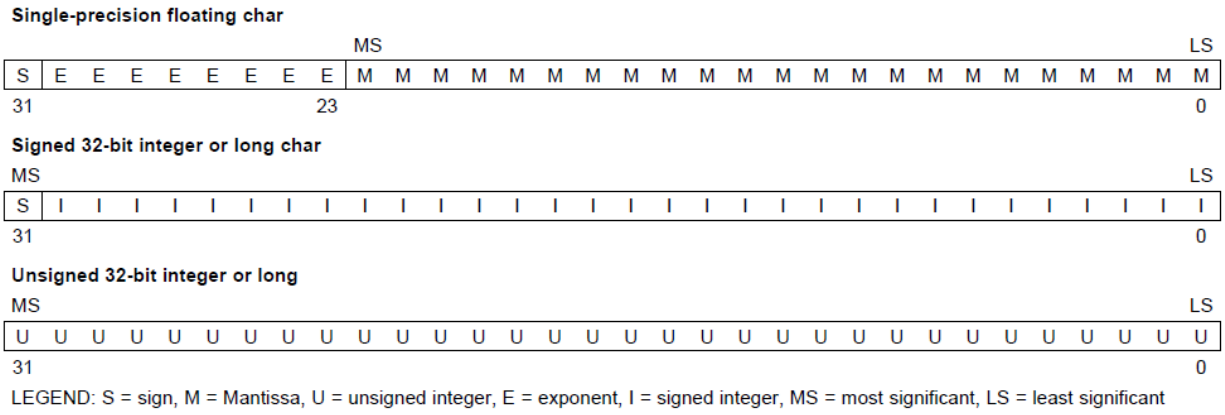


Figure 3.2: 32-Bit Data Storage Format

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.

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.

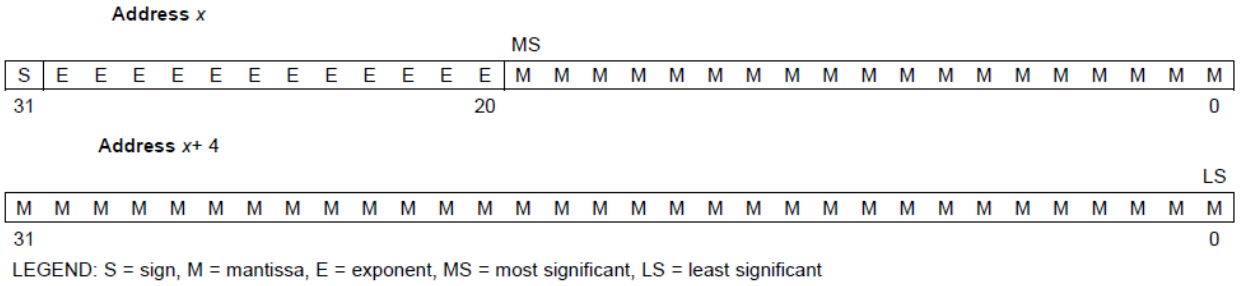


Figure 3.3: Double-Precision Floating-Point Data Storage Format

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.

Because the C29x is always in little-endian mode, bit fields are packed into registers from the least significant bit (LSB) to the most significant bit (MSB) in the order in which they are defined, and packed in memory from least significant byte (LSbyte) to most significant byte (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.

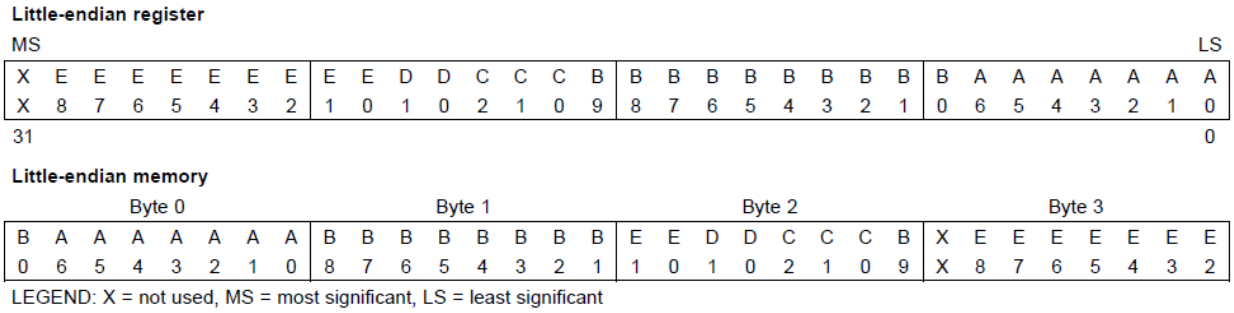


Figure 3.4: Bit-Field Packing in Little-Endian Format

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, along with a unique label that points to the string; the terminating 0 byte is included.

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 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 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.

There are a few things to be aware of when working with stack memory on TI C29x devices:

- A15 is used as the stack pointer (SP). The SP points to the next empty location.
- The stack grows "down" (from low to high addresses).
- The stack is 64-bit aligned on function entry/exit.

This document uses the following 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. This block is allocated to the largest size required of all calls in the function. It is allocated on function entry and deallocated on function exit. It is populated by moves or copies of values prior to the function call. (This is different from other targets, which push/pop onto the stack on-demand.)
- **Register save area:** The part of the local frame into which caller-saved registers are copied when the program calls a function and from which values are restored to registers when the program returns control to the caller.
- **Caller-saved (alternately, save-on-call) registers:** The callee function does not preserve values in these registers; therefore, the caller function must save them, potentially to the register save area if their values need to be preserved.
- **Callee-saved (alternately, save-on-entry) registers:** The callee function must preserve the values in these registers. If the callee function modifies these registers, it saves the value to the stack and restores the value when it returns control to the caller.
- **Calling Convention:** A calling convention is a description of how arguments are passed from caller to callee, and how values are returned from callee back to caller.

Due to the TI C29x security subsystem, there are multiple calling conventions:

- **Unprotected Calls**
  - **Caller-saved registers:**
    - D0 - D9, XD0 - XD8
    - A0 - A9, XA0 - XA8
    - M0 - M25, XM0 - XM24
    - TA0-TA4, TDM0-TDM4
  - **Callee-saved registers:**

        * D10 - D15, XD10 - XD14
        * A10 - A14, XA10 - XA12
        * M26 - M31, XM26 - XM30
    - **Argument registers:**
        * A4-A9
        * D0-D7, XD0-XD4
        * M0-M7, XM0-XM6
        * The stack may be used to store arguments beyond the above available registers listed, see *Arguments* below.
    - **Return registers:**
        * A4
        * D0, XD0
        * M0, XM0
- **Protected Calls**
    - **Caller-saved registers:**
        * All registers live across a protected call are caller-saved
    - **Callee-saved registers:**
        * No registers are saved by the callee
    - **Argument registers:**
        * A4-A9
        * D0-D7, XD0-XD4
        * M0-M7, XM0-XM6
        * The stack may not be used to store arguments beyond the above available registers, see *Arguments* below.
    - **Return registers:**
        * A4
        * D0, XD0
        * M0, XM0

The following figure shows stack usage before and after a typical function call.

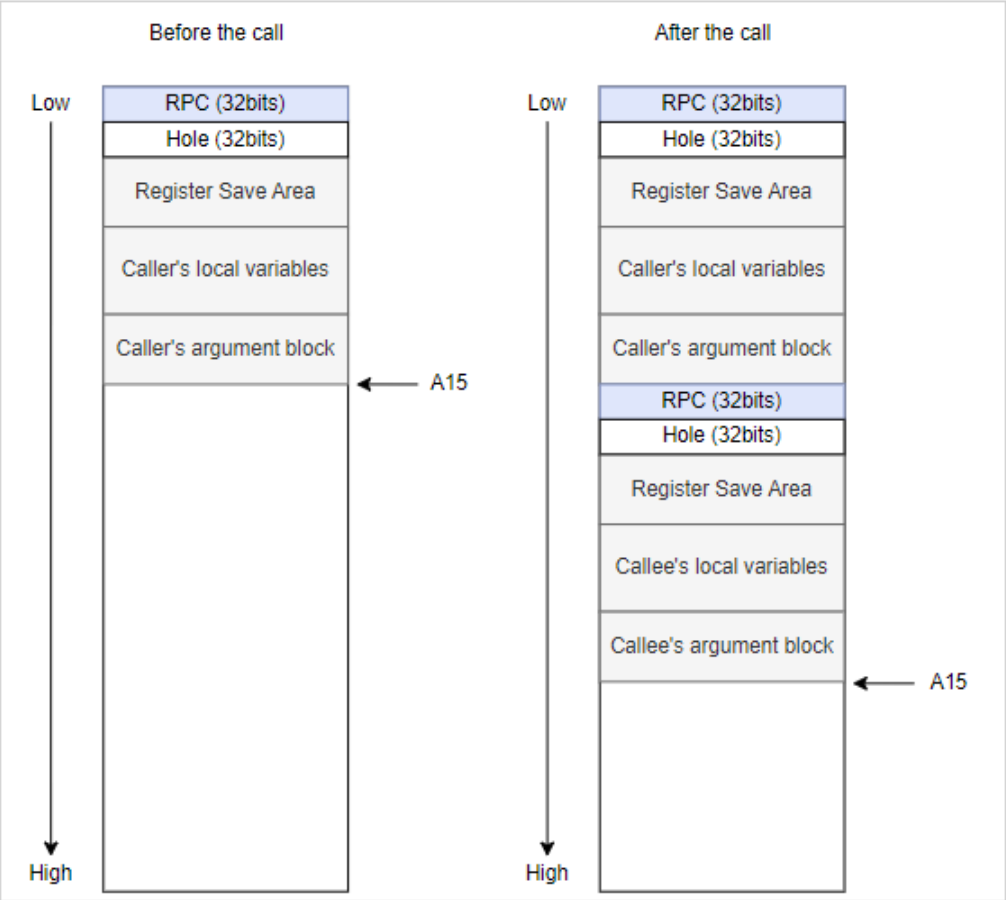


Figure 3.5: Use of the Stack During a Function Call

Note:

- The A15 register is used as the stack pointer (SP); SP points to the next empty location.
- Stack grows from low to high addresses.
- Stack is 64-bit aligned.

Arguments

Values passed from caller to callee follow a strict set of rules. A single unique location is assigned for each argument; this location does not vary program-to-program. See Examples 1 and 2.

- Arguments are assigned from first to last, and are assigned to the first valid and available register.
- Pointer arguments are assigned to A4-A9, in increasing order.

- Integer arguments are assigned to D0-D7, in increasing order. A 64-bit integer is assigned to a pair of registers.
- Floating-point arguments are assigned to M0-M7, similar to integer arguments.

Example 1:

```
void foo(int a, long long b, int c, int d, int e)
```

- `a`, a 32-bit integer, is assigned to D0.
- `b`, a 64-bit integer, must be assigned to a pair. The next available pair is XD2, so `b` is assigned to XD2 and D1 is left open.
- `c`, a 32-bit integer, is assigned to the next free 32-bit register, D1.
- `d` and `e`, 32-bit integers, are assigned to the next available registers. D2 and D3 are already assigned, so they are placed in D4 and D5, respectively.

Example 2:

```
void bar(int x, long long y, double z, char *h)
```

- `x`, a 32-bit integer, is assigned to D0.
- `y`, a 64-bit integer, is assigned to XD2, as above.
- `z`, a 64-bit float, is assigned to the first available floating-point register pair, XM0.
- `h`, a 32-bit pointer, is assigned to the first available pointer register, A4.

Variadic arguments (ellipsis "..."): Variadic arguments, such as those accepted by functions like printf(), are accessed via macros such as va_start, va_end, and va_arg. (See Variadic functions.) Every argument passed as part of a set of variadic arguments skips the register assignment phase. Instead it is assigned to the caller's argument block as if there were no valid registers remaining.

Other types: Types that do not fit into the above classifications–such as classes, structures, or union types–are assigned locations in the caller's argument block. Each is 8-byte-aligned, and values are copied into the block on call.

Running out of registers: Functions may have more arguments than there are available registers. In this case, such arguments are assign to locations in the caller's argument block. See Examples 3 and 4.

Example 3:

```
void baz(int *a, int *b, int *c, int *d, int *e, int *f, int *g)
```

- Arguments `a` through `f` are all 32-bit pointers and are assigned to A4-A9, respectively
- Argument `g` does not have a valid register in A4-A9. It is instead treated like a 32-bit integer and assigned to D0.

Example 4:

```
void fizz(long long x, long long y, long long z, long long h)
```

- Arguments `x`, `y`, and `z`, 64-bit integers, are assigned to XD0, XD2, and XD4, respectively
- Argument `h`, a 32-bit integer, has no valid register remaining. It is passed as the first 4 bytes on the caller's argument block.

Returned Values

Values returned from callee to caller follow a strict set of rules. A single unique location is assigned for the function's returned value and does not vary program-to-program.

- Pointer values are returned in A4.
- 32-bit integer values are returned in D0.
- 64-bit integer values are returned in XD0.
- 32-bit floating point values are returned in M0.
- 64-bit floating point values are returned in XM0.

Other types: Functions that return a type that does not fit into one of the above classifications–such as structures–allocate space on the caller's argument block into which the callee copies the returned value. In the following example, the returned structure is treated as a leading pointer argument in the function's argument list and is no longer treated as a returned value.

```
struct X foo(int a, char *b)
```

An equivalently handled call would be:

```
void foo(struct X *ptr, int a, char *b)
```

- The `ptr` argument, a 32-bit pointer, is assigned to the first available pointer register, A4.
- The `a` argument, a 32-bit integer, is assigned to D0.
- The `b` argument, a 32-bit pointer, is assigned to A5, not A4 as it would be normally.

The callee writes to this pointer argument, which the caller can read from on return to extract the value.

How a Function Makes a Call

A parent function) performs the following tasks when it calls another function (child function).

1. The call instruction pushes the 4-byte RPC onto the stack and increments the SP by 8 bytes, not by 4 bytes. This is so the stack remains 8-byte aligned. The RPC being saved is the return address of the caller.
2. The call instruction then sets RPC to the new return address (the return address of the callee).
3. The caller function is responsible for saving and restoring caller-saved registers whose values must be preserved across the call.
4. The caller copies arguments to registers and stack locations, as detailed in *Arguments*.
5. The caller issues a call instruction and transfers control to the callee.

How a Callee Function Responds

A function (callee function) must perform the following tasks in response to being called:

1. The callee function allocates memory for the local variables and argument block by adding a constant to the SP. This constant is the sum of:
   - The size of the register save area for callee-saved registers.
   - The size of the local variables whose lifetimes have not been optimized to be entirely in-register.
   - The maximum size of all the argument blocks for each function called by the callee.
   - Any extra size required to align to an 8-byte boundary.
2. The callee function saves each callee-saved register that it uses/modifies to the stack. This list can vary based on optimization.
3. The callee function executes the code for the function.
4. The callee function assigns the return value to a register or copies the value onto the stack, as detailed in *Returned Values*.
5. The callee function restores all registers that were saved in step 2.
6. The callee function deallocates the frame and argument block by subtracting the constant computed in step 1. Deallocation can occur either separate from the return instruction via subtraction of the constant from A15 or as part of the return instruction using the ADDR1 addressing mode. If the return instruction is the variant that does not adjust the SP after popping RPC, 8 is added to the constant computed in step 2.
7. The return instruction loads the program counter (PC) with the return address in the RPC register before loading the previous return address (saved to the stack by the call instruction) from the stack into the RPC register.

3.3.4 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.

3.3.5 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

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
  - Stack pointer (SP or A15)

  If the stack pointer 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 the stack pointer).

  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.

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.
- 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.

Likewise, to access a C/C++ function or object from assembly language, declare the C/C++ object with the .ref or .global directive in the assembly language module. This creates an undeclared external reference that the linker resolves.

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;

    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.

3.3.6 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.

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 by C/C++ functions using the __attribute__((interrupt())) function attribute. For information and examples, 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

To map 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 C29x interrupt vectors with branches to your interrupt routines. Follow these steps to use this file:

1. 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 a specific memory location.

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.

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. 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.

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.
- 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.
- 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*.

3.3.7 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 c29clang 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);
```

3.3.8 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.

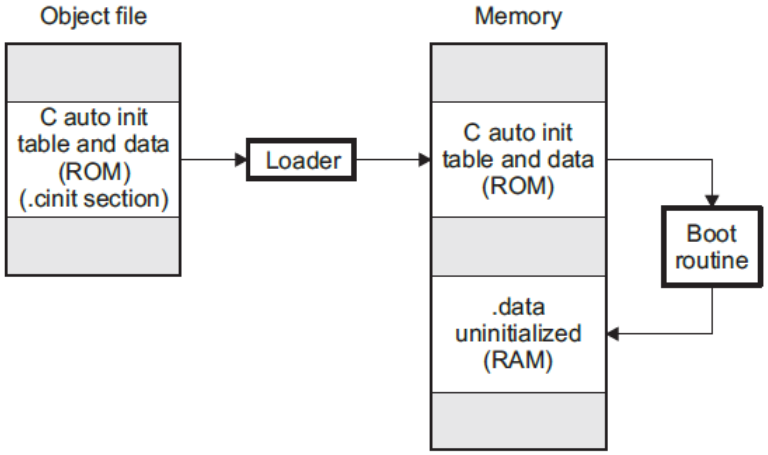


Figure 3.6: Autoinitialization at Run Time

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:

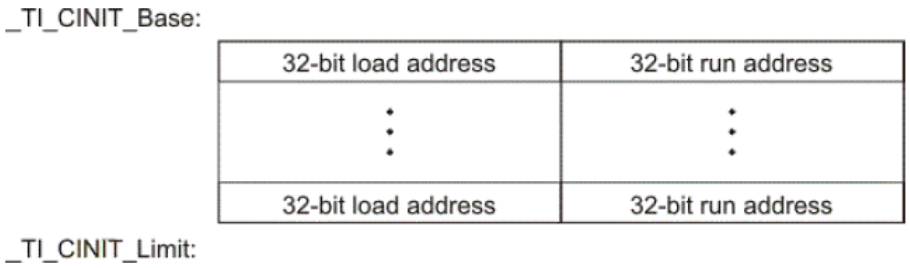


Figure 3.7: Autoinitialization Table Format

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:

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.

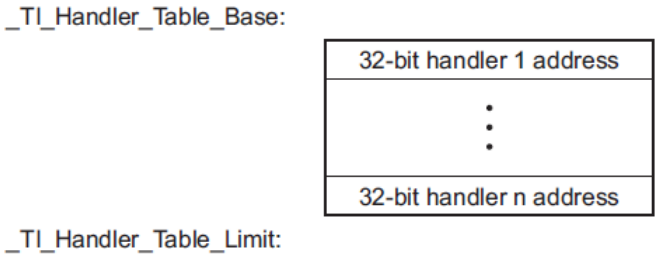


Figure 3.9: Handler Table Format

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

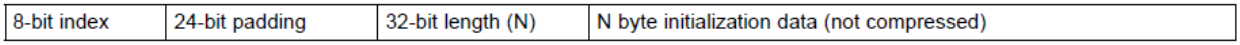


Figure 3.10: Encoded Data in Length Followed by Data Format

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

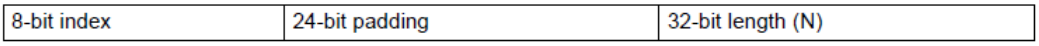


Figure 3.11: Encoded Data in Zero Initialization Format

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

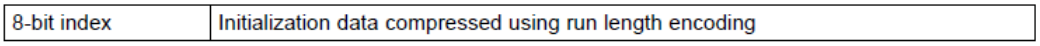


Figure 3.12: Encoded Data in RLE Format

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

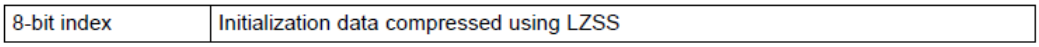


Figure 3.13: Encoded Data in LZSS Format

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;

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)
    {
```

(continues on next page)

(continued from previous page)

```
        /*----------------------------------------------------------
↪------*/
        /* 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.

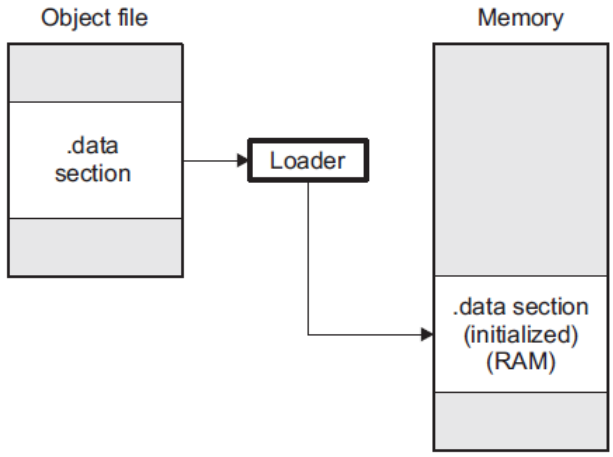


Figure 3.14: Initialization of Variables at Load Time

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.

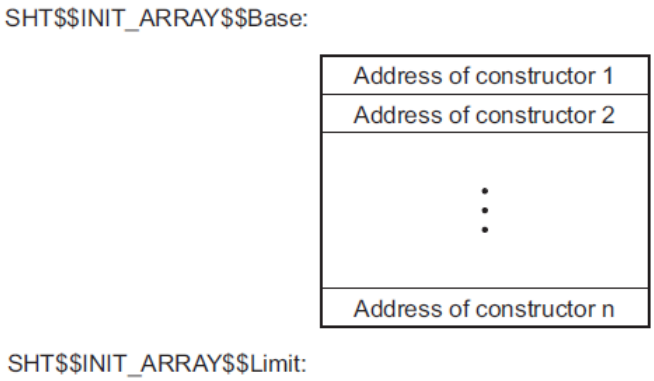


Figure 3.15: Global Constructor Address Table

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 C29x compiler installation includes pre-built run-time support (RTS) libraries that provide all the standard capabilities. Separate libraries are provided for each supported variant. 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.

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 C28x compiler, which defines time_t as an unsigned 32-bit value using a TI-defined epoch of January 1, 1900. You do not need to take any special steps or define any macros to use the time_t implementation, just be sure to #include time.h and use the standard C time functions, which automatically map to 64-bit implementations.

Leveraging the Unsigned 32-bit Representation of time_t

You may also activate the unsigned 32-bit representation of time_t by setting 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.

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.

The pre-built run-time support (RTS) libraries are as follows:

- `libc.a`: C Standard
- `libc++.a`: C++ Standard
- `libc++abi.a`: C++ Support
- `libsys.a`: Common system-level library, such as file IO and time.
- `libsysbm.a`: Common system-level library, such as file IO and time. You may provide your own copy of `libsysbm.a` to override the default IO and time functions. See *Contents of the libsysbm.a Library* for details.
- `libclang_rt.builtins.a`: Compiler support
- `libclang_rt.profile.a`: Instrumentation (code coverage)

The RTS libraries in the `<root>/lib` directory of the c29clang installation point to a library variant using the following path naming convention:

`/lib/c29`<*subtarget*> `-ti-none-eabi`<*variant*>

where the *subtargets* and *variants* are as follows:

- <*subtarget*>: `.c0` is currently the only subtarget.
- <*variant*>: * `/f64` is added to the directory path if the `-mfpu=f64` option is used, which causes native 32-bit and 64-bit floating-point hardware operations to be used. * `/fnone` is added to the directory path if the `-mfpu=none` option is used, which causes both 32-bit and 64-bit floating-point operations to be emulated in software. Object files compiled with `-mfpu=none` use different floating-point calling conventions and are not link compatible with object files compiled with `-mfpu=f32` or `-mfpu=f64`.

Variants of the `libclang_rt.builtins.a` and `libclang_rt.profile.a` libraries have `/lib/clang/<version>` added prior to this path, where <*version*> is currently "19".

For example, if you use no `-mfpu` option or the `-mfpu=f32` option, the RTS libraries in the following locations are used:

- `/lib/c29.c0-ti-none-eabi/c/libc.a`
- `/lib/c29.c0-ti-none-eabi/c/libsysbm.a`
- `/lib/c29.c0-ti-none-eabi/libc++.a`
- `/lib/c29.c0-ti-none-eabi/libc++abi.a`

- /lib/clang/<version>/lib/c29.c0-ti-none-eabi/libclang_rt.builtins.a
- /lib/clang/<version>/lib/c29.c0-ti-none-eabi/libclang_rt.profile.a

If you use the -mfpu=f64 option, the RTS libraries in the following locations are used:

- /lib/c29.c0-ti-none-eabi/f64/c/libc.a
- /lib/c29.c0-ti-none-eabi/f64/c/libsysbm.a
- /lib/c29.c0-ti-none-eabi/f64/libc++.a
- /lib/c29.c0-ti-none-eabi/f64/libc++abi.a
- /lib/clang/<version>/lib/c29.c0-ti-none-eabi/f64/libclang_rt.builtins.a
- /lib/clang/<version>/lib/c29.c0-ti-none-eabi/f64/libclang_rt.profile.a

If you use the -mfpu=none option for software emulated floating-point operations, the RTS libraries in the following locations are used:

- /lib/c29.c0-ti-none-eabi/fnone/c/libc.a
- /lib/c29.c0-ti-none-eabi/fnone/c/libsysbm.a
- /lib/c29.c0-ti-none-eabi/fnone/libc++.a
- /lib/c29.c0-ti-none-eabi/fnone/libc++abi.a
- /lib/clang/<version>/lib/c29.c0-ti-none-eabi/fnone/libclang_rt.builtins.a
- /lib/clang/<version>/lib/c29.c0-ti-none-eabi/fnone/libclang_rt.profile.a

Contents of the libsysbm.a Library

The files in the <root>/lib/src directory contain the source code for functions that may be overridden. By default, libsysbm.a contains functions from the following source files:

- add_device.c
- close.c
- host_device.c
- hostclock.c
- hostclose.c

- hostgetenv.c
- hostlseek.c
- hostopen.c
- hostread.c
- hostrename.c
- hosttime.c
- hostunlink.c
- hostwrite.c
- lseek.c
- open.c
- read.c
- remove.c
- remove_device.c
- rename.c
- time.c
- time64.c
- trgmsg.c
- unlink.c
- write.c

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:

```
#include <stdio.h>

void main()
{
    FILE *fid;

    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:

```
c29clang 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 */
O_TRUNC (0x0400) /* open with truncation */
O_BINARY (0x8000) /* open in binary mode */
```

- The *file_descriptor* is assigned by open to an opened file. The next available file descriptor is assigned to each new file opened.

Return Value The function returns one of the following values:

- *non-negative value* is file descriptor if successful
- -1 on failure

close – Close File for I/O

Syntax

```
#include <file.h>

int close (int file_descriptor);
```

Description The close function closes the file associated with *file_descriptor*. The *file_descriptor* is the number assigned by open to an opened file.

Return Value The return value is one of the following:

- 0 if successful
- -1 on failure

read – Read Characters from a File

Syntax

```
#include <file.h>

int read ( int      file_descriptor,
           char *   buffer,
           unsigned count );
```

Description The read function reads *count* characters into the *buffer* from the file associated with *file_descriptor*.

- The *file_descriptor* is the number assigned by open to an opened file.
- The *buffer* is where the read characters are placed.
- 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.

```
#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);

    /*-------------------------------------------------------
↪------------*/
```

(continues on next page)

(continued from previous page)

```
    /* 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 C29x 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

```
#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);
```

(continues on next page)

(continued from previous page)

```
main()
{
    FILE *fid;
    add_device("mydevice", _MSA, MYDEVICE_open, MYDEVICE_close,␣
 ↪MYDEVICE_read,
                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 Introduction to Object Modules

The compiler creates object modules from C/C++ code (with assembly code used internally), and the linker creates executable object files from object modules. These executable object files can be executed by an C29x device.

Object modules make modular programming easier because they encourage you to think in terms of *blocks* of code and data when you create an application. These blocks are known as sections. The linker provides directives that allow you to create and manipulate sections.

This chapter focuses on the concept and use of sections.

3.5.1 Object File Format Specifications

The object files created by the linker conform to the ELF (Executable and Linking Format) binary format, which is used by the Embedded Application Binary Interface (EABI).

The ELF object files generated by the linker conform to the December 17, 2003 snapshot of the System V generic ABI (or gABI). This specification is currently maintained by SCO.

3.5.2 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.

3.5.3 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 linker allows 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.
- **Uninitialized sections:** Reserve space in the memory map for uninitialized data. The .bss section is uninitialized.

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.

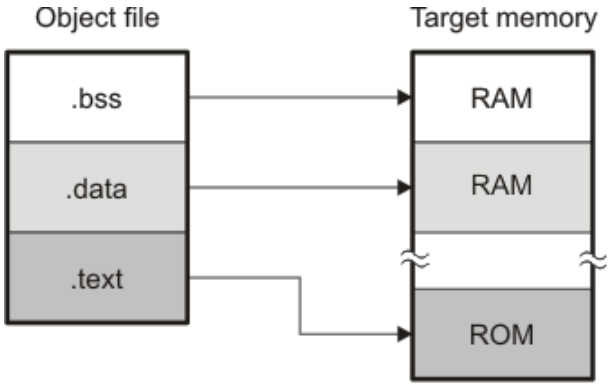


Figure 3.16: Sections in Object File and in Memory

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*.

3.5.4 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 C29x 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 C29x 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.

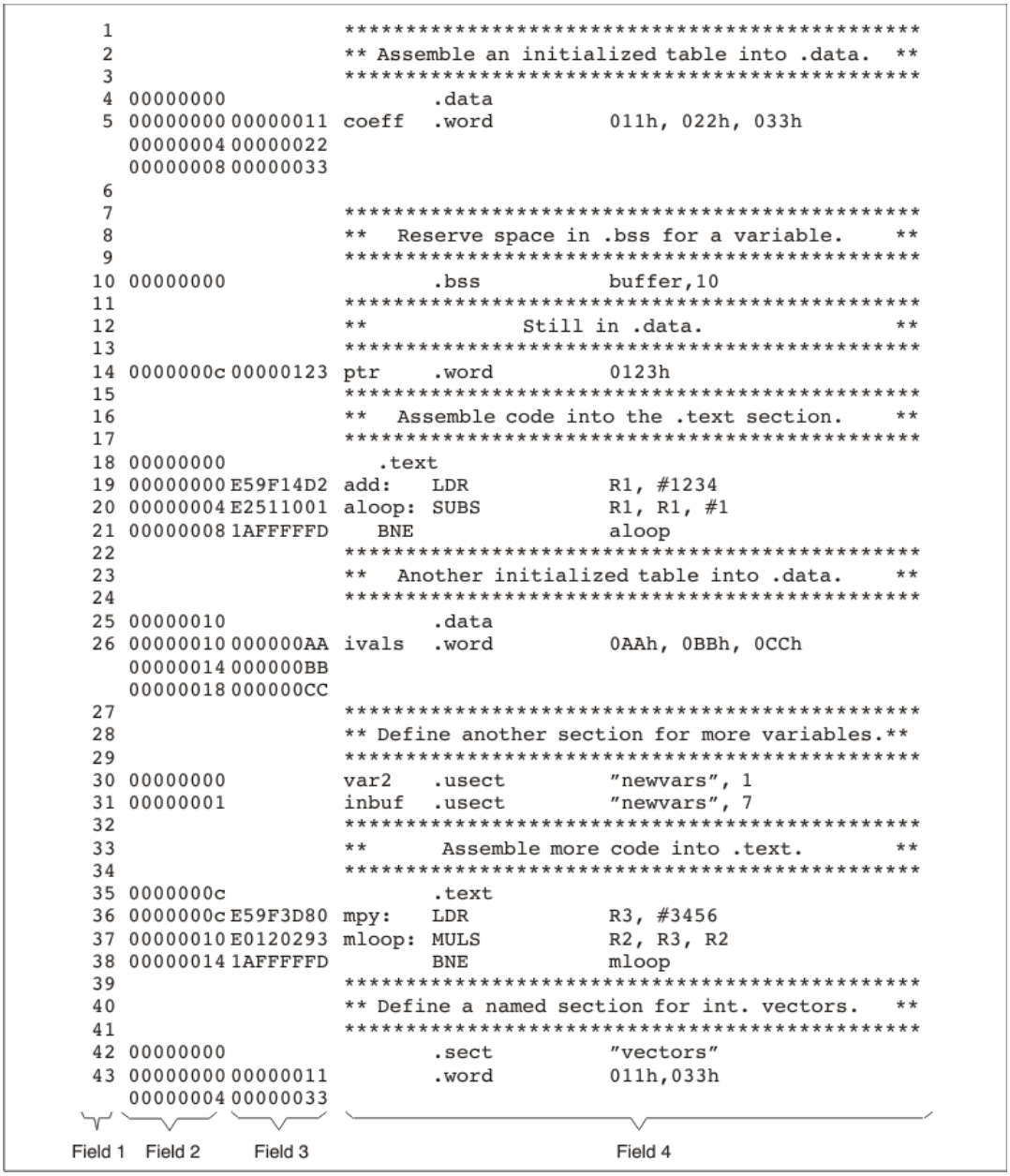


Figure 3.17: Using Sections Directives

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.
vectors	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.
newvars	is a user-named section created with the .usect directive; it reserves eight bytes in memory.

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.

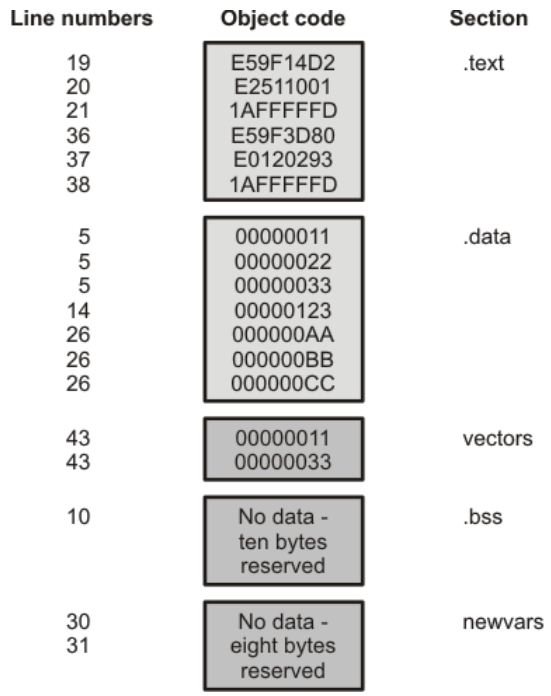


Figure 3.18: Object Code Generated by the Above Assembly Code

3.5.5 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 C29x.

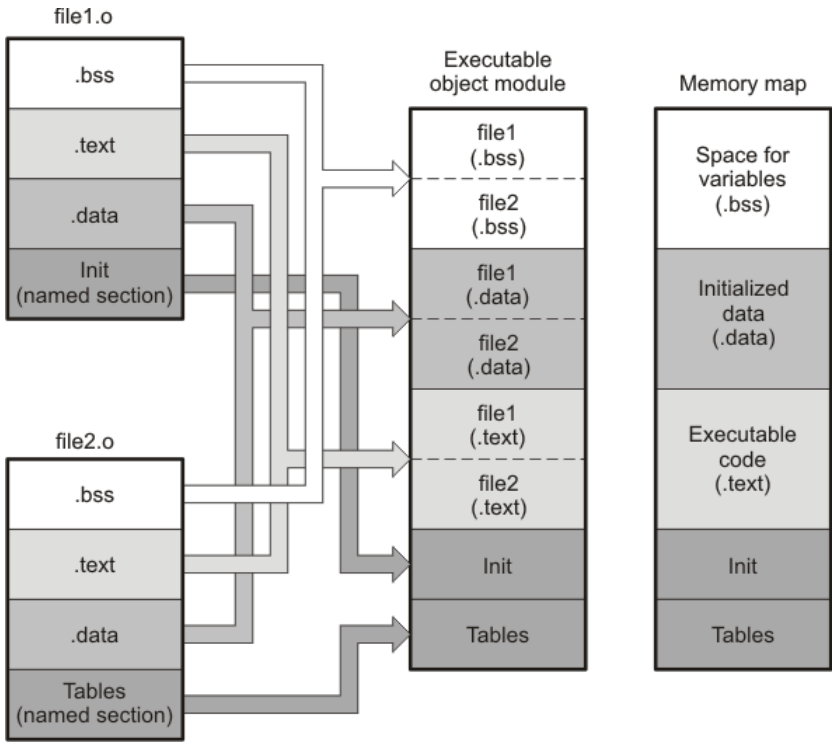


Figure 3.19: Combining Input Sections to Form an Executable Object Module

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 C29x.

3.5.6 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 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. A reference to a global symbol from any object file refers to the one and only allowed global definition of that symbol. (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.

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 the "weak" operator in the linker command file.

- **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 as described above.

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

Entries with global (external) binding are generated in the symbol table for the beginning of each section.

Entries with local binding are generated in the symbol table for each locally-available function.

For informational purposes, there are also entries in the symbol table for each symbol in a program.

3.5.7 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 a C29x 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`

3.5.8 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. For details, see *Loading*.

3.6 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.

3.6.1 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: Use the c29objcopy utility to assist with program loading. See *c29objcopy - Object Copying and Editing Tool* 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 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.

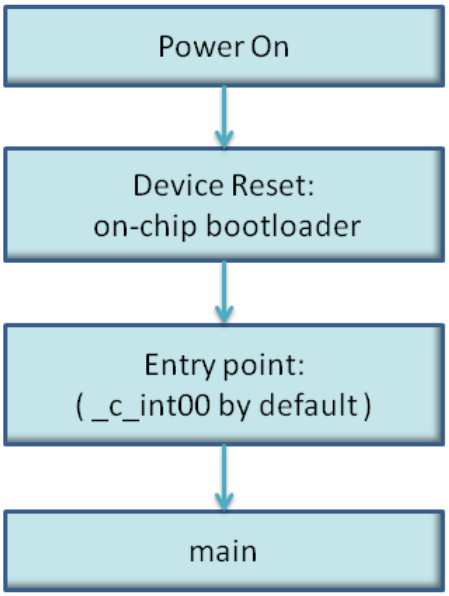


Figure 3.20: Bootloading Sequence (Simplified)

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.

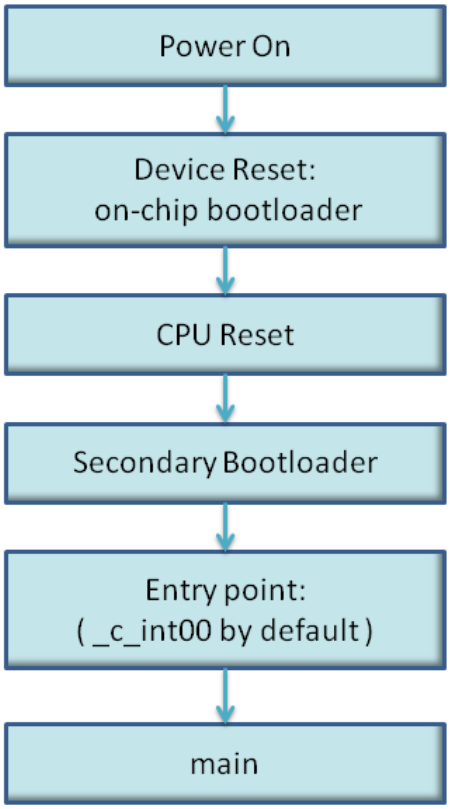


Figure 3.21: Bootloading Sequence with Secondary Bootloader

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.

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 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.

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:
```

(continues on next page)

(continued from previous page)

```
;
 ↪*****************************************************************************
;* 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
        ;
 ↪*************************************************************
        ; *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
```

(continues on next page)

(continued from previous page)

```
      ||     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
        ;
↪*************************************************************
      ||     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
```

(continues on next page)

(continued from previous page)

```
        ;
 ↪*****************************************************************
        ; *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:
```

(continues on next page)

(continued from previous page)

```
            b     .S2 b1
            nop   5
```

3.6.2 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.

3.6.3 Run-Time Initialization

After the load image is in place, the program can run. The subsections that follow describe bootstrap initialization of a C/C++ program.

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 c29clang compiler runs the linker. Therefore, either the `--rom_model` (`-c`) or `--ram_model` (`-cr`) option must be passed to the linker on the **c29clang** command line or in the linker command file. For example:

```
c29clang -mcpu=c29.c0 hello.c -o hello.out -Wl,-c,-llnk.cmd,-
↪mhello.map
```

If neither the `-c` or `-cr` option is specified to c29clang 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.

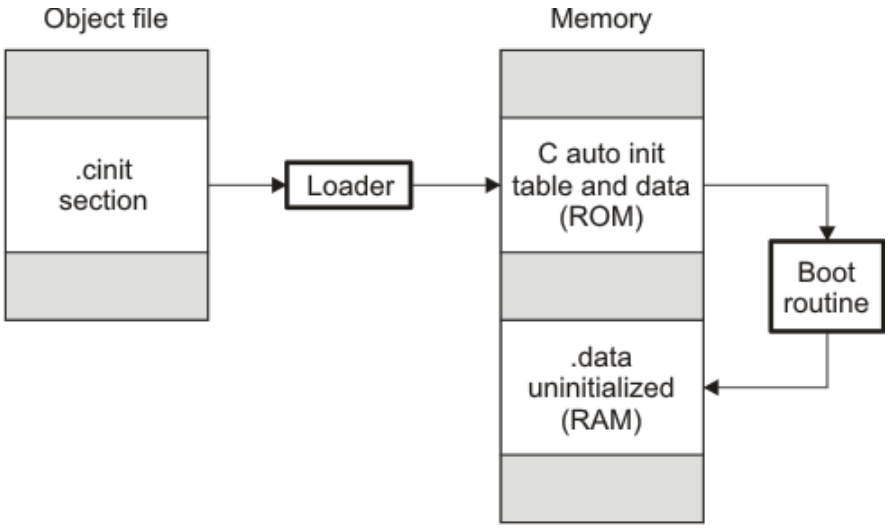


Figure 3.22: Autoinitialization at Run Time

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.

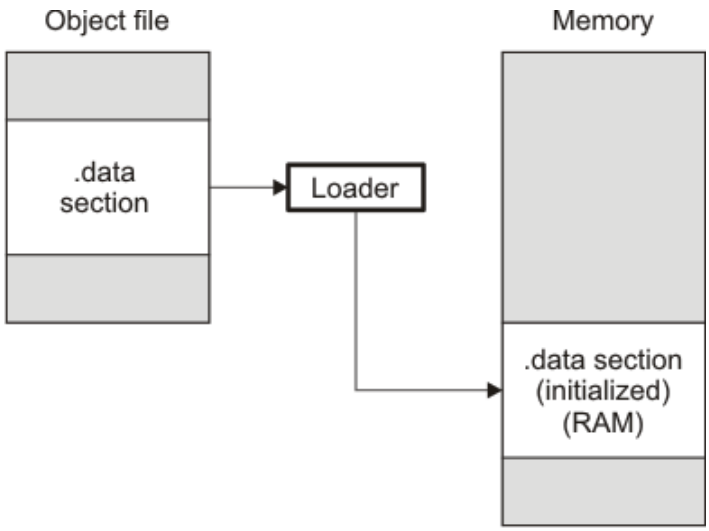


Figure 3.23: Initialization at Load Time

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.

3.6.4 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.

3.6.5 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*.

3.6.6 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*

3.7 Archiver Description

The archiver (**c29ar**) lets you combine several individual files into a single archive file. For example, you can use c29ar 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 C29x processor variants supported by the c29clang 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 **c29libinfo** 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 c29ar and c29libinfo utilities.

Contents:

3.7.1 c29ar - Archiver

The **c29ar** 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, c29ar 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 **c29ar** command is used to create an archive of LLVM bitcode files, the archive's symbol table will contain both native and bitcode symbols.

Usage

c29ar [-] <*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 c29ar 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 **c29ar** 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 c29ar prints the table of contents for the entire <*archive*>.

Operation-Specific Modifiers

- *v* - with this modifier applied to the *t* <*operation*>, c29ar 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*>, c29ar 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, c29ar 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 c29ar usage information to *stdout*.

V, --version
Display the version of the c29ar 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:

```
%> c29ar 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:

```
%> c29ar tv functions.lib
rw-r--r-- 0/0   1648 Dec 31 18:00 1969 sin.o
```

(continues on next page)

(continued from previous page)

```
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:

```
%> c29ar r functions.lib asin.o acos.o atan.o
```

Now verify the updated contents of functions.lib:

```
%> c29ar 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:

```
%> c29clang -mcpu=c29.c0 -c sin.c
%> c29ar r functions.lib sin.o
```

Exit Status

If c29ar execution is successful, it exits with a zero return code. If an error occurs during execution, c29ar exits with a non-zero return code.

3.7.2 c29libinfo - Library Information Archiver

The **c29libinfo** 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

c29libinfo [<*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_oper.c* and creating an object file library for each containing a single member, *lib_oper.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_oper.c* for compiling with either the `-mfpu=none` option or the `-mfpu=f64` option. Each version contains a definition of a global variable, *bit_oper*, that is initialized differently depending on the C29x processor option used to compile *lib_oper.c*.

```
%> c29clang <build option> -c lib_oper.c
%> c29ar r <object library name> lib_oper.o
```

results in the following list of libraries:

Target	bit_oper value	<build option>	<object library name>
Emulated 64-bit operations	0	-mfpu=none	c29_nofpu_def.a
Native 64-bit operations	1	-mfpu=f64	c29_fpu64_def.a

2 Creating an index library:

An index library called *def.a* can then be constructed with the following command:

```
%> c29libinfo -o def.a c29_nofpu_def.a c29_fpu64_def.a
```

The contents of the *def.a* index library can then be checked via the following c29ar command:

```
%> c29ar t def.a c29_nofpu_def.a.libinfo c29_fpu64_def.a.
↪libinfo __TI_$$LIBINFO
```

3 Using an index library in the link step:

A source file, *print_bit_oper.c*, containing a reference to the global variable *bit_oper* can then be linked with the index library *def.a*.

```
%> c29clang -mcpu=c29.c0 -mfpu=f64 print_bit_oper.c -o␣
↪print_lib.out -Wl,-llnk.cmd,def.a,-mprint_lib.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_bit_oper.c*. In the above case, the linker should pull in the *lib_oper.o* file from the *c29_fpu64_def.a* object file library

and the contents of the linker-generated *print_lib.map* file reveal that this is indeed the case:

```
****************************************************************************
C29 Clang Linker Unix v1.2.0
****************************************************************************
>> Linked Fri Mar 15 15:06:50 2024

OUTPUT FILE NAME:   <print_lib.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     c29_fpu64_def.
 ↪a : lib_oper.o (.data.bit_oper)
...
```

If a different `-mfpu` option had been specified on the above **c29clang** command line, then the linker would pull in the *lib_oper.o* from a different, appropriate, version of the object file library that matches the specified `-mfpu` option.

Exit Status

If c29libinfo execution is successful, it exits with a zero return code. If an error occurs during execution, c29libinfo exits with a non-zero return code.

3.8 Linker Description

The C29x 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:

3.8.1 Linker Overview

The C29x linker, c29lnk, is the proprietary linker provided by Texas Instruments.

This is the same proprietary linker use for the Texas Instruments C28x compiler. Linker commands files that were created for applications on C28x devices can generally be adapted easily for use on C29x devices.

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

3.8.2 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 C29x device.

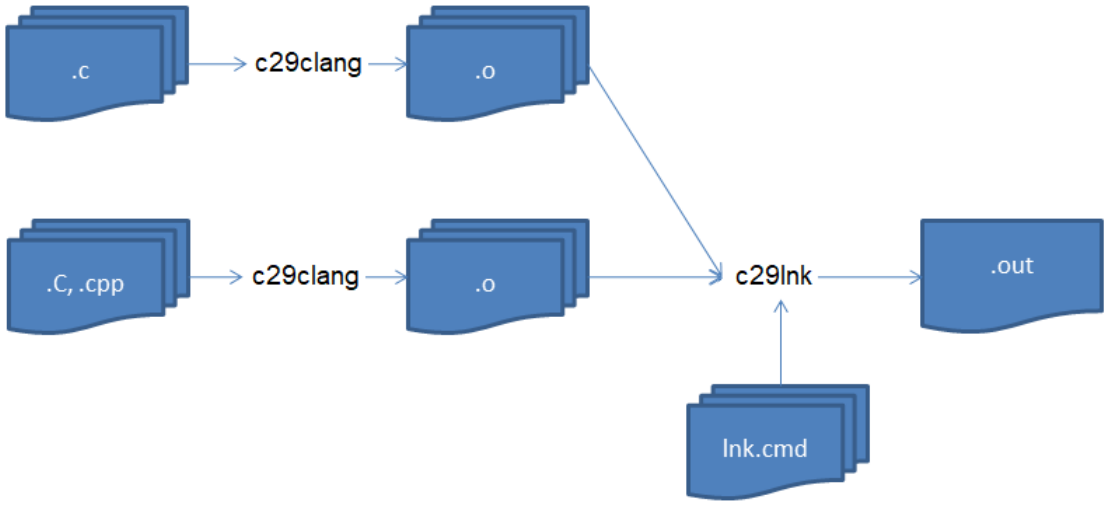


Figure 3.24: Linker's Role in Development Flow

3.8.3 Invoking the Linker

The default behavior of the c29clang compiler is to compile the specified C and/or C++ 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 c29clang 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 c29clang*
- *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:

```
c29clang [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 **c29clang** 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 c29clang runtime libraries to create an executable output file named **test.out**:

```
c29clang -mcpu=c29.c0 file1.c file2.o -o test.out -Wl,link_test.
 ↪cmd
```

Note that there is no mention of the c29clang runtime libraries on the **c29clang** command line or inside the **link_test.cmd** linker command file. When the linker is invoked from the **c29clang** command line, the c29clang compiler implicitly tells the linker where to find applicable runtime libraries like the C runtime library (libc.a). In the above **c29clang** 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 **c29clang** command, the output shows exactly how the linker (**c29lnk**) was invoked and with what options. For example, this command:

```
c29clang -mcpu=c29.c0 -v file1.c file2.o -o test.out -Wl,link_
 ↪test.cmd
```

shows the following with regards to how **c29lnk** is invoked by the c29clang compiler:

```
<install directory>/bin/c29lnk -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 c29clang 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 c29clang

When only object files are specified as input to the **c29clang** compiler command, the compiler passes those files to the linker along with any other specified options that are applicable to the link.

```
c29clang [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 **c29clang** command:

```
c29clang -mcpu=c29.c0 file1.o file2.o -o test.out -Wl,link_test.
 ↪cmd
```

invokes the linker as follows:

```
<install directory>/bin/c29lnk -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 c29clang 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 **c29clang** 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:

```
c29clang -mcpu=c29.c0 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:

```
c29clang -mcpu=c29.c0 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

  ...
}
```

```
%> c29lnk.cmd -mcpu=c29.c0 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
--------- ----------------
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
{
  ....
```

(continues on next page)

(continued from previous page)

```
  .mytext1     : { "lib-with-dashes.lib"(.text) } > 0x00010000
  .mytext2     : { "name-with-dashes.o"(.text)   } > 0x10000000

  ...
}
```

```
%> c29lnk.cmd -mcpu=c29.c0 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 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 *c29nm - Name Utility*. For information specifically about C++ symbol naming, see *c29dem - C++ Name Demangler Utility*. See *Using Linker Symbols in C/C++ Applications* for information about referring to linker symbols in C/C++ code.

3.8.4 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 **c29clang** 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=*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:

```
c29clang -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

- 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:

```
c29clang 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 C29x 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:

```
c29clang -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 *c29ofd - 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:

```
c29clang -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 **c29clang** 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)*.

--start-group / --end-group
These two linker options can be used together to identify a group of libraries passed to the linker. The options ensure that the resolution of backward references from those libraries is

constrained only to the libraries in that group. Thus, the options offer more granular control over the resolution of backward references than the –reread_libs linker option does. Nesting of groups is not supported. See *Invoking the Linker* for more about specifying runtime libraries.

For example:

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

For the runtime support libraries, the c29clang compiler implicitly adds the **--start-group/--end-group** options as shown above to the linker invocation.

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:

```
c29clang 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 **c29clang** 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:

```
%> c29clang -mcpu=c29.c0 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:

```
%> c29clang -mcpu=c29.c0 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 c29clang command lines.)

```
c29clang 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:

```
c29clang -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

% c29clang 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 **c29clang** 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:

```
c29clang --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.
- __TIME__ Expands to the compilation time in the form "*hh:mm:ss*"
- __C29__ Always defined to 1.
- __c29__ Always defined to 1.
- __C29_ARCH__ Identifies the C29x architecture version being compiled for. Currently, always defined to 0.
- __C29_C0__ Defined to 1 if the -mcpu=c29.c0 option was used when compiling.
- __C29_OPTF64__ Defined to 1 if the -mfpu option was set to f64 when compiling.
- __C29_OPTF32__ Defined to 1 if the -mfpu option was set to f32 or no fpu mode was given when compiling.

Diagnostic Options

The options listed in the subsections below control how the linker generates diagnostic messages. On the **c29clang** 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_references:format`=TEXT`
Specifies the format and file extension of the file generated when the –emit_references:file option is used. 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_references:format=TEXT
Specifies the format and file extension of the file generated when the –emit_references:file

option is used. Currently, the only format supported (and therefore, the default) is TEXT, which produces a simple ASCII text file.

--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 *c29nm - Name Utility*. For information specifically about C++ symbol naming, see *c29dem - 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 **c29clang** 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 Cor-*

recting 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:

```
c29clang -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:

```
c29clang -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:

```
c29clang -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:

```
c29clang 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():

```
c29clang 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:

```
c29clang 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
ltosymrefs	Pre-/Post-LTO symbol references	Off
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 c29clang 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>
  </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 **c29clang** 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.

```
c29clang -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 causes all symbols to 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.

```
c29clang -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:

```
c29clang -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 c29strip utility as described in *c29strip - 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 **c29clang** 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 is intended for debug purposes. It can help determine which definition of a symbol is pulled into a link from which object library, presuming that there are multiple object libraries that contain separate definitions of the same symbol name.

Consider a situation where *myfunc()* is defined in *a.lib* and is defined differently in *b.lib* and *myfunc()* is called from *main()*:

```
extern void myfunc();

int main() {
  myfunc();
  return 0;
}
```

If the application is compiled as follows:

```
%> tiarmclang -mcpu=cortex-m4 main.c a.lib b.lib -Wl,-c,--scan_
 ↪libraries
```

The linker will emit a warning that looks like this:

```
warning: Symbol "myfunc" (pulled from
   "C:\my\dev\space\a.lib<myfunc_a.o>) defined in two places:
   C:\my\dev\space\a.lib<myfunc_a.o>
   C:\my\dev\space\b.lib<myfunc_b.o>
```

The link will still succeed using the *myfunc()* definition from *a.lib*. The use of the *--scan_libraries* option provides immediate information about which definition of *myfunc()* is being linked into the application. If you had counted on the *b.lib* definition of *myfunc()* being linked into the application, perhaps because you weren't aware that there was a separate definition of *myfunc()* in *a.lib*, then the linker-generated warning would alert you that an unexpected version of *myfunc()* was included in the link.

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 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.

```
c29clang -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 **c29clang** 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)*.

--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)*.

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:

```
c29clang -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 c29clang compiler runs the linker. Therefore, either the `--rom_model` (`-c`) or `--ram_model` (`-cr`) option must be passed to the linker on the **c29clang** command line using the `-Wl` or `-Xlinker` option or in the linker command file. For example:

```
c29clang -mcpu=c29.c0 hello.c -o hello.out -Wl,-c,-llnk.cmd,-
↪mhello.map
```

If neither the `-c` or `-cr` option is specified to c29clang 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)*.

Link-Time Compression and Specialization Options

The options listed in the subsections below control how the linker handles optimization. On the **c29clang** 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 **c29clang** 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 **c29clang** 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.

3.8.5 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 **c29clang** command and follow it with the name of the command file:

```
c29clang 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:

```
c29clang link.cmd
```

You can place other parameters on the command line when you use a command file:

```
c29clang -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:

```
c29clang 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 SECTIONS 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 internally by the assembler 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:

```
%> c29clang <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 C29x 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 C29x 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 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 sequence 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*.

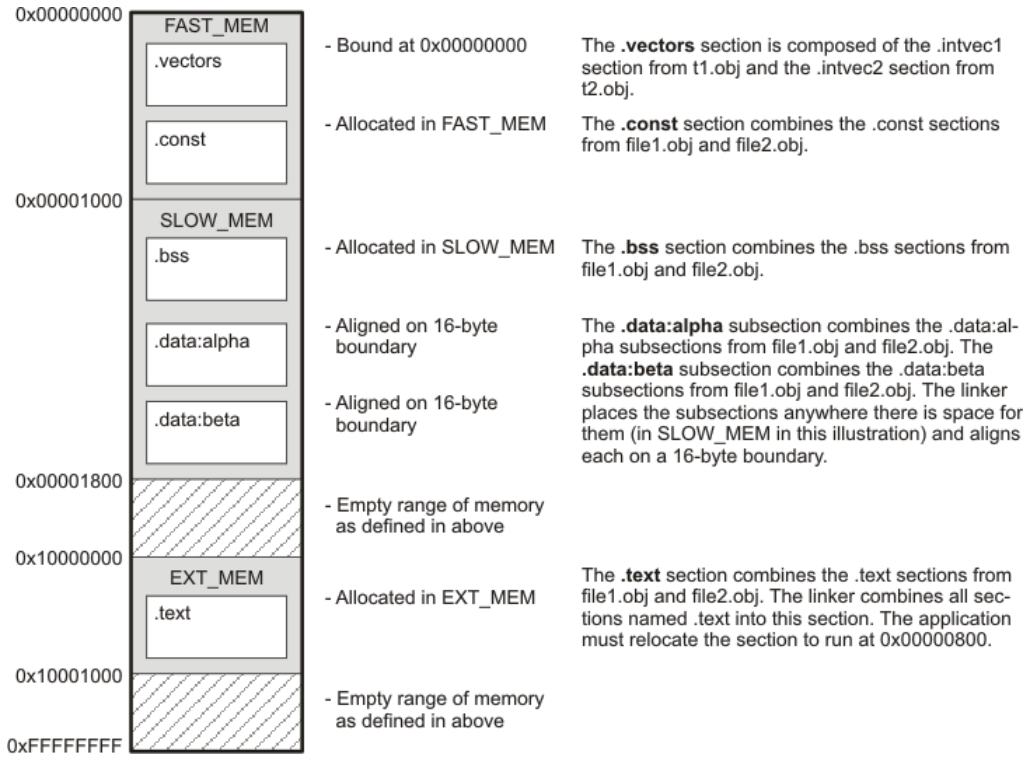


Figure 3.25: Output Sections Defined by the SECTIONS Directive

Variable-width `fill operator`

Defines the value used to fill uninitialized holes. Unlike, the *fill = value* mechanism described above, the variable-width fill operator syntax is similar to a function call that takes one or two arguments, the first being the fill value, and the optional second indicating the width of the fill value.

Syntax:

```
fill(value[, width])
```

where:

- *value* - is a required unsigned integer that can be represented in 32-bits or less. If *value* does not fit in the specified *width*, the linker will emit a warning and truncate the *value* to the indicated *width*.
- *width* - is an optional argument indicating the size, in bits, in which the *value* must fit. If the *width* argument is not specified, the linker assumes a width of 32-bits. If the *width* argument

is specified, it must be a power-of-two integer value, such that 8 <= *width* <= 32. Otherwise, the linker will emit a warning and assume a default width for the *value* of 32.

Consider a simple application defined as follows:

```
#include <stdio.h>

void func(void);

int main() {
  func();
  return 0;
}

__attribute__((section(".text:func")))
void func(void) {
  printf("Bring on the funk!\n");
}
```

The above application is compiled and linked using a linker command file that contains a **fill()** operator applied to the ".func" output section:

```
SECTIONS
{
  ...

  /* fill() operator uses 16-bit fill width */
  .func : { .+=0x0008; *(.text:func) } fill(0xfffe,0x10) > MEM

  ...
}
```

In this case, an 8-word gap was inserted at the front of the ".func" output section. The fill operator applied the ".func" output section indicates a value of 0xfffe with a size argument of 16 to instruct the linker to interpret the value as having a size of 16-bits. At link time, the 8-word gap will then be encoded with eight instances of the 16-bit value 0xfffe as shown in the following disassembly output:

```
%> c29clang -mcpu=c0 -c basic_fcn.c
%> c29clang -mcpu=c0 basic_fcn.o -o a.out -Wl,c29_16bit_fill.cmd,
 ↪-ma.map
%> c29objdump -d -S a.out
...
Disassembly of section .func:
```

(continues on next page)

(continued from previous page)

```
10002280 <.func>:
10002280:  fffe                <unknown>
10002282:  fffe                <unknown>
10002284:  fffe                <unknown>
10002286:  fffe                <unknown>
10002288:  fffe                <unknown>
1000228a:  fffe                <unknown>
1000228c:  fffe                <unknown>
1000228e:  fffe                <unknown>

10002290 <func>:
10002290:  0a44 0140 0000      MV      A4,#0x140
10002296:  0fa8 fded ffff      CALL    @printf
1000229c:  7a08                RET
...
```

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:

Binding	allocates a section at a specific address. `.text: load = 0x1000`
Named memory	allocates the section into a range defined in the MEMORY directive with the specified name (like SLOW_MEM) or attributes. `.text: load > SLOW_MEM`
Alignment	uses the align or palign keyword to specify the section must start on an address boundary. `.text: align = 0x100`
Blocking	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)
```

(continues on next page)

(continued from previous page)

```
                    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 addresses 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)
```

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
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)
```

(continues on next page)

(continued from previous page)

```
            *(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
{
    .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
```

(continues on next page)

(continued from previous page)

```
}

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.

Using a SECURE_GROUP to Place Protected Calls

The linker accepts the SECURE_GROUP attribute in the SECTIONS directive to define a section to contain protected calls. See *Security Support in the Linker* for details and examples.

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.

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)

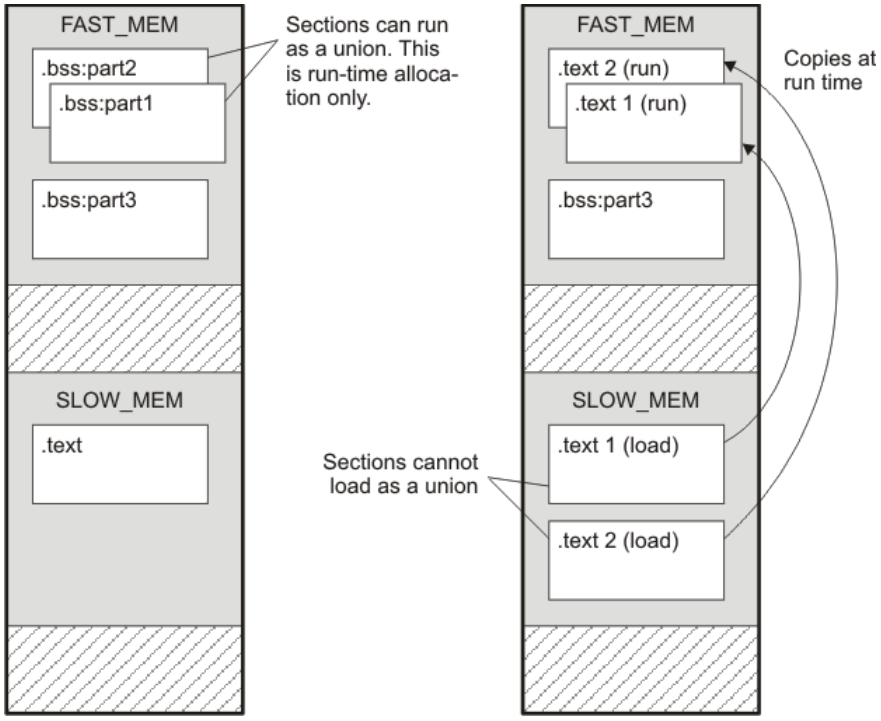


Figure 3.26: Memory Allocation for First (left) and Second (right) Examples

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.
- 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
```

(continues on next page)

(continued from previous page)

```
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 C29x 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 mod-

ule. 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:

input_range = *<range>*	The data memory range covered by this ECC data range. Required.
algorithm = *<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 can accomplish this by using 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 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.
- Numbers are recognized as decimal constants 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
Group 4	>>	Arithmetic right shift

continues on next page

Table 3.5 – continued from previous page

Precedence Group	Operator	Description
	<<	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 uninitialized 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.

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.

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   = .;
```

(continues on next page)

(continued from previous page)

```
    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 placed in them.

By default, the .bss section (see *Uninitialized Sections*) has no raw data (it is *uninitialized*). It occupies space in the memory map but has 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:

```
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 */
```

(continues on next page)

(continued from previous page)

```
    }
}
```

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:

   ```
   c29clang -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.

3.8.6 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, 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:

```
c29clang 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:

```
c29clang 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:

```
c29clang -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.

3.8.7 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.

3.8.8 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++ 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);
...
```

(continues on next page)

(continued from previous page)

```
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.

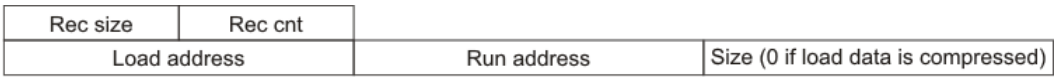


Figure 3.27: Compressed Copy Table Layout

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:

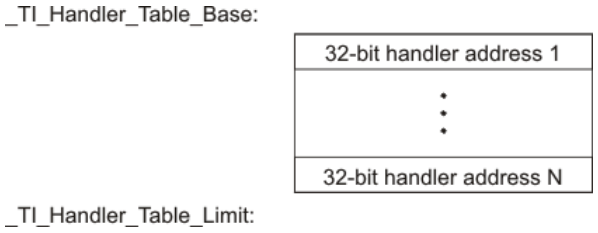


Figure 3.28: Handler Table

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. C29x 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 C29x 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 C29x 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)

(continued from previous 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.                             ⌴
 ↪             */
/
 ↪*************************************************************************
 ↪
```

(continues on next page)

(continued from previous page)

```
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;
```

(continues on next page)

(continued from previous page)

```
    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);
        }
    }
}
```

3.8.9 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)

(continued from previous 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:

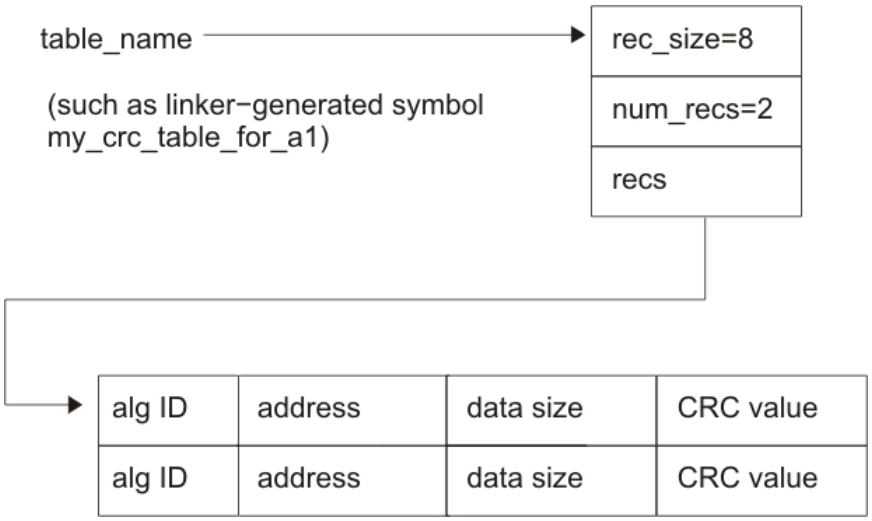


Figure 3.29: CRC Table Format

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         */
```

(continues on next page)

(continued from previous page)

```
/* 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                         */
```

(continues on next page)

(continued from previous page)

```
/* 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.  */
        /*************************************************/
```

(continues on next page)

(continued from previous page)

```
    }
    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 :...
```

(continues on next page)

(continued from previous page)

```
    } 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
{
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
    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:

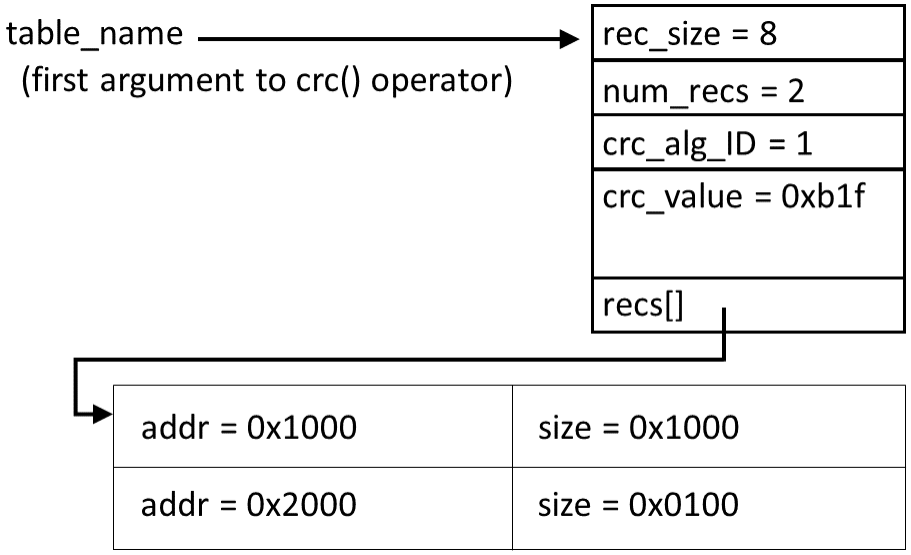


Figure 3.30: CRC Storage Format

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:

```
/
 ↪*************************************************************************
 ↪
/* MEMRANGE CRC Record Data Structure                                      ␣
 ↪                */
/
 ↪*************************************************************************
 ↪
typedef struct memrange_crc_record
{
    uint32_t          addr;           /* Starting address                  ␣
 ↪                */
    uint32_t          size;           /* size of data in 16-bit␣
 ↪addressable units */
} MEMRANGE_CRC_RECORD;

/
 ↪*************************************************************************
 ↪
/* MEMRANGE CRC Table Data Structure                                       ␣
 ↪                */
```

(continues on next page)

(continued from previous page)

```
/
 ↪*****************************************************************************
 ↪
typedef struct memrange_crc_table
{
    uint16_t              rec_size;       /* 16-bit addressable␣
 ↪units                */
    uint16_t              num_recs;       /* how many records are␣
 ↪in the table      */
    uint16_t              crc_alg_ID;     /* CRC algorithm ID      ␣
 ↪                  */
    uint16_t              padding;        /* padding               ␣
 ↪                  */
    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.

3.8.10 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.

```
c29clang -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.

```
c29clang -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.

```
c29clang -Wl,--map_file=final.map,--output_file=final.out␣
 ↪tempout1.out tempout2.out
```

3.8.11 Linking C/C++ Code

The C/C++ compiler produces object files that can be linked. For example, a C program consisting of modules prog1, prog2, etc., can be linked to produce an executable file called prog.out:

```
c29clang -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 C29x 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.

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 *c29ofd - 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.

3.8.12 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 */
```

(continues on next page)

(continued from previous page)

```
--output_file=demo.out /* Name the output file */
--map_file=demo.map /* Create an output map file */
/
 ↪*************************************************************************
 ↪
/*** Specify the Input Files ***/
/
 ↪*************************************************************************
 ↪
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)
```

(continues on next page)

(continued from previous page)

```
        . = 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 */
}
/
↪*************************************************************************
↪
/*** End of Command File ***/
/
↪*************************************************************************
↪
```

Invoke the linker by entering the following command:

```
c29clang demo.cmd
```

This creates the following map file and an output file called demo.out that can be run on an C29x 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)
```

(continues on next page)

(continued from previous page)

```
                  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]
```

3.8.13 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>
```

(continues on next page)

(continued from previous page)

```
        <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>
```

(continues on next page)

(continued from previous page)

```
    <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>
        <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>
```

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>
```

3.9 Code Coverage

Contents:

3.9.1 Source-Based Code Coverage in c29clang

The TI C29x Clang Compiler Tools (c29clang) 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 c29clang 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 c29clang 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 c29clang.

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 c29clang 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 c29clang compiler, which is used to produce counter instrumentation, the tools used to produce and visualize code coverage data are **c29profdata** and **c29cov**.

Useful Coverage Profile Merging Options (c29profdata)

```
USAGE: c29profdata 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 (c29cov)

```
USAGE: c29cov {subcommand} [OPTION]... --sources [SOURCES]...
```

Subcommands

- **export** - Export instrprof file to structured format either as text (JSON) or as LCOV.
  - JSON: c29cov export --format=text
  - LCOV: c29cov export --format=lcov
  - CSV: c29cov 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.
- **--sources [SOURCES]** Show code coverage only for specified source files.

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 "c29cov 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.
- **--show-functions** - Show coverage summaries for each function.

Generating Instrumented Binaries

Source code must be built using **c29clang** 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:

```
c29clang -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 c29clang, 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:

```
var scriptEnv = Packages.com.ti.ccstudio.scripting.environment.
↪ScriptingEnvironment.instance();
var server = scriptEnv.getServer("DebugServer.1");
var session = server.openSession("Texas Instruments USB␣
↪DebugProbe_0");

var cntStart = session.symbol.getAddress("__start___llvm_prf_cnts
↪");
var cntStop = session.symbol.getAddress("__stop___llvm_prf_cnts
↪");

var cntContent = session.memory.readData(0, cntStart, 8, cntStop␣
↪- cntStart);

var executable = session.symbol.getSymbolFileName();
var outFile = new Packages.java.io.RandomAccessFile(executable +
↪".cnt" , "rw");
```

(continues on next page)

(continued from previous page)

```
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**.

```
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 **c29profdata** 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.

```
c29profdata merge -sparse -obj-file=app1.out app1.profcnts -o␣
↪app1.profdata
c29profdata merge -sparse -obj-file=app2.out app2.profcnts -o␣
↪app2.profdata
c29profdata 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*.

```
c29profdata 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 c29cov 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.

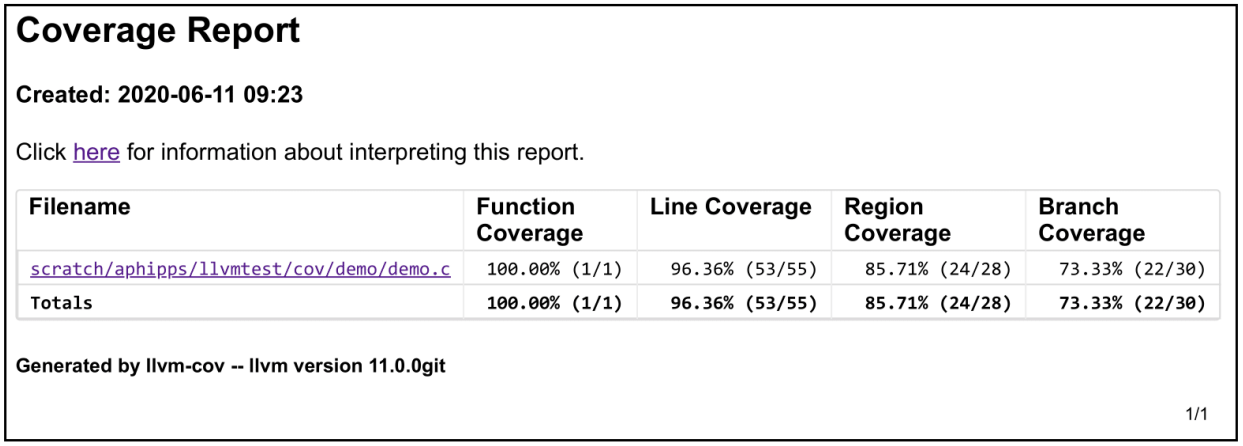
HTML Format

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.

If you need to focus on code coverage for specific source files, you may list the source files following the *--sources* option.

The following example visualizes coverage in HTML with macros and templates expanded; it also includes detailed branch coverage in terms of execution count.

```
c29cov show --format=html --show-expansions --show-
↪instantiations --show-branches=count
  --object=./app1.out --object=./app2.out --object=./app3.out -
↪instr-profile=app-merged.profdata --sources demo.c
  --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]
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++)

7/6/2020

```
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;
```

2/3

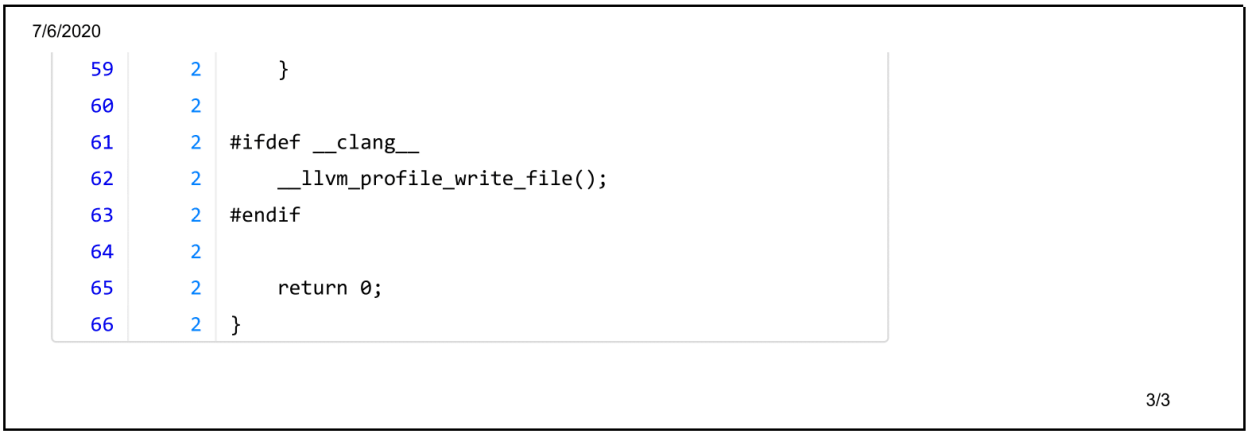


Figure 3.31: Code Coverage HTML Output Format

When generating HTML output with *--show-mcdc-summary*, the summary coverage report includes an additional column with the MC/DC coverage data.

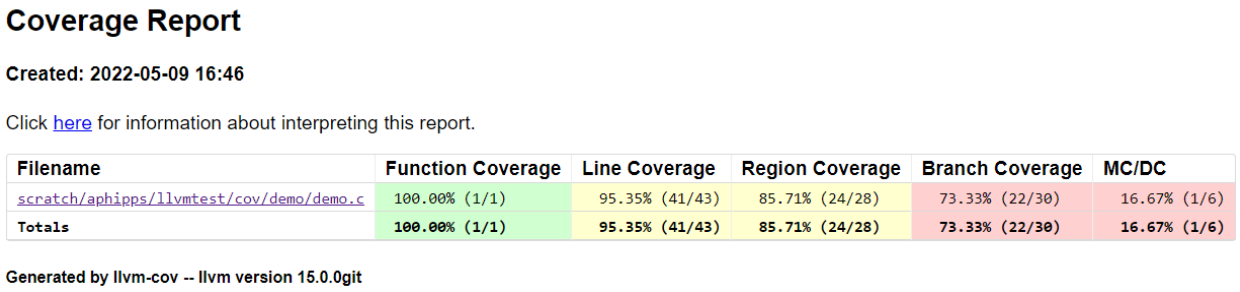


Figure 3.32: Code Coverage HTML Summary Report

Text Format

When generating Text output, the summary coverage report is generated using a separate **c29cov report** option. For example, to view the source-based coverage view:

```
c29cov 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)

(continued from previous 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)
------------------
```

(continues on next page)

(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)

(continued from previous 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:

```
c29cov 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:

```
c29cov 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.

```
c29cov export --format=csv --object=./multidemo.out -instr-
 ↪profile=default.profdata --sources 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.

Important Considerations for MC/DC

- 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)
|
|  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)
```

(continues on next page)

(continued from previous page)

```
|  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 c29clang 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)
|   |   |   |   |   |   ------------------
```

(continues on next page)

(continued from previous page)

```
|   |   |   |   |   |   |  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]
```

(continues on next page)

(continued from previous page)

```
------------------
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 c29clang 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 c29profdata tool. When the counter data is merged, c29profdata 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 c29cov 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 c29cov 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 c29cov 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 c29cov 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 c29cov 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 c29cov on a different machine where you have the same files on a different path.

- Source Filtering
  - The source filtering facility implemented by c29cov 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 (c29cov **--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.

3.9.2 c29profdata - Profile Data Tool

The **c29profdata** tool can be used to work with profile data files.

Usage

c29profdata *command* [*options*] <*filenames*>

- **c29profdata** - Command to invoke the profile data tool.
- *command* - One of the available c29profdata 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 **c29profdata merge** command takes several profile data files generated by c29clang 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, c29profdata 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 **c29profdata 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 **c29profdata 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

c29profdata 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.

3.9.3 c29cov - Emit Coverage Information

The **c29cov** tool is used to show code coverage information for programs that are instrumented to emit profile data.

Usage

c29cov *command [arguments]*

- **c29cov** - Command used to invoke the code coverage display tool.
- *command* - One of the available c29cov modes of operation: *show*, *report*, or *export*
- *arguments*

Commands

show

c29cov show [*options*] -instr-profile *profile binary* --sources [*sources*]

The **c29cov 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* using the *--sources* option.

A *binary* can be an executable, an object file, or an archive.

To use **c29cov show**, you need a program that is compiled with instrumentation to emit profile and coverage data. To build such a program with **c29clang** use the *-fprofile-instr-generate* and *-fcoverage-mapping* flags. If linking with using the **c29clang** 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 **c29cov 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 **c29profdata 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, **c29cov** 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 **c29cov** on a different machine where you have the same files on a different path.

report

c29cov report [*options*] -instr-profile *profile binary* --sources [*sources*]

The **c29cov 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* using the *--sources* option.

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

c29cov export [*options*] -instr-profile *profile binary* --sources [*sources*]

The **c29cov 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* using the *--sources* option.

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 **c29cov 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, c29cov auto-detects an appropriate number of threads to use. This is the default.

Exit Status

c29cov 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.

3.9.4 Code Coverage for Functional Safety

The TI C29x 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 c29clang*.

For more information:

- How to apply the TI Compiler Qualification Kit for functional safety development (TI App Note)

3.10 Compiler Security

Contents:

3.10.1 Stack Smashing Detection

- *Introduction*
- *Stack Smashing Detection Options*
  - *Enabling Stack Smashing Detection*
  - *Stack Smashing Detection Example*

Introduction

The TI C29x Clang Compiler Tools (c29clang) 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.

Enabling Stack Smashing Detection

To enable stack smashing detection in your application, you need to provide definitions of:

__stack_chk_fail() - This function is called from an instrumented function when a check against the stack guard value, *__stack_chk_guard*, fails. A simple definition of this function might look like this:

```
void __stack_chk_fail(void) {
  printf("__stack_chk_guard has been corrupted\n");
```

(continues on next page)

(continued from previous page)

```
  exit(0);
}
```

__stack_chk_guard - This is a globally visible symbol whose value can be copied into a location at the boundary of a function's allocated stack on entry into the function, and loaded just prior to function exit to perform a check that the local copy of the *__stack_chk_guard* value has not been overwritten. A simple definition of this symbol might look like this:

```
unsigned long __stack_chk_guard = 0xbadeebad;
```

You can then compile a file containing both of these definitions to produce an object file that can be linked into an application that is instrumented for stack smashing detection.

Stack Smashing Detection Example

Here is a simple example to summarize and demonstrate how the stack smashing detection capability can be used:

- The first source file presents the definitions of *__stack_chk_fail()* and *__stack_chk_guard* (stack_check.c):

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

void  __stack_chk_fail(void);
unsigned long __stack_chk_guard = 0xbadeebad;

void __stack_chk_fail(void) {
  printf("ERROR: __stack_chk_guard has been corrupted\n");
  eixit(0);
}
```

- The second source file presents a use case where a function, *foo*, writes past the end of a local buffer (stack_smash.c):

```
#include <string.h>

void foo(void);

int main() {
  foo();
  return 0;
}
```

(continues on next page)

(continued from previous page)

```
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*:

```
%> c29clang -mcpu=c29.c0 -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:

```
%> c29clang -mcpu=c29.c0 -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
```

3.10.2 C11 Secure Functions in C Runtime Support Library

- *C11 Secure Function Constraint Violations*
- *Setting Up a Constraint Violation Handler Function*
- *Enabling Use of Secure Functions via __STDC_WANT_LIB_EXT1__ Definition*
- *Example*
- *The Secure Functions*

The TI C29x Clang Compiler Tools (c29clang) provides an implementation of a subset of the "secure" functions that were introduced as optional extensions to the C11 language standard. You can find a full description of the C11 secure functions in Annex K of a recent C Language Standard.

The following C11 secure functions **are** supported in the c29clang compiler tools (annotated with the C11 language standard section number where function is described):

- abort_handler_s() - K.3.6.1.2
- gets_s() - K.3.5.4.1
- ignore_handler_s() - K.3.6.1.3

- memcpy_s() - K.3.7.1.1
- memmove_s() - K.3.7.1.2
- memset_s() - K.3.7.4.1
- set_constraint_handler_s() - K.3.6.1.1
- strcat_s() - K.3.7.2.1
- strcpy_s() - K.3.7.1.3
- strncat_s() - K.3.7.2.2
- strncpy_s() - K.3.7.1.4
- strnlen_s() - K.3.7.4.4

The Annex K C11 secure functions that are **not** supported in the c29clang sre listed in alphabetical order below (annotated with C11 language standard section number where function is described):

- asctime_h() - K.3.8.2.1
- bsearch_s() - K.3.6.3.1
- ctime_s() - K.3.8.2.2
- fopen_s() - K.3.5.2.1
- fprintf_s() - K.3.5.3.1
- freopen_s() - K.3.5.2.2
- fscanf_s() - K.3.5.3.2
- fwprintf_s() - K.3.9.1.1
- fwscanf_s() - K.3.9.1.2
- getenv_s() - K.3.6.2.1
- gmtime_s() - K.3.8.2.3
- localtime_s() - K.3.8.2.4
- mbrsrtowcs_s() - K.3.9.3.2.1
- mbstowcs_s() - K.3.6.5.1
- printf_s() - K.3.5.3.3
- qsort_s() - K.3.6.3.2
- scanf_s() - K.3.5.3.4
- snprintf_s() - K.3.5.3.5
- snwprintf_s() - K.3.9.1.3

- sprintf_s() - K.3.5.3.6
- sscanf_s() - K.3.5.3.7
- strerror_s() - K.3.7.4.2
- strerrorlen_s() - K.3.7.4.3
- strtok_s() - K.3.7.3.1
- swprintf_s() - K.3.9.1.4
- swscanf_s() - K.3.9.1.5
- tmpfile_s() - K.3.5.1.1
- tmpnam_s() - K.3.5.1.2
- vfprintf_s() - K.3.5.3.8
- vfscanf_s() - K.3.5.3.9
- vfwprintf_s() - K.3.9.1.6
- vfwscanf_s() - K.3.9.1.7
- vprintf_s() - K.3.5.3.10
- vscanf_s() - K.3.5.3.11
- vsnprintf_s() - K.3.5.3.12
- vsnwprintf_s() - K.3.9.1.8
- vsprintf_s() - K.3.5.3.13
- vsscanf_s() - K.3.5.3.14
- vswprintf_s() - K.3.9.1.9
- vswscanf_s() - K.3.9.1.10
- vwprintf_s() - K.3.9.1.11
- vwscanf_s() - K.3.9.1.12
- wcrtomb_s() - K.3.9.3.1.1
- wcscat_s() - K.3.9.2.2.1
- wcscpy_s() - K.3.9.2.1.1
- wcsncat_s() - K.3.9.2.2.2
- wcsncpy_s() - K.3.9.2.1.2
- wcsnlen_s() - K.3.9.2.4.1
- wcsrtombs_s() - K.3.9.3.2.2

- wcstok_s() - K.3.9.2.3.1
- wcstombs_s() - K.3.6.5.2
- wctomb_s() - K.3.6.4.1
- wmemcpy_s() - K.3.9.2.1.3
- wmemmove_s() - K.3.9.2.1.4
- wprintf_s() - K.3.9.1.13
- wscanf_s() - K.3.9.1.14

C11 Secure Function Constraint Violations

The intent behind the "Bounds-checking interfaces" described in Annex K of the C language standard is to provide a means for a developer to detect, at run-time, unintended behavior in C runtime library functions that write to memory.

For example, consider the memcpy() C runtime library function:

```
void *memcpy(void *dest, const void *src, size_t count);
```

A typical C runtime library implementation of the memcpy() function is optimized to execute as efficiently as possible. Most likely, the memcpy() implementation does not check the validity of the *dest* and *src* arguments before attempting the copy. There is also no information available to memcpy() about the size of the buffer pointed to by *dest*, and therefore, there is no way to check whether the copy may write past the end of the *dest* buffer. Consequently, an issue like writing past the end of the buffer that the *dest* points to goes undetected and could lead to run-time behavior that is difficult to debug.

The c29clang compiler tools provide the C11 secure version of the memcpy() function, memcpy_s(),

```
errno_t memcpy_s(void *dest, rsize_t destsz, const void *src,␣
↪rsize_t count);
```

which is implemented as follows:

```
/
↪*****************************************************************************
↪
/* memcpy_s.c                                                                 ␣
↪            */
/*                                                                            ␣
↪            */
/* Copyright (c) 2024 Texas Instruments Incorporated                          ␣
↪            */
```

(continues on next page)

(continued from previous page)

```
/* http://www.ti.com/                                                    ␣
 ↪            */
/*                                                                       ␣
 ↪            */
/*  Redistribution and  use in source  and binary forms, with ␣
 ↪or without   */
/*  modification,  are permitted provided  that the  following␣
 ↪conditions   */
/*  are met:                                                             ␣
 ↪            */
/*                                                                       ␣
 ↪            */
/*     Redistributions  of source  code must  retain the  above␣
 ↪copyright   */
/*     notice, this list of conditions and the following␣
 ↪disclaimer.        */
/*                                                                       ␣
 ↪            */
/*     Redistributions in binary form  must reproduce the above␣
 ↪copyright   */
/*     notice, this  list of conditions  and the following ␣
 ↪disclaimer in   */
/*     the  documentation  and/or   other  materials  provided ␣
 ↪with  the   */
/*     distribution.                                                     ␣
 ↪            */
/*                                                                       ␣
 ↪            */
/*     Neither the  name of Texas Instruments Incorporated  nor␣
 ↪the names   */
/*     of its  contributors may  be used to  endorse or  promote␣
 ↪products   */
/*     derived  from   this  software  without   specific  prior␣
 ↪ written   */
/*     permission.                                                       ␣
 ↪            */
/*                                                                       ␣
 ↪            */
/*  THIS SOFTWARE  IS PROVIDED BY THE COPYRIGHT  HOLDERS AND␣
 ↪CONTRIBUTORS   */
/*  "AS IS"  AND ANY  EXPRESS OR IMPLIED  WARRANTIES, INCLUDING,␣
 ↪ BUT NOT   */
```

(continues on next page)

(continued from previous page)

```
/*  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND␣
 ↪FITNESS FOR   */
/*  A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE␣
 ↪COPYRIGHT   */
/*  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,␣
 ↪INCIDENTAL,   */
/*  SPECIAL,  EXEMPLARY,  OR CONSEQUENTIAL  DAMAGES  (INCLUDING,␣
 ↪BUT  NOT   */
/*  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;␣
 ↪LOSS OF USE,   */
/*  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED␣
 ↪AND ON ANY   */
/*  THEORY OF  LIABILITY, WHETHER IN CONTRACT, STRICT  LIABILITY,
 ↪ OR TORT   */
/*  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT␣
 ↪OF THE USE   */
/*  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH␣
 ↪DAMAGE.    */
/*                                                                ␣
 ↪           */
/
 ↪*****************************************************************************
 ↪

#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <errno.h>
#include "_c11_secure_private.h"

/
 ↪*****************************************************************************
 ↪
/* MEMCPY_S()                                                     ␣
 ↪           */
/*    Read count characters from a source buffer and write them␣
 ↪to a        */
/*    destination buffer. The C11 secure version of memcpy will␣
 ↪enforce     */
/*    security constraints as prescribed by the C11 standard.␣
 ↪Details of    */
/*    the constraints to be enforced are described in the␣
 ↪comments below.   */
```

(continues on next page)

(continued from previous page)

```
/
 ↪*****************************************************************************
 ↪
#define MAX_EMSG_SZ         100
errno_t
memcpy_s(void * __restrict dest, rsize_t destsz,
         const void * __restrict src, rsize_t count)
{
  char emsg[MAX_EMSG_SZ] = "memcpy_s : ";

  /*-----------------------------------------------------------
 ↪-----------*/
  /* Destination buffer pointer cannot be NULL.
 ↪            */
  /*-----------------------------------------------------------
 ↪-----------*/
  if (dest == NULL)
    strcat(emsg, "dest is NULL");

  /*-----------------------------------------------------------
 ↪-----------*/
  /* Source buffer pointer cannot be NULL.
 ↪            */
  /*-----------------------------------------------------------
 ↪-----------*/
  else if (src == NULL)
    strcat(emsg, "src is NULL");

  /*-----------------------------------------------------------
 ↪-----------*/
  /* Indicated destination buffer size cannot be > RSIZE_MAX.
 ↪            */
  /*-----------------------------------------------------------
 ↪-----------*/
  else if (destsz > RSIZE_MAX)
    strcat(emsg, "destsz > RSIZE_MAX");

  /*-----------------------------------------------------------
 ↪-----------*/
  /* Indicated count cannot be > RSIZE_MAX.
 ↪            */
  /*-----------------------------------------------------------
 ↪-----------*/
```

(continues on next page)

(continued from previous page)

```
  else if (count > RSIZE_MAX)
    strcat(emsg, "count > RSIZE_MAX");

  /*------------------------------------------------------------
↪-----------*/
  /* Indicated count cannot be > indicated destination buffer␣
↪size.          */
  /*------------------------------------------------------------
↪-----------*/
  else if (count > destsz)
    strcat(emsg, "count > destsz");

  /*------------------------------------------------------------
↪-----------*/
  /* Source and destination buffers may not overlap.          ␣
↪            */
  /*------------------------------------------------------------
↪-----------*/
  else if ((((unsigned char *)src >= (unsigned char *)dest) &&
            ((unsigned char *)src < ((unsigned char *)dest +␣
↪destsz))) ||
           (((unsigned char *)src <= (unsigned char *)dest) &&
            ((unsigned char *)dest < ((unsigned char *)src +␣
↪count))))
    strcat(emsg, "src and dest overlap");

  /*------------------------------------------------------------
↪-----------*/
  /* All constraint violation checks have been cleared. Do the␣
↪copy.         */
  /*------------------------------------------------------------
↪-----------*/
  else {
    memcpy(dest, src, count);
    return (0);
  }

  /*------------------------------------------------------------
↪-----------*/
  /* If any constraint violations are detected, the C11 standard␣
↪prescribes */
  /* that zeroes are written to dest[0] through dest[destsz-1]. ␣
↪            */
```

(continues on next page)

(continued from previous page)

```
  /*------------------------------------------------------------------
↪-----------*/
  if (dest && (destsz <= RSIZE_MAX))
    memset(dest, 0, destsz);

  __throw_constraint_handler_s(emsg, EINVAL);
  return (EINVAL);
}
```

In contrast to a typical memcpy() implementation, the memcpy_s() implementation performs several run-time checks on the validity of the arguments, including:

- Neither the *dest*, nor the *src* pointers may be NULL
- Both the *destsz* and the *count* arguments must be less then RSIZE_MAX
- The *count* argument must be less than or equal to the *destsz* argument
- The *src* buffer must not overlap with the *dest* buffer

A failure of any of these run-time checks constitutes a *constraint violation*, in which case, the memcpy_s() implementation does not perform the copy operation. Instead, the first *destsz* bytes of the *dest* buffer is filled with zero, and the type of the constraint violation is communicated to a constraint handler function. Also, in the event of a constraint violation, the memcpy_s() function returns a non-zero error type value to indicate to the calling function that a constraint violation has been detected.

Setting Up a Constraint Violation Handler Function

Included with the C11 secure functions themselves, the C runtime support library API provides a function that allows a developer to designate their own function as the one to be called when a constraint violation is detected by one of the implemented secure functions.

set_constraint_handler_s <stdlib.h>

```
constraint_handler_t set_constraint_handler_s(constraint_
↪handler_t handler);
```

where the *constraint_handler_t* type is defined in stdlib.h as follows:

```
typedef typedef void (*constraint_handler_t)(const char *,␣
↪void *, errno_t);
```

and the *errno_t* type is also defined in stdlib.h as follows:

```
typedef int errno_t;
```

The constraint handler registration helper function registers a custom constraint handler function to be called when a constraint violation is detected, set_constraint_handler_s() must be called with a pointer to the constraint handler function that is to be called when a violation is detected.

A custom implementation of a constraint handler function must match the signature as specified in the definition of the *constraint_handler_t* type. That is,

```
void <name of constraint handler function>(const char *,␣
↪void *, errno_t);
```

Two example constraint handler function implementations are provided in the C Runtime Support Library:

- **abort_handler_s()** - a constraint violation causes the application to abort
- **ignore_handler_s()** - a constraint violation goes unreported

If set_constraint_handler_s() is not called before the first constraint violation is detected, then ignore_handler_s() is assumed to be the default constraint handler.

Enabling Use of Secure Functions via __STDC_WANT_LIB_EXT1__ Definition

In the c29clang C runtime library header files, all C11 secure function prototypes are guarded by a pre-processor directive. In order for a given C11 secure function prototype to be made known to the compiler before encountering a call to the secure function in the C source file, the following conditions must be true:

- The compiler must be invoked assuming the C11 (or later) language standard. The compiler assumes a C language standard of C17 with GNU extensions (**-std=gnu17**) by default.
- The C source file must define __STDC_WANT_LIB_EXT1__ to "1" prior to including the header file that contains the prototype of the secure function to be called.
- The C source file must include the C runtime support library header file that contains the prototype of the secure function to be called.

Please see the example below for further details.

Note: __STDC_WANT_LIB_EXT1__ must be defined to "1" when using C11 secure functions

An attempt to call a C11 secure function without defining the __STDC_WANT_LIB_EXT1__ compile-time symbol to 1 is likely to cause unpredictable behavior if the linked application is loaded and run. If no appropriate prototype of the C11 secure function is provided prior to a call to that function, the compiler emits a warning diagnostic like this:

```
ex_n.c:17:3: warning: call to undeclared function 'strcat_s';␣
↪ISO C99 and later do not support implicit function␣
↪declarations [-Wimplicit-function-declaration]
   17 |   strcat_s(dest_string, 10, source_string);
      |
1 warning generated.
```

Consider using the *-Werror=implicit-function-declaration* option to instruct the c29clang compiler to interpret a call to an undeclared function as an error instead of a warning.

Example

To summarize the different aspects of using a C11 secure function properly, consider a function that declares a char buffer of fixed length and then attempts to concatenate the string from an incoming buffer to the string that currently exists in the fixed length local buffer. To guard against several unintended effects, the strcat_s() function could be used as follows:

```
/
↪*****************************************************************
↪
/* safe_strcat_ex.c                                                ␣
↪            */
/*                                                                 ␣
↪            */
/* Example of proper setup and use of strcat_s() C11 secure␣
↪function.        */
/*                                                                 ␣
↪            */
/
↪*****************************************************************
↪
#define __STDC_WANT_LIB_EXT1__        1

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

/
↪*****************************************************************
↪
/* dump_error_msg() - a custom constraint handler function.       ␣
↪            */
```

(continues on next page)

(continued from previous page)

```
/
 ↪*****************************************************************************
 ↪
void dump_error_msg(const char * __restrict emsg,
                    void * __restrict ptr,
                    errno_t eval)
{
  printf("Constraint violation detected:\n");
  printf("\t%s\n", emsg);
}

/
 ↪*****************************************************************************
 ↪
/* safe_strcat()                                                              ␣
 ↪             */
/
 ↪*****************************************************************************
 ↪
errnot_t safe_strcat(const char *source_string) {
  errno_t ret_val = 0;
  char dest_string[10] = "aaa";

  /*-------------------------------------------------------------------
 ↪------------*/
  /* Register custom constraint handler function.                        ␣
 ↪             */
  /*-------------------------------------------------------------------
 ↪------------*/
  set_constraint_handler_s((constraint_handler_t)&dump_error_
 ↪msg);

  /*-------------------------------------------------------------------
 ↪------------*/
  /* Concatenate source_string buffer contents to end of dest_
 ↪string buffer  */
  /* contents with safety checks.                                        ␣
 ↪             */
  /*-------------------------------------------------------------------
 ↪------------*/
  return ret_val = strcat_s(dest_string, 10, source_string);
}
```

Observations of note include:

- The __STDC_WANT_LIB_EXT1__ compile-time symbol is defined to "1" prior to inclusion of any of the C runtime support header files, making C11 secure function prototypes that are declared in any of the following include files visible to the compiler.
- The C source file includes an implementation of a constraint hander function, dump_error_msg(), whose signature matches the *constraint_handler_t* type that is defined in stdlib.h.
- The set_constraint_handler_s() function is called to register dump_error_msg() as the constraint handler function to be called when a constraint violation is detected.
- The call to the strcat_s() C11 secure function returns a non-zero *errno_t* type value to the caller to indicate whether a constraint violation is detected.
- In the definition of strcat_s(), there are a few constraint violations that may occur depending on the value of the *source_string* argument that is passed into the safe_strcat() function, including:
  - if the value of *source_string* is NULL,
  - if the buffer pointed to by *source_string* contains a string > 7 bytes long, which would cause the concatenation operation to write past the end of the *dest_string* buffer, or
  - if the value of *source_string* happened to be an address that falls within the bounds of the *dest_string* buffer.
- If a constraint violation is detected in the execution of strcat_s(), then the dump_error_msg() constraint handler function would be called to print the type of the violation out to stdout.

The Secure Functions

Included below is a summary description of each of the C11 secure functions that are implemented in the c29clang C runtime library. In the majority of cases, the C11 secure functions takes an extra *destsz* argument – or *_size* in the case of gets_s() – with which the caller indicates the total size of the buffer that data is being written into.

gets_s <stdio.h>

```
char *gets_s(char *_ptr, rsize_t _size);
```

Writes input from stdin to *_ptr* buffer. The following constraint violations are detected and reported:

- storage buffer pointer *_ptr* is NULL
- specified *_size* is 0
- specified *_size* is > RSIZE_MAX

- input from stdin is truncated

memcpy_s <string,h>

```
errno_t memcpy_s(void *dest, rsize_t destsz, const void *src,
 ↪ rsize_t count);
```

Copy *count* bytes from *src* buffer to *dest* buffer. The following constraint violations are detected and reported:

- *src* or *dest* pointer is NULL
- specified *destsz* > RSIZE_MAX
- specified *count* > RSIZE_MAX
- specified *count* > specified *destsz*
- *src* and *dest* buffers overlap

memmove_s <string.h>

```
errno_t memmove_s(void *dest, rsize_t destsz, const void␣
 ↪*src, rsize_t count);
```

Copy *count* bytes from *src* to *dest*. The following constraint violations are detected and reported:

- *src* or *dest* pointer is NULL
- specified *destsz* > RSIZE_MAX
- specified *count* > RSIZE_MAX
- specified *count* > specified *destsz*

memset_s <string.h>

```
errno_t memset_s(void *dest, rsize_t destsz, int ch, rsize_t␣
 ↪count);
```

Write *count* instances of specified character *ch* to *dest*. The following constraint violations are detected and reported:

- *dest* pointer is NULL
- specified *destsz* > RSIZE_MAX
- specified *count* > RSIZE_MAX

- specified *count* > specified *destsz*

strcat_s <string.h>

```
errno_t strcat_s(char *dest, rsize_t destsz, const char␣
↪*src);
```

Concatenate contents of *src* to the end of the *dest* buffer. The following constraint violations are detected and reported:

- *src* or *dest* pointer is NULL
- specified *destsz* is 0 or > RSIZE_MAX
- no null terminator character is present in the first *destsz* bytes of the *dest* buffer
- the contents from *src* are truncated
- *src* and *dest* buffers overlap

strcpy_s <string.h>

```
errno_t strcpy_s(char *dest, rsize_t destsz, const char␣
↪*src);
```

Copy contents of *src* buffer into *dest* buffer. The following constraint violations are detected and reported:

- *src* or *dest* pointer is NULL
- specified *destsz* is 0 or > RSIZE_MAX
- the contents from *src* are truncated
- *src* and *dest* buffers overlap

strncat_s <string.h>

```
errno_t strncat_s(char *dest, rsize_t destsz, const char␣
↪*src, rsize_t count);
```

Concatenate a maximum of *count* characters from the *src* buffer to the end of the content in the *dest* buffer. The following constraint violations are detected and reported:

- *src* or *dest* pointer is NULL
- specified *destsz* is 0 or > RSIZE_MAX
- specified *count* > RSIZE_MAX

- no null terminator character is present in the first *destsz* bytes of the *dest* buffer
- the contents from *src* are truncated
- *src* and *dest* buffers overlap

strncpy_s <string.h>

```
errno_t strncpy_s(char *dest, rsize_t destsz, const char␣
↪*src, rsize_t count);
```

Copy a maximum of *count* characters from the *src* buffer into the *dest* buffer. The following constraint violations are detected and reported:

- *src* or *dest* pointer is NULL
- specified *destsz* is 0 or > RSIZE_MAX
- specified *count* > RSIZE_MAX
- the contents from *src* are truncated
- *src* and *dest* buffers overlap

In addition, the following C11 security-related helper function is included in the C runtime support library (libc.a):

strnlen_s <string.h>

```
size_t strnlen_s(const char *string, size_t maxLen);
```

A variation on the strlen() function. In this case, if the length of *string* is longer than the specified *maxLen*, then strnlen_s() returns *maxLen* instead of the actual length of *string*.

3.11 C29x Security Model

3.11.1 The Safety and Security Unit (SSU)

The TI C29x may take advantage of the special Safety and Security Unit (SSU), which protects code and data within an application. The Implementing Run-Time Safety and Security With the C29x Safety and Security Unit (SPRADK2 <http://www.ti.com/lit/pdf/SPRADK2>__**) application note describes this unit in depth. This page provides an introduction to the SSU.

3.11.2 Key Terms and Constructs

The SSU is configured with a hierarchy of control:

- An Access Protection Range (APR) is a region of memory with associated read/write permissions.
- The LINK is one or more regions of executable code that control data accesses (reads and writes) to both memory and peripherals.
- The STACK isolates execution contexts and the stack pointer address (A15).
- The ZONE is configured to control debugging and breakpoint support, as well as firmware update permissions.

A ZONE contains one or more STACKs. A STACK contains one or more LINKs. A LINK is a member of one and only one STACK. A STACK is a member of one and only one ZONE.

APRs are granted or denied read/write access to LINKs via hardware configurations.

In this document, the all-capital APR, LINK, STACK, and ZONE terms refer to the SSU concepts instead of other definitions.

3.11.3 Inter-STACK Calls

A call from one LINK in a STACK to a different link in another STACK is called an inter-STACK call. Such calls are heavily restricted:

- A special "protected" call instruction must be used to transfer control from caller to callee.
- The first instruction fetched from the callee must be a special handshake instruction packet.
- A special "protected" return instruction must be used to transfer control from callee to caller.
- The first instruction fetched from the caller upon return must be a special handshake instruction packet.
- All registers are cleared by both call and return except registers passed to or returned from the callee.
- The stack pointer (A15) is not shared by the caller and callee.
  - Functions with variadic arguments or those that pass or return structures and/or non-fundamental types use the stack for storage according to the rules of the C29x ABI. Such functions can't be the callee of an inter-STACK call because the stack pointer changes.

Any call whose caller and callee are in the same STACK may be referred to as an intra-STACK call. Such calls are less restricted. None of the restrictions above apply.

Compiler Support for Inter-STACK Calls

C/C++ functions can be declared/defined using the `c29_protected_call` function attribute. This attribute causes the c29clang compiler to act as if calls to that function or function type are inter-STACK calls. At the assembly level, the compiler emits "protected" calls and returns with the proper registers preserved in addition to the handshake instructions at both the callee's entry and at the return address. See *Function Attributes* for more about function attributes.

For example:

```
void my_function() __attribute__((c29_protected_call)) {
    return; // The definition of my_function, along with the␣
 ↪attribute,
            // results in the correct handshake and protected␣
 ↪return.
}
void call_my_function() {
    my_function(); // my_function is declared protected, so this␣
 ↪results in
                   // the correct protected call and return␣
 ↪handshake.
}

void normal_function();
void (*protected_fn_ptr)() __attribute__((c29_protected_call)) =␣
 ↪&normal_function; // Suppressible warning in C
void call_normal_function {
    normal_function(); // Normal call instruction with no␣
 ↪handshake

    // Note: normal_function will not have the handshake and␣
 ↪protected
    // return unless its definition also has the c29_protected_
 ↪call
    // attribute.
    protected_fn_ptr(); // Results in protected call and return␣
 ↪handshake
}
```

The c29clang compiler identifies and produces an error for any protected function that cannot be called due to stack requirements from the ABI. Example causes for such errors include passing excessive numbers of integer, pointer, or floating-point arguments in registers, returning a structure type by value, and use of variadic arguments.

3.11.4 Security Support in the Linker

The SECURE_GROUP Memory Range Attribute

The linker accepts the SECURE_GROUP attribute in the MEMORY region of the linker command file. This attribute declares the type of secure behavior allowed in code placed in that memory range. The syntax is as follows:

SECURE_GROUP (*name* [, (public|private)][, reads=(a, b. . .)][, writes=(a, b,. . .))

The following example assigns the .text_caller section to the CALLER_GROUP **call group**:

```
MEMORY {
    RO_CODE1: origin=0x20, length=0x1000, SECURE_GROUP(CALLER_
 ↪GROUP)
}
SECTIONS {
    .text_caller : { *(.text.caller) } > RO_CODE1
}
```

A **call group** is a functional name for a group of code within a single secure module. The linker assumes that code placed in the same **call group** can freely call to and return from one another without protection, including sharing the same stack pointer.

The following example extends the above linker command file snippet by adding .text_callee in the CALLEE_GROUP **call group**:

```
MEMORY {
    RO_CODE1: origin=0x20, length=0x1000, SECURE_GROUP(CALLER_
 ↪GROUP)
    RO_CODE2: origin=0x1020, length=0x1000, SECURE_GROUP(CALLEE_
 ↪GROUP)
}
SECTIONS {
    .text_caller : { *(.text.caller) } > RO_CODE1
    .text_callee : { *(.text.callee) } > RO_CODE2
}
```

If a caller and callee might be placed in such a way that they are in different **call groups**, the linker assumes that it is an inter-STACK call and requires protection.

- If the callee was annotated with the `c29_protected_call` attribute in the source code, the linker identifies that protection has already been added and continues.
- If the callee was not annotated with the `c29_protected_call` attribute in the source code, the linker will issue an error, refusing to allow insecure transfer of control.

Linker-generated Secure Calls (Bridging)

If it's not feasible to annotate the callee with the `c29_protected_call` attribute, the linker provides a way to insert the proper CALL.PROT and RET.PROT sequences at the cost of code size, stack usage, and execution cycles. The following example introduces the **PUBLIC** SECURE_GROUP attribute:

```
MEMORY {
    RO_CODE1: origin=0x20, length=0x1000, SECURE_GROUP(CALLER_
 ↪GROUP)
    RO_CODE2: origin=0x1020, length=0x1000, SECURE_GROUP(CALLEE_
 ↪GROUP, PUBLIC)
}
SECTIONS {
    .text_caller : { *(.text.caller) } > RO_CODE1
    .text_callee : { *(.text.callee) } > RO_CODE2
}
```

The default, and opposite of the **PUBLIC** attribute is the **PRIVATE** attribute, which can be specified for readability, but is not required.

A **PUBLIC** SECURE_GROUP instructs the linker to transform an insecure transfer of control into a secure form. This is done by inserting veneers called trampolines and landing pads.

- The linker inserts a trampoline if the call site is not secure. A trampoline contains code to:
  - Save every register on the stack.
  - Securely transfer control to and receive the protected return from the callee.
  - Restore the saved registers.
  - Transfer control back to the caller.
- A landing pad is inserted if the callee is not secure. A landing pad contains code to:
  - Receive the protected call from the caller.
  - Insecurely transfer control to the callee.
  - Securely transfer control back to the caller.

The linker generates both veneers if the call site or callee are insecure. However, if either the call site's function pointer type or the callee's definition are marked with the `c29_protected_call` attribute, the associated veneer is not necessary, saving a portion of the overhead in cycles and memory usage. This is especially true for the call site, as the register saves take a considerable amount of time and space.

Memory Access Protections

Outside of secure calls and returns, the C29 Safety and Security Unit (SSU) provides memory access protections. While two functions might be part of the same STACK, they may not necessarily have read/write access to the same APRs. The **READS** and **WRITES** attributes of SECURE_GROUP describe an abstract form of these access permissions:

```
MEMORY {
    RO_CODE1: origin=0x20, length=0x1000, SECURE_GROUP(SAME_
 ↪GROUP, READS=(DATA1))
    RO_CODE2: origin=0x1020, length=0x1000, SECURE_GROUP(SAME_
 ↪GROUP, READS=(DATA2))
    DATA1: origin=0x10000000, length=0x1000
    DATA2: origin=0x10002000, length=0x1000
}
SECTIONS {
    .text_caller : { *(.text.caller) } > RO_CODE1
    .text_callee : { *(.text.callee) } > RO_CODE2
}
```

In this example, the caller and callee are within the same group, but the linker observes that the caller may access DATA1 while the callee may access DATA2. While a call between caller and callee need not be protected, there is an expectation that the code in callee should not be executed in caller, and vice versa. This has consequences when considering interprocedural optimization.

Note that the arguments to the **READS** and **WRITES** list are informative strings and may be any valid identifier. For simplicity, it's often useful to define them in terms of MEMORY ranges that will be accessible.

SECURE_GROUP and Link-time Optimization

Link-time optimization is a powerful optimization strategy that allows the compiler to optimize while being aware of as much of the program as possible. This is similar to taking all source files, headers, and libraries used to build the program and creating one large source file. Definitions of functions are visible to call sites for inlining, global constants are known when used, etc.

These types of optimizations must be done carefully when security is active. Many of these optimizations move potentially proprietary code and inaccessible data through the boundary of a call site.

- In the case of inter-STACK calls, no such movement should be possible because the system configuration has specifically disallowed the caller and callee from sharing data that isn't part of the ABI.
- In the case of intra-STACK calls, there is still a risk of code containing a data access moving across the boundary into an APR that does not have that permission, causing a security fault.

The linker will not move code or data accesses through the secure boundary of an inter-STACK call during link-time optimization.

The linker refers to the **READS** and **WRITES** lists of SECURE_GROUPs for intra-STACK calls, deciding to allow or prohibit optimization across the call using the following rules:

- If the caller may be allocated to multiple memory ranges, the linker assumes it only has access to memory that is shared by all possibilities. That is, the assumed access is the intersection of all **READS** or all **WRITES** of associated MEMORY ranges.
- If the callee may be allocated to multiple memory ranges, the linker assumes it will access all possibilities. That is, the assumed access is the union of all **READS** or all **WRITES** of associated MEMORY ranges.
- Optimization is prohibited if the caller's sum of access doesn't contain all of the callee's potential accesses.

In the following example from linker command file snippets, MEM_* are MEMORY regions and caller/callee are output sections, where code in .caller calls into code in .callee:

```
// Not optimized: MEM_A can't read MEM_2, required by MEM_B
MEM_A: SECURE_GROUP(GROUP1, READS=(MEM_1))
MEM_B: SECURE_GROUP(GROUP1, READS=(MEM_2))

.caller {} > MEM_A
.callee {} > MEM_B
```

```
// Not optimized: MEM_A can't write MEM_1, only read it
MEM_A: SECURE_GROUP(GROUP1, READS=(MEM_1))
MEM_B: SECURE_GROUP(GROUP1, WRITES=(MEM_1))

.caller {} > MEM_A
.callee {} > MEM_B
```

```
// Not optimized: if callee is placed into MEM_B2, then caller␣
 ↪can't cover
//                 its READ of MEM2
MEM_A: SECURE_GROUP(GROUP1, READS=(MEM_1))
MEM_B1: SECURE_GROUP(GROUP1, READS=(MEM_1))
MEM_B2: SECURE_GROUP(GROUP1, READS=(MEM_2))

.caller {} > MEM_A
.callee {} > MEM_B1|MEM_B2
```

```
// Not optimized: if caller is placed in MEM_A1 and callee is␣
 ↪placed in
```

(continues on next page)

(continued from previous page)

```
//                 MEM_B2, or caller in MEM_A2 and callee in MEM_
 ↪B1,
//                 accesses are not covered.
MEM_A1: SECURE_GROUP(GROUP1, READS=(MEM_1))
MEM_A2: SECURE_GROUP(GROUP1, READS=(MEM_2))
MEM_B1: SECURE_GROUP(GROUP1, READS=(MEM_1))
MEM_B2: SECURE_GROUP(GROUP1, READS=(MEM_2))

.caller {} > MEM_A1|MEM_A2
.callee {} > MEM_B1|MEM_B2
```

```
// Optimized: take the accesses shared by all of caller's␣
 ↪possibilities:
//   READS=(MEM_1, MEM_2, MEM_3) WRITES=(MEM_4), where
//   MEM_X, MEM_Y, and MEM_Z are not shared between MEM_A1 and␣
 ↪MEM_A2
// Take all accesses from all possible placement callee:
//   READS=(MEM_1, MEM_3) WRITES=(MEM_4)
// Since the caller's READS and WRITES sets are supersets of the␣
 ↪callees,
// optimization is allowed.
MEM_A1: SECURE_GROUP(GROUP1, READS=(MEM_1, MEM_2, MEM_3, MEM_X),
                             WRITES=(MEM_4))
MEM_A2: SECURE_GROUP(GROUP1, READS=(MEM_1, MEM_2, MEM_3, MEM_Y),
                             WRITES=(MEM_4, MEM_Z))
MEM_B1: SECURE_GROUP(GROUP1, READS=(MEM_1))
MEM_B2: SECURE_GROUP(GROUP1, READS=(MEM_3))
MEM_B3: SECURE_GROUP(GROUP1, WRITES=(MEM_4))
MEM_B4: SECURE_GROUP(GROUP1)

.caller {} > MEM_A1|MEM_A2
.callee {} > MEM_B1|MEM_B2|MEM_B3|MEM_B4
```

Optimization and Security Caveats

Within a single file (after preprocessing, so including headers), the only guaranteed security control is the `c29_protected_call` attribute. For example, a function marked with this attribute will not be inlined at any call site.

Because there is no guarantee of security, the data and code seen within a single compilation unit (a file with all includes expanded and macros replaced) is considered visible and accessible to all code in the same unit. Projects should be structured such that sensitive or proprietary code and data

is not visible to untrusted modules. For example, avoid defining sensitive or proprietary functions in header files that may be included from untrusted modules.

3.12 Name and C++ Name Demangler Utilities

The compiler tools include the name utility, **c29nm**, and the C++ name demangling utility, **c29dem**. The **c29nm** utility is useful for listing the symbols referenced and defined within a file, and the **c29dem** utility can translate mangled C++ symbol and type names into human-readable form.

Contents:

3.12.1 c29nm - Name Utility

The **c29nm** utility can be used to print a list of the names of symbols from c29clang compiler-generated bitcode files, object files, object file libraries, or static executables.

Usage

c29nm [*options*] [*input files*]

- *options* - affect how the c29nm utility behaves in processing any specified *input files*.
- *input files* - a list of one or more c29clang compiler-generated bitcode files, object files, object file libraries, or static executables. If there is no *input files* list specified, **c29nm** attempts to read *a.out* as an input file. If **-** is specified in place of the *input files* argument, then **c29nm** expects a user-specified input file from *stdin*.

Output Format

The c29nm utility outputs a list of symbols including the symbol name along with some simple information about its provenance. The default output format from c29nm 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", c29nm 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

--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 **c29nm** executable. This option causes c29nm 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):

```
#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:

```
%> c29clang -mcpu=c29.c0 -c hello.cpp
```

We can output the list of symbols in the symbol table for *hello.o*:

```
%> c29nm 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 c29nm output to only include symbols that are not defined in *hello.o*:

```
%> c29nm -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*:

```
%> c29clang -mcpu=c29.c0 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*:

```
%> c29nm 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 **c29nm** as input to **c29dem**:

  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:

```
c29clang -mcpu=c29.c0 -c test.cpp
```

  We can then use the **c29nm** utility to write out the symbol names in test.o:

```
%> c29nm 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 **c29nm** to **c29dem** to demangle the mangled names that are present in the **c29nm** output:

```
%> c29nm test.o | c29dem
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 **c29dem** by just using the *--demangle* option or the *-C* option with **c29nm** to instruct **c29nm** to demangle the symbol names that it prints out:

```
%> c29nm --demangle test.o
00000000 T f()
00000000 D NS::ns_my_num
00000000 B g_my_num
00000000 T main
```

Exit Status

The **c29nm** utility should exit with a return code of zero.

3.12.2 c29dem - C++ Name Demangler Utility

The **c29dem** 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

c29dem [*options*] [*mangled names*]

- *options* - affect how the c29dem 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 c29dem utility, then c29dem 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 c29dem utility is to pipe the output of the c29nm name utility to a c29dem 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 **c29dem** 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 c29dem C++ name demangler utility on the command line as in the following:

```
%> c29dem _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 c29dem as follows:

```
%> c29dem < sym_names.txt
foo()
bar(int)
not_mangled
```

- Piping output of c29nm as input to c29dem:

  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:

```
c29lang -mcpu=c29.c0 -c test.cpp
```

  We can then use the c29nm utility to write out the symbol names in test.o:

```
%> c29nm 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 **c29nm** to **c29dem** to demangle the mangled names that are present in the **c29nm** output:

```
%> c29nm test.o | c29dem
00000000 T f()
00000000 D NS::ns_my_num
00000000 B g_my_num
00000000 T main
```

Exit Status

The c29dem utility returns 0 unless it encounters a user error, in which case a non-zero exit code is returned.

3.13 Object File Utilities

Several object file utilities are included with the c29clang compiler toolchain installation. Some of these, specifically **c29objcopy** and **c29strip**, can be used to edit the content of an ELF object file. Others, like **c29objdump** and **c29ofd**, 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 c29clang installation can be found in the sections listed below.

Contents:

3.13.1 c29objcopy - Object Copying and Editing Tool

The **c29objcopy** 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

c29objcopy [*options*] *input file* [*output file*]

- **c29objcopy** - is the command that invokes the object copying and editing tool.
- *options* - affect the behavior of c29objcopy.
- *input file* - identifies an object file or archive of object files. If **-** is specified as the *input file* argument, then c29objcopy 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 c29objcopy output is written. This is typically a file name, where the named file is created or overwritten with the c29objcopy 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 c29objcopy output is written to *stdout*.

Options

The following c29objcopy 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 **c29objcopy** 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 c29objcopy 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 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 **c29objcopy** 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 c29objcopy 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, c29objcopy 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 **c29objcopy** 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 c29objcopy for the <*format*> argument to the *--input-target*, *--output-target*, and *--target* options.

- *binary*
- *elf32-c29*
- *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 c29objcopy attempts to guess the input format.

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.

ihex Object Format

The ihex (Intel HEX) 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

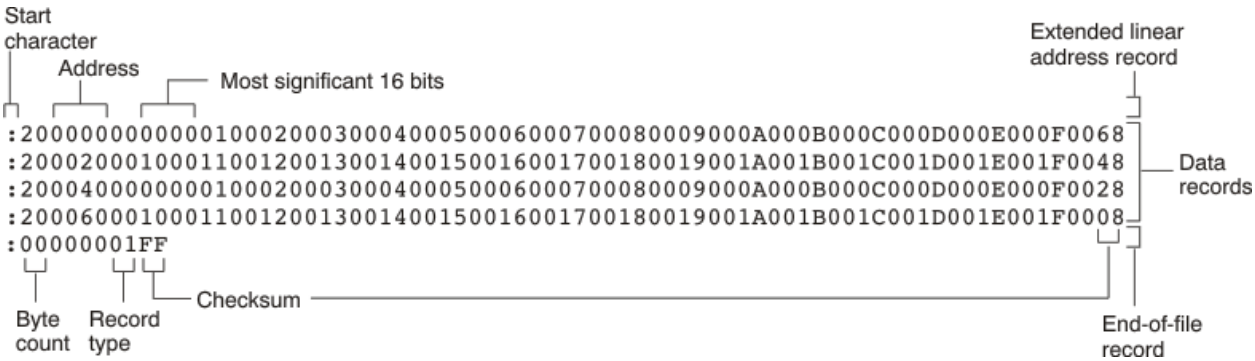
Record type 00, the data record, begins with a colon (:) and is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. The address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 04 (extended 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



ti-txt Format

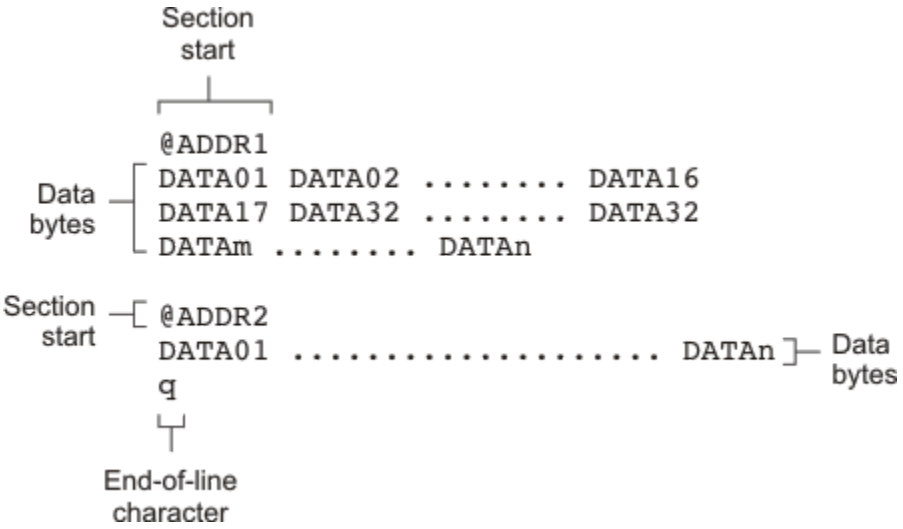
The *ti-txt* format supports 8-bit hexadecimal data. It consists of section start addresses, data byte, and an end-of-file character. The following restrictions apply:

- 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.

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
```

Exit Status

The c29objcopy utility exits with a non-zero exit code if there is an error. Otherwise, it exits with code 0.

3.13.2 c29objdump - Object File Dumper

The **c29objdump** utility can be used to print the contents of object files and linker output files that are named on the **c29objdump** command line.

Usage

c29objdump [*commands*] [*options*] [*filenames* . . .]

- **c29objdump** - is the command used to invoke the object file dumper utility.
- *commands* - are option-like commands that dictate the c29objdump mode of operation.
- *options* - affect the behavior of c29objdump in a particular mode of operation.
- *filenames* - identify one or more input object files. If no input file is specified, then c29objdump attempts to read from *a.out*. If **-** is used as an input file name, c29objdump 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 demangled 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 c29objdump 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*
- *–syms*

Options

The c29objdump 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

c29objdump 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.

3.13.3 c29ofd - Object File Display Utility

The object file display utility, **c29ofd**, can be used to print the contents of object files, executable files, and/or archive libraries in both text and XML formats.

Usage

c29ofd [*options*] *filename*

- **c29ofd** - is the command used to invoke the object file display utility.
- *options* - affect the behavior of c29ofd.
- *filename* - identifies an ELF input object file to be read by c29ofd. If an archive file is specified as an input file, then c29ofd 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 c29ofd 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 c29ofd 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 **c29ofd** 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 c29ofd 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 **c29ofd** 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 c29ofd 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

c29ofd 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.

3.13.4 c29readelf - Object File Reader

The **c29readelf** utility displays low-level, format-specific information about one or more object files.

Usage

c29readelf [*options*] [*filenames* . . .]

- **c29readelf** - is the command used to invoke the object file reader.
- *options* - affect the behavior of c29readelf.
- *filenames* - identifies one or more ELF object files as input to c29readelf. If **-** is specified as the input file, c29readelf 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 c29readelf 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 c29readelf 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 c29readelf to display relocations for each section shown.

--section-symbols
When used with the *--sections* option, the *--section-symbols* option causes c29readelf 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 **c29readelf** executable.

--version-info, -V
Display version sections.

@<file>
Read command-line options from specified <*file*>.

Exit Status

The **c29readelf** utility returns 0 under normal operation. It returns a non-zero exit code if there were any errors.

3.13.5 c29size - Print Size Information

The **c29size** tool prints size information for binary files.

Usage

c29size [*options*] [*input* . . .]

- **c29size** - is the command used to invoke the tool.
- *options* - affect the behavior of the c29size tool.
- *input* - identify one or more input object files, for which c29size prints the size information for each *input* file specified. If no input is specified, c29size attempts to print size information for *a.out*. If **-** is specified as an input file, c29size reads a file from the standard input stream. If an *input* is an archive, size information is displayed for all its members.

The c29size 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:

```
$ c29size --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:

```
$ c29size --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
.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
.C29.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
.rodata.str1.1              18      0
.comment                   121      0
.note.GNU-stack              0      0
.C29.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:

```
$ c29size --radix=8 ref_global.o
   text    data     bss     oct     hex filename
    076       0       0      76      3e ref_global.o

$ c29size --radix=10 ref_global.o
   text    data     bss     dec     hex filename
     62       0       0      62      3e ref_global.o

$ c29size --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.

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 **c29size** executable.

-x
Equivalent to the *--radix=16* option.

@<file>
Read command-line options from specified <*file*>.

Exit Status

The c29size utility exits with a non-zero exit code if there is an error. Otherwise, it exits with code 0.

3.13.6 c29strip - Object File Stripping Tool

The **c29strip** tool removes the symbol table and debugging information from object and executable files. To invoke the **c29strip** tool, use the following command line:

c29strip [*options*] *filename* [*filename*]

- **c29strip** - is the command that invokes the object file stripping tool.
- *options* - control the behavior of the c29strip tool. Options are not case sensitive and can appear before or after the files to be stripped. Precede each option with a hyphen (-).
- *filename* - identify one or more object files (.obj) or executable files (.out) to be processed by c29strip. By default, the files are modified in-place.

Options

--buffer_diagnostics,-pdb
Line buffer diagnostic output

--diag_wrap[=on,off]
Wrap diagnostic messages (argument optional, defaults to on)

--emit_warnings_as_errors,-pdew
Treat warnings as errors

--help,-h
Display help information

--outfile,-o=file
Write the stripped output to the specified new file. When the strip utility is invoked without the -o option, the input object files are replaced with the stripped version.

--postlink,-p
Remove all information not required for execution. This option causes more information to be removed than the default behavior, but the object file is left in a state that cannot be linked. This option should be used only with static executable files.

--rom
Strip readonly sections and segments.

Exit Status

The c29strip utility exits with a non-zero exit code if there is an error. Otherwise, it exits with code 0.

CHAPTER

FOUR

NOTE ON LINUX INSTALLATIONS

4.1 c29clang Shared Library File Dependencies

It is useful to be aware of the shared library files that c29clang 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/c29clang
```

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)
```

CHAPTER

FIVE

ADDITIONAL MATERIAL

See the following additional documentation for more about the TI C29x processors from Texas Instruments and the TI C29x code generation tools.

- *C2000 C29x CPU and Instruction Set User's Guide* (SPRUIY2), which is available through your TI Field Application Engineer

Build Automation Tools

- Using CMake with TI Clang Compilers and SysConfig

Generic Clang and LLVM Documentation

- Clang Compiler User's Manual
- Clang Overview
- The LLVM Compiler Infrastructure

CHAPTER

SIX

SUPPORT

Post compiler-related questions to the TI E2E™ design community forum and select the TI device being used.

CHAPTER

SEVEN

VERSION HISTORY

Version	Date	Summary
v0.1	February 2024	Initial documentation draft.
v0.1.0.STS	July 22, 2024	Documentation updated to include Getting Started Guide, Migration Guide and Compiler Tools User Manual.
v1.0.0.LTS	October 2024	Documentation updated for LTS release.
v2.0.0.STS	June 26, 2025	Documentation updated for STS release.
v2.1.0.STS	November 2025	Documentation updated for STS release.

CHAPTER

EIGHT

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN 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 MERCHANTABILITY, 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, validating 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

For offline use, a PDF version of the guide is available here: TI C29x Clang C/C++ Compiler Tools User's Guide