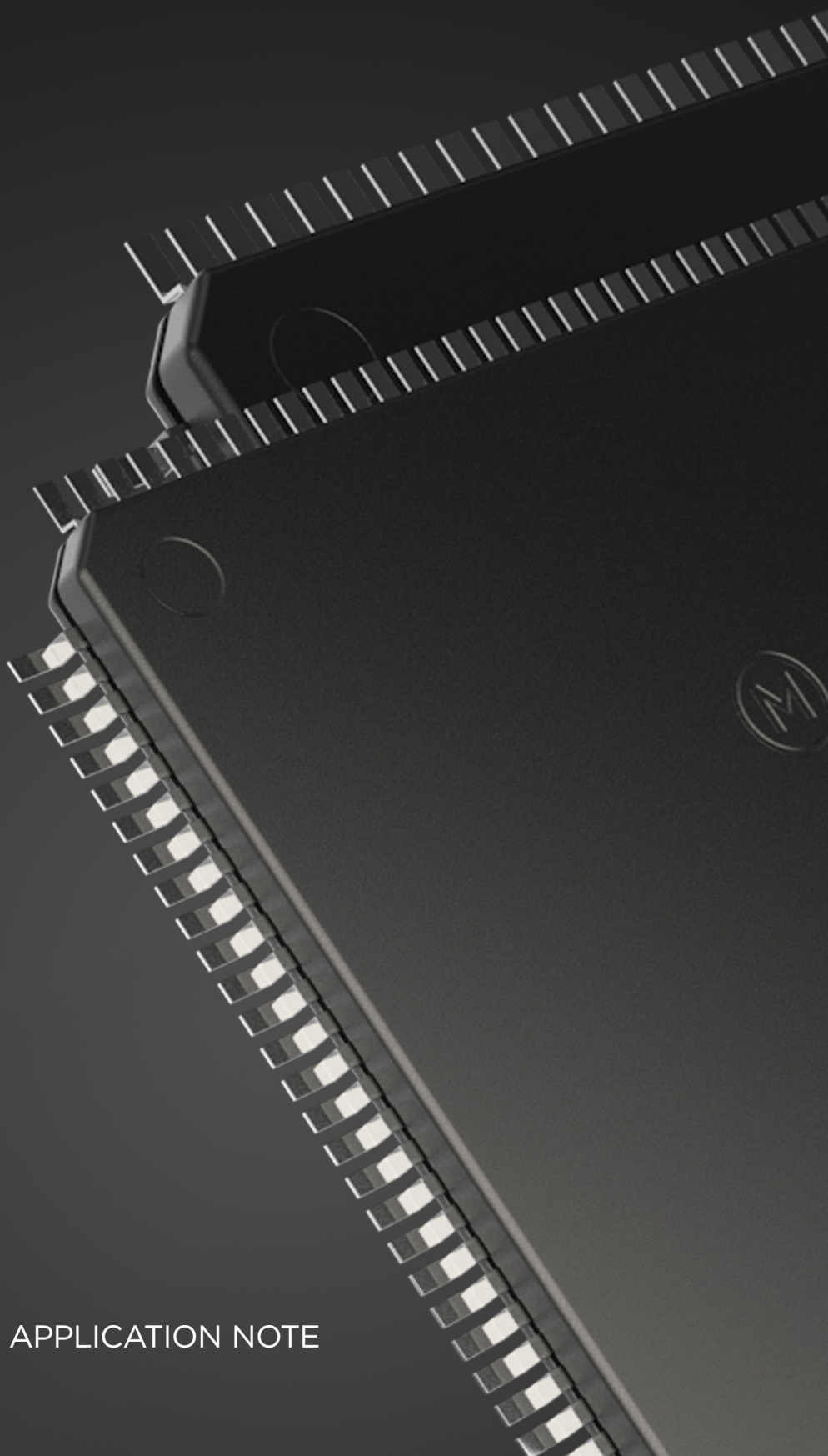
TASKING®

GCC TO TASKING MIGRATION GUIDE
FOR INFINEON AURIX



APPLICATION NOTE

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

1. INTRODUCTION

This migration guide describes the porting of an AURIX/TriCore software project built with GCC to the TASKING toolset. It covers the changes to be made to the C/C++ source code, the assembly source code, the startup code, and the linker script file.

Each section in this guide briefly explains a topic, optionally followed by a reference to the TASKING User Guide, and concludes with a summary of the actions that you should take. The actions do not necessarily have to be executed in the order listed. It is assumed that you are familiar with the GCC tools and are not yet familiar with the TASKING tools.

The TASKING tools can either be configured and executed via the Eclipse Integrated Development Environment (IDE), or via the command line. You probably already have an existing makefile to build the project and are used to configure the GCC tools from the command line, therefore this guide describes the tool configuration and execution via the command line.

You can invoke each TASKING tool (compiler, assembler and linker) separately, or via the control program. It is recommended to use the control program, except for larger projects it is recommended to call the linker directly. More information about the control program can be found in the TASKING User Guide, section 8.1 Control Program.

Summary of actions:

- Read section 8.1 Control Program of the TASKING User Guide.

2. C/C++ SOURCE CODE MIGRATION

2.1 ISO-C compliant code - data elements

Source code that complies with ISO-C (ISO/IEC 9899:1999) can be ported without modification. The behavior of the code remains the same, since the size and value range of all data and pointer types are equal. Also both compilers treat characters as signed, unless the default settings are changed.

- Check GCC command line settings for use of unsigned characters.
- If GCC option `-funsigned-char` or `-fno-signed-char` is set, then set TASKING option `--uchar`

2.1.1 Implementation defined behavior

Implementation defined behavior does not affect the build process, but may affect the behavior of the executable code. Once you can build and execute your software it is advised to verify whether differences in implementation defined behavior between GCC and TASKING cause unwanted side effects.

Summary of actions:

As a minimum, verify whether potential differences in the following implementation defined behavior affect your code:

- The results of some bitwise operations on signed integers. (This behavior may vary between GCC versions.)
- Whether a "plain" int bit-field is treated as a signed int bit-field or as an unsigned int bit-field, and use the TASKING option `--signed-bitfields` when your code relies on a signed interpretation.
- What constitutes an access to an object that has volatile-qualified type.

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

2.2 Set appropriate command line options

The command line options should be compatible with the GCC project. It is obvious that include paths and preprocessor macros shall be specified properly, and that GCC specific files should be removed from the project.

Run `cctc -?` to get a list of all compiler options accessible via the control program. Run `ctc -?` to see all options supported by the compiler. Use option `--help=o` to see an extended option description.

You may have used GCC options to specify the maximum size of objects that shall be addressed using the small addressing mode. Consider to copy these settings from the GCC project, the relevant TASKING options are: `--default-a1-size`, and `--default-near-size`.

Summary of actions:

- Specify appropriate command line options.

2.3 Enable GCC compatibility mode

The TASKING toolset provides an option to enable GNU C extensions as well as GNU C++ extensions. For more information see the TASKING Tricore User Guide section 2.2.2 GNU C++ mode.

Summary of actions:

- Set C compiler option `--language=+gcc`
- Set C++ compiler option `--g++`
- Optionally set C++ compiler option `--gnu-version=`*`<version>`*

2.4 Using the preprocessor

If the source code specifically targets the Infineon AURIX devices and will be built with TASKING tools then it is advised to update the source files and use TASKING syntax to implement AURIX specific features.

However, if you do not want to modify the source files you can use the preprocessor to make the necessary changes to the source code. Use the `--include-file=`*`<file>`* command line option to insert an include file into the source file being compiled, and add the include file to the dependencies in the makefile. For example GCC style intrinsics (see section **Intrinsic functions)** can be easily converted to TASKING syntax using the preprocessor directive `#define _enable() __enable()`.

Summary of actions:

- Consider to use C/C++ compiler option `--include-file=`*`<file>`*

2.5 Intrinsic functions

Every intrinsic function available in GCC is also provided by TASKING. GCC however uses just one underscore while TASKING uses two. To migrate your intrinsic function calls, simply add a second underscore, e.g. `_enable()` becomes `__enable()`.

There is one exception, the `_rstv()` intrinsic function is not provided by TASKING. Create the following inline function to implement this intrinsic:

```
 inline void __rstv ( void )
{
   __asm("rstv");
}
```

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

Intrinsic functions that have not been updated will show up as unresolved references during the build.

Summary of actions:

- Add a second underscore to the intrinsic function name.
- Add definition of `__rstv()`: `inline void __rstv ( void ) { __asm("rstv"); }`

2.6 Attributes

TASKING supports the GCC `__attribute__` syntax, however not all attributes provided by GCC are supported. The TASKING compiler emits a warning message when an unsupported attribute occurs.

All, except the following GCC function attributes are supported: `constructor`, `deprecated`, `destructor`, `format_arg` and `no_instrument_function`.

All, except the following GCC variable attributes are supported: `fardate`, `mode`, `nocommon` and `vector_size`.

All, except the following GCC type attributes are supported: `deprecated` and `transparent_union`.

The following TriCore specific attributes are not supported: `asection`, `interrupt`, `interupt_handler`, and `long_call`. However, TASKING provides equivalent functionality using a different syntax.

In opposition to the GCC tools, the TASKING toolset is specifically designed for the development of embedded software executing on resource constrained devices. This has let to a different philosophy regarding section naming, linking and locating, and the support of interrupt functions. More details can be found in the sections below.

Summary of actions:

It is advised to read the following sections of the TASKING User Guide:

- 1.2.1. Memory Qualifiers;
- 1.12. Compiler Generated Sections;
- 1.11.4. Interrupt and Trap Functions.

Optionally read sections:

- 1.7. Attributes;
- 1.8. Pragmas to Control the Compiler.

2.6.1 Section allocation (__attribute__ asection)

By default the TASKING compiler creates a named section for each function and variable and it often is not needed to explicitly specify section names in the source code, this makes it easier to read and maintain the software.

GCC provides the `__attribute__ ((asection("<name>", "a=<align>", "f=<flags>")))` syntax to control the allocation of functions and variables. The parameter <name> specifies the name of the section in which the object is located, <align> specifies the alignment of the section, and <flags> specifies whether the section is 'a' allocatable and 'x' executable.

If section renaming in the source code is required then use the TASKING syntax:

```
#pragma section type|all "<name>"
```

For more information see the TASKING User Guide, section 1.12.1 Rename Sections.

The section alignment can be specified using attribute `__aligned__(<value>)`. Parameter <*value*> specifies the alignment in bytes, and must be a power of two or 0. For more information see the TASKING User Guide, section 1.1.4. Changing the Alignment: __unaligned, __packed__ and __align().

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

2.6.2 Pragma section & Relative addressing

GCC also offers the pragma section as another way to set the section of an object. This pragma is also supported by TASKING.

However, TASKING offers a third and better way to place variables in specific sections using memory qualifiers. Using these qualifiers increases the readability and maintainability of the source code. For more information see the TASKING User Guide, section 1.2.1. Memory Qualifiers. Consider to use these C language extensions.

Summary of actions:

- Consider to remove all asection attributes.
- Consider to use memory qualifiers.
- Alternatively replace

```
      __attribute__ ((asection("<name>", "a=<align>", "f=<flags>")))
```

with

```
      #pragma section type|all "<name>"
      __align(<value>)
```

2.6.3 Interrupt and trap functions (__attribute__ interrupt & interupt_handler)

The GCC compiler requires you to install a service handler for an interrupt by passing the function pointer of it to the interrupt service handler function.

TASKING provides a much simpler method. There are specific keywords to be used for a function which will turn it into an interrupt routine. For example,

```
 void __interrupt(2) rxint(void)
{
   dosomething…
}
```

will turn `rxint` into an interrupt routine with the ISR priority. Of course you can also implement fast interrupts directly in the vector table and other advanced features.

For more information see the TASKING User Guide, section 1.11.4. Interrupt and Trap Functions.

Summary of actions:

- Use TASKING syntax to implement interrupt and trap functions.

2.7 Inline functions

The GCC and TASKING compilers support function inlining. Once your application has been built you have to verify whether functions are inlined in accordance with your specific requirements, and adapt inlining parameters when needed. For details see the TASKING User Guide, section 1.11.3. Inlining Functions: inline.

2.8 Inline assembly

TASKING uses the same syntax for defining inline assembly. However the underlying assemblers are not 100% compatible. The TASKING assembler will issue an error if GNU specific assembly syntax is used in an inline assembly statement.

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

TASKING provides an elaborate set of intrinsic functions to support TriCore specific functionality that cannot be mapped at ISO-C language constructs. For a list of all intrinsic functions see the TASKING User Guide, section 1.11.5. Intrinsic Functions. Often you can replace your inline assembly code by one or more calls to intrinsic functions and avoid the potential pitfalls of inline assembly. Consider to do this, otherwise implement the changes listed below.

Summary of actions:

- A different syntax is used for numeric labels, rewrite label definitions and references.
- Replace `__attribute__ ((section(".zdata"))` with `__near`
- The first parameter `%A0 of ld .d, st .d, st .w` instructions should be replaced with `%0`.
- Replace offsets `%A0,[%a]0` with `%0,[%a]`
- When using extended register names you can e.g. replace `ld.d %%e0` with `ld.d d0/d1`, or use the `e` constraint character. See section 1.6. Using Assembly in the C Source: __asm() in the TASKING User Guide.
- When needed convert 32-bit instructions to their 16-bit equivalent, e.g. `subs %0,%0,%2` could be replaced by `subs16 %0, %2`

2.9 Built-in Functions

The GCC built-in functions `__builtin_tricore_<name>` insert the instruction as indicated by the built-in name.

To migrate your built-in functions you simply remove the prefix `__builtin_tricore` and add an second underscore before the function name, e.g. `__builtin_tricore_nop()` becomes `__nop()`.

Summary of actions:

- Replace built-in function prefix `__builtin_tricore_` with a __ (double underscore).

2.10 Circular addressing

TASKING supports circular addressing using the `__circ` type modifier whereas GCC supports circular addressing through an instance of the type `circ_t`, and macros to initialize the ring buffer and access the ringbuffer. For more information see the TASKING User Guide, section 1.3.1. Circular Buffers: __circ.

Summary of actions:

- Either include the GCC file `machine/circ.h` into the TASKING project and leave your code as is, or rewrite your code using TASKING's syntax for circular addressing.

2.11 Startup code

It is advised to replace the startup code of your project with the default TASKING startup code. You can use the startup code from the library or use the Eclipse IDE to configure the startup code to your needs. For more information see the TASKING User Guide, section 4.3. The C Startup Code.

Subsequently you have to migrate the extensions you have made to the GNU startup code into the TASKING startup code.

Summary of actions:

- Adapt startup code.

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

2.12 Unsupported GNU C/C++ extensions

The following GNU extensions are not supported and have to be rewritten in ISO-C:

- The forward declaration of function parameters (so they can participate in variable-length array parameters).
- GNU-style complex integral types (complex floating-point types are supported).
- Nested functions.
- Local structs with variable-length array fields.

3. ASSEMBLY SOURCE CODE MIGRATION

3.1 Assembler design philosophy

The TASKING toolset design is based on the principle that the compiler optimizes the source code and that when you write assembly code you want to have precise control over the generated instructions sequence. Therefore the assembler does not optimize the code, it does not rewrite 32-bit instructions into their 16-bit equivalent as the GNU assembler does and it does not reorder instructions.

3.2 Set appropriate command line options

Set the appropriate command line options for your specific project. Run `astc -?` to get a list of all assembler options, use `--help=o` to see an extended option description.

3.3 Assembly syntax

3.3.1 Special characters # and ;

Special characters # (hash) and ; (semicolon) have different semantics. GCC accepts a ; to separate statements, whereas TASKING requires a \n (new line). GCC uses a # symbol to start a comment wherease TASKING uses a ; symbol.

Summary of actions:

- Replace the GCC inline comment character # with ;
- Replace the GCC statement separator ; with \n

3.3.2 Register and SFR names

GCC uses the % (percentage) symbol as prefix for register names whereas TASKING does not use a prefix.

GCC uses the $ (dollar) symbol to prefix special function registers (SFRs) whereas TASKING uses the # (hash) prefix. For instance, `mfcr %d8,$psw` should be replaced with `mfcr d8,#psw`.

Summary of actions:

- Remove the register name prefix `%`
- Replace the SFR prefix `$` with a `#`

3.3.3 Location counter

GCC uses the dot character to mark the current program counter (PC). See 3.2. Assembler Significant Characters.

Summary of actions:

- Replace location counter symbol . (dot) with *.

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

3.3.4 Literals

Summary of actions:

- Prefix literals with the `#` character.

3.3.5 Operand prefixes

TASKING offers an elaborate expression syntax plus built-in assembly functions that provide a superset of the GCC operand prefix functionality. For more information see the TASKING User Guide, section 3.6. Assembly Expressions, and section 3.1 Built-in Assembly Functions. Replace GCC specific operand prefixes with an expression that has equivalent semantics.

Summary of actions:

- Replace `hi:<symbol_or_expression>` with `@HI(<expr>)`
- Replace `lo:<symbol_or_expression>` with `@LO(<expr>)`

3.4 TriCore opcodes

Both GCC and TASKING support the opcodes (mnemonics) that are described in the 'TriCore Architecture Manual'. No conversion is required.

3.5 Miscellaneous

- Add section definitions using the `.SDECL "<name>",<type>[,<attribute>]... [AT <address>]` directive.
- `%0.0` should be replaced by `%LO` and `%0.1` by `%HO`

3.6 Assembler Directives

GCC directives and TASKING directives having the same syntax are semantically equivalent.

Some GCC directives have optional parameters that are not supported by TASKING. The TASKING assembler will emit an error if such parameters are used. You have to analyze the error and act appropriately. For example, the second and third parameter of directive `.align <boundary>,<abs-expr>, <maximum>` are optional in GCC and are not supported by TASKING.

Some GCC directives are not supported by TASKING. The assembler emits an error if such directive is used. You have to analyse the error and act appropriately. TASKING may provide a different directive with equal semantics; Or the directive has trivial functionality and can be easily rewritten using TASKING assembly syntax. Some examples:

Trivial replacements:

- Replace `.hword` with `.half`
- Replace `.short` with `.half`
- Replace `.rept` with `.dup`
- Replace `.section` with `.sdecl` and add a `.section` directive.

GCC controls start with a . (dot) character whereas TASKING uses a `$` (dollar) prefix.

- Replace `.list` with `$LIST`

The GCC `.eject` directive forces a page break when generating assembly, use the TASKING $PAGE control instead.

- Replace `.eject` with `$PAGE`

Conditional assembly using the `.ifgt <absolute expression>` is not supported but can be easily implemented using the `.if` directive and including the greater than comparison in the expression.

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

Summary of actions:

- Fix error messages caused by directives based on the info above.

3.7 Pseudo-opcodes for TriCore

GCC pseudo-opcodes for TriCore directives and TASKING directives having the same syntax are semantically equivalent.

Some GCC pseudo opcodes like `.code16`, `.code32`, `.optim`, and `.noopt` are not supported by TASKING since it's TASKING's philosophy that the compiler should optimize the code and if you decide to write assembly you want to have perfect control over assembly output.

Summary of actions:

- Remove unsupported pseudo opcodes.
- Use 16-bit instruction mnemonics if 16-bit instructions are required.

4 MIGRATION OF THE LINKER CONFIGURATION

At the conceptual level the GCC and TASKING linker offer identical functionality. However these tools do not share a common history and the script languages used to control the link-and-locate process are quite different. Typically one only uses a small subset of the linker functionality. This guide explains how to convert frequently used Hightec linker features into TASKING linkers syntax.

4.1 Set appropriate command line options

For larger projects you probably want to call the linker directly instead of via the control program. However it is difficult to figure out which options to set and which libraries to use. You can use the the control program to find a good set of initial option settings.

First execute

```
cctc --cpu-list
```

to show a list of all supported processors, and select the "cpu type" that is used in your project.

Next execute

```
cctc main.c --dry-run --cpu=<type> --library-directory=<dir> --lsl-file=<file>
```

to see how the linker is called to create executable code.

You should take the listed linker options as the starting point for defining your project specific options. Add the compiler options you have set for your project to the above command, since this may affect the linker option settings, e.g. which libraries to use.

Run `ltc -?` to get a list of all linker options. Add the options that you consider as useful, however at first instance probably no additional options are needed. The following list shows the most used GCC linker command line options and their TASKING equivalent (in both short and long formats). Be aware that a GCC linker script can also contain commands that have the same functionality as a GCC command line option, such commands must be converted to TASKING linker command line options.

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

GCC option	TASKING option	Remarks
-(, --start-group <*archives*> -), --end-group	None	Grouping of archives is not needed since the TASKING linker rescans libraries by default (without performance penalty). Use --no-rescan to disable the default behavior.
--core=<*n*>	No direct equivalent: LSL can be used to select code and data sections by module name and locate these in a specific (core-associated) memory	See also control program option --core=<*name*>, compiler options --code-core-association=private<*n*> and --data-core-association=private<*n*>.
-e <*address*>, --entry <*address*>	-DRESET=<*address*>	Use <address> as the explicit symbol for beginning execution of your program, rather than the default entry point.
--extmap=<*output-option*> -Map <*file*> --cref	-M --map-file[=<*file*>,...] -m --map-file-format[=<*flags*>...]	Generates an extended map file with additional information.
-ffunction-sections [1]	None	TASKING places every function in a separate section by default.
-fdata-sections [2]	None	TASKING places every variable in a separate section by default.
--gc-sections, --no-gc-sections	-Oc --optimize=+delete-unreferenced-sections -Ox --optimize=+delete-duplicate-code -Oy --optimize=+delete-duplicate-data	Enable garbage collection of unused input sections. TASKING also allows removal of duplicate code and data sections.
--help	-? --help	Print a summary of the command-line options on the standard output and exit.
-i, -r, --relocateable	-r --incremental	Perform an incremental link (same as option -r).
-L <*directory*>, --library-path <*directory*>	-L --library-directory[=<*dir*>,...]	Add <*directory*> to the list of paths to search for archive libraries. TASKING specific: this option does not affect the search path for linker control scripts.

[1] This is a compiler option but listed here since it related to the linker.

[2] This is a compiler option but listed here since it related to the linker.

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

-l <*library*>, --library <*library*>	-l --library-name=<*name*>	Add archive file <*library*> to the list of files to link.
--mcpu=<*core*>	-d<device>.lsl -D --define=__CPU__=<*core*> -D --define=__PROC__<*core*>__	Note that the TASKING core names differ from GCC core names.
-nostdlib	-L --ignore-default-library-path	Only search library directories explicitly specified on the command line.
-o <*file*>, --output <*file*>	-o --output=[<*filename*>][:<*format*>[:<*addr-size*>][,<*space_name*>]]...	Use <*file*> as the name for the object file produced by the linker.
--relax	--long-branch-veneers	Relax branches. The maximum allowed displacement for local branches is +/-16 MB. call , fcall , j and jl instructions branching to global functions can be relaxed by the linker to reach any valid code address within TriCore's entire 32-bit address space.
--task-link	--new-task	Do task level linking
-u <*symbol*>, --undefined <*symbol*>	--extern=<*symbol*>,...	Force <*symbol*> to be entered in the output file as an undefined symbol.
--verbose	-v --verbose	Verbose information
@file	-f --option-file=<*file*>,...	Read command line options from file.

4.2 Setting the entry point, stack size, heap size, and more

The symbol `__START` specifies the entry point and is defined in the default startup code (file `cstart.c`). You can specify the start address using command line option `--define=RESET=<address>`. Effectively you assign a new value to the preprocessor symbol RESET that is defined in the linker script file that is associated with the cpu-type you selected when you specified option `--cpu=<type>`.

The size and and the location of the context save area, the user and system stacks, the heap, and the interrupt and trap tables can be specified in a similar way. See the TASKING User Guide, section 7.9.3. Preprocessor Macros in the Linker Script Files for all possibilities and associated macro names.

Summary of actions:

- Use linker option `-D --define=<macro>[=<def>]` to change the default settings for start address, stack size, heap size, CSA size, ...

4.3 The linker script file

First build the project with the default linker script file that is included in the TASKING toolset. Likely your project will link and locate without any problem. In a next step the memory layout of your software is adapted to your specific requirements.

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

4.3.1 TriCore Sections

At the conceptual level the GCC and TASKING tools use a similar naming conventions for sections (.text, .data, .bss, .sdata, .zdata, ...). Notice however that TASKING by default creates separate named sections for each function and variable. By default the TASKING compiler generates sections with the following names where the section prefix is often equivalent to the corresponding GCC section name:

```
section_type_prefix[.uncached][.core_association].module_name.symbol_name
```

When you use a section renaming pragma the following naming convention will be applied:

```
section_type_prefix[.uncached][.core_association][.module_name][.symbol_name][.pragma_value]
```

See section 1.12 Compiler Generated Sections in the TASKING User Guide for details.

4.4 Linker script concepts

Once you want to describe the memory layout of your application you have to familiarize yourself with the TASKING Linker Script Language, abbreviated as LSL.

It is advised to quickly scan through chapters "7 Using the Linker", and chapter "16 Linker Script Language (LSL)" of the TASKING User Guide in order to get an overview about the linker its capabilities. The LSL is used for two purposes. First to describe hardware architecture in order to tell the linker how an address as encoded in an instruction is mapped at a physical memory address. Second, to describe where to place sections in memory.
Only the latter part is currently of interest.

The standard LSL files delivered with the toolset describe the hardware architecture including the available memory of all AURIX (and predecessor) devices released by Infineon. You do not have to modify these LSL files, and do not have to understand the details of these files.

Since you want to define the section layout you should read (versus scan) section 7.9.9 and and the subsequent sections of chapter 7, and also look at section 18.8 which lists the precise syntax and semantics of LSL keywords that could be relevant for you. First, see also the next section **The location counter**.

The remainder of this guide contains examples of how frequently used GCC linker script language constructs should be converted to TASKING linker script language (LSL).

4.4.1 The location counter

Like most modern linkers the TASKING LSL does not support the concept of a "location counter". This is a result of the different philosophies behind the GCC and TASKING linker design. The TASKING linker is an optimizing multi-core linker. The linker will automatically fit all sections in memory, and optimally places sections in non-shared and shared memories. It does not necessary place sections sequentially from low to high addresses as the GCC linker does.
As such, in order to create a specific section layout in memory the optimization process is *restricted* by the section layout description.

Consider the following GCC linker script fragment. The `.text` section from `file1` is located at the beginning of the output section `output`. It is followed by a 1000 byte gap. Then the `.text` section from `file2` appears, also with a 1000 byte gap following before the `.text` section from `file3`. The notation `= 0x12345678` specifies what data to write in the gaps.

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

```
SECTIONS
{
       output :
       {
              file1(.text)
              . = . + 1000;
              file2(.text)
              . += 1000;
              file3(.text)
       } = 0x12345678;
}
```

In TASKING LSL syntax you use the keyword reserved to create a gap. A named section is created and located in this gap. You can specify a fill pattern for each individual gap.

```
section_layout ::linear (direction=low_to_high)
{
       group (ordered, contiguous)
       {
              select ".text*file1*";
              reserved "reserved_text_1" (size=1000, fill=0x12345678);
              select ".text*file2*";
              reserved "reserved_text_2" (size=1000, fill=0x12345678);
              select ".text*file2*";
       }
}
```

4.5 Linker script examples

4.5.1 GCC SECTION command

The GCC command `SECTIONS` is used to describe the memory layout of the output file. TASKING uses the `section_layout` keyword for this purpose.

Assume a program consists only of code, initialized data, and uninitialized data. These will be in the .text, .data, and .bss sections, respectively. Further assume that these are the only sections which appear in the input files. For this example, the code should be loaded at address 0x10000, and the data should start at address 0x8000000. This is a GCC linker script which will do that:

```
SECTIONS {
       . = 0x10000;
       .text : {
              *(.text)
       }
       . = 0x8000000;
       .data : {
              *(.data)
       }
       .bss : {
              *(.bss)
       }
}
```

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

The corresponding TASKING LSL syntax is as follows. Consult the TASKING User Guide for explanations about the used keywords. Be aware that the GCC GROUP command and the TASKING group keyword provide totally different functionalities.

```
section_layout ::mypace (direction = low_to_high)
{
       group (ordered, run_addr=0x10000) {
              select ".text*";
       }
       group (ordered, run_addr=0x8000000) {
              select ".data*";
              select ".bss*";
       }
}
```

Every loadable or allocatable output section has two addresses. GNU uses the term Virtual Memory Address (VMA) which is the address the section will have when the output file is executed, and load memory address (LMA), which is the address at which the section is loaded/stored in flash memory. The TASKING equivalent for the LMA is `load_addr`.

4.5.2 GCC MEMORY command

Typically you do not have to describe the memory in your systems since this is done in the LSL files that are shipped with the product. Via command line option `--cpu=<type>` you have selected the proper LSL file that contains the memory description of your AURIX device. Most AURIX devices do not support external memory and only in exceptional cases you may have to specify the size of the external memory.

GCC defines the memory command as follows,

```
MEMORY {
       <name> [(<attr>)] : ORIGIN = <origin>, LENGTH = <len>
       ...
}
```

The corresponding TASKING LSL syntax is as follows. Consult the TASKING User Guide for explanation about the used keywords.

```
memory <mem_name> {
       type = rom | ram | nvram | blockram;
       mau = <size>;
       size = <size>;
       [priority = <number>;]
       [exec_priority = <number>;]
       [fill = <value>;]
       [write_unit = <size>;]
       map <map_name> ( <map_description> );
}
```

The `map` keyword is used to attach the memory to a bus. Notice that there is no TASKING LSL equivalent for the `GCC` command `REGION_ALIAS`. Such keyword is not needed since the memory is mapped to a bus, which is subsequently mapped to one or multiple address space(s) of one or multiple, potentially heterogeneous, cores. As such the linker can deduce all "region_aliasses".

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

4.5.3 GCC Symbol assignment

Here is a GCC example showing the three different places that symbol assignments may be used. In this example, the symbol `floating_point` will be defined as zero. The symbol `_etext` will be defined as the address following the last `.text` input section. The symbol `_bdata` will be defined as the address following the `.text` output section aligned upward to a 4 byte boundary.

```
floating point = 0;
SECTIONS {
       .text : {
               *(.text)
               etext = .;
       }
       bdata = (. + 3) &  ~ 3;
       .data : { *(.data) }
}
```

There is no simple 1-to-1 mapping of this example onto TASKING LSL. In LSL you can create preprocessor symbols, which do not occur in the output file. LSL does not support a location counter, instead the linker creates symbols, so called linker labels, that mark the start and end address of a section. See section 7.10 Linker Labels in the User Guide. This is the corresponding TASKING LSL.

```
#define preprocessorsymbol = 10;
section_layout ::mypace ()
{
       "floating_point" = 0;
       group code { select ".text*"; }
       group vars { select ".data*"; }
       "etext" = "__lc_ge_code";
       "bdata" = ( "_lc_gb_vars" + 3 ) & ~3;
}
```

4.5.4 GCC PROVIDE command

In some cases, it is desirable for a linker script to define a symbol only if it is referenced and is not defined by any object included in the link. The PROVIDE keyword may be used to define a symbol, such as `etext`, only if it is referenced but not defined. The syntax is `PROVIDE(<symbol> = <expression>)`.

Here is a GCC example of using PROVIDE to define etext:

```
SECTIONS
{
       .text : {
               *(.text)
               _etext = .;
               PROVIDE(etext = .);
       }
}
```

GCC TO TASKING MIGRATION GUIDE FOR INFINEON AURIX

TASKING LSL offer identical functionality, see section 16.8.4 Creating Symbols. This is the corresponding TASKING LSL.

```
section_layout :vtc:linear {
      group code { select .text*; }
      "etext" := __lc_ge_code;
}
```

4.6 Copy Table and Clear Table

At program startup sections are copied from their storage location in flash to a location in RAM and uninitialized data sections are filled with zeros. This program initialization code is "created" by the linker and covers all sections, including sections with names specified by the user.

If you have your own initialization code then use command line option --user-provided-initialization-code.

4.7 Additional resources

In addition to the information supplied in the TASKING User Guide you can consult the application note **Linker Script Language (LSL) Tips & Tricks for TASKING TriCore Toolset** to learn more about the LSL language and its application.

Please do not hesitate to contact **support@tasking.com** if you have questions, or suggestions to improve this guide.