

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

INTRODUCTION

In Spring 2012, Infineon introduced the fifth generation of TriCore™. This family line, called AURIX™, was the first to implement up to three 32-bit TriCore™ CPUs and aimed to meet the highest safety standards while significantly increasing performance at the same time¹. Yet with great architecture comes increased complexity, and if you're a developer about to embark on an AURIX™-based project, then you need the right tools to get the job done. During the same time period, TASKING® launched the next major release of the TriCore™ VX-toolset featuring a powerful palette of language extensions specifically tailored to meet those needs². Since then, the TASKING® VX-toolset has matured into the product that we know today. Not just another compiler, but a future-proof developer platform equipped for fast-paced development through its ACT (AURIX™ Configuration Tool) driven technology.

This application note aims to address the majority of multi-core aspects that come into play while working on an AURIX™-based project. Using a minimalistic homogeneous sample case, it will familiarize you with all of the nitty-gritty details. Both novice users and professionals should find this application note a comfortable read and, in the end, have a well-formed understanding to which extent the architectural features of AURIX are supported by the TASKING® VX-toolset for TriCore™ v6.0r1.

1. THE DESIGN

Pictured below is a flowchart of the sample case that we'll work on throughout this application note. Its design is fictitious in as much that it doesn't reflect anything that might be considered practical. Like with any example, it's been toned down so as to not draw attention away from the play's main characters.

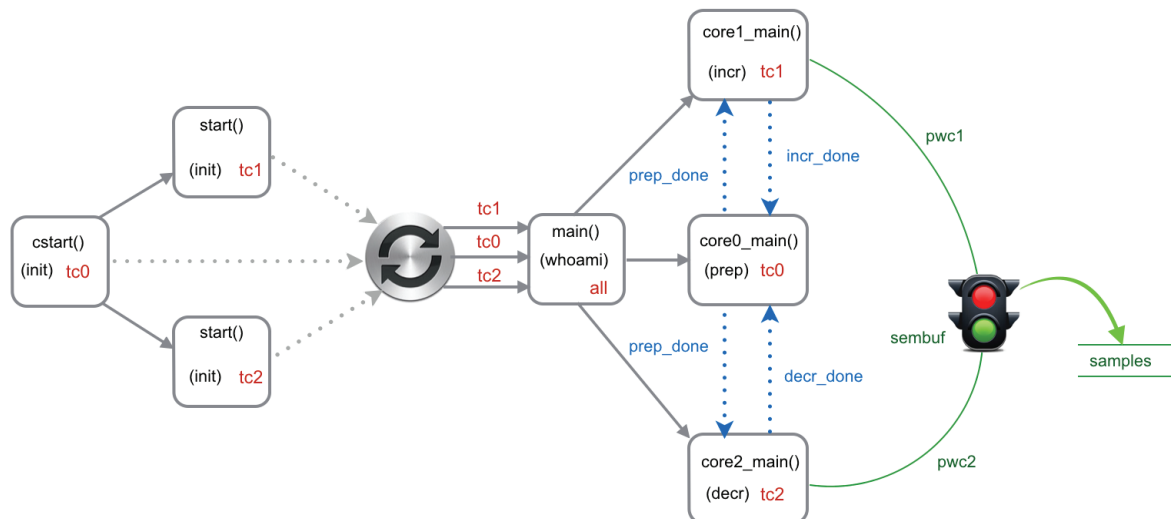


Figure 1 - Example Flowchart of Functions in Embedded Application

The flowchart reads from left to right. Boxes represent a function within the application, either system or user³. The function name is printed within the box along with a single braced keyword that hints to its purpose. Red tags represent the core it must run on. Dotted lines represent some form of synchronization, either by semaphores or status bits. The next three sections provide a closer look. It's still relatively high level before we start discussing the actual implementation.

¹ "highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications", Infineon

² TASKING® VX-toolset for TriCore™ v4.0r1 was released in April 2012

³ System code is code facilitated by the toolset, such as for example libraries and startup code. User code is functional code of the application under development.

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

1.2 SYNCHRONIZING STARTUP CODES (SYSTEM CODE)

The flowchart begins with the startup code⁴ for core `tc0`, which is the only AURIX™ TriCore™ core that runs after a reset. The first thing it does is initialize the stack pointer and start cores `tc1` and `tc2`, at which point these start executing their respective startup codes as well. There are now three startup codes running in parallel which simultaneously start processing the system copy table, but only for variables that have been associated to their specific core⁵. Consider the implications of this. Since the amount of variables associated to each core is not necessarily evenly weighed, the startup codes will not finish at the same time. If the ‘winner’ immediately starts with its associated application part then this poses a risk for variables that it shares with other cores. If these are initialized by one of the other cores, then there’s a risk that they are not yet initialized and the application can potentially break. It is for this reason that within the TASKING® TriCore™ multi-core environment the startup codes wait for each other to finish via a system variable called `_tcx_end_c_init`. If you are evaluating toolsets, be sure to check how these solve it. It is seemingly trivial but also crucial.

1.3 THE MAIN ENTRY POINT CONUNDRUM (SYSTEM/USER CODE)

Here’s a bit of a puzzle: if a single core application has one single program entry point called `main`, then how many entry points does a multi-core application have? The answer is still one, for the simple reason that there can only be one matching definition for any function declaration⁶. It is for this reason that after startup code synchronization, all startup codes converge into `main`. This is the point where your brain might stall for a moment because it’s being trapped into believing that multiple cores are about to do the same thing, which kind of defeats the purpose. Yet the confusion is only momentary when realizing the AURIX™ architecture implements a core special function register that can identify the currently executing core⁷. Using this feature, the `main` body can check by which core it is being executed and diverge into tailored code paths defined for those specific cores. Since these can be considered core specific entry points to `main`, they have been called `core0_main`, `core1_main` and `core2_main`, as observed in the flowchart above.

1.4 INCREMENTS, DECREMENTS AND STATES (USER CODE)

We’ve now reached the actual functional part of the program consisting of three tasks distributed across three cores. Task `core0_main` keeps track of the overall state of the application. It is responsible for starting tasks `core1_main` and `core2_main`, both of which have been assigned a simple calculous task. Both tasks walk along circular buffer `samples` for an equal amount of iterations. However, where one adds a small value to each cell, the other does the opposite using the same amount. While during calculous all cells essentially are in a state of flux, of one thing we can be sure: on completion the aggregate amount added and subtracted to each cell is zero. To verify this, `core0_main` prints the contents of the `samples` buffer when both tasks have signalled that they have completed their work. Each task has its own private circular pointer to access the circular `samples` buffer. These are represented by `pw1` and `pw2`. Note, however, that access is arbitrated via semaphore `sembuf`. In chapter 3, we’ll explain its significance.

2. THE IMPLEMENTATION

What follows next is the implementation of our AURIX™ TriCore™ multi-core design. We’ll walk through it step-by-step and discuss relevant project settings, language extensions, intrinsics and what you need to do in terms of the Linker Script Language. We’ll also discuss the necessity of certain constructs and their alternatives if they exist. Note that for some items we’re just scratching the surface. Checkout the annotations if you want to dive into the deep.

⁴“TASKING® VX-toolset for TriCore™ User Guide”, section 4.3, “The C Startup Code”

⁵“TASKING® VX-toolset for TriCore™ User Guide”, section 16.4.3, “Copy tables”

⁶ ISO/IEC 9899:1999 5.1.2.2.1 Program startup specifies: The function called at program startup is named `main`.

⁷ We’ll discuss this further up.

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

2.1 PROJECT SETTINGS

When you start a new project, a wizard will guide you through three initial project settings that must be chosen. These have been depicted below.

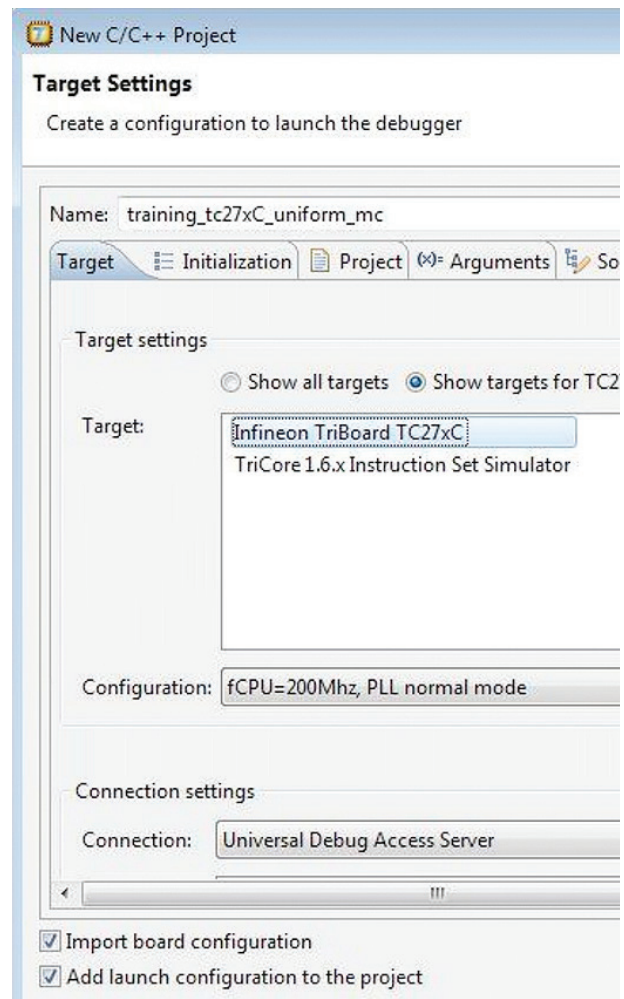
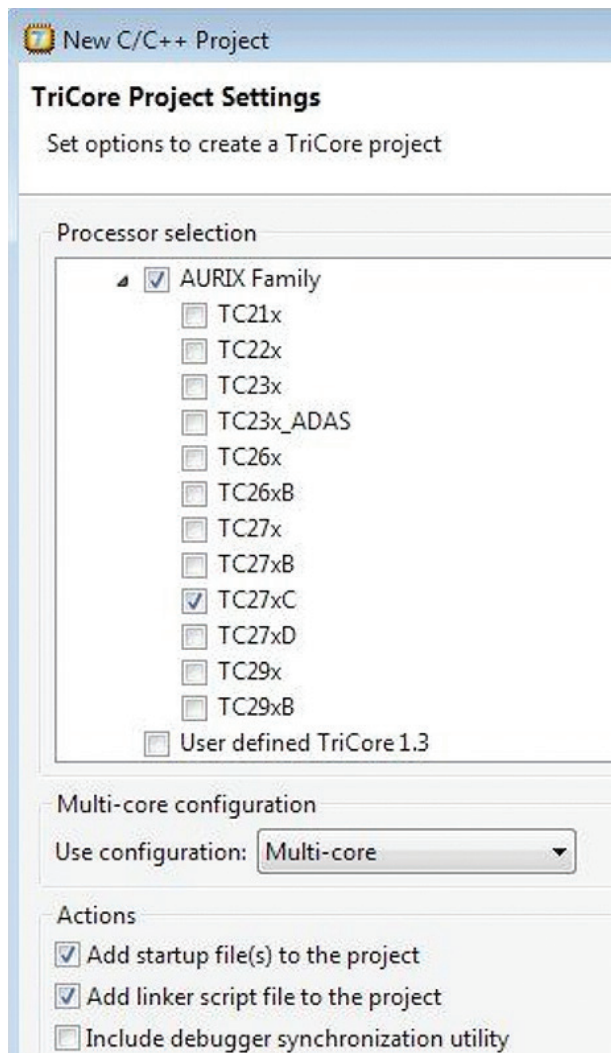
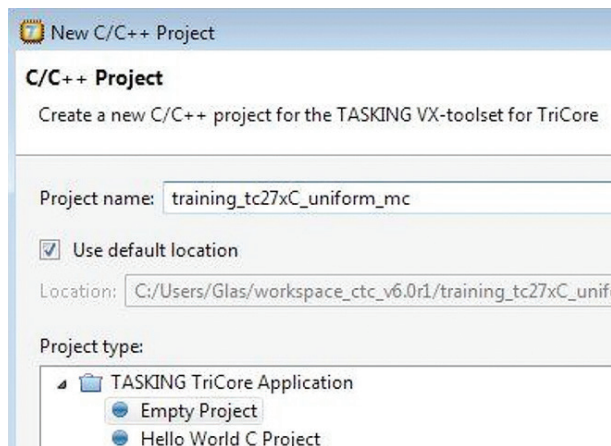


Figure 2 - New Project Wizard Settings for AURIX™ TriCore™

As always, one of the first things you do is assign a project name. Next, you decide which processor type to use, in this case TC27xC. The wizard will recognize this is a multi-core processor and will ask you whether you want to run it single core or multi-core. For obvious reasons, we choose multi-core. The next dialog lists all available execution environments for your application. This can be the simulator, but it can also be a specific development board.

The wizard will show you which ones are available based on your selected processor type. Therefore, in this case it is populated with Infineon TriBoard TC27xC. Select it and click Finish.

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

The next step is to determine which specific cores need to be enabled. For this you need to go to the startup configuration settings dialog as depicted below.

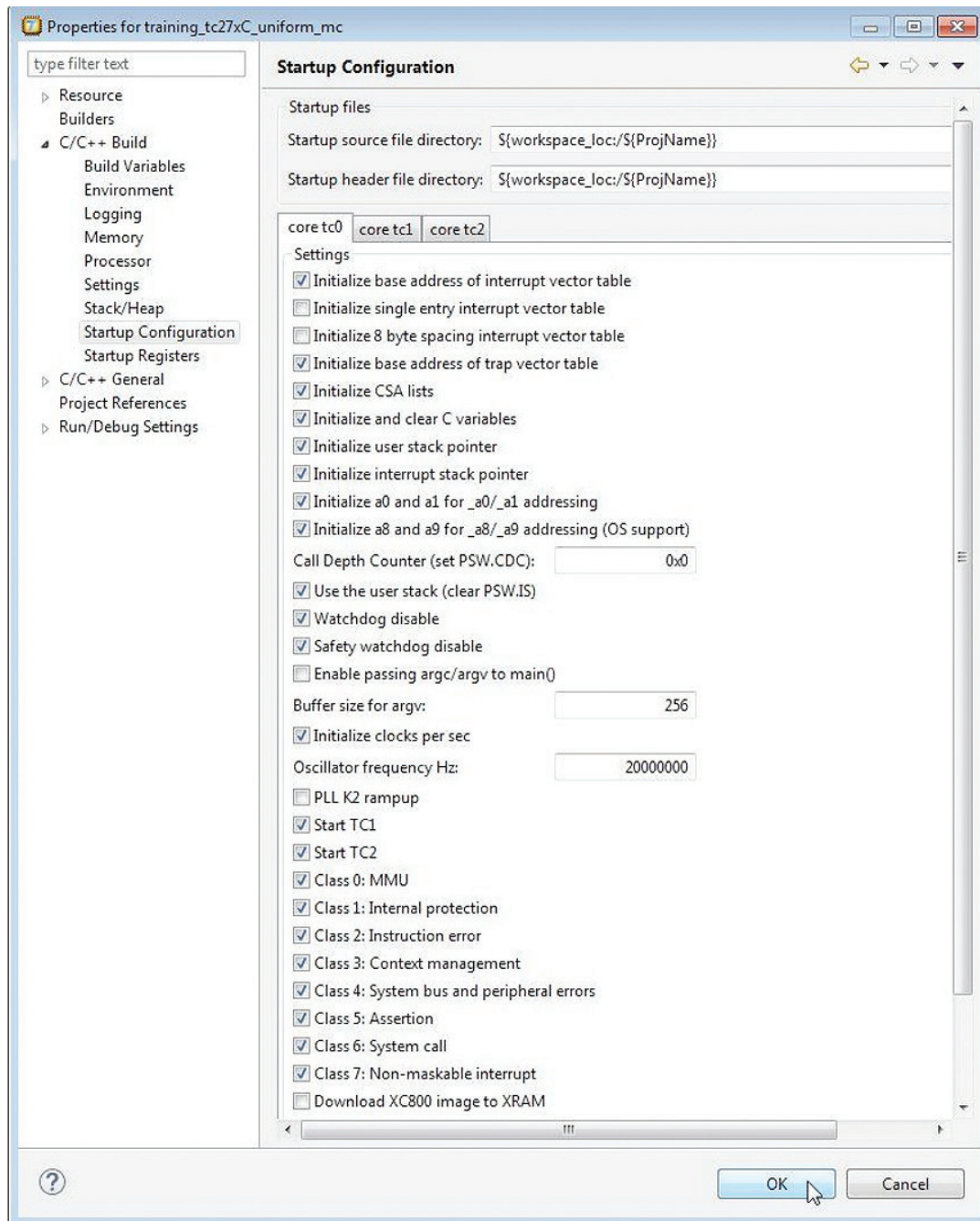


Figure 3 - Enabling Specific Cores in Startup Configuration Settings

In the snapshot, `Start TC1` and `Start TC2` have already been enabled, but by default they're not. You might wonder why, specifically because you chose a multi-core project while running the wizard. The wizard only added the multiple startup codes to the project and enabled system multi-core LSL⁸ files rather than single core. It did not,

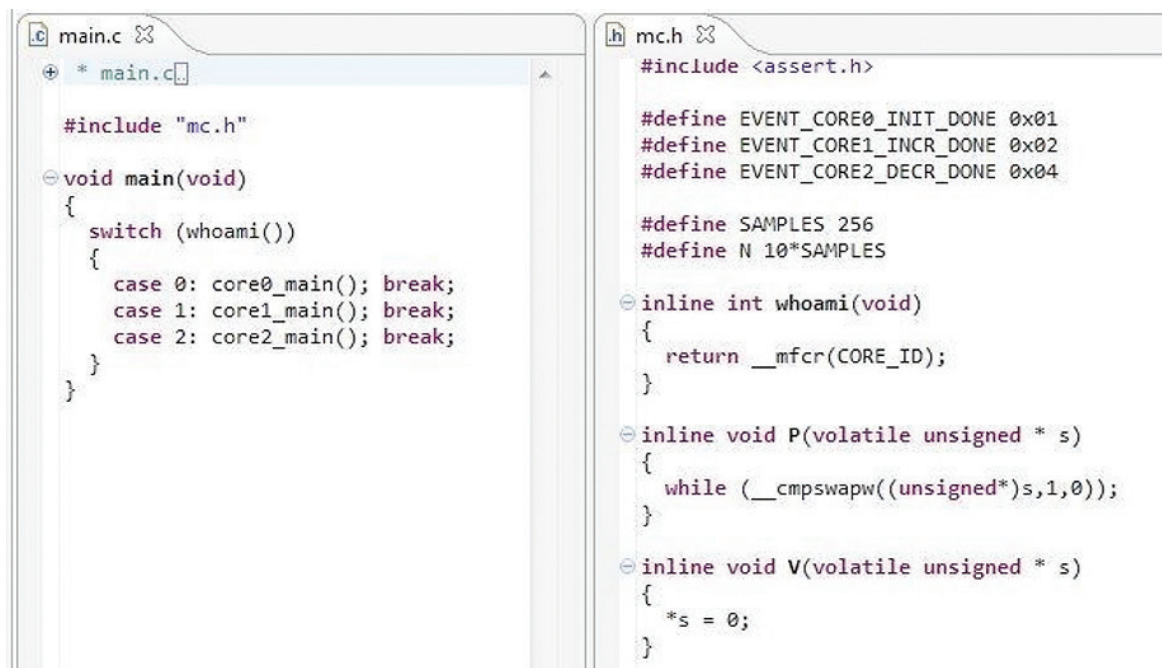
⁸ "TASKING® VX-toolset for TriCore™ User Guide", chapter 13, "Linker Script Language"

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

however, automatically enable all TriCore™ cores, reason being that only the developer knows which ones are going to be actually used. Keep this in mind for your first multi-core application. If your application ever appears to be running just one core, this should jog your memory to inspect your startup configuration.

2.2 RUN-TIME CORE IDENTIFICATION

Here's the `main` program already referred to in section 1.3. All startup codes converge into this area, before `main` itself distributes the application across all available cores.



```

main.c
#include "mc.h"

void main(void)
{
    switch (whoami())
    {
        case 0: core0_main(); break;
        case 1: core1_main(); break;
        case 2: core2_main(); break;
    }
}

mc.h
#include <assert.h>

#define EVENT_CORE0_INIT_DONE 0x01
#define EVENT_CORE1_INCR_DONE 0x02
#define EVENT_CORE2_DECR_DONE 0x04

#define SAMPLES 256
#define N 10*SAMPLES

inline int whoami(void)
{
    return __mfcr(CORE_ID);
}

inline void P(volatile unsigned * s)
{
    while (__cmpswapw((unsigned*)s,1,0));
}

inline void V(volatile unsigned * s)
{
    *s = 0;
}
    
```

Figure 4 - Using Inline Functions to Retrieve the Core Identification Register

Crucial in this setup is the use of inline function `whoami()` which you see defined in header file `mc.h`. This function makes use of intrinsic function `__mfcr`⁹ which retrieves the Core Identification Register¹⁰. Using the switch-statement, the code can now self-check which core it is being executed by and follow its assigned code path.

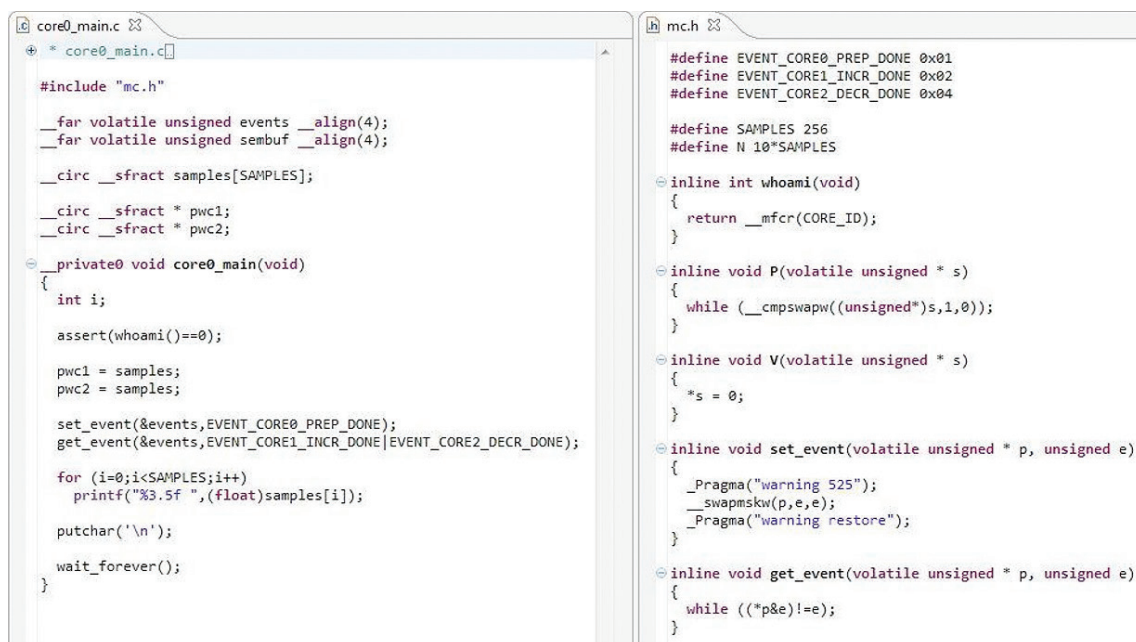
2.3 SIGNALLING EVENTS

As mentioned in section 1.4, function `core0_main` keeps track of the overall state of the application.

⁹"TASKING® VX-toolset for TriCore™ User Guide", section 1.11.5.6 "Register Handling"

¹⁰"Infineon TriCore™ TC1.6P & TC1.6E Core Architecture 32-bit Unified Processor Core User Manual (Volume 1)", chapter 13, "Core Register Table"

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS



```

core0_main.c
#include "mc.h"

__far volatile unsigned events __align(4);
__far volatile unsigned sembuf __align(4);

__circ __sfract samples[SAMPLES];

__circ __sfract * pwc1;
__circ __sfract * pwc2;

__private0 void core0_main(void)
{
    int i;

    assert(whoami()==0);

    pwc1 = samples;
    pwc2 = samples;

    set_event(&events, EVENT_CORE0_PREP_DONE);
    get_event(&events, EVENT_CORE1_INCR_DONE|EVENT_CORE2_DECR_DONE);

    for (i=0; i<SAMPLES; i++)
        printf("%3.5f ", (float)samples[i]);

    putchar('\n');

    wait_forever();
}

mc.h
#define EVENT_CORE0_PREP_DONE 0x01
#define EVENT_CORE1_INCR_DONE 0x02
#define EVENT_CORE2_DECR_DONE 0x04

#define SAMPLES 256
#define N 10*SAMPLES

inline int whoami(void)
{
    return __mfcn(CORE_ID);
}

inline void P(volatile unsigned * s)
{
    while (__cmpswapw((unsigned*)s, 1, 0));
}

inline void V(volatile unsigned * s)
{
    *s = 0;
}

inline void set_event(volatile unsigned * p, unsigned e)
{
    _Pragma("warning 525");
    __swapmskw(p, e, e);
    _Pragma("warning restore");
}

inline void get_event(volatile unsigned * p, unsigned e)
{
    while ((*p&e)!=e);
}

```

Figure 5 - Keeping Track of an Application's State with Variable Events

Variable `events` keeps track of the overall state of the application. The states are defined in header file `mc.h`. Note that `events` is declared as `__far volatile` and has word alignment through the use of the `__align` keyword. The use of `__far` assures a predictable section name¹¹. The `volatile` keyword is familiar to most and assures that the compiler performs explicit read and write operations. The `__align`¹² keyword needs some elaboration. Its reason becomes clear when briefly returning to `mc.h` and noticing signals are sent through the use of `__swapmskw`¹³. This intrinsic is particularly suitable for implementing bit semaphores within a word, and maps to equally named assembly instruction `swapmsk.w`.¹⁴ It comes with the restriction that it must be aligned to word boundaries. Therefore, you must override the TASKING® default half-word alignment for integers¹⁵.

Note how `core0_main` uses inline functions `set_event` and `get_event` to communicate with its neighbouring cores. After having assigned `pwc1` and `pwc2` with the base address of `samples`, it signals the other cores to do their calculus. It subsequently waits for them to finish and then prints the `samples` for inspection. Bear in mind that the function definition for `core0_main` includes the `__private0`¹⁶ keyword. This associates it to memory `PSPR0`, but since this is RAM memory, a copy section needs to be added to the copy table, allowing the startup code to install it. We'll check this once we've built the application and review the project map file. We use this keyword for educational purposes. Its application has nothing to do with the fact that the code for `core0_main` runs on `tc0`. That is merely determined by the `switch` statement used in `main` and thus by you as a programmer. It's essentially a keyword that effects locating behavior similar to the way `__at()` does and with the same considerations that you can alternatively use LSL to achieve the same thing¹⁷.

¹¹ The compiler uses a threshold `--default-near-data` to determine the default memory type of data. Since section names incorporate the memory type it complicates matters when wanting to locate certain variables to specific addresses or address ranges.

¹² "TASKING® VX-toolset for TriCore™ User Guide", section 1.1.4, "Changing the Alignment: `__align()`"

¹³ "TASKING® VX-toolset for TriCore™ User Guide", section 1.11.5.8, "Miscellaneous intrinsic Functions"

¹⁴ "Infineon TriCore™ TC1.6P & TC1.6E Instruction Set 32-bit Unified Processor Core User Manual (Volume 2)", section 1.1, "CPU Instructions"

¹⁵ Another solution would be to enable EABI compliancy through the use of the `--eabi-compliant` command line option.

¹⁶ "TASKING® VX-toolset for TriCore™ User Guide", section 1.4.2, "Code Core Association"

¹⁷ Take a look at the `tc_blink_aurix` example included in the toolset. Like this application note, it's a multi-core example and demonstrates the LSL alternative to code/data core association.

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

2.4 THE DOG RACE

Let's take a look at cores `tc1` and `tc2` running `core1_main` and `core2_main` respectively.

```

core1_main.c
+ * core1_main.c
#include "mc.h"

extern __far volatile unsigned events __align(4);
extern __far volatile unsigned sembuf __align(4);

extern __circ __sfract * pwc1;

__private1 void core1_main(void)
{
    int i;

    assert(whoami()==1);

    get_event(&events,EVENT_CORE0_PREP_DONE);

    for (i=0;i<N;i++)
    {
        P(&sembuf);
        *(pwc1++) += 0.0625;
        V(&sembuf);
    }

    set_event(&events,EVENT_CORE1_INCR_DONE);

    wait_forever();
}
    
```

```

mc.h
#define EVENT_CORE0_PREP_DONE 0x01
#define EVENT_CORE1_INCR_DONE 0x02
#define EVENT_CORE2_DECR_DONE 0x04

#define SAMPLES 256
#define N 10*SAMPLES

inline int whoami(void)
{
    return __mfc(CORE_ID);
}

inline void P(volatile unsigned * s)
{
    while (__cmpswapw((unsigned*)s,1,0));
}

inline void V(volatile unsigned * s)
{
    *s = 0;
}

inline void set_event(volatile unsigned * p, unsigned e)
{
    _Pragma("warning 525");
    __swapmskw(p,e,e);
    _Pragma("warning restore");
}
    
```

Figure 6 - Cores `tc1` and `tc2` Running `core1_main` and `core2_main`

```

core2_main.c
+ * core2_main.c
#include "mc.h"

extern __far volatile unsigned events __align(4);
extern __far volatile unsigned sembuf __align(4);

extern __circ __sfract * pwc2;

__private2 void core2_main(void)
{
    int i;

    assert(whoami()==2);

    get_event(&events,EVENT_CORE0_PREP_DONE);

    for (i=0;i<N;i++)
    {
        P(&sembuf);
        *(pwc2++) -= 0.0625;
        V(&sembuf);
    }

    set_event(&events,EVENT_CORE2_DECR_DONE);

    wait_forever();
}
    
```

Initially, they don't do much other than a self-check and waiting for `core0_main` to give the green light. But as soon as that happens they immediately start doing their dedicated calculous tasks. While one iteratively adds a small fractional value (`GAMMA`) to each cell, the other does the reverse for an equal amount of iterations. Its effect is similar to a dog race whereby `pwc1` and `pwc2` are competing to finish first. But unlike dogs simultaneously chasing an imaginative buck on a dusty track, `pwc1` and `pwc2` instead only have intermittent access to the `samples` buffer, for it being arbitrated by the `P()` and `V()` synchronization primitives¹⁸. In terms of granularity, this is perhaps a little bulky but the alternative would have been to have a semaphore for each individual cell which, for an example, would be overdoing it a bit.

Note that the implementation of `P()` makes use of the `__cmpswapw` intrinsic. Similar to `__swapmskw`, it can be used - and in this case is - to implement semaphores with the added advantage that the conditional check of acquiring a semaphore is built into the `swapmsk.w` instruction.

¹⁸ The `P()` and `V()` primitives were first coined by Dutch computer scientist Edsger W. Dijkstra in his paper "Hierarchical Ordering of Sequential Processes".

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

But why the arbitration, you may ask. The reason is that updating the cell along with the post-increment of either `pwc1` and `pwc2` must be an atomic operation, but the generated assembly is not. Why then must it be atomic, your next question might be. Because if it is not, it might cause interference when the pointers are at a close pace, potentially leading to non-deterministic results. In chapter 3, we will see this proven at run-time when we purposely drop the semaphores. We'll then take a closer look at the assembly to see if it can be explained.

After their calculus, both `core1_main` and `core2_main` signal to `core0_main` that they're done. At this point, `core0_main` will wrap things up by printing the contents of the `samples` buffer. We'll review those results in chapter 3 as well.

2.6 LSL - CACHE COHERENCE

By default, the linker will try to locate code and data in cached memory. Generally, that's a good thing, but not for variables shared between cores. For these it's important that all cores have a consistent representation of a variable's state. However, since AURIX™ TriCore™ cores all have their own private cache without a supporting coherence mechanism¹⁹, you must make sure such critical variables are not being cached. This can be achieved in two steps. First, decide in which memory you want them to be located, for example `lmuram`. Next, inspect the Eclipse properties of this memory to find out its memory mappings.

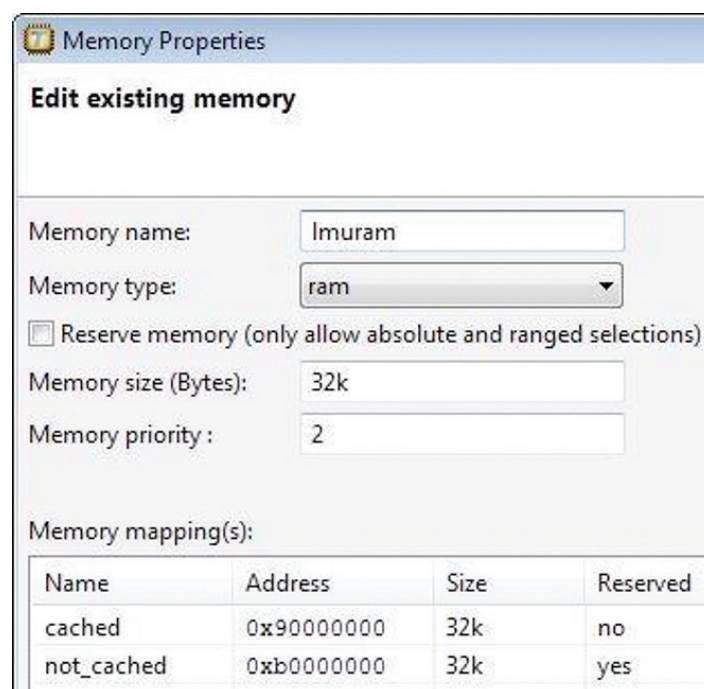


Figure 7 - Reviewing the Memory Properties for `lmuram`

Note that `lmuram` is mapped to a caching and non-caching area, the labels of which you need when locating your critical variables. Next, find out what section names have been assigned to variables `events`, `sembuf` and `samples`. For this you need to familiarize yourself with the naming convention explained in the user manual²⁰. It then follows that each variable has a unique section name that starts with the type of data (in this case `.bss` for cleared far data) followed by the name of the module that declares them (`core0_main`) and ends with the name of the variable. Knowing your section names, you can proceed to locate them using the following section layout definition.

¹⁹ https://en.wikipedia.org/wiki/Cache_coherence

²⁰ "TASKING® VX-toolset for TriCore™ User Guide", section 1.12, "Compiler Generated Sections"

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

```

training_tc27xC_uniform_mc.lsl
// TASKING VX-toolset for TriCore
// Eclipse project linker script file
//
- #if defined(__PROC_TC27XC__)
+ #include "tc27xc.lsl"
  derivative my_tc27xc extends tc27xc
  {
  }
+ #else
-
  section_layout :vtc:linear
  {
    group not_cached (run_addr=mem:mpe:lmuram/not_cached)
    {
      select ".bss.core0_main.events";
      select ".bss.core0_main.sembuf";
      select ".bss.core0_main.samples";
    }
  }
  }
  }
  
```

Figure 8 - Locating Shared Variables in Non-Cached Memory

The key here is that the run-time address is set to `mem:mpe:lmuram/not_cached`, which assures that variables are mapped into `lmuram` using its non-cached memory mapping²¹.

2.7 INSPECTING THE MAP FILE FOLLOWING A BUILD

You can now build your application and review the map file. Earlier, we briefly touched on the fact that because `core0_main` is associated to `PSPR0` it implies that copy sections must be created automatically to allow the startup code to install them. Copy sections are braced with `[]` pairs and have the same name as the section they aim to install. The section names themselves honor the naming convention referred to in the previous section. So, in this case, you'll be looking for a section name `.text.private0.core0_main.core0_main` and a matching copy section `[.text.private0.core0_main.core0_main]`. See if you can spot them.

```

training_tc27xC_uniform_mc.map
* Sections
*****
+ Space mpe:vtc:linear (MAU = 8bit)
-----
| Chip | Group | Section | Size (MAU) | Space addr | Chip addr | Alignment |
-----+-----+-----+-----+-----+-----+-----+
| mpe:dspr0 | | .bss.cstart_tcx_end_c_init (71) | 0x00000004 | 0x70000000 | 0x0 | 0x00000004 |
| mpe:pflash0 | | [.text.private0.core0_main.core0_main] (762) | 0x000000a2 | 0x8000002c | 0x0000002c | 0x00000002 |
| mpe:pflash0 | | [.text.private1.core1_main.core1_main] (767) | 0x0000007e | 0x800000ce | 0x000000ce | 0x00000002 |
| mpe:pflash0 | | [.text.private2.core2_main.core2_main] (772) | 0x0000007e | 0x8000014c | 0x0000014c | 0x00000002 |
+-----+-----+-----+-----+-----+-----+
| mpe:lmuram | not_cached | .bss.core0_main.samples (4) | 0x00000200 | 0xb0000000 | 0x0 | 0x00000008 |
| mpe:lmuram | not_cached | .bss.core0_main.events (2) | 0x00000004 | 0xb0000200 | 0x00000200 | 0x00000004 |
| mpe:lmuram | not_cached | .bss.core0_main.sembuf (3) | 0x00000004 | 0xb0000204 | 0x00000204 | 0x00000004 |
+-----+-----+-----+-----+-----+-----+
training_tc27xC_uniform_mc.map
-----
| Chip | Group | Section | Size (MAU) | Space addr | Chip addr | Alignment |
-----+-----+-----+-----+-----+-----+
| mpe:dspr0 | | istack_tc0 (760) | 0x00000400 | 0x70000008 | 0x00000008 | 0x00000008 |
| mpe:dspr0 | | ustack_tc0 (759) | 0x00004000 | 0x70005000 | 0x00005000 | 0x00000008 |
| mpe:pspr0 | | .text.private0.core0_main.core0_main (1) | 0x000000a2 | 0x70100000 | 0x0 | 0x00000002 |
  
```

Figure 9 - Inspecting Map File for Correct Memory Mapping

²¹ "TASKING® VX-toolset for TriCore™ User Guide", section 16.8.2, "Creating and Locating Groups of Sections"

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

The map file confirms that `core0_main` is located in `PSPR0` and that it has a rom copy residing in `pflash0`. Also, note that shared critical sections have indeed been located in `lmmuram` using its non-cached memory mapping.

3. EXECUTION

We're all set to start testing the run-time behavior. For our testing we used the integrated TASKING® debugger and the Infineon AURIX™ Application Kit for TC277²². Let's look at two circumstances. One with `sembuf` in place, and the other without.

3.1 DETERMINISTIC BEHAVIOR

When semaphore `sembuf` is in place the run-time results are as follows.

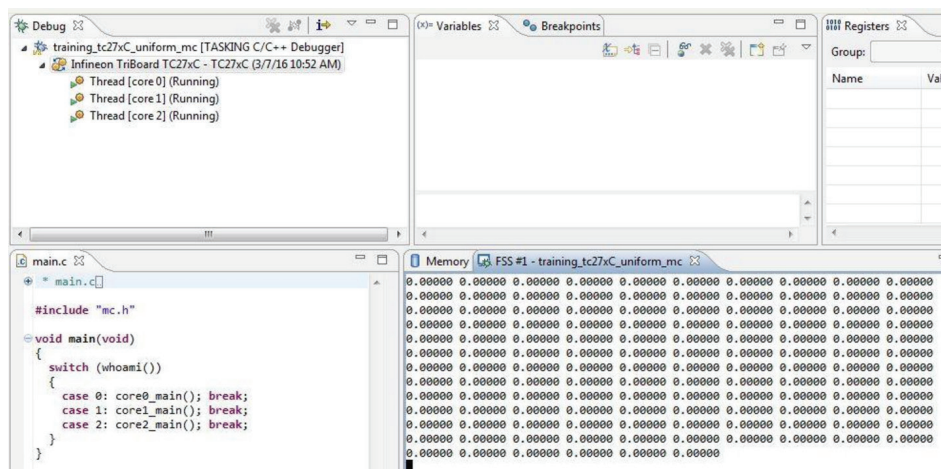


Figure 10 - Run-Time Results for Semaphore `sembuf`

These results are in line with what we predicted them to be; the aggregated addition of two tasks (cores) doing opposite additions for an equal amount of operations must be zero. Since all cells with `samples` were initially cleared, it means that afterwards they will still be cleared or fractionally zero.

3.2 NONDETERMINISTIC BEHAVIOR

Now look at what happens when commenting synchronization primitives `P()` and `V()` from your source code.

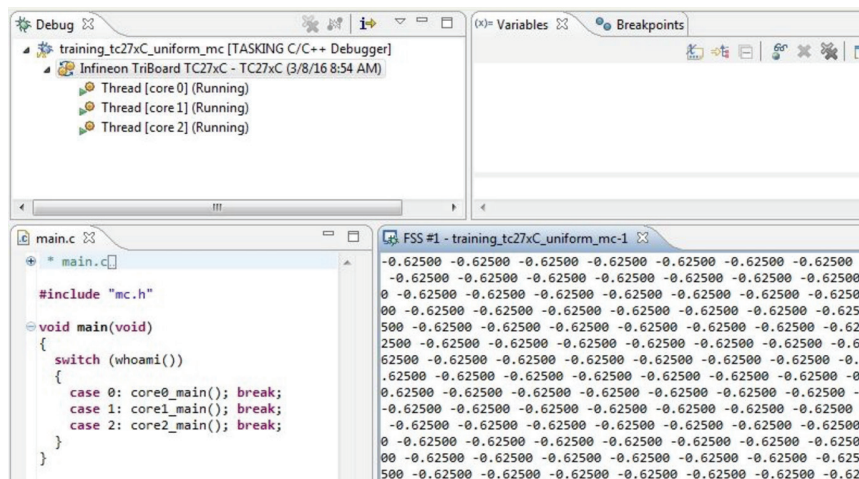


Figure 11 - Run-Time Results After Commenting Synchronization Primitives

²² For your own testing we advise to use any of the officially supported AURIX™ TriBoards since this application kit is not yet supported.

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

Note that in this case all cells end up to be fractionally -0.625 which is an exact multiple of -10 times the value of GAMMA listed in `mc.h`. Let's try to explain this race condition by dissecting the assembly of the following two C statements:

```
* (pwc1++) += GAMMA; /* tc1 */
* (pwc2++) -= GAMMA; /* tc2 */
```

For the former, the assembly looks like this:

```
ld.da  a6/a7,pwc1
ld.q   d0,[a6/a7+c]0
.L66:
movh   d1,#2048
.L67:
adds   d0,d1
st.q   [a6/a7+c]2,d0
.L68:
st.da  pwc1,a6/a7
.L41:
```

For the latter, it looks like this:

```
ld.da  a6/a7,pwc2
ld.q   d0,[a6/a7+c]0
.L66:
movh   d1,#2048
.L67:
subs   d0,d1
st.q   [a6/a7+c]2,d0
.L68:
st.da  pwc2,a6/a7
.L41:
```

Now suppose both `pwc1` and `pwc2` point to address `0xb0000000`. Let's further suppose that `tc1` has just loaded `pwc1` in `a6/a7` and `tc2` is about to do the same for `pwc2`, as indeed might be the circumstance if they start concurrently. Then, you can draw up the following sequence:

tc1		tc2		tc1				tc2				shared
		pwc1	a6:a7	d0	d1	pwc2	a6:a7	d0	d1	[0xb0000000]		
ld.da	a6/a7,pwc1	0xb0000000	0xb0000000	-	-	0xb0000000	-	-	-	0.0000		
ld.q	d0,[a6/a7+c]0	0xb0000000	0xb0000000	0.0000	-	0xb0000000	0xb0000000	-	-	0.0000		
movh	d1,#2048	0xb0000000	0xb0000000	0.0000	0.0625	0xb0000000	0xb0000000	0.0000	-	0.0000		
adds	d0,d1	0xb0000000	0xb0000000	0.0625	0.0625	0xb0000000	0xb0000000	0.0000	0.0625	0.0000		
st.q	[a6/a7+c]2,d0	0xb0000000	0xb0000002	0.0625	0.0625	0xb0000000	0xb0000000	-0.0625	0.0625	0.0625		
st.da	pwc1,a6/a7	0xb0000002	0xb0000002	0.0625	0.0625	0xb0000000	0xb0000002	-0.0625	0.0625	-0.0625		

Figure 12 - Results of tc1 and tc2 Loaded into pwc1 and pwc2

AURIX™ MULTI-CORE TRICORE™ PROGRAMMING ESSENTIALS

Note that on the far right you see the contents of address `0xb0000000`, which is just one cell of the `samples` buffer. Also, note that the value of `0.0625`, written back by `tc1`, is immediately overwritten by `-0.0625` of `tc2`. This behavior will repeat itself for each sweep through the `samples` buffer. Thus, after 10 iterations it becomes `-0.625` as you see in the slides.

Some might say that this also is deterministic behavior since we can deduce what happens and why it happens. However, it is nondeterministic nonetheless because the slightest change in relative speed between the cores makes it go away, as we have also observed during our testing. So, conversely it can be dormant for a long time before suddenly being triggered. It shows the necessity for semaphores for crucial non atomic operations.

CONCLUSION

In this application note, we showed the design steps that come into play when developing an AURIX™ multi-core TriCore™ application. We observed how the New Project wizard assists in making your best initial project settings, only leaving a few tweaks to do for yourself. We pondered some of the philosophical aspects, such as why TASKING® implements synchronizing startup codes. Or why a single main entry point still applies, even when having multiple cores running in parallel. We implemented a small multi-core design and in doing so had to deal with synchronization and arbitration. For this we used intrinsics and language extensions specifically tailored for the AURIX™ architecture. We subsequently used the integrated debugger to debug the application running on all cores, and investigated the consequences of using and not using semaphores.

In summary, we hope that it has been educational and that this multi-core essentials application note has given you a first glance into the challenging and sometimes complex world of multi-core development.