GETTING STARTED WITH FREERTOS APPLICATION PROGRAMMING ON XCORE USING THE AIOT SDK

WHITEPAPER

JUNE 2021

RATIONALE

Traditionally, xcore multi-core processors have been programmed using the XC language. The XC language allows the programmer to statically place tasks on the available hardware cores and wire them together with channels to provide inter-process communication. The XC language also exposes "events," which are unique to the xcore architecture and are a useful alternative to interrupts.

Using a combination of tasks statically placed on hardware cores, channels, and events, it is possible to write software with deterministic timing, and with very low latency between I/O and software, as well as between tasks.

While XC elegantly enables the intrinsic, unique capabilities of the xcore architecture, there often needs to be higher level application type software running alongside it. The features and approaches that make lower level deterministic software possible may not be best suited for those parts of an application that do not require deterministic timing. Where strict real-time execution is not required, higher level abstractions can be used to manage finite hardware resources, and provide a more familiar programming environment.

A symmetric multiprocessing (SMP) real time operating system (RTOS) can be used to simplify xcore application designs, as well as to preserve the hard real-time benefits provided by the xcore architecture for the lower level software functions that require it.

This document assumes familiarity with real time operating systems in general. Familiarity with FreeRTOS specifically should not be required, but will be helpful. For current up to date documentation on FreeRTOS see the following links on the FreeRTOS website.

- Overview
- Developer Documentation
- API

1. SMP FREERTOS

To support this new programming model for xcore, XMOS has extended the popular and free FreeRTOS kernel to support SMP (now upstreamed to Amazon Web Services). This allows for the kernel's scheduler to be started on any number of available xcore logical cores per tile, leaving the remaining free to support other program elements that combine to create complete systems. Once the scheduler is started, FreeRTOS threads are placed on cores dynamically at runtime, rather than statically at compile time. All the usual FreeRTOS rules for thread scheduling are followed, except that rather than only running the single highest priority thread that is ready at any given time, multiple threads may run simultaneously. The threads chosen to run are always the highest priority threads that are ready. When there are more threads of a single priority that are ready to run than the number of cores available, they are scheduled in a round robin fashion.

SMP SPECIFIC CONSIDERATIONS

Programming an application for a multiprocessor environment using an SMP RTOS is very similar to programming for a single processor environment using an RTOS. Most of the time, it is almost

identical and the fact that there are multiple cores available to the threads is a detail that the programmer does not need to worry about. However, there are some differences that the programmer must take into account to avoid race conditions which do not exist when there is only a single processor core available to the RTOS.

It is possible for multiple threads to run simultaneously on different cores. This is obvious, and is the point of an SMP RTOS. But it may not be immediately obvious why this requires special consideration above what must already be considered when programming for a multi-threaded, but single processor, environment.

The first big difference that this introduces is that it is now possible for threads with different priority levels to run simultaneously. In a single core environment, when two threads with different priorities share a data structure, it is not necessary for the higher priority one to enter a critical section when using it. This is no longer true in a multiprocessor environment. Any instance where a thread assumes that a lower priority thread will not run is no longer valid when using an SMP RTOS.

The second big difference is with interrupt service routines (ISRs). In a single processor environment, ISRs cannot run simultaneously either with each other (although there is a similar issue for architectures that support interrupt priority levels and nesting) or with application threads. Of course, all of this is possible in a multiprocessor environment. So, there must be a way to ensure mutual exclusion for access to data structures that are shared both between multiple ISRs, as well as between ISRs and threads.

FreeRTOS already provides the macro functions taskENTER_CRITICAL_FROM_ISR() and taskEXIT_CRITICAL_FROM_ISR() for use with ports for architectures that support interrupt nesting. The SMP FreeRTOS port for xcore makes use of these and uses an xcore hardware lock under the hood. Be sure to remember to use these in ISRs around access to data that is shared with threads and requires mutual exclusion. The corresponding task version macro functions taskENTER_CRITICAL() and taskEXIT_CRITICAL() must be called by threads that access this shared data. The task version both disables interrupts on the calling core, as well as obtains the lock.

NEW FEATURES

Two new APIs have been added to FreeRTOS to support SMP and xcore. Similar capability is also found in other RTOSes that support SMP.

1. The first allows a FreeRTOS thread to be excluded from any number of cores. This is done with a core exclusion mask. This supports various scenarios.
    - One common scenario is having a task that fully utilises the xcore architecture and requires deterministic execution. Most FreeRTOS applications, however, require a timer interrupt that runs periodically, typically once every 1 or 10 milliseconds. The xcore SMP FreeRTOS port always places this timer interrupt on core 0[1]. When execution of this interrupt's service routine breaks the timing assumption made by tasks that require deterministic execution, and it is not feasible to disable interrupts around its critical sections, then it can make sense to exclude these tasks from core 0.

1. This is not necessarily core 0 as returned by get_logical_core_id() found in xs1.h. SMP FreeRTOS maintains its own core ID numbering for the cores that it resides on. For the SMP RTOS core ID value, use rtos_core_id_get() instead.

- Another scenario is when there are two or more “legacy” threads written with the assumption that they are running in a single core environment. It is common to find that the higher priority threads will often not enter a critical section when modifying data structures shared with lower priority threads, as it is not possible for the lower priority threads to pre-empt the higher priority threads. While this is still true in an SMP environment, it is possible that the lower priority thread can run simultaneously in another core. Therefore, additional protection must be added (see the discussion above about this). When it is not possible to modify the code to add this protection, for example when the functions are part of a third party library, then it can make sense to lock all of these threads to a single core, ensuring that they do not run simultaneously.

The two new functions to support this are:

```
void vTaskCoreExclusionSet( const TaskHandle_t xTask, UBaseType_t uxcoreExclude )
```

This function sets the specified thread’s core exclusion mask. Each bit position represents the corresponding core number, supporting up to 32 cores. Subsequent to the call, the task will be prevented from running on any core whose corresponding bit in the mask is set to 1.

```
UBaseType_t vTaskCoreExclusionGet( const TaskHandle_t xTask )
```

This function returns the specified thread’s current core exclusion mask.

The second new feature allows pre-emption to be disabled at runtime on a per thread basis. Global pre-emption may still be disabled at compile time with the configuration option configUSE_TASK_PREEMPTION_DISABLE.

This allows threads to ensure that they are never pre-empted by another lower or same priority task. This can be useful for tasks that require deterministic execution but that do not necessarily need to be run at the highest priority level. For example, a thread that spends much of the time blocked in a waiting state, but once woken up and running must not be interrupted. Disabling interrupts within these tasks may also be required, but by additionally disabling pre-emption the scheduler will not even attempt to pre-empt it, ensuring that other threads continue running as they should.

The two new functions to support this are:

```
void vTaskPreemptionDisable( const TaskHandle_t xTask )
```

This function disables pre-emption for the specified thread.

```
void vTaskPreemptionEnable( const TaskHandle_t xTask )
```

This function enables pre-emption for the specified thread.

Aside from the above additions, the API is identical between the single core FreeRTOS kernel and the SMP FreeRTOS. Any code that has been written for single core FreeRTOS should compile and work under SMP FreeRTOS. Just be aware of the single core assumption that is occasionally made and account for it as necessary.

2. XCORE RTOS DRIVERS

To help ease development of xcore applications using an SMP RTOS, XMOS provides several SMP RTOS compatible drivers. These include, but are not necessarily limited to:

- Common I/O interfaces
  - GPIO
  - I²C
  - I²S
  - PDM microphones
  - QSPI flash
  - SPI
  - USB
- xcore features
  - Intertile channel communication
  - Software defined memory (xcore.ai only)
- External parts
  - Silicon Labs WF200 series WiFi transceiver

These drivers are all found in the AIoT SDK under the path modules/rtos/drivers.

Documentation on each of these drivers can be found under the References/RTOS Drivers section in the AIoT SDK documentation pages.

It is worth noting that these drivers utilize a lightweight RTOS abstraction layer, meaning that they are not dependent on FreeRTOS. Conceivably they should work on any SMP RTOS, provided an abstraction layer for it is provided. This abstraction layer is found under the path modules/rtos/osal. At the moment the only available SMP RTOS for xcore is the SMP FreeRTOS, but more may become available in the future.

XMOS also includes some higher level RTOS compatible software services, some of which the aforementioned drivers. These include, but are not necessarily limited to:

- DHCP server
- FAT filesystem
- HTTP parser
- JSON parser
- MQTT
- SNTP client
- TLS
- USB stack
- WiFi connection manager

These services are all found in the AIoT SDK under the path modules/rtos/sw_services.

3. RTOS APPLICATION DESIGN

A fully functional example application that demonstrates usage of a majority of the available drivers can be found in the AIoT SDK under the path examples/freertos/independent_tiles. In addition to being a reference for how to use most of the drivers, it also serves as one example for how to structure an SMP RTOS application for xcore.

This example application runs two instances of SMP FreeRTOS, one on each of the processor's two tiles. Because each tile has its own memory, which is not shared between them, this can be

viewed as a single asymmetric multiprocessing (AMP) system that comprises two SMP systems. A FreeRTOS thread that is created on one tile will never be scheduled to run on the other tile. Similarly, an RTOS object that is created on the tile, such as a queue, can only be accessed by threads and ISRs that run on that tile and never by code running on the other tile.

That said, the example application is programmed and built as a single coherent application, which will be familiar to programmers who have previously programmed for the xcore in XC. Data that must be shared between threads running on different tiles is sent via a channel using the RTOS intertile driver, which under the hood uses a streaming channel between the tiles.

Most of the I/O interface drivers in fact provide a mechanism to share driver instances between tiles that utilizes this intertile driver. For those familiar with XC, this can be viewed as a C alternative to XC interfaces.

For example, a SPI interface might be available on tile 0. Normally, initialization code that runs on tile 0 sets this interface up and then starts the driver. Without any further initialization, code that runs on tile 1 will be unable to access this interface directly, due both to not having direct access to tile 0's memory, as well as not having direct access to tile 0's ports. The drivers, however, provide some additional initialization functions that can be used by the application to share the instance on tile 0 with tile 1. After this initialization is done, code running on tile 1 may use the instance with the same driver API as tile 0, almost as if it was actually running on tile 0.

The example application referenced above, as well as the RTOS driver documentation, should be consulted to see exactly how to initialize and share driver instances.

The AIoT SDK provides the ON_TILE(t) preprocessor macro. This macro may be used by applications to ensure certain code is included only on a specific tile at compile time. In the example application, there is a single task that is created on both tiles that starts the drivers and creates the remaining application tasks. While this function is written as a single function, various parts are inside #if ON_TILE() blocks. For example, consider the following code snippet found inside the task vApplicationDaemonTaskStartup():

```
#if ON_TILE(I2C_TILE)
{
    int dac_init(rtos_i2c_master_t *i2c_ctx);
    if (dac_init(i2c_master_ctx) == 0) {
            rtos_printf("DAC initialization succeeded\n");
            dac_configured = 1;
    } else {
            rtos_printf("DAC initialization failed\n");
            dac_configured = 0;
    }
    chan_out_byte(other_tile_c, dac_configured);
}
#else
{
    dac_configured = chan_in_byte(other_tile_c);
}
#endif
```

When this function is compiled for tile I2C_TILE, only the first block is included. When it is compiled for the other tile, only the second block is included. When the application is run, tile I2C_TILE performs the initialization of the DAC, while the other tile waits for the DAC initialization to complete.

I2C_TILE is defined at the top of the file. Because the I2C driver instance is shared between the two tiles, it may in fact be set to either zero or one, providing a demonstration of the way that drivers instances may be shared between tiles.

The AIoT SDK provides a single XC file that provides the main() function. This provided main() function calls main_tile0() through main_tile3(), depending on the number of tiles that the application requires and the number of tiles provided by the target xcore processor. The application must provide each of these tile entry point functions. Each one is provided with up to three channel ends that are connected to each of the other tiles.

The example application provides both main_tile0() and main_tile1(). Each one calls an initialization function that initializes all the drivers for the interfaces specific to its tile. These functions also call the initialization functions to share these driver instances between the tiles. These initialization functions are found in the board_init.c source file.

Each tile then creates the vApplicationDaemonTaskStartup() task and starts the FreeRTOS scheduler. The vApplicationDaemonTaskStartup() task completes the driver instance sharing and then starts all of the driver instances. Additional application demo tasks are created before vApplicationDaemonTaskStartup() completes by deleting itself.

The application may be experimented with by modifying the *RPC_ENABLED macros in board_init.h, as well as the *_TILE macros at the top of main.c. RPC here stands for Remote Procedure Call, and is what allows for driver instances to be shared. Provided RPC is enabled for a particular driver, it may be used by either tile and the corresponding *_TILE macros for it may be set to either tile. However, if RPC is disabled then note that when the corresponding *_TILE macro is not set to the tile that owns the instance, the application will fail.

Consult the RTOS driver documentation for the details on what exactly each of the RTOS API functions called by this application does.

For a more interesting application that does more than just exercise the RTOS drivers see the example application under the path examples/freertos/explorer_board. This application does not provide as complete an example of how to use and share all of the drivers, but does utilize many of the software services.

4. BUILDING RTOS APPLICATIONS

RTOS applications using the AIoT SDK are built using CMake. The AIoT SDK provides many drivers and services, all of which have .cmake files which can be included by the application's CMakeLists.txt file. The application's CMakeLists can specify precisely which drivers and software services within the AIoT SDK should be included through the use of various CMake options.

The example applications also provide a Makefile that actually runs CMake and then runs make with the generated CMake makefiles. This is done to automate the steps that must be taken to build for more than one tile. The Makefile actually runs CMake once per tile. Each tile is built independently, and the two resulting binaries are then stitched together by the Makefile.

By simply running:

```
$ make -j
```

in the example application directories, all the steps necessary to build the entire application are taken, and a single binary that includes both tiles will be found under the bin directory. If the xcore board is connected to the computer via an xTag, running:

$ make run

will run it on the board with xscope enabled so that all debug output from the application will be routed to the terminal.

To access the SMP FreeRTOS, please visit https://www.freertos.org/. To find out more about our xcore processors, please visit https://www.xmos.com/processors/.