HighIntegritySystems

WITTENSTEIN

Building on FreeRTOS for Safety Critical Applications

Issue 1 - November 12, 2020

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600

email: sales@highintegritysystems.com
web: www.highintegritysystems.com

WITTENSTEIN

CONTENTS

List of Figures

List of Notation

API	Application Programming Interface
DAP	Design Assurance Pack
DHF	Design History File
MCU	Microcontroller Unit
MPU	Memory Protection Unit
MMU	Memory Management Unit
RTOS	Real Time Operating System
SIL	Safety Integrity Level
TCB	Task Control Block

CHAPTER 1 Introduction

FreeRTOS™ is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use.

Increasing safety regulations in markets such as automotive, medical, and industrial means that designers require a RTOS that is certifiable to the relevant industry standard, however this can be an expensive outlay at the start of a long safety project.

This then, is where the strength of FreeRTOS lies in prototyping. Start work with FreeRTOS, and migrate to safety critical SAFE**RTOS**® when ready. SAFE**RTOS** is a safety critical RTOS that has been pre-certified to IEC 61508 and ISO 26262 ASILD by TÜV SÜD. It follows the same functional model as FreeRTOS, but has been built for safety for the outset, meaning that it is familiar to use, but is entirely reliable and pre-deterministic.

This white paper uses a very brief example project to illustrate the basics of migrating a FreeRTOS application to a SAFE**RTOS** platform.

The example projects are all targeted at the NXP Freedom K64F development board, chosen for its cost effectiveness and wide availability. It is well-supported by a comprehensive toolchain and IDE that is both free and multi-platform. The projects were created using NXP's Windows based MCUXpresso IDE in conjunction with online SDK support for the Freedom board, using the GCC compiler.

This white paper will run through three projects in turn. These projects are freely available for download from www.highintegritysystems.com SAFE**RTOS** Download Centre.

The three projects:

- 1_FreeRTOS_Demo – the original FreeRTOS project.
- 2_RTOS_Demo – the privileged mode SAFE**RTOS** version of the project.
- 3_RTOS_UnprvDemo – The SAFE**RTOS** project with some unprivileged tasks.

CHAPTER 2 Comparing FreeRTOS™ and SAFE**RTOS**®

2.1 A Shared History

Richard Barry created FreeRTOS while an employee at WHIS. Seeing its potential, WHIS engineers worked closely with Richard. They took the FreeRTOS functional model, exposed it to a full HAZOP, identified all areas of weakness within the functional model and API, mitigated all areas of weakness, and took the resulting requirements set through an IEC 61508 SIL 3 development life cycle, the highest possible for a software only component.

In doing so SAFE**RTOS** and its accompanying Design Assurance Pack was created: the renowned safety critical RTOS that delivers superior performance and safety critical dependability whilst consuming minimal resources.

The success of these endeavours can be judged by the fact SAFE**RTOS** was independently certified on the first iteration by TÜV SÜD back in 2007. SAFE**RTOS** has continuously evolved since its initial creation and the list of supported processors and toolsets is constantly expanding.

As the FreeRTOS kernel and SAFE**RTOS** share the same functional model upgrading is easy. Many developers prototype using the FreeRTOS kernel, and convert to SAFE**RTOS** at the start of their formal development phase.

2.2 Key Differences

SAFE**RTOS** and FreeRTOS share a functional model, so feel very similar to use. There are however some key differences. Compared with FreeRTOS, SAFE**RTOS**:

- Has fewer Application Programming Interface (API) functions.
- Does more error checking.
- Returns a status code from most API calls (with other return data passed by reference).
- Requires the application to supply all stack, task control block and queue buffers.
- Strongly encourages 100% static allocation and provides no “heap” functions.
- Uses a processor’s Memory Protection Unit (MPU) by default.
- Has been completely re-engineered to meet safety critical software needs.

Because of this, when migrating a FreeRTOS project to SAFE**RTOS** there is work to do to get the kernel up and running.

FreeRTOS hides much routine memory management “under the hood”, dynamically allocating stack buffers as tasks are created, allocating key kernel buffers when the kernel is started, and so on. It is possible to configure FreeRTOS to have all these things statically allocated and supplied by the application, but most people settle for the simpler approach of letting FreeRTOS do it.

FreeRTOS also provides a lot of compile-time options, and allows the application designer to insert extra functionality into the kernel via strategically placed “hook macros”, so that code can be run as tasks are switched in or out, for example, or when they are deleted or created.

2.2.1. API Differences

WHIS offer a free Application Note that goes into some detail about the main differences between the FreeRTOS and SAFE**RTOS** API, that is freely available from the WHIS website Download Centre www.highintegritysystems.com

For a complete description of the SAFE**RTOS** API, the product- specific manual that ships with every licensed copy of SAFE**RTOS** is definitive.

The RTOS-defined abstract type names differ between the two platforms, as do the necessary RTOS #include files. With FreeRTOS, an application file needs to #include headers for each of the main parts of the API that it uses – task functions, queue functions, semaphore functions, etc. – while with SAFE**RTOS**, an application file need only #include a single RTOS header file, ***`SafeRTOS_API.h`***.

2.2.2. Static Allocation and the MPU

SAFE**RTOS** requires the application engineer to supply all the buffers that are needed to maintain task and object state, up to and including things that might reasonably be regarded as very much involved with the internal workings of the kernel.

For example, both FreeRTOS and SAFE**RTOS** create a kernel "idle task" at the lowest possible task priority to ensure that there is never a situation where there is no ready task that can be scheduled. In both cases, the idle task mustn't ever block, and in both cases there are mechanisms by which the idle task can be used for "background" application functions, or for power saving measures, or other application-specific purposes.

Obviously the idle task needs a task stack, and also a task control block. FreeRTOS allocates these from heap (unless explicitly configured for static allocation) while SAFE**RTOS** requires the application to supply these (and other) buffers.

Part of the reason for this is that static allocation is generally preferred in safety critical systems as it's easier to make a case that there is sufficient memory for all run-time circumstances, but another consideration is the fact that the vast majority of SAFE**RTOS** ports presume the use of some kind of MPU.

Working with the MPU implies that the application designer has a conscious oversight of the exact placement in address space of all memory structures – and the kernel task and queue buffers are obviously an important part of that. Moreover, an MPU usually imposes restrictions on the alignment and size of protection regions, and the application engineer needs the flexibility to meet these restrictions, which often means arranging things carefully to avoid inefficient use of the available space.

So where with FreeRTOS, it's usually enough to make sure there's plenty of free space on the heap before calling xTaskCreate, with SAFE**RTOS**, you need to pre-allocate and explicitly locate a carefully sized and aligned stack buffer, and statically allocate and locate a buffer for the task TCB, then pass pointers to these structures into xTaskCreate along with the usual task parameters.

2.2.3. Task Privilege and Kernel Function Wrappers

Each SAFE**RTOS** task is assigned an operating privilege, of which there are usually two. "Privileged" tasks have the same basic powers as kernel code. Most CPUs that allow privileged and unprivileged modes have a restricted range of instructions available in unprivileged modes, and a limited mechanism – usually some kind of software trap, exception or interrupt – for moving between un- privileged and privileged mode operation.

It's generally regarded as good practice to code as much of an application as possible to run in unprivileged mode, and to that end, each task is provided with a set of MPU parameters that are switched in and out of (usually) a fixed range of MPU region descriptor slots whenever the task is switched in and out by the scheduler.

The SAFE**RTOS** task creation adds an additional MPU region for each task that grants it access to its own stack, and this is automatic, although the application programmer still has to ensure that the task's stack buffer conforms to the restrictions imposed by the MPU regarding alignment and size.

Unprivileged tasks also need to have execute access to the kernel API functions. Since these are usually expected to be called only from privileged mode, every API function with SAFE**RTOS** has a privilege-escalating wrapper. Typically, this works by raising an exception of some kind – most often using a "system call" – a synchronous interrupt or trap facility provided by the CPU for this purpose. The API wrapper will raise a task's privilege if need be, execute the API function, then

lower the privilege again if it was raised.

The way in which this is done in the SAFE**RTOS** code can, depending on the debugger, make debugging awkward at times, as the actual API functions all have names that differ from the name by which they are usually called.

Although FreeRTOS can support the MPU using a similar scheme of privilege-escalating wrappers, this is generally limited to a few MPU-aware portable layers, and the way in which privileged and unprivileged tasks are regarded differs. In SAFE**RTOS**, we presume that application tasks will normally run unprivileged. With the FreeRTOS MPU-aware ports, tasks are presumed generally to run at kernel privilege, as they do in non-MPU ports, but tasks may optionally be created “restricted”, that is, unprivileged.

CHAPTER 3 Getting Started

The example projects could readily be adapted to other platforms and development tools, although obviously there would be more work involved, and, if the target MCU is changed away from the Kinetis K64F or a closely similar platform, the RTOS portable layer will need to be changed. In the case of the FreeRTOS example, this would simply involve obtaining the relevant FreeRTOS port. The SAFE**RTOS** examples, however, use a library version of the kernel and that is fixed on the Kinetis K64F target – if you don't have a SAFE**RTOS** license for your proposed alternative target, then you'll need a library version of SAFE**RTOS** that is designed for that target, perhaps from one of the other web demos.

Note that although the K64F is an ARM Cortex-M4 based device, it doesn't have the standard ARM Memory Protection Unit (MPU), but rather a Kinetis MPU, so the supplied kernel library will not run on generic Cortex-M based microcontrollers.

For the purposes of this Application Note, it will be assumed that you are using the intended target and IDE/toolchain.

3.1 Setting up your Example Project

3.1.1 The Workshop Projects

The example projects referred to through this white paper are available to download from the WHIS Download Centre at www.highintegritysystems.com. For simplicity, this white paper shall refer to this downloadable as the "Workshop Demo" from now on.

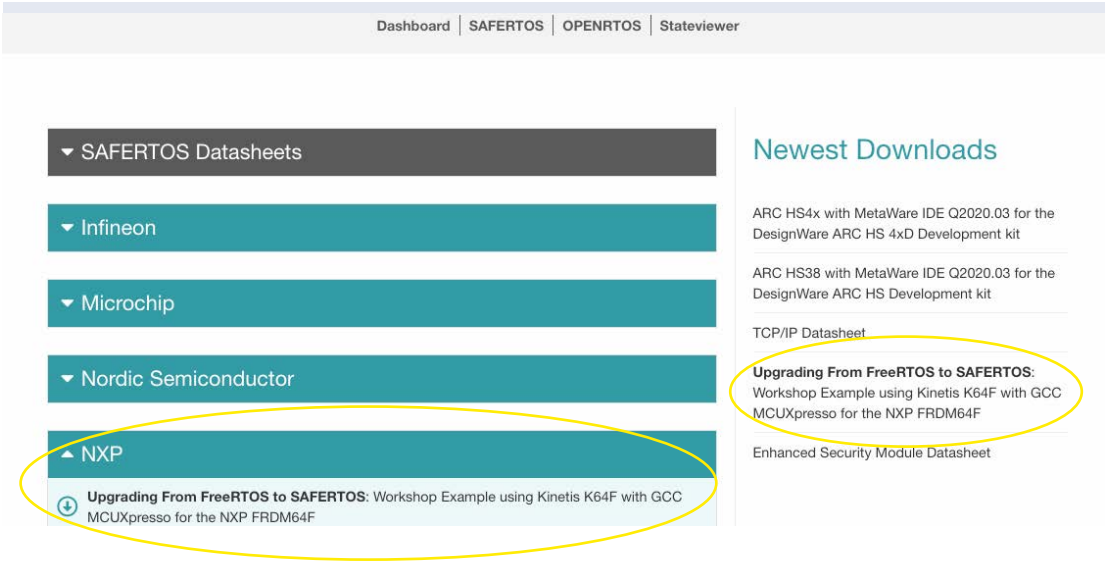

Figure 3-1 *The Workshop Demo in the WHIS Download Centre*

3.1.2. The Development Board

To follow along with this project you will need:

- An NXP FRDM K64F development board. These are available from various vendors and cost in the region of 40GBP at time of writing. There are cheaper FRDM boards available but the K64F has an MPU and may be used to develop for a variety of other Kinetis microcontrollers.
- A micro-USB cable for debug & power.
- The example projects included in this package.
- MPUXpresso IDE. The example projects were created using v11.1 of this IDE.

Before starting this project it is advisable to download MCUXpresso and familiarize yourself with basic project creation using the device-aware project wizard.

Figure 3-2 *The NXP FRDM K64F*

CHAPTER 4 Start Development with FreeRTOS

The example projects included in the Workshop Demo all began as wizard-generated projects that included "Amazon FreeRTOS" among the board support software/middleware. As an alternative you could simply start with a bare C project featuring just startup code and an empty main(), and add in FreeRTOS yourself, but the generated basic FreeRTOS project is usually quicker to set up. The following refers to Project 1 in the Workshop Demo: "1_FreeRTOS_Demo – the original FreeRTOS project."

4.1 The Auto-generated Linker Files

Having built the basic project once, the generated linker files were moved out of the "Debug" build directory into their own directory, and the project options altered to turn off the automatic generation of linker files, and to point at the new linker file directory. This is because for the SAFE**RTOS** projects, the linker files need to be modified, and we don't want them to be automatically regenerated and overwritten (see Figure 5-1).

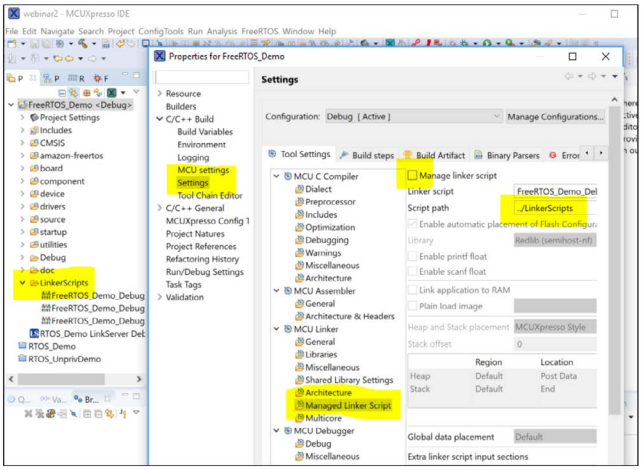

Figure 4-1 *Modifying the Linker Files*

4.2 The Application Code

The "application" code was then added to the FreeRTOS project, and the project built and run on the development board to verify that the code functioned as expected.

The "application" is a very simple program that cycles the tri-colour LED on the Freedom board through various shades, by controlling the brightness of each of the component red, green and blue LEDs via pulse-width modulation, using a separate task for each LED.

This is done in a deliberately simplistic and slightly convoluted way – the chip features various PWM peripherals, but the control tasks just do a rather slow and crude form of PWM by simply toggling a GPIO line on and off. The reason for this is purely to have a few tasks running and making use of the RTOS to switch between tasks at timed intervals and so on. An additional "control" task updates a global array of "target brightness" values for each of the three LEDs, and a mutex is used to police access to this shared global array. This is not by any means the best approach to communicating between tasks – indeed it could arguably be said to be among the worst – but the object here is to provide a very simple example that illustrates some key aspects of converting from FreeRTOS to SAFE**RTOS**.

Each of the LED tasks uses the same task code, but each task is created with a different "task parameter" that tells it which LED it is responsible for.

4.3 The LED Tasks

Each LED task maintains a local variable, `PulseWidth`, that indicates the "on" time for its LED, in milliseconds. The total PWM frame time is set by a hard-coded constant `LED_PWM_PERIOD_MS`, again in milliseconds.

Each time around its main loop, each LED task notes the current system tick count, then, If `PulseWidth` is non-zero, it turns its LED on, and then calls the RTOS task delay function to delay for `PulseWidth` milliseconds after its start time. This causes that task to enter the blocked state, allowing another task to run. When next the LED task wakes, it tries to obtain the mutex that controls access to the "brightness request" global array and, if successful, it adjusts `PulseWidth` to reflect the "brightness request" value (this will take effect next time around the loop).

Finally, if `PulseWidth` was originally less than the entire frame, i.e. if the LED was set to anything less than full brightness, then the LED is switched off and the task sleeps for the remainder of the frame time, before continuing around the loop for the start of the next frame.

4.4 The Control Task

The control task simply loops around, waking at fixed intervals determined by the constant `CTRL_RAMP_INTERVAL_MS`, and, according to three different "step" counts, it varies the target brightness of each of the three LEDs from minimum to maximum and back – the effect is that each LED sweeps from fully off to fully on at a fixed rate, but the rate is different for each LED, so the colour and brightness of the RGB LED changes in reasonably a smooth and apparently random way, as sometimes the peaks will coincide, and sometimes they won't.

CHAPTER 5 Migrating from FreeRTOS to SAFE**RTOS**

5.1 Two Stage Approach

The approach used to convert this very simple FreeRTOS project into a SAFE**RTOS** one is shown in two basic stages:

1. Convert the project to SAFE**RTOS**, with everything running in privileged mode.
2. "Lock down" tasks that needn't run in privileged mode.

5.1.1. Converting the Project to use SAFERTOS

For a project as simple as this, most of the work is in stage 1. We will:

1. Replace the FreeRTOS kernel code files with the SAFE**RTOS** ones;
2. Edit the linker file to export symbols required by the SAFE**RTOS** kernel;
3. Ensure that the SAFE**RTOS** interrupt and exception handlers are installed;
4. Create mandatory hook functions and static buffers for kernel tasks and queues;
5. Create a scheduler initialization struct and function;
6. Create required static buffers for application task TCBs, stacks, queues, etc;
7. Convert FreeRTOS API calls to equivalent SAFE**RTOS** ones;
8. Convert FreeRTOS-specific types to SAFE**RTOS** ones;
9. Make all tasks privileged and set default MPU parameters.

Note that steps 1 through 6 inclusive are essentially unavoidable one-off overhead for any SAFE**RTOS** application, whether converted from FreeRTOS or not. This is more work than for a FreeRTOS application (especially a wizard-generated one) but it follows a basic template that is easy enough to carry from one project to another.

Steps 7 and 8, particularly 8, can to some extent be speeded up using an IDE's search & replace functions, though beware of fundamental differences between the APIs that mean care is needed.

5.1.2. Making Tasks Unprivileged

Making tasks run in an unprivileged manner varies in complexity according to the tasks. It can be as simple as marking a task unprivileged, or it can involve considerable extra work.

5.2 Starting to Convert the Application

5.2.1. Replace the FreeRTOS Code

The first step is straightforward, we just delete everything under ***amazon-freertos*** and replace it with the SAFE**RTOS** kernel source. Or, in the case of this web-demo version of SAFE**RTOS**, replace it with the SAFE**RTOS** headers and library file. Here, we'll create a folder called ***SAFERTOS*** containing subfolder ***kernel***.

In the project options, all references in include paths to ***amazon-freertos/freertos_kernel*** need to be replaced by ***SAFERTOS/kernel.*** There are the expected entries in the compiler and assembler include paths under ***C/C++ Build/Settings*** (see Figure 6-1).

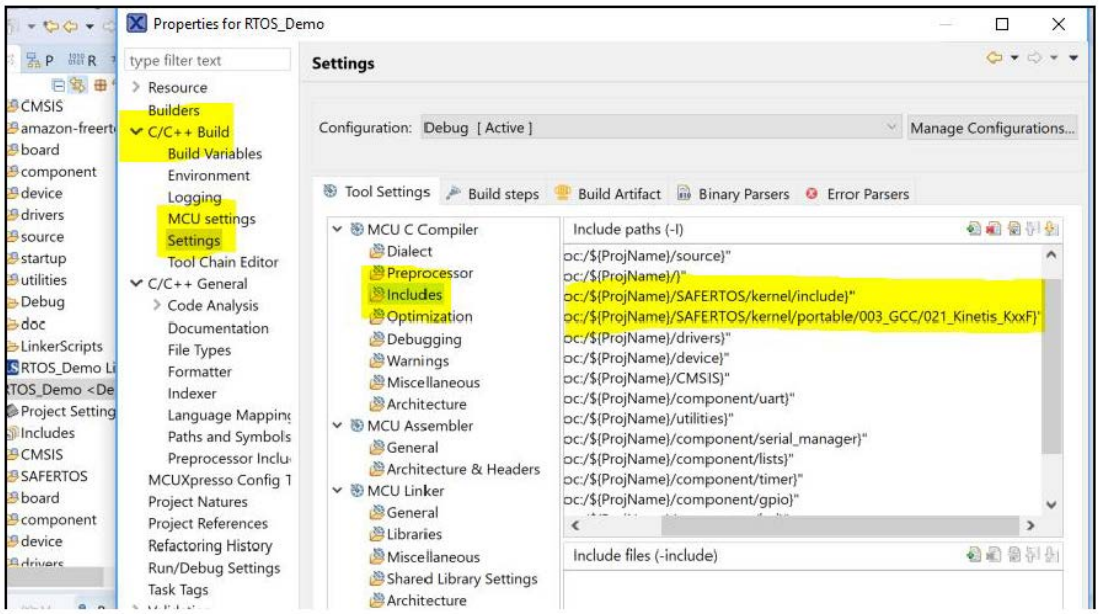

Figure 5-1 *Expected Entries in the Compiler and Assembler Include Paths*

In a wizard-generated project like this there are also some references under ***C/C++ General/Paths and Symbols***.

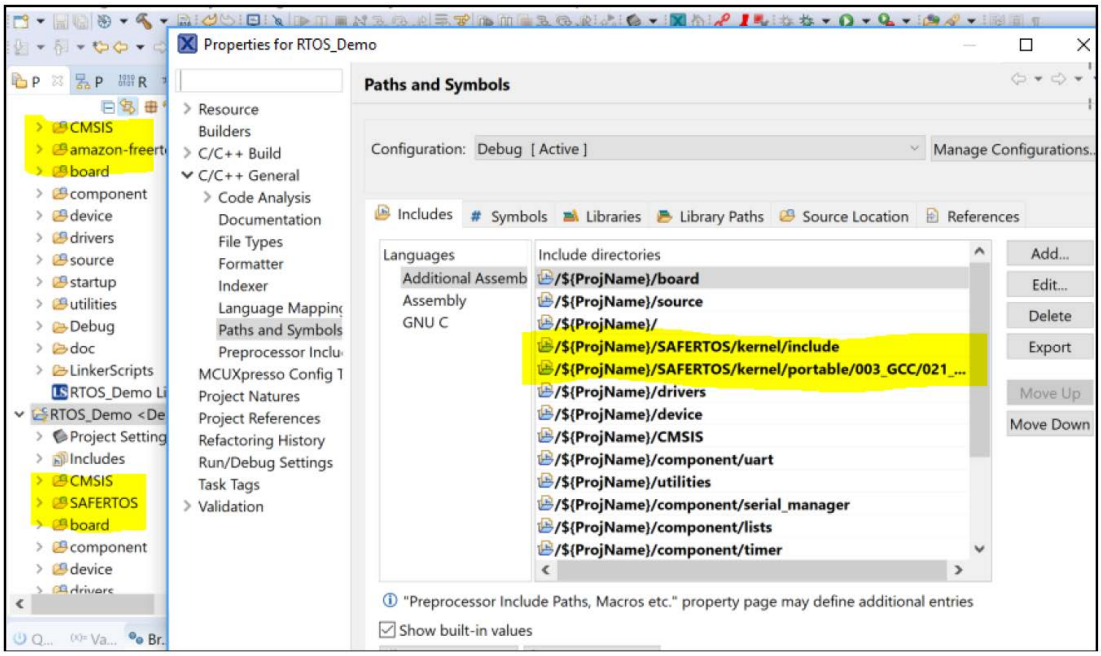

Figure 5-2 *References under C/C++ General/Paths and Symbols*

5.2.2 Edit the Linker File

SAFE**RTOS** requires some named sections to be defined in, and various symbols to be exported from, the linker file in order to set up initial MPU regions to protect kernel data and to allow access to general code.

The sections and symbols required are port specific, but you can see the MPU settings (and hence the required symbols) in the kernel portable layer header file ***portmpu.h.***

We can see we need a section for kernel functions called “kernel_func” and another for kernel data called “kernel_data”. The kernel functions and data are placed in these by section attributes.

The kernel functions are typically placed alongside the vector table so that both can be included in the MPU region that protects them, and the kernel data are placed somewhere in RAM that can be readily aligned to the needs of the MPU. In this case, the Kinetis MPU doesn’t have especially strict alignment requirements – regions need to be a minimum of sixteen byte aligned – so the placement and size are quite flexible.

In addition to the linker sections, the linker file needs to export symbols indicating where these sections start and finish, and we also need symbols for the start and end or size of ROM and RAM.

Again, from ***portmpu.h*** we can see we need:

- `lnkStartFlashAddress`
- `lnkEndFlashAddress`
- `lnkStartKernelFunc`
- `lnkEndKernelFunc`
- `lnkStartKernelData`
- `lnkEndKernelData`
- `lnkRAMEnd`
- `lnkRAMStart`

Each of these is declared as if it were an array of unsigned longs. When a symbol is defined and exported in the linker file, it represents an address, as with any other symbol. If you declare it as a variable and then reference it, you’ll get the contents of that address. We want the address itself, though, so we can either remember to use an address-of operator with these linker symbols, or declare them as arrays whereupon their unadorned name is a pointer to the declared type, i.e. the address assigned to that symbol.

In the linker file, we have statements like those shown in Figure 6-3.

Figure 5-3 *Linker File Statements*

5.2.3. Exception and interrupt handlers

SAFE**RTOS**, like FreeRTOS, makes use of certain interrupts and exceptions. The Kinetis K6xxF port needs to use the SysTick, PendSV and SVC interrupts, all of which are generally hooked into simply by using the CMSIS names for the handlers for these. If using CMSIS compliant platform support, these symbols are weakly linked and appear in the default vector table where they need to be, so that the RTOS simply has to name its own handlers according to the CMSIS convention.

In SAFE**RTOS**, we generally achieve this by simply #defining the SAFE**RTOS** names for the necessary handlers to be the CMSIS names instead, and we typically do this in ***SafeRTOSConfig.h.*** However, for this particular example, we're using a library build of SAFE**RTOS**, so the handler names are already fixed in the library, so we just reverse the trick and redefine the CMSIS names in the startup file where the vector table is defined, instead.

In this example project, the vector table is defined in ***startup_mk64f12.c,*** and we simply insert the following few lines near the beginning of that file:

```
/* SAFERTOS system tick, SVC and PendSV handlers */
#define SysTick_Handler         vTaskProcessSystemTickFromISR
#define SVC_Handler             vSafeRTOSSVCHandler
#define PendSV_Handler          vSafeRTOSPendSVHandler
```

Unlike FreeRTOS, SAFE**RTOS** checks the vector table entries at startup and will not run if they aren't present and correct.

5.2.4. Kernel hook Functions

SAFE**RTOS** doesn't have FreeRTOS's hook macros in the kernel. Instead, it provides for a limited set of "hook functions" that, if present, are called by the kernel when, for example, a task is deleted or the idle task runs or the tick interrupt fires.

Of the various hook functions, only one is mandatory, and that is the Error Hook Function. It's a requirement of safety critical systems that they can enter a safe failure state if an unrecoverable error is detected. If the kernel detects an error condition like a corrupted task control block or an overflowed stack, then it calls the Error Hook Function, which is responsible for making the system safe overall – turning off any motors or H-bridges, for example – and must not return. For our simple example application, we'll have an error hook function that simply loops indefinitely having made information about the error accessible to the debugger.

5.2.5. Kernel Task Buffers, Kernel Configuration and Startup

In addition to the address of one or more hook functions, we need to pass in the address and size of a stack buffer for use by the kernel idle task. There's also a second stack buffer required for the kernel timer task. Both the idle and timer tasks have some additional parameters that must be passed in, including their own MPU parameters, and there's also a timer command queue that needs a queue buffer of sufficient size, alignment etc.

All in all, the scheduler needs enough different buffers and pieces of information to get going, that there's a dedicated structure that is passed to a dedicated API call to configure the kernel.

Arguably, the most complicated part of the move to SAFE**RTOS** with a minimal application is getting the scheduler configured and started.

To avoid obfuscating the tiny demo application with so much more code, we've chosen to place the error hook function in a "hook functions" source file, and all the kernel configuration, including the stack and TCB allocations for kernel tasks, into another source file which we've named ***SafeRTOSConfig.c***, since it's a sort of extension, in a sense, of the kernel configuration header file ***SafeRTOSConfig.h.***

At each stage of the kernel startup, that is, when the configuration struct is passed in to the kernel configuration function and when the kernel is subsequently started, the API calls potentially return an error code.

Should either of these functions return an error code, it can be looked up in ***projdefs.h*** in the kernel include directory. The error codes for the entire API are listed, making it relatively easy to work out what went wrong if the kernel didn't start, for example. Usually if this happens it's because the proper IRQ handlers haven't been set up in the vector table(s), or one or other of the static kernel task structures – stack of TCB – is mis-aligned or too small, or a non-optional hook function – usually only the Error Hook – is missing, misaligned or mis-located.

5.2.5.1. Global MPU Regions

One final thing worth doing for this particular application is to use one or other of the "spare" MPU regions (that many SAFE**RTOS** ports provide) to set up a global, permissive region that allows access to the GPIOs that we're using to drive the on-board LEDs. This saves us having to grant explicit access to those to every task or call-back that might want to use them – the number of task- specific MPU regions is limited. If you look at the kernel configuration and startup code in SafeRTOSConfig.c, you'll see where we've done just that. There's a call to `xMPUConfigureGlobalRegion()` that sets up a "global" MPU region granting r/w access to the GPIO register space. (Which is the majority of peripheral address space in this case.)

5.2.6. Application Task TCBs and Stacks

We next need to set up the stack and TCB buffers for our individual application tasks, and to fill in the other task parameters.

Again, this involves allocating static buffers, having a suitable size and alignment for their intended use given MPU and other port-specific considerations.

In FreeRTOS, when we create tasks, we just pass all the necessary task parameters via six arguments to the `xTaskCreate()` function, which returns either the handle of the newly created task, or pdFAIL if it can't create the task. With SAFE**RTOS**, there are more task parameters, and some of them are port-specific, so they're passed to xTaskCreate() using single argument pointing to a port-specific `xTaskParameters` structure. Like most of the SAFE**RTOS** API, `xTaskCreate()` returns either pdPASS or an error code, so a second argument gives the address of a task handle variable to receive the handle of the newly-created task. If you don't need the task handle, this argument can be NULL.

In our example, many of the task parameters remain the same for all four of the tasks our application creates, so we save some space and time by re-using the same parameters structure and just changing the parameters that need to change.

This isn't best-practice as it would be relatively easy to overlook some vital change – for instance, it is imperative that each task has unique stack and TCB buffers (for obvious reasons).

Here, we see that the task name and parameter differs in each case, too, as in the FreeRTOS version.

Also among the task parameters are each task's privilege level and MPU region settings. As described earlier, in the first place we're going to make all tasks privileged, so none of them require any additional MPU parameters – not with this application on this target anyway – so we leave those set to zero/null.

5.2.7. API Function Call Changes

More or less the last thing that needs doing is for every API call, to check to see that the SAFE**RTOS** version has the same name and whether, or how, its prototype differs. In most cases, void functions will have become functions returning the architecture's signed base type, in order to report an error or pass code. Where the equivalent FreeRTOS function also returns a value, typically the SAFE**RTOS** version will return the value via a pointer type argument, and the function return value will usually be a status code. Some of the function names will have changed, and some of the FreeRTOS functions won't be available, so the application programmer will sometimes have to work around such omitted functions by using the underlying core functions in perhaps a slightly more long- winded way.

For the sake of form, where functions in SAFE**RTOS** return a value that we discard, the function should generally be prefixed with a void cast to indicate that this is deliberate.

5.2.8. Type Name Changes

As well as some of the API functions having different names, many of the port-layer defined type names also differ. Some are common to both FreeRTOS and SAFE**RTOS**, but many of the longer- established ones are not.

If developing FreeRTOS applications with a specific view to converting to SAFE**RTOS**, picking the type name options that are shared between the two, or are at least readily search-and-replaceable, will save some time.

5.2.9. Wrapping Up

After following the steps to now, we should have arrived at a point where the application will compile again and will run on the target system essentially the same as the FreeRTOS version.

If, as often happens in practice, the kernel refuses to start at this point, or a task creation call fails, the error codes returned should help to identify the problem. These codes are listed in the kernel header file ***projdefs.h***. As an example, the function that starts the kernel, `xTaskStartScheduler()`, is not supposed to return. If, rather than starting the scheduler, it instead returned the value -27, we could refer to ***projdefs.h*** and see that this is the code for errERROR_IN_VECTOR_TABLE, which means that the kernel start-up configuration checks have determined that one of the interrupt vectors essential to the SAFE**RTOS** kernel is misconfigured in some way.

5.3 Making Application Tasks Un-Privileged

In the final stage of conversion, we'll look at "locking down" the privileged tasks so that they can run un-privileged.

We've already granted access for any task to the GPIO registers we're using, via a "global MPU region."

Now let's see what else the tasks might need to have explicit access granted for if they are converted to unprivileged.

One worthwhile exercise is simply to mark one or other of the tasks as unprivileged and then load and run the application and note that (in this case) it winds up in the MPU fault handler. You can examine registers with the debugger to identify the faulting address, and you could try using the debugger to force a return from the handler as a quick way to identify the instruction that gave rise to the MPU fault. (Though with the fault uncorrected, the CPU will likely machine-check at that point and probably drop the debug connection.)

In this case, we can see we need the individual LED tasks need access to the shared "brightness request" array, and also to the handle of the mutex that is used to control access to said array. The mutex buffer, on the other hand, is only ever accessed via privileged kernel code so can remain out of reach of the tasks.

By the same token, if we'd used a queue to pass information between the tasks, rather than shared memory, we would need no additional MPU settings to grant access as the queue itself would be accessed only in privileged mode (by the kernel API) and the queue handle could be communicated via each task's parameter or thread-local storage pointer.

Recall that earlier on in this document, we commented that the shared memory communication between tasks was less than ideal but allowed us to illustrate a technique? Well, this is it. Given a need to access a shared region of RAM, how do we do it?

We go back to the linker file and create another named section, complete with exported start and size symbols, somewhat similar to the kernel sections we looked at earlier. Then we add placement attributes (compilers vary, but with GCC, we typically use:

```
__attribute__ ( ( section ( "section_name")))
```

To specify the placement of a variable or function.

Finally, we use the symbols from the linker file to define MPU region specifications that can be added to task MPU parameters in order to gain access.

Remember that the exported linker symbols are addresses, so we can take the symbol's address after declaring it as an `extern` variable, or we can declare it as an array type whereby its unadorned name will be an address automatically.

CHAPTER 6 Some Potential Pitfalls

Some of the differences between the FreeRTOS and SAFE**RTOS** APIs or behaviour can introduce subtle errors or unexpected beahviour, and this section will briefly outline a couple of the ones that are perhaps more obscure than many.

6.1 Return Codes are more Detailed in SAFERTOS

At first sight, there's nothing terribly tricky about the fact that SAFE**RTOS** almost always returns a status code and places any other return data in variables passed by reference.

Where the differing behaviour opens up the possibility of some very subtle errors, though, is in the few functions were both FreeRTOS and SAFE**RTOS** return an error code. In just about all such cases, FreeRTOS will return `pdPASS` or `pdFAIL` – usually defined in the port layer to be 1 and 0, resp.

SAFE**RTOS**, on the other hand, will usually return `pdPASS` or some negative number indicating the error in more detail, per ***`projdefs.h`***. If you have FreeRTOS code that explicitly checks for `pdFAIL`, or zero, then, it will likely behave under SAFE**RTOS** as if calls that result in an error code actually returned successfully, because the SAFE**RTOS** error codes are usually non-zero.

This can result in some very subtle errors, like now and then losing information being passed via a queue, say, instead of re-trying after a pause, or presuming that a semaphore or mutex has been acquired when it has not.

6.2 MUTEX Behaviour Differs in SAFERTOS

In FreeRTOS, mutexes embody a priority inheritance mechanism, which, until relatively recently, SAFE**RTOS** did not implement. Priority inheritance is intended to ensure that a priority inversion deadlock can't arise should a low priority task be holding a mutex that a high priority task has blocked on, while an intermediate priority task runs to the exclusion of the low-priority task. In this situation, the low priority mutex-holder can't run, and can't therefore release the mutex, so, in effect, the medium priority task is blocking the high-priority task – a form of priority inversion.

Priority inheritance means that when a high priority task blocks on a mutex held by a low priority task, the low priority task inherits the priority of the task trying to acquire the mutex, meaning that it can run regardless of any intermediate priority tasks, and hence can eventually release the mutex. With FreeRTOS, if a task has multiple mutexes, and inherits a high priority through one of them, it will only return to its base priority once it releases all the mutexes it holds.

The inheritance mechanism in SAFE**RTOS** is more thorough – each time a task releases a mutex through which it has inherited, it re-assesses whether it can drop its priority, and how far – so it might be able to revert immediately to its base priority, or it might hold another mutex through which it has inherited a slightly lower priority and therefore need to drop only as far as that.

This has a couple of potential effects – one is that it is obviously more time-consuming to be more thorough, and the other is that the behaviour will be different. Not any more or less correct – both behaviours are as intended and documented for their respective platform – but different nonetheless.

It might emerge that an application depends on the longer priority retention under FreeRTOS, or simply that things operate in a different order because of the different priority reversion. Again, the effects could be tremendously subtle and very hard to identify and pin down.

So it is as well at all times to keep in mind that SAFE**RTOS** is not the same as FreeRTOS, even though the two share a common heritage and functional model.

CHAPTER 7 Conclusion

This white paper has explored the similarities and differences of FreeRTOS and SAFE**RTOS**, illustrated with a walkthrough migration project.

At each stage of the conversion of this simple example project from FreeRTOS to SAFE**RTOS**, it is possible to 'diff' the pre-configured project source code and run each version in turn, with and without your own modifications, to explore the differences and similarities between the application on its original FreeRTOS platform and the eventual un-privileged version of it running under SAFE**RTOS**.

It should be apparent that even with this very simple code, there are a number of different ways of doing things, and early design decisions can have a bearing on the simplicity or otherwise of such a conversion. Minimise use of FreeRTOS-only API, use kernel-managed inter-task communication and use type names that FreeRTOS and SAFE**RTOS** share, and you can make things a lot easier.

Design with unprivileged execution in mind from the outset, and it'll be much simpler when you come to upgrade.

Full API changes are documented in the Manual "Upgrading from FreeRTOS to Safe**RTOS**", which is freely available from the WHIS Download Centre. If you have a licensed copy of SAFE**RTOS** then you will have a complete API reference to go with it.

A SAFE**RTOS** DAP or DHF release also comes with a comprehensive safety manual that goes into considerable detail about safety-related aspects of the integration of application code with SAFE**RTOS** for a safety-critical application.

WITTENSTEIN

RESOURCES

All components of this walkthrough are available for you to try yourself. The WHIS website in particular has a range of content that is free to download, including manuals, datasheets, and SAFE**RTOS** demos for a wide range of platforms. Access this content by creating a free account with our Download Centre, and gain unlimited downloads.

The Workshop Demo and Upgrade Manual

Download the workshop demo explored in this white paper, and complimentary manual, from the WHIS Download Centre. Create an account to access these resources, as well as many others. The Demo explored in this white paper is titled "**Upgrading From FreeRTOS to SAFERTOS: Workshop Example using Kinetis K64F with GCC MCUXpresso for the NXP FRDM64F**". The Manual is "**Upgrading from FreeRTOS to SAFERTOS User Manual"**

- https://www.highintegritysystems.com/down-loads/manuals-datasheets/safertos-datasheet-downloads/

Video Tutorials on Using the SAFERTOS Demo

WHIS offer a variety of useful videos on the WHIS YouTube Channel. In Particular, the playlist "**Using the SAFERTOS Demo**" is a three-part series of videos that will help walk you through how to use the free demos on the WHIS website.

- https://www.youtube.com/user/HighIntegritySystems/videos

FreeRTOS

Developed in partnership with the world's leading chip companies over a 15-year period, and now downloaded every 170 seconds, FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of IoT libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use.

- https://www.freertos.org/
- https://github.com/FreeRTOS

NXP Freedom-K64F

The Freedom-K64F from NXP is an ultra-low-cost development platform for Kinetis® K64, K63, and K24 MCUs. The MCUXpresso SDK Builder Software Development Kit contains optional FreeRTOS™ and MCU Bootloader

- https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F
- https://mcuxpresso.nxp.com/

SAFERTOS®

SAFE**RTOS**® is a pre-certified safety Real Time Operating System (RTOS) for embedded processors. It delivers superior performance and pre-certified dependability, whilst utilizing minimal resources.

- https://www.highintegritysystems.com/safertos/

CONTACT INFORMATION

User feedback is essential to the continued maintenance and development of SAFE**RTOS**. Please provide all software and documentation comments and suggestions to the most convenient contact point listed below.

Contact WITTENSTEIN high integrity systems

Address: WITTENSTEIN high integrity systems
Brown's Court, Long Ashton Business Park
Yanley Lane, Long Ashton
Bristol, BS41 9LB
England

Phone: +44 (0)1275 395 600
Email: support@HighIntegritySystems.com

Website www.HighIntegritySystems.com