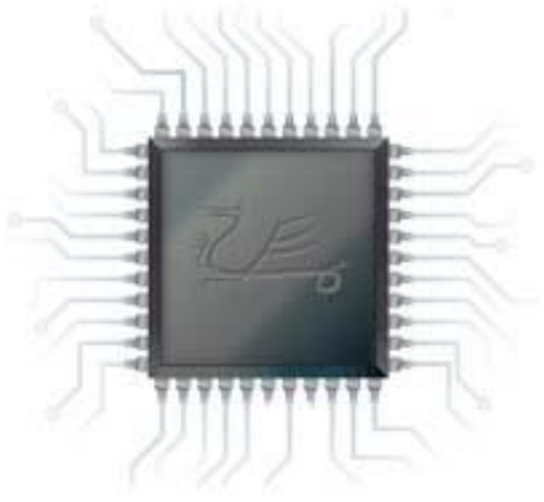
Introduction to Real-Time Operating Systems



Simon Southwell

August 2023

Preface

This document brings together four articles written in July and August 2023 and published on LinkedIn, that is a look at real-time operating systems, using FreeRTOS as a case-study, and porting and configuring this to run on a RISC-V C++ instructions set simulator virtual model.

Simon Southwell
Cambridge, UK
August 2023

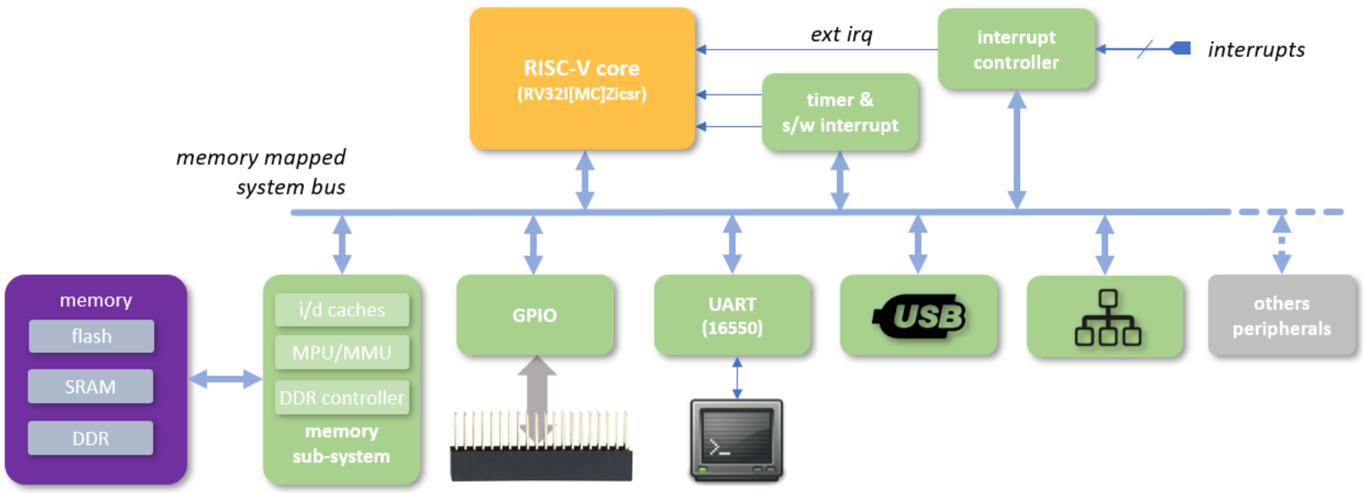
Contents

Part 1: Real-time OS and Tasks

Introduction

In previous articles I have talked about microprocessors, what they are, what they do and looked at a case study of designing one from scratch based on the RISC-V ISA specifications. I recommend reading these first, if you are unfamiliar with processor design and RISC-V. Assuming we've done all that, and have a processor in our hands (and some of my mentees have done just such a thing), what can we do with it now?

Well, a processor by itself can't even run without some additional hardware. At the very least it will need some sort of memory to house a program and have RAM for reading and writing data values. More likely though, a whole other set of hardware will make up the environment in which the processor, or processors, sit. A System-on-Chip (SoC) is a typical embedded environment in which processors might find themselves. These will vary in design from one implementation to the next, depending on the particular needs, but they all share a set of characteristics. The diagram below shows a simplified generic SoC block diagram based around a RISC-V core:



Here we have a core connected to some sort of memory mapped bus infrastructure. It has a memory sub-system, with caches, memory protection unit or memory management unit, a DDR memory controller, and might have other features for FLASH, ROM or tightly coupled memory—you get the idea. A processor can be interrupted, so there is an interrupt controller for gathering and prioritizing interrupts, and a timer which can also be a source of interrupts, as might any other present peripherals. An SoC is not much good if it can't interact with the outside world to sample, measure, control and communicate, and so a set of peripherals would be present to implement this functionality. Which sets are present would be based on what the function of the system is meant to achieve. This could be quite limited and specific if required to control a particular scenario, to quite broad in scope as a single-board computer, such as the Raspberry Pi. What is common to most systems is that they have one or more processors, a bus system, memory, sources of interrupts (which could be the peripherals) and a set of peripheral functions to interact with the outside world. So, we've designed our processor core and we have our hardware environment. Is this all we need?

In the same set of articles on processor design, I finished off by looking at the RISC-V assembly language so that code could be constructed and run on the core. In other words, we started looking at software. Software is the key to a processor based system, as the whole point of the processor is to execute programs. With the hardware environment we have constructed, the software will have the task of configuring and using the peripherals to affect some overall system function. We could write a single program, as a single thread of code, to do all of this. We could even do all of this in assembly language. This might be a good approach for a very small system, such as one monitoring a signal and raising an alarm on some state of that signal. More generally, though, multiple peripherals will need to be running in parallel with each other, with the software taking care of all of them. This still might be done as a single program, written specifically for the function, but it is likely that each peripheral (or a sub-set group) will need to be treated independently, and so we would need a multi-tasking software environment via a multi-tasking operating system (OS), so that individual tasks can be assigned to each peripheral or sub-group, running independently from each other (though they might still interact).

In embedded systems, the interaction with the real world would normally need to have a sense of actual time passing. For example, with the signal monitoring I mentioned earlier, it may need to sample this at a certain rate to ensure that the latency from a 'dangerous' value to the alarm being raised meets a certain minimum requirement. This system would then said to be a 'real-time' system and the operating system running on it would be a real-time operating system (RTOS). The timer in the SoC diagram would 'tick' at some constant known real-time rate and be configurable to interrupt when some set time has passed.

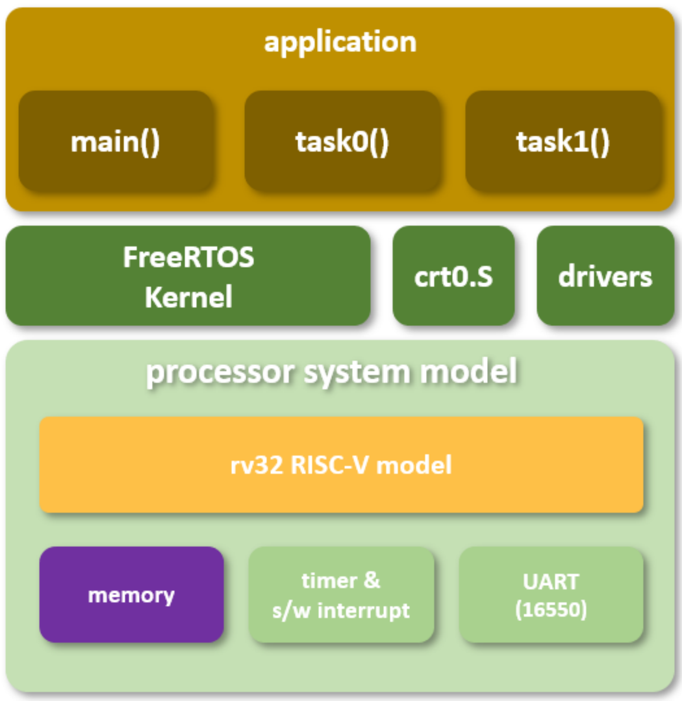
In the rest of this first part, I want to look at the basics of a real-time operating system, what it is and how it is used. We will be looking at the FreeRTOS operating system as an example, which is an open-source RTOS maintained by Amazon Web Services (AWS). This was chosen as it is lightweight, fairly simple to use and contains just the functionality required to have a useful RTOS in an embedded environment. In this first part, some groundwork will be covered and then we will look at 'tasks', in relation to a multi-tasking OS. In the next part we will look at how tasks communicate and synchronise, how they work across multiple processor cores, and then look at how FreeRTOS was ported to the rv32 RISC-V instruction set simulator (ISS). This last has been done so that the RTOS can be explored without the need for any additional and specific hardware. The configuration and porting requirements are also documented as an introduction and example for migrating to other potential target systems.

The aim of the document is to give the reader some insights into the basic common features of an RTOS and how they might be used, as well as exploring a real-world example in FreeRTOS, ported to a RISC-V based virtual system so you can start exploring it right away.

Fundamentals

On top of the hardware layer, then, we need a software layer that will consist of the operating system, along with driver software to configure and use the peripherals. In addition to this, in order to run a 'main' program, we need a C run-time program (CRT). This initialises state and gets things ready before calling the user's main program and we will look at our CRT program in detail, when discussing port FreeRTOS to the ISS, in a later part of the document.

On top of these things, we can run an application, with a main program and any tasks that are created from this to manage the individual independent operations. This stack of hardware and software now looks something like the diagram below:



Here we have the hardware as the bottom layer—assume it looks like the first diagram, with all the other peripherals. The FreeRTOS operating system sits on top of this as a 'bare-metal' program. That is, there is no other software layer between it and running on the processor. Any driver code sits at this same level, as it will be peeking and poking registers of the peripherals that they drive in the memory mapped space without going through the OS necessarily. I've also placed the CRT program (`crt0.S`) at this same level for similar reasons, with it setting things up before any call to FreeRTOS is made. The application software sits on top of these, with a main program that creates a number of tasks and then starts the OS scheduler to manage the running of these tasks.

Some common features of real-time operating systems can be summarised as listed below:

- Tasks/co-routines
  - Creation, deletion/abortion, suspending/resuming
- Task communications
  - Notifications/events, messages/streams, queues
- Synchronisation
  - Semaphores, mutexes
- Timing
  - Delays, software timers

There will be many other features not listed above that can be found in various RTOS's, but those listed are fairly fundamental and constitute the main features of the FreeRTOS kernel, and so we will concentrate on these in our discussions.

FreeRTOS is written largely in C. Certainly the code that is common to all the platforms on which it can run (the bulk of the code) is written in C. It also has platform and processor specific portions that differ from each other, and these can be a mixture of C and assembly programs. We will, of course, stick to the RISC-V specific code for this discussion, and the demonstration programs will be written in C. The application code can be written in C++, if desired, with C linkage for the OS's API, and for any user provided functions, called by the OS. For simplicity, we will stick to C for this exercise.

The FreeRTOS kernel common code can be found in its repository under `FreeRTOS/Source`, with headers found in the `include` sub-directory in that location. A `portable` sub-directory is where the processor and platform specific code can be found. For our purposes, this is `portable/GCC/RISC-V`. For those that are curious, have a look in these directories. We will be alluding to this code in the discussions to follow.

Tasks

Since running tasks is the most obvious feature of a multi-tasking RTOS, we will start with this functionality. In the list above the first item mentions both tasks and coroutines. So, what's the difference?

In general, a co-routine is just like a function but can be suspended to run other code and then resumed from where it left off. This would normally be because the co-routine 'yielded' control to some other routine (the caller, for example) to be 'resumed' at a later point. This is known as 'co-operative scheduling'. A task runs freely and can be suspended at any point by an external routine (in kernel code), usually because of some 'event', such as an interrupt. This is known as preemptive scheduling.

In FreeRTOS, there are facilities for co-routines and cooperative scheduling, where co-routines cannot be suspended by other co-routines (but yield themselves), although they can be preempted by tasks. To keep things simple, we will stick to the most commonly used method, which is preemptive scheduling and tasks. The

FreeRTOS website has more information about co-routines for those wishing to delve deeper.

Each task will have its own space within memory to store its local data state (its 'context'), which is how it can pick up from where it was suspended. When switching between tasks the processor register state will need saving to the suspended task's memory space, on a 'stack', and then the processor register state is updated with the saved state of the resuming task, which will also include updating registers pointing to the memory area for that task's local data, via the stack pointer (`sp`) and, for global data, global pointer (`gp`) registers for RISC-V. Thus, the new task can start running from where it left off, with the register state and the local memory state returned to the values where it was suspended. It is often the case that when a user task is suspended, a kernel task is the one to be resumed. When the kernel task has completed its operation, it can then suspend and resume a user task—possibly a different one from that which was originally suspended.

Time Slicing

Having discussed in general terms how tasks can be suspended and resumed, the question is *when* does this happen? Given that this is a document on preemptive real-time operating systems, the most common reason for swapping out a task for another is due to a timing event. In a multi-tasking operating system, a set of running tasks will 'share' the processor run-time so that each makes progress. We briefly mentioned co-operative scheduling, but our main focus is on preemptive scheduling.

In an RTOS, this involves having a real-time clock that can be configured to generate an interrupt at regular intervals. For RISC-V this involves the `mtime` and `mtimecmp` memory mapped registers (see the "Timer Registers" section in part 2 of my article on processor design). We will talk about priorities shortly, but let's assume, for now, that all the user tasks are the same priority and need to have similar access to the processor execution time. The timer will be set up to interrupt at some regular interval that is long enough to allow a task to progress more than just a few instructions, but short enough so that a task is active enough to respond to real-world events with a sufficiently low latency (this being dependent on the needs of the required function of the system). Usually this might be in the millisecond range—say from 1ms to 100ms. Each time the timer interrupts, the running user task will be suspended, and a new task resumed. It will cycle through each active task, giving each a proportion of the processor run time. This is known as ‘time-slicing’ and the tasks are said to be running ‘concurrently’. In this context, concurrently is different from running in parallel. The latter would be the case if the tasks were running on separate processors (which we will discuss later), but when running on a shared processor they are executing concurrently. From an external point of view, it looks like each task is running in parallel, but this is an illusion, though with a very similar result.

Priority

In the above discussion I made the assumption that all the tasks are running at the same priority and thus get an equal share of the processor time. However, a feature of an RTOS is that tasks can have different priorities. This is useful to distinguish between tasks that have to respond to events quickly, or capture data that is removed in a short time over tasks that need only respond more slowly, such as updating a display.

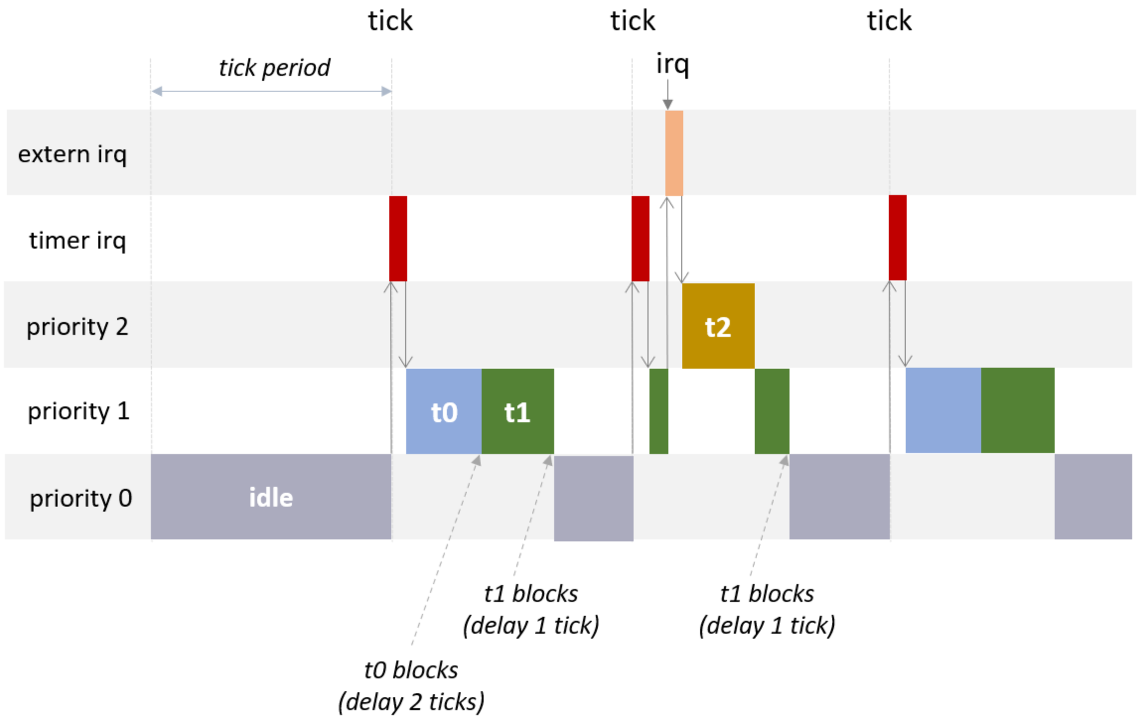
When a set of tasks are running with different priorities, at each time slice, the active task with the highest priority will always be selected—even if it is the task that is already running. Of course, if there is more than one task at this high priority, the processor time will be shared out between those tasks. The lower priority task, though, will not be run. Won’t this lock out the lower priority tasks forever?

Well, yes, if things are not set up correctly. We have discussed tasks, up until now, as if they are always active—i.e., they have something to do and are running code. This can only be the case if all the tasks are the same priority, and the time-slicing will sort out a fair share of run time. In a real-time system, however, tasks do not have things to do all the time. For example, a task might be responsible for capturing an analogue-to-digital’s measurement and storing it away for processing. This may happen infrequently, but the capture may have a small time window for when the data needs to be read and stored away, and so this task has high priority. When it is waiting for the next ADC reading, it can ‘sleep’. This mechanism could be that, when it calls to a function to read an ADC function, the code suspends the tasks, making it inactive. It could also setup an interrupt from the ADC when the next value is available, and this interrupt causes the kernel to make the task active once more. The task, now the highest priority, will be resumed and can capture the value before going to sleep again.

Tasks can also voluntarily suspend for a period of time. Calls to delay functions will suspend a task, to be resumed after some delay, or at some specific future time. The task will be inactive during that period and so will not be scheduled until that time has passed, where it will be put back on the list of active tasks. This use of delays is somewhat like cooperative scheduling where a task yields execution, but not quite. A coroutine specifically yields execution, but it does not know when it will be resumed. With tasks, calls to blocking functions that ultimately suspend the task have some criteria for resumption—a delay, or some result being available etc.

Interrupt service routines, though not tasks (and more like coroutines), have priorities even higher than for tasks. That is, if an unmasked interrupt occurs, the relevant interrupt service routine will be called, regardless of what the priority of active tasks are. Because of this, ISRs would normally be as short as possible so as not to lock out high priority tasks.

The diagram below shows a simple scenario with three tasks, timer interrupts and an external interrupt:



In this example there are three tasks: t0, t1 and t2. The t0 and t1 tasks both have the same priority, whilst the t2 task has a higher priority. In addition to these user tasks, the kernel has an idle task which will be automatically started. If no user task is active, the processor must still do ‘something’, and so it runs the idle task, which basically loops on a no-operation (nop). If the processor supports it, it can go into a low powered state. For example, the RISC-V processor can implement a wait-for-interrupt (`wfi`) instruction which puts the processor I low powered mode and will only power back up if an interrupt is received.

In the diagram, when the timer first generates an interrupt, the timer ISR is executed, and t0 is scheduled and run. It is assumed that t2, a higher priority task is suspended. For this example, we will say it is initially blocked waiting on an external interrupt. At some point t0 blocks, waiting on a 2 tick delay. Because t1, of the same priority, is ready to go t0 is no longer ready, it is run. It eventually blocks waiting on a single tick delay. The flow then returns to the idle task. At the next tick, t1 is resumed as it was waiting on the next tick, rather than the two ticks of t0. Before it blocks, an external interrupt event happens, running its ISR, and t2 now gets run, suspending t1. When t2 blocks again, t1 is the resumed…and so on.

Thus, in summary, all the tasks would normally have natural suspend points, allowing periods where lower priority tasks will be run when active. At each point where an interrupt occurs and runs an ISR (timer or other), or where a task is blocked by making a call to a blocking kernel function, such as a delay, kernel code is run to assess who gets to run next based on a task being ready to run and its priority over other ready tasks.

Within FreeRTOS, this is done with calls to `xTaskSwitchContext()`, found within the `tasks.c` source file. Part of the task's context data (`tskTaskControlBlock`) is state that indicates the priority, ready state, and blocked state. An event list is kept of tasks, with a ready list and a delayed list, and ordered by priority. As events occur and time advances, tasks are added and removed from the event list as they are resumed or suspended. Thus, choosing and then switching between tasks becomes a matter of ordered list management. Some of the FreeRTOS Kernel API functions for tasks are listed below:

- `xTaskCreate`, `xTaskDelete`
- `vTaskStartScheduler`, `vTaskEndScheduler`
- `vTaskSuspendAll`, `vTaskResumeAll`
- `uxTaskPriorityGet`, `vTaskPrioritySet`
- `vTaskDelay`, `vTaskDelayUntil`

This list is not exhaustive, but overlaps the main subjects that we have covered so far and are hopefully now obvious in what they do. More details of these functions and how to use them can be found in the FreeRTOS API reference (under "Task

Creation” and “Task Control”), but we will be looking at some examples that use the API functions when discussing running FreeRTOS on the RISC-V ISS.

Conclusions

This first part of the document introduced what a real-time operating system is and some of its basic features, with reference to FreeRTOS. In particular the concept of a task as an independently running functions, running concurrently with other tasks. This was achieved with each task having its own private data space, that included information about its status as active or inactive. This allows switching between tasks to be efficient and simple by saving off register state and then updating the stack-point to point to a new task’s local data and restoring the register state for that task.

The concept of time-slicing was explained and the RTOS switching between active tasks at set ‘tick’ intervals. Tasks can have different priorities, with higher priority tasks being selected in preference to lower priority tasks. Also discussed was the fact that tasks can be suspended for other reasons apart from being deselected on time-slicing. This is when a task makes a call to a ‘blocking’ functions, which might be to read a value that is not yet available, or if it makes a call to a delay function which will suspend the task for a given number of ticks. The RTOS’s scheduler functionality provides the management for all of this task execution co-ordination.

In the next part will be discussed how tasks communicate and synchronise with each other, and even how this might be done when running on separate processor cores, in a multi-core environment. Then we will look at the porting of FreeRTOS to the rv32 RISC-V instructions set simulator, and some demonstrations of using the features.

Part 2: RTOS Task Communication and Synchronisation

Introduction

In the previous part of the document an overview of real-time operating systems was given, how they fit on top of processor based system, and the sort of common features they have. That part then looked at one of these features, namely 'tasks', in more detail, discussing what they were. We looked at how they were scheduled and how they were prioritised so that a scheduler can make decisions about when to swap from running one task to running another. By default, this was done using 'time-slicing' via a real-time clock, but other reasons were seen that could make a task 'block', such as calling API functions delaying for some number of ticks, or waiting on data becoming available, and the scheduler would then run another active, unblocked task. This was all discussed under the examples of a RISC-V processor based system and the FreeRTOS real-time operating system, and we will continue the discussions of this part of the document in the same vein.

Now we are able to run separate tasks, each of which can control one (or more) of the functions in our SoC, we have a powerful solution for an embedded system. If each of the functions truly ran independently from each other, then this would be the only functionality we'd need. Each task would read and write to the memory mapped registers of the logic it was controlling, via driver software, using simple pointer access:

```
*ptrCtrlReg = newCtrlRefVal; // Write to control register
newStatus   = *pStatusReg;   // Read status register
```

In a real-world system, however, it is unlikely to be the case that all the tasks act alone. Therefore, they need to communicate with each other in various ways, from quite low level, to exchanging quite sophisticated data structures. At an even lower level, tasks may need to synchronise with each other without exchanging any further data. In this part of the document I want to look at these mechanisms, and what FreeRTOS features are provided by way of examples.

Inter-Task Communication

In a multi-tasking system, in order to create a system as a whole, the running tasks may fall into different categories, and have priorities to match. For example, a task may be a “data collecting” task with high priority to quickly fetch data from a peripheral, say, when it becomes available, and send this to another lower priority tasks that is a “data processing” task, to use that data in some form, perhaps delayed until a certain amount has been gathered. Therefore, the tasks need to be able to exchange data. In FreeRTOS kernel, the features made available are in the following categories:

- Queues
- Messages
- Streams

When and where the different types are used will be discussed in the rest of this section.

Queues

Queues are the basic form of inter-task data communication feature and are exactly what you’d expect them to be. They are basic first-in, first-out (FIFO) objects, with new data being sent to the back of the queue from a sender, and a receiver receiving the data in the order in which it was sent. In FreeRTOS, though, data can also be sent to the front of the queue—useful if, say, some high priority data needs to bypass lower priority data sitting in the queue. The task itself would have to keep track of what is already in the queue to make this assessment and choose to put at the front.

The Queue size will be finite, with a fixed number of items, and each item a fixed size. This is a defining characteristic of a queue, that all items in the queue are the same size, and defined when the queue is created. Therefore, a sending task will block if there is no space in the queue, and thus be descheduled until space becomes available. Similarly, a task trying to receive an item from an empty queue will also be blocked until one becomes available.

FreeRTOS provides a rich set of API calls for queues, with some of the highlights listed below:

- `xQueueCreate`, `xQueueCreateStatic`

- `xQueueSendToBack`, `xQueueSendToFront`, `xQueueSendToBackFromISR`, `xQueueSendToFrontFromISR`
- `xQueueReceive`, `xQueueReceiveFromISR`
- `uxQueueMessagesWaiting`, `uxQueueMessagesWaitingFromISR`
- `uxQueueSpacesAvailable`

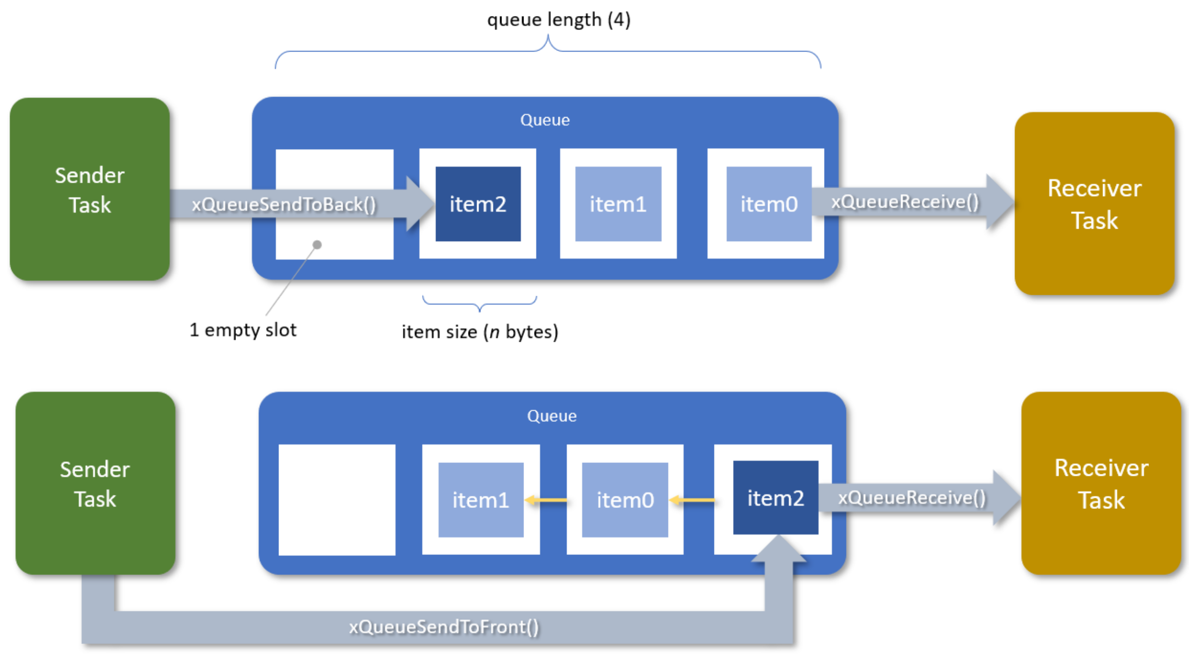
Obviously, a queue needs to be created before we can use it, and this is done with `xQueueCreate`. The arguments for this define the number of items the queue can hold, plus the size of each item (in bytes). It returns a handle to the queue for use in identifying it when sending or receiving data etc.

Note that there is an alternative version, `xQueueCreateStatic`. The former requires that dynamic allocation is configured for FreeRTOS (we will look at these when discussing configuration in the next part of the document). If not, then `xQueueCreateStatic` must be used, and pointers to a couple of buffers of appropriate size provided—one to store all the items, and one to store the queue's data structure. For the most part, dynamic allocation is enabled, and so buffering is handled by the OS. Other objects have similar variants in their creation API functions, but I will assume that dynamic allocation is enabled, as it is the simplest to use.

Once created, data can be sent, as we saw, either to the back (the normal situation) or to the front of the queue. The API function takes a pointer to the queue to send to as an argument, a pointer to a buffer containing the data to be sent, and also a number of ticks to wait. This last defines the maximum time the task should be blocked, before returning. If the data was sent successfully, the returned value indicates this (`pdTRUE`) or it will return with a 'queue full' status (`errQUEUE_FULL`) if the call timed out. If FreeRTOS is configured in a certain way, then the call can be set to block indefinitely.

You'll notice that there are variants of these API calls with a suffix of `FromISR`, so a quick note on this is in order. Interrupt service routines, as we saw in the first part of the document, have the highest priority, trumping all the task priorities. Because of this, ISRs should not block, potentially starving progress on the tasks. Many (but not all) of the API calls have variants for use from within ISRs. Taking `xQueueSendToBackFromISR` as an example, it behaves similarly to the non-ISR version, but takes no time out argument. Instead, it will return immediately if there is no space, with `errQUEUE_FULL`. To aid in scheduling, it does have another argument (which can be `NULL`) that is a pointer to return status as to whether sending the item unblocked a higher priority task, in which case the ISR should send a request to do a context switch (co-operatively yielding) before exiting.

The two receive API calls behave in a similar way, with identical arguments. Received queue items are copied, rather than sent by reference, and so a buffer pointer is still needed to receive the data. The basic operations on the queues are summarised by the diagram below:



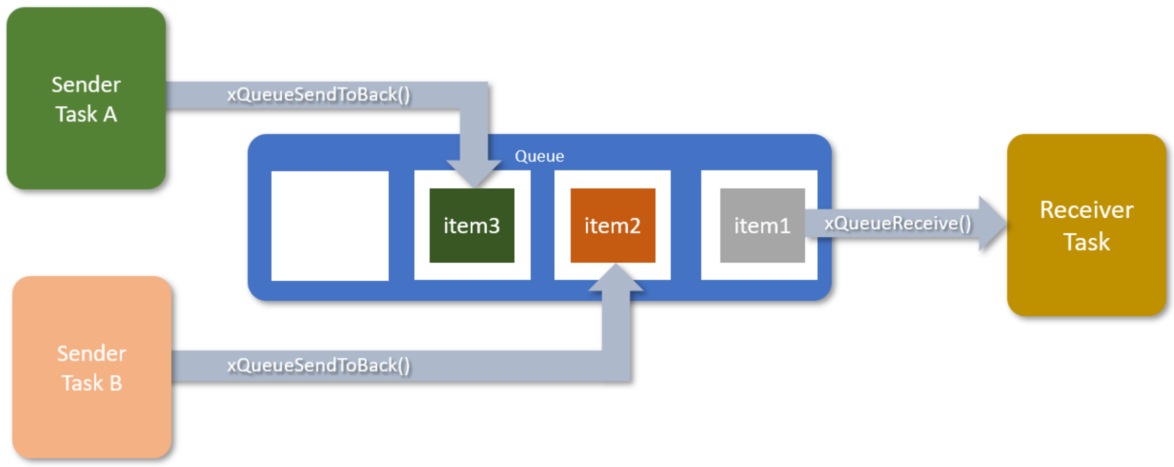
Here are two cases of sending data to the back of a queue and to the front. The example shows a case where two items already exist in a queue of length 4, so has two used and two empty slots. The size of the items is unspecified (so let's call it *n*) and will be the same for each item. In the first (upper) case, a third item (item2) is sent to the back of the queue and is placed logically behind item1. In the second (lower) case, a third item is sent to the front of the queue, which then sits as the first item of the queue, and the old items logically moved one place back. In both cases, a single space remains after the operation. The receiving task, if it subsequently calls the `xQueueReceive` API function, will get item0 in the first case, and item2 in the second case.

Another way to handle blocking or not, is to inspect the contents of the queue before sending or attempting to receive data. The `uxQueueMessagesWaiting` function (and its ISR equivalent), for instance, returns the number of items in the queue so and receiver can decide whether to call the receive API function or not. (These are, confusingly, called messages but are different from 'messages' that we will look at shortly. This is why I have used the name ‘item’). Another API call returns the number of spaces available in the queue so a sender can decide whether to make a call to the send API function, or not. Whether a task uses these, or simply allows itself to block, will depend on whether it can usefully do something else if it would block on a send or receive call.

As stated earlier, queues have fixed sized items. This could be just fixed ‘blocks’ of raw data, but can also be complex structures, so long as those structures are of fixed size—fundamentally, this is the same thing, with a structure just an interpretation of the data overlaid on top. Therefore, a queue would be used in just these circumstances, of fixed size sets of data to send from one task to another. One could, of course, send smaller amounts of data, perhaps with the first bytes indicating the size of the payload, and pad the rest of the items with some null value, such as 0. But this would be inefficient, and a better type of communication object would be more suitable, as we shall see.

There are other queue API functions not detailed here, with ‘peek’ facilities to fetch an item from the queue without deleting it, or the ability to reset the queue, deleting any items that were in it, as well as some other functionality. (See the FreeRTOS API documentation for queues.)

Queues can safely be written to from multiple tasks, allowing data transfer from multiple sources. The OS takes care of making this safe operation when writing from two tasks and will block one task if another task send operation is already in progress. If the task sources of write operations are truly asynchronous, the order in which items are written may not be deterministic, and external factors will determine the order of items written to the queue, such as the relative task priorities, the timing of any external events that instigate sending data to the queue etc. If the receiving task needs to know the origin of the data it is receiving, then the item data structures might contain a field with a task ID, or handle, that can be used to differentiate the data—but this is implementation specific. The diagram below summarises sending data from multiple sources:



One last topic on queues I want to briefly discuss is queue sets. This allows blocking on reading from multiple queues. A queue set can be created, and multiple queues added to the set (they must be empty at this point). A receiving task can then block on the set—that is, it will wait until there is at least one item in each of the queues associated with the set. Why is this useful? Well, it is useful, for example, when the control information is separated from the data itself, and maybe being received over different channels in the hardware, and even that the data may arrive before its corresponding control information. Then it is useful to have two separate queues for each data type. However, the receiver may need both sets in order to start processing the information, and so these two queues can be added to a queue set, and the task can block on the set until a pair of items is available. The queue set API calls are documented in detail, in the FreeRTOS API documentation.

Message and Stream Buffers

Alternative means to send information between tasks is the use of message and stream buffers. These are very closely related and so I want to discuss them together.

The most obvious difference between messages and streams over queues is that they *not* fixed in size. The difference between messages and streams is that messages are passed as units of data, much like a queue item, but that each message can be of different size, whereas stream are a continuous set of bytes that can be sent and received. Actually, according to the FreeRTOS documentation, messages are in fact built on top of streams. Another difference is that the OS assumes that there is a single source (task or ISR) of data being sent to a single destination—in other words a point-to-point connection. There can be multiple sources set up but requiring the user code to make sure only one task/ISR is involved at any one time. API highlights are listed below, where `TYPE` is ether `Stream` or `Message`:

- `xTYPEBufferCreate`, `xTYPEBufferCreateStatic`, `xTYPEBufferCreateWithCallback`, `xTYPEBufferStaticCreateWithCallback`
- `xTYPEBufferSend`, `xTYPEBufferSendFromISR`
- `xTYPEBufferReceive`, `xTYPEBufferReceiveFromISR`
- `xTYPEBufferBytesAvailable`, `xTYPESpacesAvailable`, `xTYPEBufferIsFull`, `xTYPEBufferIsEmpty`
- `xTYPEBufferReset`, `vTYPEBufferDelete`

As for queues, streams and message buffers must be created, though they have slightly different arguments, but both return a handle for the buffer object. For both streams and messages, the first argument is a buffer size in bytes. This is the total number of bytes that the buffer can hold. For messages, this will include a length value added to each message, which might be 4 bytes on a 32-bit machine. So, if, for example, two messages are sent, one with 16 bytes, and another with 24 bytes, assuming they have not yet been read, this will occupy 20 + 28 = 48 bytes in the message buffer. An additional parameter is needed for the stream buffer creation to specify the trigger level number of bytes. This dictates the number of bytes in a stream buffer before a blocked receiving task is unblocked (or is unblocked on a time out).

There are variants of the buffer creation functions with the suffix `WithCallback`. These allow call back functions to be specified for when a send has completed (when reached the trigger level for streams, or when the message has been fully written to the buffer) and when data is received (either bytes read for streams, or a message read for messages). This, as we shall see in a later part of the document, is useful when communicating between tasks on different processor cores in a multicore environment. For those who may be unfamiliar with the concept of callback functions, I will have a discussion on these when we look at multi-core systems (in a later part of the document) and these become useful.

Sending is similar to that for queues, with arguments of a handle, a buffer pointer, a size in bytes (new from queues) and a tick to wait before timing out. Receiving calls have the same arguments as sending calls. The number of bytes written or read is returned by the API functions. Again, as for queues, there are ISR versions.

The buffers can be inspected for the number of bytes it contains and the space remaining, as well as whether full or empty. It can also be reset back to its initial empty state or deleted completely.

Inter-task Synchronisation

In the last section, we looked at how we can send data between tasks. Another basic feature required is to be able to synchronise between tasks. We do this with semaphores and mutexes. These two are very much related and we will look at these together in this section.

Semaphores

It might be helpful if we have a look at what a semaphore actually is. Obviously, the original meaning was the use of flags to enable non-verbal communication over a distance, particularly between ships, spelling out letters with two flags held in different positions relative to the sender's body. (This is not strictly relevant, but I am originally from Portsmouth, the home of the Royal Navy.) In the context of an RTOS, it is a means of blocking and unblocking tasks based on the value of the semaphore. In that sense, they are like flags indicating whether a task can advance or not. They come in two flavours:

- Binary semaphores
- Counting semaphores

The binary semaphore is, as the name indicates, is an on/off, true/false, 1/0 kind of arrangement, and is ostensibly used to synchronise tasks, allowing one task to be halted until another task has reached some point, either complete the construction data, or some event has happened etc. A counting semaphore builds on this with a count value, where it can be incremented and decremented and (usually) will block a task when it is zero. The counting semaphores are used to synchronise some access to a shared resource, which has multiple, but finite, capacity. For example, suppose we have a DMA block that has two channels which are shared between multiple tasks (more than 2). It is okay if one channel is in use and a task requires some DMA operation as it can use the free channel, but if a third task requires DMA services it must block until a channel becomes free. If a counting semaphore is created with an initial value of 2 and is decremented when a task requests use of a DMA channel, then the first two tasks will continue, which will decrement the semaphore to 0, making the third task block when it makes a request. If the semaphore is incremented when a DMA has completed, then the third task will be unblocked when the semaphore is incremented to 1, though it will immediately decrement it back to 0 as it takes the channel, blocking any other task that comes along to use the DMA engine, and so on. In may respects, a binary semaphore works like a counting semaphore with an initial value of 1.

A key feature of all semaphores is that changing of its state (either 1/0 or incrementing/decrementing) is 'atomic'. That is, happens as a single uninterruptable operation. This is critical for it to work, and hardware often provides facilities to help with this as it's so important. For example, the RISC-V provides the "A" atomic extensions to allow atomic read-modify-writes between *harts* (hardware threads) sharing the same memory, and the AXI bus provides optional "exclusive" bus transaction signalling to aide in the implementation of semaphores. These kinds of features may not be necessary to implement semaphores in a system, depending on its architecture, so long as the semaphore state can be updated atomically by some means. This is a lot easier in single core solutions.

FreeRTOS provides facilities to create and use semaphores. Before we look at these, just a quick word on nomenclature. In most of my dealings with semaphores the terms 'post' and 'wait' have been used for the operation of incrementing a semaphore, and for decrementing a semaphore or blocking if it's at 0. In FreeRTOS, the terms 'give' and 'take' are used but mean exactly the same thing.

Some API highlights are listed below for the FreeRTOS semaphore functionality:

- `xSemaphoreCreateBinary`, `xSemaphoreCreateCounting`, `xSemaphoreCreateBinaryStatic`, `xSemaphoreCreateCountingStatic`
- `vSemaphoreDelete`
- `xSemaphoreTake`, `xSemaphoreTakeFromISR`, `xSemaphoreGive`, `xSemaphoreGiveFromISR`
- `uxSemaphoreGetCount`

Creating a binary semaphore is a simple matter of calling the API function `xSemaphoreCreateBinary`, which takes no arguments, and returns the handle. For counting semaphores, a maximum count and an initial count are passed in as arguments. Both functions have static equivalents, which require an additional buffer pointer argument. A semaphore can be deleted with `vSemaphoreDelete`, passing in the handle of the surplus-to-requirements semaphore.

The ‘take’ functions have arguments for the semaphore handle and a tick timeout count, and return `pdTRUE` if semaphore obtained or `pdFALSE` if it timed out. Like the other API functions, the ISR versions replace the time out value with pointer to return whether a higher priority task was unblocked by obtaining the semaphore. The ‘give’ equivalent API functions simply take the semaphore handle as an argument, with the ISR equivalent having the same addition argument as the take ISR function.

Finally, `uxSemaphoreGetCount` returns the current count of a semaphore. In the case of binary semaphores this is 1 if the semaphore is available, or 0 if not.

Mutexes

A mutex (from mutually exclusive) is a special kind of semaphore used to mark a ‘critical’ section of code. By this I mean a piece of code that is shared by more than one task, but that must not be interrupted when entered. This might be because it has multiple instructions to perform an operation that would fail if a higher priority task preempted the task running this critical section. For example, suppose a peripheral can be set up to send some data, but this takes multiple steps to configure, send and acknowledge, and so this must always start from some idle type state, returning to idle when the operation is completed. If a task were part way through this operation when it was preempted by a higher priority task, say, then the new task might configure this incorrectly, since state has already been advanced by the preempted task. But even if it reset the device to idle and completed its operation, when the preempted task was resumed, the state of the device would be wrong for the point where it was suspended, and thus the operation will almost certainly fail.

So why don’t we just use a semaphore? Well, there is a problem to solve called ‘priority inversion’. In the scenario above, when the higher reaches the critical section and tries to take the semaphore already taken by the lower priority task, it will block until the lower priority task gives it back—this is the behaviour that we want. The lower priority task will effectively block a higher priority task, but only in the shared critical section. Thus, critical sections need to be kept as short as possible. However, if a third task, with some middling priority between the lower and higher priority tasks, becomes active whilst the lower priority task is in its critical section, it would preempt the lower priority task. If this middling task never goes through this critical section, it will never block on it, and could run for a long time, blocking the higher priority task artificially long—this is the behaviour that we *don't* want.

The solution is to use mutexes for critical sections, which add 'priority inheritance' over ordinary semaphores. When a task successfully takes a mutex, its priority is temporarily raised to be higher than all other tasks, returning to normal when the mutex is given back. This ensures that critical sections are exited in all haste, and avoids other tasks, not associated with the critical section, starving higher priority tasks, blocked on the mutex. It is not a bullet proof solution, and if large amounts of code are encompassed by mutexes, higher priority tasks can still be starved beyond design intention. Careful coding is required.

Highlights of the API calls relating to FreeRTOS mutexes are listed below:

- `xSemaphoreCreateMutex, xSemaphoreCreateMutexStatic, xSemaphoreCreateMutexRecursive, xSemaphoreCreateMutexRecursiveStatic`
- `xSemaphoreTakeRecursive, xSemaphoreGiveRecursive`
- `xSeamphoreGetMutexHolder`

The first two listed API functions take exactly the same arguments as for their semaphore counterparts. In addition, there are a pair of mutex creation functions with `Recursive` in their name. These allow a task that has already successfully taken the mutex to retake it any number of times. Only when it gives it back the same number of times it has taken it, is the mutex freed for other tasks. I have been racking my brains to think of a use case for using recursive mutexes, but couldn't think of one, and I have never used them myself (let me know if you have any use case examples). The recursive mutexes have their own give and take functions (and the API functions mustn't be mixed with the wrong type of mutex) but work the same way as the non-recursive functions. For non-recursive mutexes, the semaphore give and take API functions, mentioned before, are shared with mutexes. Lastly, the `xSeamphoreGetMutexHolder` function returns the handle of the task that currently holds a mutex (with the mutex handle passed in as the argument).

Conclusions

In the first part of the document was introduced the concept of an RTOS which then focused on creation and manipulation of free running, independent tasks. In this part of the document, the concepts of inter-task communication and synchronisation were introduced.

We looked at binary semaphores to synchronise tasks, and counting semaphores to protect shared resources with multiple, but finite, capacity. Mutexes were described as special forms of semaphore for encapsulating ‘critical’ sections of code that mustn’t be interrupted. These are needed over ordinary semaphores to solve the issue of ‘priority inversion’ by using ‘priority inheritance’.

So, we leave this part of the document with the ability of our tasks to exchange information and to synchronise with each other under various conditions. In the next part I want to finish up the main features of an RTOS, looking at the remaining functionality of the FreeRTOS kernel, then look at running on multiple processor cores, and finish up with a discuss how FreeRTOS is configured with reference to the porting of the OS to the RISC-V instruction set simulator.

Part 3: Notifications, Software Timers and Multi-Core

Introduction

In the previous two parts of this document we introduced real-time operating systems, using FreeRTOS as a case-study, and had a look at multi-tasking before looking at inter-task communication (and it's worth reading these parts before tackling this one, as it is a natural follow-on and assumes that you have). In this third part we will finish off with the last few basic features I want to cover in this introductory set, including notifications, software timers, user ISRs and 'hooks', before finishing with a look at how this all works when in a multi-core environment.

A fourth part of the document will use all the knowledge gained from these first three in order to document the porting of FreeRTOS to a RISC-V instruction set simulator based virtual system. So let's start by talking about notifications.

Direct to Task Notifications

In the previous part of the document we talked about inter-task communication using queues, streams and message buffers. These all had in common the fact that an object was created, references with a returned handle, and tasks either sent data or received data via those objects.

Another way to communicate with a task is via a 'notification', and have each task be able to receive one or more such notifications directly. In other words, the state of the notifications is part of the task's own state, and no intermediary object is involved. This is much more efficient in communicating between tasks. A task can, just as for a semaphore (for example), block waiting for it to be available, and it can also block waiting on receiving a notification. The kind of data that can be sent with a notification is much more restricted than that for the other inter-task communication methods (it's basically a number) but if this is all that's required for the application functionality, then this communication method will be done more efficiently with notifications.

With FreeRTOS, each task can be configured with a number of notifications (discussed in the next part of the document) and also has API functions to be able to send and receive notifications from the tasks or ISRs. Some highlights of these API functions are listed below.

- `xTaskNotify, xTaskNotifyIndexed, xNotifyFromISR, xNotifyFromISRIndexed`
- `xTaskNotifyAndQuery, xTaskNotifyAndQueryIndexed, xTaskNotifyAndQueryFromISR, xTaskNotifyAndQueryFromISRIndexed,`
- `xTaskNotifyWait, xNotifyWaitIndexed`
- `xTaskNotifyGive, xTaskNotifyGiveIndexed, vNotifyGiveFromISR, vNotifyGiveFromISRIndexed`
- `ulTaskNotifyTake, ulTaskNotifyTakeIndexed`
- `xTaskNotifyStateClear, xTaskNotifyStateClearIndexed`
- `ulTaskNotifyValueClear, ulTaskNotifyValueClearIndexed`

As mentioned above, notifications can be sent directly to a task and those tasks may have 1 or more notification receivers (or 0 if notifications aren't being used). The notifications receivers are in an array indexed from 0 to one less than the number of configured notifications. To set a particular notification of a task to be pending, `xTaskNotifyIndexed` is used with the task's handle, along with an index to select the task's particular notification receiver, and also an update value to be used to change a value associated the indexed notification. This is followed by an action argument which dictates what happens to this value. If it's `eNoAction`, the notification receives the event and becomes pending, but the value remains unchanged. If it's `eSetBits`, then the value argument is ORed with the notification's old value to give the new value. If `eIncrement`, then the value is simply incremented, and the value argument ignored. If `eSetValueWithoutOverwrite` the notification is updated with the argument only if the old value has been read, whilst if `eSetValueWithOverwrite` then the value is overwritten regardless. The `xTaskNotify` version of the function is for backwards compatibility (when each task had a maximum of one notification). It effectively has an index of 0. There are also `FromISR` versions of these functions. A variation on the notify methods are the `xTaskNotifyAndQuery` versions which, in addition, return the old notification value (via a pointer passed in) before the update. All these functions return `pdPASS` in all cases, except when using function `eSetValueWithoutOverwrite` and it didn't overwrite the value.

The `xTaskNotifyWait` and `xTaskNotifyWaitIndexed` will wait for a notification with a timeout value (which can be set to never time out). The arguments can specify which bits in the notification's value are cleared both on entering the function (and it may block) and on exiting the function, where a 1 in the argument bits indicate a bit to be cleared. The next argument is a pointer to return the notification value that was updated by sending call (depending on the action). Note that this returned value will include any bits cleared by the clear-on-entry argument (assuming not set back to 1 by the sent notification), but the changes done with the clear-on-exit will not be included.

The `xNotifyGive` and `xNotifyTake` version of the functions are provided to be able to implement a notification based fast and lightweight semaphore mechanism, and their arguments are the same as for the corresponding semaphore functions (see the second part of the document). The give functions have `FromISR` versions, but not the take functions as only tasks have notifications, not ISRs.

The `xTaskNotifyStateClear` and `xTaskNotifyStateClearIndexed` functions will force a notification to clear any pending state. The `ulTaskNotifyValueClear` and `ulTaskNotifyValueClearIndexed` functions will clear bits within a notification's value using a bitmask as in the `xTaskNotifyWait` functions.

With these API functions, then, notifications can be sent with some limited data information, but direct to the task and efficiently. These can block tasks, as for other communication methods, and even be used to implement efficient semaphore functionality.

Software Timers

We have already met functionality that uses the real-time timer, accessible on the RISC-V processor through the memory mapped `mtime` and `mtimecmp` CSR registers, to do the scheduler's time-slicing, and to provide task delay API functions. In addition to this functionality, 'software timers' can be set up which allow a user provided function to be called at some point in the future. The function to be executed is called the timer 'callback' function. (The callback term keeps cropping up and more details of this will be discussed very shortly, I promise.) In many ways, the timer callback functions are like interrupt service routines and, like ISRs, it is essential that they do not block—for example, by calling `xTaskDelay`. This is because they are part of the OS's timer task which must keep running to service all the scheduling etc.

FreeRTOS provides a rich set of API functions to setup and use software timers. Some of these are highlighted in the list below, though this is not exhaustive.

- `xTimerCreate, xTimerCreateStatic`
- `vTimerSetReloadMode`

- `xTimerStart, xTimerStop, xTimerReset, xTimerChangePeriod, xTimerStartFromISR, xTimerStopFromISR, xTimerResetFromISR, xTimerChangePeriodFromISR`
- `xTimerIsTimerActive, xTimerGetPeriod, xTimerGetExpiryTime, xTimerGetReloadMode`

Like other objects we have met before, a software timer must be created and `xTimerCreate` is used to just that (or its static variant). As arguments it takes a name string (to aid debugging), a period value (in units of ticks), an auto-reload flag (more in a bit), a pointer to return a unique ID, and a pointer to the user callback function that will be called when the timer fires.

The auto-reload flag, when true (`pdTRUE`) will reactivate the timer when it expires such that it will trigger continuously at a rate as specified by the period argument. When not true, the timer will only activate once. To use some hardware analogy, this is like the auto-trigger or one-shot trigger modes on an oscilloscope. This mode can be changed once a timer has been created using the `xTimerSetReloadMode` API function.

A timer can be started using `xTimerStart`, for instance, when it is on-shot and already fired. If the timer is already active this starts the timing period to the beginning again and carries on. A timer also can be stopped with `xTimerStop`. The `xTimerReset` has functionality that is almost identical to `xTimerStart` and will start a dormant timer or, of active, reset it relative to the time `xTimerReset` was called and continue. The period can also be changed using `xTimerChangePeriod`, which will also start a dormant timer. All these functions have `FromISR` equivalents.

Various status values can be retrieved from a timer, including whether it's active, its period, its expiry time and its reload mode.

With these features, then, we can set up to run some code both periodically, or as a single event at some future point. In many embedded systems there are hardware timers available which can be configured to generate periodic signals to high accuracy or generate events at very specific times. These may be limited in number, or even not present. The use of software timers is an alternative, especially if the precision requirements are not high. Low frequency clock signals, where some jitter is tolerable, or event pulses with a set delay on some input, with some allowance for timing accuracy, can all be generated using software timers. The number of software timers that can be used is only limited by the amount of memory available for the OS timer code requirements, and the necessary timer state in RAM. There are many other uses of software timers, and this is just a brief overview.

User Extensions

In this last section summarising RTOS functionality using FreeRTOS, I want to look at some extensions that can be provided by the user code to override some default functionality. In particular, we will look at how to add our own interrupt service routine code, and some 'hook' (aka callback) functions to handle some error conditions as well as called when some internal OS tasks are activated.

Interrupt handlers

ISRs are processor architecture specific but, since we are focusing on RISC-V, we will use just the relevant information for this architecture as an example. The FreeRTOS RISC-V port built-in handler will only handle interrupts from the CLINT timer to do its scheduling and timers etc. If nothing else is done it will call, by default, an internal function if any other type of interrupt occurs (i.e., for RISC-V, an external interrupt of a software interrupt). This simply loads the values of the `mcause`, `mepc` and `mstatus` CSR registers into some CPU registers for any attached debugger to inspect. In a real application, these interrupts could be serviced using user provided code. If a function `freertos_risc_v_application_interrupt_handler` is compiled in the application, this will be called by the OS if any unhandled interrupt is fired. It takes no arguments and returns no value, but this is where the application can run code to handle any interrupts in whatever way the system needs.

Similarly, if any exception is fired (and the OS won't handle any of them) then if a function called `freertos_risc_v_application_exception_handler` is included in the application (no arguments or return value), this will be called on encountering any exceptions, such as misaligned memory access, or illegal instruction etc., and the code run from there can attempt to handle these situations in whatever way is relevant.

Both of these functions are called from an interrupt service routine, and therefore must follow the ISR condition not to block, using the `FromISR` API function variants if OS functionality is required.

Hooks

The ‘hook’ functions are a set of optional callback functions to be called when certain events happen within the OS. The are:

- `vApplicationTickHook`
- `vApplicationIdleHook`
- `vApplicationMallocFailedHook`
- `vApplicationStackOverflowHook`

The first of these (if included in the application code) is called whenever a tick event occurs to allow extra operations to be executed at each tick to meet application needs. Similarly, the second function is called whenever enters the idle task. As with ISRs, these functions must not block, with the tick hook actually called from an ISR, and so must use `FromISR` API functions only.

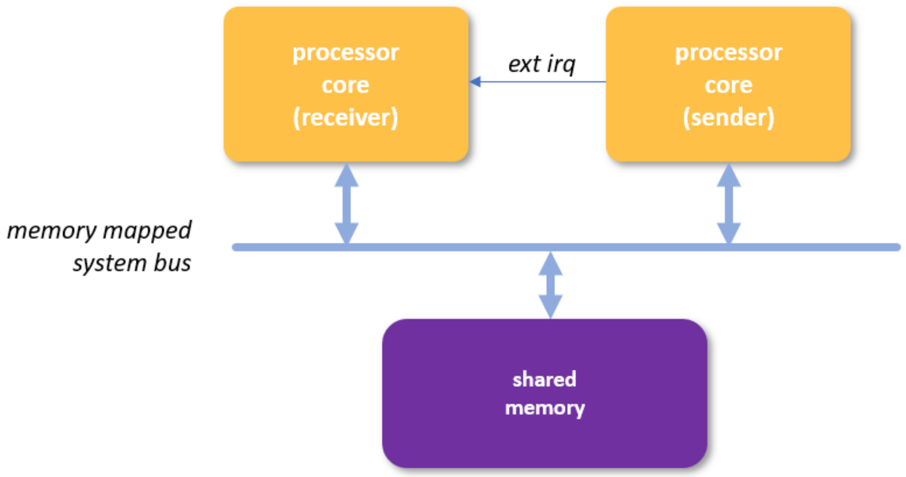
A couple of functions can be called from internal OS error events. If a memory allocation failed (due to insufficient remaining memory), this function will be called, if present ,to attempt to handle and recover from this situation. Similarly, if a stack overflows the last function is called, giving the chance to the application to fix this situation—or ‘crash’ in an elegant way.

RTOS on Multi-Core

In all the discussions we’ve had so far there has been the assumption that only one processor core is running FreeRTOS and the application tasks. A common, even likely, scenario is that there is more than one processor cores present. If those multiple cores are to run tasks, spread over the available cores, for a single given application function and need to communicate then there are two architectures that can enable this to happen. The first is an ‘asymmetric multi-processor’ (AMP) arrangement, where each core is running its own copy of the OS, whilst the second is a ‘symmetric multi-processor’ arrangement (SMP), where only a single version of the OS is running, which schedules tasks across each core. In the former, the processors can be of differing architectures (one ARM, the other RISC-V, say), whilst in the latter all the cores must of the same architecture.

Asymmetric Multi-Processor

With the AMP arrangement, two or more cores will be sitting on the memory mapped bus or interconnect and have access to some shared memory. This is a likely scenario for any multicore system. In addition, a core running a task that needs to send information to a task running on a different core also needs to be able to generate an interrupt on the core that is running the receiving task. The diagram below shows this scenario (the communication is unidirectional, from right to left, in this simple example).



The shared memory, which might be some or all of the memory, would normally have some sort of protection via a memory-protection unit (MPU) or the protection functionality provide via a memory management unit (MMU), so that only OS kernel code would have access to that particular area, but this is not a hard requirement. In addition, if the cores have private memory caches, the shared memory might also be marked as non-cachable, or have cache write-through enabled, or have means for cache coherent accesses. (See my article on caches for more details.)

This arrangement will have separate OS code running on each core, each with their own scheduler to select which tasks to run. If a task running on a processor sends data via shared memory to a task on another processor that's blocked waiting for that message, the second processor's scheduler can't know that the data has been sent from the first core. This is where the interrupt comes in. If the sending task raises an interrupt on the receiving task's processor, the interrupt service routine can then inform the scheduler that data has arrived. If that unblocks a task, then the second processor's scheduler can recalculate which of the tasks to run and either run the receiving task, or at least make it queued for running if a higher priority task is active.

With FreeRTOS, support for AMP is given using messages, as discussed in the previous part of the document. Assuming a message buffer has been created, and that both processors know its handle, then the sending task code will write a message to it with `xMessageBufferSend`, as discussed in the last part. It will then generate an interrupt for the other processor, using either a customised version of the `sbSEND_COMPLETED` macro, which will be common to all message buffers without callback functions configured, or use an application supplied 'send completed' callback function if one was set up when the message buffer was created using `xMessageBufferCreateWithCallback` (as discussed in the previous part of the document) for buffer specific functionality. The receiving processor is interrupted and runs an ISR which calls the API function `xMessageBufferSendCompletedFromISR` (using the ISR version as we are in an ISR) to notify the scheduler and unblock the task blocked on that buffer. It does this using the state returned `xHigherPriorityTaskWoken` argument, indicating whether a task was unblocked or not.

It won't be detailed here, but this can be scaled to have multiple message buffers, with one reserved for control messages. This will send information about which message buffer has been updated, passing in the handle for that buffer. The receiving ISR can read that data and process the particular message buffer, just as for the previous case.

The `xMessageBufferCreateWithCallback` function discussed above, and in previous parts of the document, mentions 'callback' functions without much qualification, and I promised to talk about these in more detail in the previous part, so now might be a good time to talk about these. If you are already aware of what callback functions are then you can skip to the end of the section.

The code for all functions in C must reside somewhere in memory so that the processor can fetch and execute the instructions. Therefore, they have an associated start address and thus can have a pointer that points to them. There is an added complication that the functions might have arguments and might return a value. Nonetheless, we can define a pointer to a function with a particular set of arguments and return type. This would normally be via a C `typedef` type definition (though can be done without, but using a `typedef` is simpler). For example, if we want a pointer to a function whose prototype is a single integer argument and returns an integer, then we can define a type that is a pointer to these types of functions as, for example: `typedef int (*pFunc_t)(int)`

This defines a type, called `pFunc_t`, which is a pointer to a function whose prototype is `int <funcname>(int)`. We can use this to create a pointer to a function, and then call that function using the pointer.

```
typedef int (*pFunc_t)(int); // Define a pointer to a function type

int func (int value) {       // A function
    return value + 1;
}
int main(int argc, char* argv[]) {
   int valp1;
   pFunc_t pFunc;            // A pointer to a function

   pFunc = func;             // Set the point to the func function
   valp1 = (*pFunc)(10);     // Call the function
   printf("%d", valp1);
}
```

In the above example, the program, when compiled and run, will print `11`. So, now that we can create a pointer to one of our own functions, that pointer can be passed to another function which can use it to call our function at certain times from within its own code. And this is what the idle and tick hook functions are. The OS takes care of the pointers (the name of a function, without arguments, is the pointer to that function, as seen in the above example when assigning `pFunc`) and, when the hooks are enabled, the user code must supply the functions with the correct name and prototype (which is just `void` *`name`* `(void)`). We will look at these when discussing the demonstration program in the next part of the document. So, this is what callback functions are. Just ordinary functions, which can have a pointer point to them, and that pointer can be passed to another piece to call our function at certain points.

Symmetric Multi-Processor

With SMP, a single instance of the OS is run, and so has only a single scheduler for all cores, and no interrupts between cores are required to support this. The OS must now handle running tasks across multiple cores, and this changes the rules slightly. Before, with a single core, tasks with a higher priority would always preempt lower priority tasks (if using preemptive scheduling. This might not be the case when multiple cores are available as the lower priority task might be able to run on another core, if no higher priority task is running there. It might also be prudent to restrict which of the available cores a particular task is run, and so a 'mask' might be set to indicate which cores are allowed for a given task, from a single core to all available cores.

Apart from this, the tasks are run just as in the single core case. The scheduler and memory (perhaps just a partial region) are common, and so each task running on whatever core has the same access to OS objects, such as queues and semaphores etc., and can use them in the same way as for the single core case.

FreeRTOS provides SMP support, and some addition API functions are provided. The `xTaskCreateAffinitySet` is much like the `xCreateTask` function but allows specification of an 'affinity mask', which is a bitmapped value of the core on which the task can be run. This can be changed at run-time with `vTaskCoreAffinitySet`. Specific SMP `FreeRTOSConfig.h` settings can also be configured. The number of available cores is set with the `configNUMBER_OF_CORES` definition and the affinity API functions are included when the `configUSE_CORE_AFFINITY` definition is set to 1. A `configRUN_MULTIPLE_PRIORITIES` definition, when set to 1, allows different priority tasks to run simultaneously to allow higher and lower cores to run in parallel on different cores.

Use of Linux in Embedded Systems

In this document we have been looking at real-time operating systems and their use in embedded systems. In more recent times what might be considered embedded system applications have used complex operating systems such as Linux, and this has been the case on projects I have worked on in recent years, so are worth a mention here, for context. Strictly speaking, though, these Linux based systems tend to be on what would be called single board computers. Many FPGA devices will have a 'hard processor system', which is dedicated, non-reprogrammable, logic that might include one or more multi-core processors such as ARM Cortex, along with other SoC features such as memory sub-systems, including MMUs and DDR interfaces, and dedicated peripherals and input and output interfaces.

A Linux operating system might normally expect to be run on a traditional computer and to boot, via a boot loader, from a storage drive such as an HDD or SSD. It will initialise and start up some OS processes (daemons) until it gets to a state to have user input, such as from a console in the simplest case. When used in an embedded system, the storage would now come from a RAM drive and, instead of waiting for user input to run one or more programs, would be configured to automatically start the embedded application executable to run the desired embedded functionality. Nonetheless, the Linux OS is still a fully-fledged virtual memory based system, running multiple processes (daemons and programs) with, potentially, multiple threads (tasks), just as if running on a PC, say. So when would Linux be used over the kind of RTOS systems we have been talking about?

An OS like Linux is a very mature OS with in-built support for a whole host of systems, including for multi-core and multi-processors, networking like Ethernet, and drivers for a whole host of standard peripherals, such as USB. In other words, a lot of functionality that might be needed for an embedded system solution is already provided. Therefore, only the functions specific to the application need to be developed. This comes at a cost though. The ICs with the logic to be able support Linux will be larger and more costly. Linux's memory requirements, both for the OS code (flash) and the data, will be much larger, possibly requiring external DDR memory (with accompanying interface logic from the IC to drive it). All adding cost. Linux will also have more requirements regarding processor speed, requiring faster clock rates and/or more processors. Again, this makes the logic larger but also increases the power requirements. Therefore, there is a trade-off between complexity, cost of the hardware, and power requirements versus the reduction in development time for functionality provided by such a complex OS. This will make sense for larger more complex applications, and this is why one sees Linux based systems being deployed in many systems.

RTOS's, such as FreeRTOS, target systems with requirements that have lower processor performance and complexity requirements, with well-defined interfaces to measure and control the external hardware. Driver code may be bundled with the OS for a variety of functionalities that can be used by the application, but more software for the application may have to be written if this is not available in the accompanying libraries. It does mean, though, that only code needed for the application has to be present, and the requirements on hardware are much less, particularly memory, which can be orders of magnitude smaller. So, applications that are deeply embedded, simple in scope, sensitive to cost, and light on hardware requirements would be suitable for these kinds of real-time operating systems.

Conclusions

In this third part of the document, we finished up looking at the basic features of an RTOS and looked at FreeRTOS examples for these. In particular, notifications provided a simpler but more efficient means of communication between tasks, whilst software timers provided a means to have timing functionality that does not have such strict timing requirements as for a hardware timer. FreeRTOS also provides means for user code to be called to process interrupts and exceptions, and hooks to be called when the internal tick and idle tasks are run.

We concluded with a look at two architectures for multi-core systems—asymmetric and symmetric multi-processor. It was seen that not much is needed to support this, with only AMP needs extra, but simple, hardware giving the means for a core to send an interrupt to another core to indicate that a message has been sent.

In this third part, we have finished the introduction to real-time operating system functionality, though this is by no means all that is available in differing RTOS code (including FreeRTOS). In the next part a more practical look at the specifics of configuring and porting FreeRTOS to a RISC-V ISS based C++ system model will be undertaken. This may not be of interest to everyone but does discuss more details of some of the concepts covered, cross referencing what has been discussed up until now with applying it to the practicality of getting FreeRTOS to run on a system, even if a virtual one.

Part 4: FreeRTOS porting

Introduction

In this last part of the document on the series on real-time operating systems I want to look at the configuring of FreeRTOS for a given platform, using the set up for a RISC-V instruction set simulator (ISS) based C++ system model. This part will look at the configuration files in some detail, discussing some of the concepts used along the way. We will then look at the demonstration application code and hardware drivers, its use of FreeRTOS, and how to compile it, using a linker script, along with some startup code, before explaining how to run it. Reference to a real-world RISC-V FreeRTOS based embedded system example finishes up the discussions.

By the end of this part, along with the previous part, it is hoped that there is now an appreciation of what an RTOS's main features are and a general sense of how they work, as well as how to configure FreeRTOS, and use the RISC-V `rv32` ISS as a platform for further experimentation, starting from the provided simple demonstration application.

FreeRTOS on rv32 ISS

So now we have had a look at FreeRTOS and its features in the previous parts of the document, I want to back that information up with an actual example that can be run, and one that requires no specific platform or hardware. Therefore, we will use the instruction set simulator that is part of my riscV repository. With this setup you can experiment with the operating system and its API in a virtual environment. The ISS is extensible, so the intrepid reader can add models of different peripherals to the base system to try out interacting with these from the RTOS. More details of how to do this can be found in the manual for the ISS.

The versions used for this discussion, and in the examples, are listed below, though later versions will probably work just as well.

- FreeRTOS: version202212.01
- rv32: version 0.28

My recommendation is to compile and run under a Linux environment (I have used Ubuntu 22.04LTS), but it has also been tried out in Windows under Msys2 with mingw64. The ISS requires a gcc/g++ toolchain to compile and relies on 'make', and

FreeRTOS, and the application code, require the RISC-V `gcc` toolchain to compile the OS and example code. At the time of writing, pre-built toolchains can be found for Windows and Linux, though I can't guarantee these links won't go stale at some point. For those wanting to get started straight away, pre-built ISS executables for Windows and Linux are checked in to the repository (in the `iss/` directory), as is a compiled binary for the FreeRTOS OS and the demonstration application (in the `freertos/` directory).

Minimum Requirements for FreeRTOS Kernel

The minimum requirements to run FreeRTOS are surprisingly small. Obviously, we need at least one processor—in our case this is a RISC-V core, but this need only be to the base RV32I specification standard. It also needs a real-time timer. This can be any timer, and the OS can be configured to use a user supplied real-time clock source, but the user must then supply the driver code. The RISC-V ISA defines the `mtime` and `mtimecmp` memory mapped registers as part of the *Zicsr* extensions. A "core local interrupt" (CLINT) specification, defines which other CSR registers must be present, relating to interrupts in the machine privilege mode, required to allow the timer to generate interrupts. These are:

- **`mstatus`**: contains the master interrupt enable control, and some interrupt status bits.
- **`mie`**: contains the individual enables for external, s/w and timer interrupts.
- **`mip`**: contains the pending status for external, s/w and timer interrupts.
- **`mvtec`**: Pointer to the base address of the ISR vector table.
- **`mepc`**: Used to store the address of the running program when the exception was taken.
- **`mcause`**: Contains the cause ID of the exception.

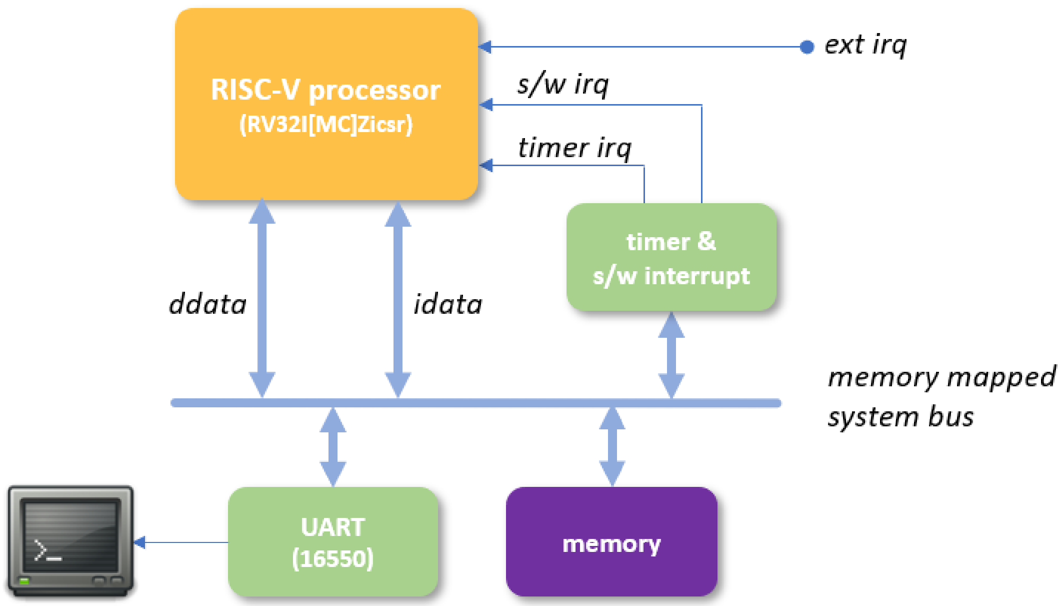
Necessarily, the CSRxx instructions of the *Zicsr* extensions must be implemented to access these registers. The OS also uses the `mscratch` CSR register and might use the `mtval` register (though I haven't found an instance of this). This set of *Zicsr* registers, and the functionality behind them, constitute the CLINT functionality that FreeRTOS can then make use of for RISC-V support.

We now have the minimum processor and CLINT functionality for timer and interrupts (external, software and timer), and let's assume that we also have memory for the OS, application programs and data—what else is required? Actually, perhaps unexpectedly, nothing. FreeRTOS can be configured and run under this environment, albeit not very usefully. Depending on the system requirements, at least one connection to the outside world would be required to make the system useful. One could connect an external interrupt to a signal as one source of input, and memory map a register bit, connected to an output (say a GPIO pin), and this might be all that's needed for the alarm example mentioned in the first part of the document. The application would then wait on the interrupt and set the GPIO output when triggered. A more realistic system would, of course, have more inputs and outputs, but FreeRTOS does not dictate what these are, which remains simply a function of the system requirements. So, as you see, the minimum requirements are few.

Virtual RISC-V System

The RISC-V instruction set simulator provides a minimum environment on which to port FreeRTOS. It implements the RV32I base specification and the minimum Zicsr extension to cover CLINT functionality, as well as supporting some other extensions. The most useful of these extensions are the C (compression instructions) and M (integer multiply and divide instructions) extensions, as using these reduce the size of the code footprint in memory—though FreeRTOS can be successfully compiled and run without the use of these.

The ISS also has a built in model of a UART, which can allow us to interact with the outside world, ostensibly to display information from the running programs. It also has a sparse memory model and can be interrupted externally via a registering a 'callback' function. This virtual machine, then, looks something like the diagram shown below.



This is the platform on to which FreeRTOS will be ported. In order to do that, we will have to provide some software files to configure FreeRTOS for this system, which is discussed in the next section.

Configuring FreeRTOS

FreeRTOS provides two files that must be customised for the system being targeted. Namely, `FreeRTOSConfig.h` and `freertos_riscv_chip_specific_extensions.h`. The former is the main configuration, whilst the second allows for any platform specific actions to be taken on certain operations, and platform specific configurations. These would normally be defined for a given platform and even for a particular application, as memory availability and usage may differ between different programs. In this section I will discuss the configuration files specific for the RISC-V rv32 ISS and its demonstration program.

`FreeRTOSConfig.h` consists of a set of definitions to be set in order to configure some basic features of the OS and to define which features will be compiled into the kernel as a given system may not wish to use them all and would want to minimise the size of the code. FreeRTOS is constructed in such a way that, by configuring the features to use though this file, one can compile all the kernel code files and allow the configuration to include or exclude code, depending on the settings, much simplifying the compilation process. The main configuration categories in this file are listed below:

- Memory configuration

- Task configuration
- Timer configuration
- Semaphores and mutex configuration
- Error checking
- API function inclusion

Memory Configuration

For memory configuration, there are various definitions to configure things like ‘stack’ sizes, size of the ‘heap’ and whether to use static or dynamic allocation. The main thrust of this set of configurations is to configure just the amounts of memory required for the target system. If this is constrained in the amount of memory available, the choice of settings will be critical in fitting the code into the system. Before starting on the memory configuration particulars, just a short word on what a stack and a heap actually are. If you already know this, then you can skip the next paragraph.

In C (and other languages), there are functions which have local variables, that must reside in memory somewhere for use by the functions’ code. Functions can call other functions, so the calling function’s data must be maintained, along with any processor register state being used, whist the called function (with its own bit of memory for its variables) is executing. When the called function returns, the memory for its local variables is no longer required and can be freed, whilst the calling functions registers are restored, and the local data is made active once more. In a multi-tasking environment, a separate stack would be kept for each task, sometimes known as ‘contexts’ and for interrupt service routines. The mechanism used is to create a ‘stack’, starting from the top of available RAM (usually) and allocate a suitable chunk when a function is called. If that function calls another function, then another chunk is allocated immediately below the caller’s data, of a suitable size for the called function. This is what the stack pointer is used for. Each time a function is called, the stack pointer, initialised to the top of RAM (say), will be moved to point to just below the allocated chunk, which is where any new chunk will be added, and so on. When a function returns, the stack pointer is moved to the top of the chunk for that function, freeing up the RAM for any new function call. When tasks are switched, the state is updated to point to, and use, the resuming task’s stack. Thus, the stack grows down as the call hierarchy deepens, and frees upwards, as functions return. Choosing the right stack size is a function of local variable requirements and call depths of the application functions. The heap is an area of memory used when dynamically allocating memory as code is executed. It usually starts at the lower end of memory, above areas that hold the global data and uninitialised data (more later). It is allocated when required, but is not so organised as a stack, as previously allocated blocks of data can be freed, creating fragmented memory usage.

The interrupt service routines, as mentioned above, have their own stack, and its size is defined with `configISR_STACK_SIZE_WORDS`. Note that the units are words, which is dependent on the processor architecture. For a 32-bit architecture, such as RISC-V RV32, a word is 4 bytes. Similarly, the `configMINIMAL_STACK_SIZE` defines a stack size for application code to use as the smallest task stack size, though larger stacks can be used for a task if needed. The only place within the OS I can find this definition used (for the RISC-V port at least) if for the idle task stack. The `configTOTAL_HEAP_SIZE` definition sets the allocated RAM for all the heap requirements. Since the heap is only used when dynamic memory allocation is used, this is only active if in that mode. The `configSUPPORT_DYNAMIC_ALLOCATION` definition is used to enable or disable dynamic allocation.

For the porting of FreeRTOS to the ISS and for the particular demonstration code, the memory will be configured as follows:

- `configISR_STACK_SIZE_WORDS`: 128
- `configMINIMAL_STACK_SIZE`: 128
- `configTOTAL_HEAP_SIZE`: 6Kbytes
- `configSUPPORT_DYNAMIC_ALLOCATION`: 1

Task Configuration

When configuring the tasks, we need to consider things like whether using pre-emptive or co-operative scheduling, how many priority levels we want and how many notification channels the tasks should have. In addition, there are some related settings to do with system tasks and co-routines.

The two scheduling types of ‘preemption’ and ‘cooperative’ where detailed in the first part of the document and are selected using the `configUSE_PREEMPTION` definition. The number of priorities available is defined with `configMAX_PRIORITIES`. Note that the system idle task always sits at priority 0 and consumes this priority number, with the lower numbers the lowest priorities. The timer task priority will use one of these priorities as well and would normally be expected to be the highest priority task (and not shared with any other task). So, this needs to be taken into consideration when allocating the number of priorities to be made available.

When discussing task notifications earlier in the previous part of the document, it was noted that a task can have more than one notification channel (as of FreeRTOS v10.4.0). The number is set with the `configTASK_NOTIFICATION_ARRAY_ENTRIES` definition.

For system idle and timer tick tasks some configurations modify their behaviour. The `config_IDLE_SHOULD_YIELD` is used when preemptive scheduling is enabled and there are tasks running at the same priority as the idle task (for some reason!). Normally, tasks at the same priority time slice to give a fair share of run time. If this definition is set to 1 then the idle task will immediately yield to another task at its priority. Actually, this situation can cause 'unfairness' issues and it is not recommended to run tasks at the same priority as the idle task. The definition `configUSE_TICKLESS_IDLE` is used for systems that have a low power mode where it is useful to shut down the tick interrupt and go into a sleep mode. Not all ports support this feature and for those that do, the application is responsible for putting the system to sleep and compensating for the lost ticks before it was woken up. This is usually done by enabling an idle hook function, provided by the application (`vApplicationIdleHook`), which is called when the OS runs the idle task. This is enabled by the `configUSE_IDLE_HOOK` definition. Similarly, there is a tick hook which is enabled using `configUSE_TICK_HOOK` and calls an application supplied function `vApplicationTickHook` when a tick interrupt occurs.

There are just a few last configurations to talk about before we move on from tasks. The `configMAX_TASK_NAME_LEN` definition sets the maximum length for a descriptive name string given to a task when it is created using the `xTaskCreate` API function. The descriptive name is mostly used for debugging to identify the different tasks but can also be used to get a handle for the task using the `xTaskGetHandle` API call. There is also a `configUSE_APPLICATION_TASK_TAG` definition where an application, if this is defined at 1, can set a tag value to a task (`vTaskSetApplicationTaskTag`). Its use is application specific and is usually used for trace and debug. Finally, if using coroutines, these can be enabled with `configUSE_CO_ROUTINES` set to 1, and the number of priorities set with the definition `configMAX_CO_ROUTINE_PRIORITIES`.

For the porting of FreeRTOS to the ISS and for the particular demonstration code, the tasks will be configured as follows:

- `configUSE_PREEMPTION`: 1
- `configMAX_PRIORITIES`: 5
- `configTASK_NOTIFICATION_ARRAY_ENTRIES`: 4
- `config_IDLE_SHOULD_YIELD`: 0
- `configUSE_IDLE_HOOK`: 0
- `configUSE_TICK_HOOK`: 0
- `configMAX_TASK_NAME_LEN`: 16
- `configUSE_APPLICATION_TASK_TAG`: 0
- `configUSE_CO_ROUTINES`: 0
- `configMAX_CO_ROUTINE_PRIORITIES`: 2

Timer Configuration

For a RISC-V system that has CLINT functionality, as discussed before, we must tell the OS where the memory mapped timer registers are located in the address space, and `configMTIME_BASE_ADDRESS` and `configMTIMECMP_BASE_ADDRESS` are used for this. We than have to define at what rate the timer is advancing. This is done with the definition `configCPU_CLOCK_HZ`. The naming here is a little confusing. If the timer is counting CPU clock cycles, then this naming is fine. But the timer may be coming from a different source running at a different rate to the CPU clock. Indeed, a CPU clock may be unreliable as a timer source as it might be variable in rate if it is slowed when the processor is only lightly loaded, or even suspended if it is in low power modes. So the value to set this to is the rate at which the *timer* is advancing, which may or may not be the CPU clock rate. The next timer configuration is the tick rate set with `configTICK_RATE_HZ`. This is effectively the time-slice rate (the inverse of the time-slice period).

If wanting to use software timers, then these must be enabled with the definition `configUSE_TIMERS`. As mentioned before, the timer task would normally be the higher priority, but it is still configurable to be any valid priority using the definition `configTIMER_TASK_PRIORITY`. The timer task's stack size also needs specifying with `configTIMER_TASK_STACK_DEPTH`. Application tasks can send commands to the timer task via API functions, and these are queued in a command queue. The depth of this queue is configured with `configTIMER_QUEUE_LENGTH`. When full, this will block a task attempting to send a command, and the length set in this definition dictates the blocking probabilities.

For the porting of FreeRTOS to the ISS and for the particular demonstration code, the timer functionality will be configured as follows:

- `configMTIME_BASE_ADDRESS`: `0xAFFFFFE0`
- `configMTIMECMP_BASE_ADDRESS`: `0xAFFFFFE8`
- `configCPU_CLOCK_HZ`: 1,000,000
- `configTICK_RATE_HZ`: 100 (ISS only, else 1000)
- `configUSE_TIMERS`: 0
- `configTIMER_TASK_PRIORITY`: (`configMAX_PRIORITIES` – 1)
- `configTIMER_TASK_STACK_DEPTH`: `configMINIMAL_STACK_SIZE`
- `configTIMER_QUEUE_LENGTH`: 4

Note that the `configTIMER_TASK_PRIORITY` definition still needs to be set even if software timers are disabled, as this is also used for the time slicing timer.

Semaphore and Mutex Configuration

There are only a few configurations for semaphores and mutexes. If counting semaphores are needed then `configUSE_COUNTING_SEMAPHORES` is set to 1, whilst mutexes are enabled with `configUSE_MUTEXES` set to 1 and recursive mutex features enabled with `configUSE_RECURSIVE_MUTEXES` set to 1. For the porting of FreeRTOS to the ISS and for the particular demonstration code, the semaphore and mutex functionality will be configured as follows:

- `configUSE_COUNTING_SEMAPHORES`: 1
- `configUSE_MUTEXES`: 0
- `configUSE_RECURSIVE_MUTEXES`: 0

Error Checking Configuration

Some error checking can be configured to do with stack and heap errors. For the stack it has a couple of methods to choose from, and these are chosen by setting the `configCHECK_FOR_STACK_OVERFLOW` value either 1 or 2 (with a value of 0 disabling overflow checks). The first method simply checks at each point a task is suspended whether the stack pointer is in a valid range, as dictated by the size of the stack set when the task was created. Although quick, this is not a guarantee that an overflow will be detected. The second method will fill a task's allocated stack space with a known value when the task is created. When a task is suspended the OS will check the last 16 bytes of the task's stack to ensure they are still at their initial known value. This is slightly less efficient than the first method, but better at catching overflows. If the overflow checks are enabled with either method, then if a check fails it will call a user supplied callback function, `vApplicationStackOverflowHook`, which is passed, as arguments, the handle of the failing task and the task name string. The application callback function must then try and handle this situation—perhaps deleting the task or, at the very least, flagging an error externally.

A callback for a memory allocation (malloc) failure can also be set up using the definition `configUSE_MALLOC_FAILED_HOOK`. There is only one check method here, so this is either 1 or 0. The application must supply a callback function, when enabled, which must be called `vApplicationMallocFailedHook`, that has no arguments. This callback is called if a malloc doesn't have enough space on the heap to complete the request. This is only used if using one of the FreeRTOS five sample memory allocation scheme source code files, named `heap_1.c` to `heap_5.c`. Each of these schemes varies in complexity and features, and trades off the size of the code. There is no configuration to select between these schemes (and the application may use its own memory allocation code), but one of the 5 source files is selected to be compiled along with the rest of the OS kernel. As with the semaphore callback, the malloc callback can attempt to handle and recover from this situation.

For the porting of FreeRTOS to the ISS and for the particular demonstration code, the error checking functionality will be configured as follows:

- `configCHECK_FOR_STACK_OVERFLOW`: 0
- `configUSE_MALLOC_FAILED_HOOK`: 0

API Configuration

Any given application may not use all of the API functionality supplied with the FreeRTOS kernel, and if a feature is not required, it can be disabled by setting the appropriate configuration to 0 instead of 1 to save code space. The list of API include definitions is given below along with their settings for the porting of FreeRTOS to the ISS and demonstration code:

- `INCLUDE_vTaskPrioritySet: 0`
- `INCLUDE_uxTaskPriorityGet: 0`
- `INCLUDE_vTaskDelete: 0`
- `INCLUDE_vTaskCleanUpResources: 0`

- `INCLUDE_vTaskSuspend: 0`
- `INCLUDE_vTaskDelayUntil: 1`
- `INCLUDE_vTaskDelay: 1`
- `INCLUDE_eTaskGetState: 0`
- `INCLUDE_xTimerPendFunctionCall: 0`
- `INCLUDE_xTaskAbortDelay: 0`
- `INCLUDE_xTaskGetHandle: 0`
- `INCLUDE_xSemaphoreGetMutexHolder: 0`

Most of these should be self-explanatory by now, so I won't go into any more details here.

Remaining Configuration

For the remaining configurations, we don't need to go into a great deal of detail for our purposes, but simply summarise what configurations are left.

FreeRTOS provides hooks for users to add trace facilities to collect data on how the application is behaving. These come as a set of macros that a user can redefine to add their trace functionality and are enabled through `configUSE_TRACE_FACILITY`. Accompanying this is `configUSE_STATS_FORMATTING` which enables the generation of stats tables of executed code.

The tick counters can be set to be 16-bit (`configUSE_16_BIT_TICKS`), to improve performance on 8 or 16 bit processors, but at the expense of limiting the maximum size of specifiable time period, for things like delays, to a maximum of 65535 ticks. The `configQUEUE_REGISTERY_SIZE` is only applicable when using a kernel aware debugger and specifies the maximum number queues and semaphores that can be registered for visibility with the debugger.

The `configUSE_PORT_OPTIMIZED_TASK_SELECTION` definition selects between a supplied platform independent generic task selection method, written in C, and a port specific method, written in assembler. The former is available for all ports, whilst the latter is available only for some ports. The RISC-V port does, fortunately, supply a processor specific implementation.

Finally, a macro can be defined (`configASSERT(x)`) to take certain platform specific actions if an assertion in the OS kernel fails. In the macro definition for the ISS and demonstration program, this disables interrupts and executes the `ebreak` instruction. As for the other configurations for the ISS, stats and tracing are disabled, and a queue registry size of 0 is set. Since there is a port optimized tasks selection method for RISC-V, this is enabled.

Chip Specific Extensions

The `freertos_risc_v_chip_specific_extensions.h` header is the file for RISC-V specific configurations, written in assembler. In it there is a definition to specify whether the platform has the `mtime`/`mtimecmp` registers implemented. This is the `portasmHAS_MTIME` definition. Related to this is the `portasmHAS_SIFIVE_CLINT`. This may be obsolete now, as I can't find any reference to this in the FreeRTOS repository except in the example configuration file headers. For the ISS, only `portasmHAS_MTIME` needs to be set to 1. Lastly, two macros are defined to save and restore any additional registers onto the stack when swapping tasks.

- `portasmSAVE_ADDITIONAL_REGISTERS`
- `portasmRESTORE_ADDITIONAL_REGISTER`

This might be done if custom extensions have been added to the RISC-V processor with their own registers in addition to the regular RISC-V set. Related to this is the definition `portasmADDITIONAL_CONTEXT_SIZE`, to specify the extra space required for these registers. For the ISS there are none, so the macros are left doing nothing and the additional context size is 0.

C run-time Startup Code

Before we can run a C main program certain things must be prepared. This is done with a startup assembly program called `crt0.S`, where *crt* stands for C run-time. The particular one used with the ISS and demo code (located in `freertos/rv32`) has been modified from one in the NEORV32 processor project. The code will contain the code run at start of execution from reset and follows certain steps before calling our main program. These are:

- Initialise all the RISC-V registers (`sp` and `gp` set for data supplied by the linker, discussed later)
- Copy all the initialised data in the `.data` section from its location in the binary image to its location in RAM
- Set the uninitialised/initialised-to-zero RAM data (`.bss` section) to 0
- Set the function argument registers (`a0` and `a1`) for `argc` and `argv`
- Call the main function

A few of things to note here. The code provides a basic trap handler in case any exceptions occur before calling `main`, and setting this up is the very first operation it does. This will be overridden by the user application program to set up a different trap handler, supplied by the OS. The copying of the `.data` section from the binary image to RAM is not required for the ISS, as the ISS loader sends this data directly to its final RAM location. This is also true of the code that calls C++ constructor initialisation and destructor functions, as we are not using C++. So these sections of code are not included if `RV32_ISS` is defined (as it will be in the `makefile`).

If the main program returns (and with FreeRTOS this is not usually the case), the code disables interrupts and places the return value in the `mscratch` register for any debugger to inspect, and `ebreak` is executed (the ISS can be configured to exit when an `ebreak` is executed).

Drivers

If you recall from earlier, the virtual system defined by the ISS has a UART transmitter model included within it. This has registers that match a 16550 UART, and so can be programmed to send characters to the console from which it is run. Some simple driver code is provided in `uart_driver.c` to read and write these registers to put out a character and provides an `int out_byte(int c)` API function for use by the application.

When called to output a character, the code reads the line status register (LSR) in a loop until it indicates it is ready to take a new byte. When it is okay to write, the provide character argument is written to the transmit holding register THR (with the same address as the receive buffer register—RBR).

Thus, we have a blocking UART output function, and this is all the driver code we need for our purposes.

The Demonstration Code

We now have all the foundations to compile our application, along with the FreeRTOS source code, the C run-time startup code, and the drivers we need. We now need to provide a `main()` function to be called once the system is initialised and call the FreeRTOS API functions to set up and run some tasks.

The simple demonstration code is in `freertos/demo/main.c`. This application makes use of a `printf` function to display formatted data over the UART. This open source code, in `freertos/rv32/printf.c`, is taken from Marco Paland's Tiny printf repository.

The main function initialises some parameter structures, for use by the task code, and then writes to the RISC-V processor's `mvtec` register to point to the FreeRTOS trap handler code for RISC-V. In the `freertos/rv32/rv32_freertos.h` header, two inline functions are defined to read (`csr_read()`) and write (`csr_write()`) CSR registers. Although these are C function calls, the code inside these functions is RISC-V assembly, making use of the `csrw` and `csrr` instructions of the *Zicsr* extensions. The main program then sets up two tasks, using `xTaskCreate`, both of which use the same function as the task code to run (`task()`), but with different parameters, supplied by the initialised structures, and passed into the `xTaskCreate` calls as pointers. A stack size of 256 bytes is also specified for each task, and a matching priority of 1, so these tasks will be time sliced with each other. The scheduler loop is then started with a call to the FreeRTOS API `xTaskStartScheduler` function, which, barring something going wrong, runs forever.

The function run by each task sets up an initial count value, extracted from the parameter structure passed in, along with a task ID, and then loops for a set count printing the task ID and the current count value, decrementing the count by 1 each iteration, and then delaying for one second. When the loop is complete, the function then loops forever calling `vTaskDelay` with a large delay time.

The `main.c` source code also provides tick and idle hooks, which just print out a message when called. By default, the hooks are disabled, but can be enabled in the `FreeRTOSConfig.h` file. Similarly, malloc failed and stack overflow hooks are provided and can be enabled in the configuration file. These basically just print a message and then call `ebreak` if executed.

The demo source code also defines interrupt and trap handler code in two specifically named functions:

- `freertos_risc_v_application_interrupt_handler()`
- `freertos_risc_v_application_exception_handler()`

For interrupts, the user function is called for any interrupt that is not the timer interrupt, as that is handled by the OS. All exceptions are sent to the application handler function. Both of the functions in the demo extract the value of the `mcause` CSR register and print it for debug purposes. In the normal execution of this simple demonstration, neither of these functions are expected to be called.

Compiling the Code

To compile FreeRTOS and the application code, a `makefile` is provided and from the `freertos/` directory, if `make` is executed, the OS and application is compiled into a `main.exe` RISC-V 32-bit executable. From a usage point of view this is all that's required, but it will be instructive to look in a little more detail at the steps to compile the operating system.

For our purposes, we only want to compile the FreeRTOS kernel. There are other libraries bundled alongside the FreeRTOS kernel, but this is outside of the scope of this particular document. There are surprising few files for the common part of the kernel. These all reside in `<FreeRTOS root>/FreeRTOS/Source` and are all C files. As mentioned before, the configuration will include or exclude functionality as required, so we can safely compile in all the C files in this directory. The headers for these files are all in an `include/` subdirectory, which must be added to the compile search path.

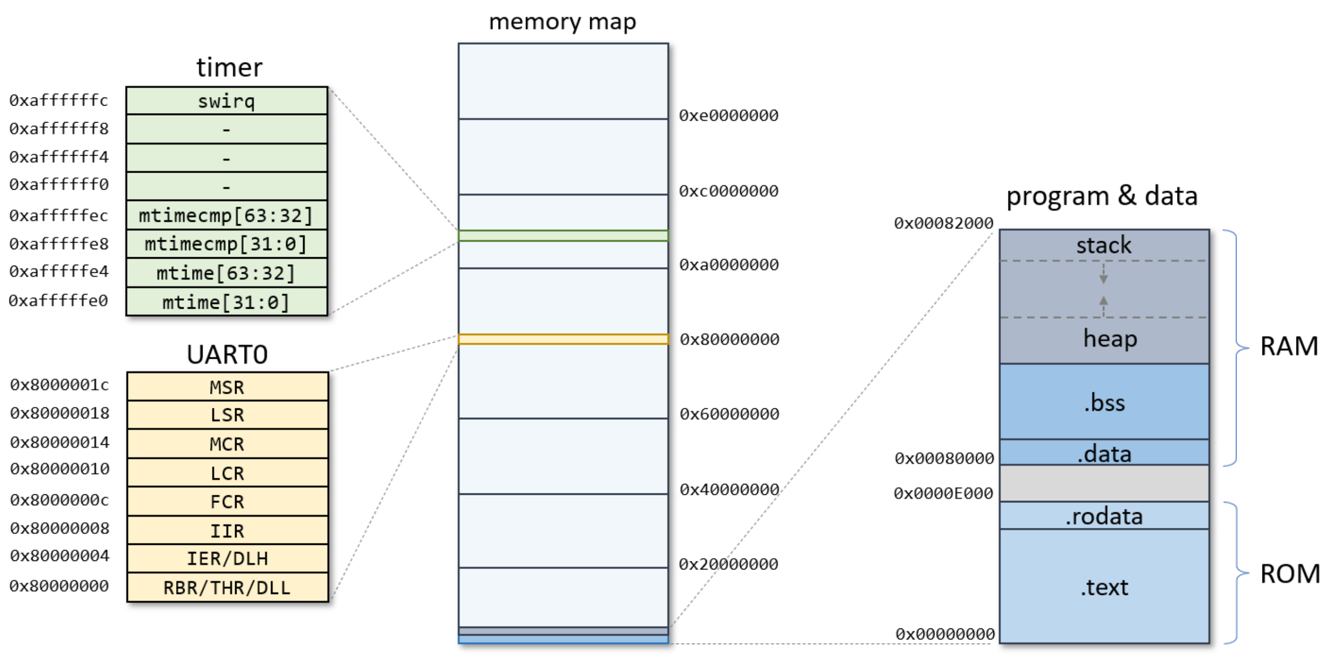
All the platform and compiler specific code is under the sub-directory `portable/`. For this demonstration and RISC-V system model, the relevant directory is `portable/GCC/RISC-V`. This contains both C (`port.c`) and assembly (`portASM.S`) source code, specific to the RISC-V processor which must be compiled/assembled with the rest of the OS code, and the directory included in the compilation search path to pick up the headers.

It was mentioned before that we have a choice of FreeRTOS provided memory allocation code. The source for this is in `portable/MemMang`, and a choice from `heap_1.c` to `heap_5.c` is available (and only one must be selected and compiled)—for the demonstration, `heap_1.c` is used (see the FreeRTOS Memory Management documentation for details on the different functionalities).

Linker script

One last important piece required to compile the code is a linker script to define all the parameters needed to place the compiled code in the correct places for our system. The `rv32.ld` file is the linker script for the demonstration. The starting point for this script was, again, taken from the NEORV32 project but I have stripped this right down to the bare bones to the point where it is unrecognisable from the original (but credit where it's due for the original source). This will make it much easier to understand what's going on. It might be helpful to have this file open in an editor whilst reading this section.

Below is a diagram of the memory map for the RISC-V ISS model, showing the location of the UART and timer peripherals, and the area where the program and data will reside, with details of the program sections. Note that none of this is to scale.



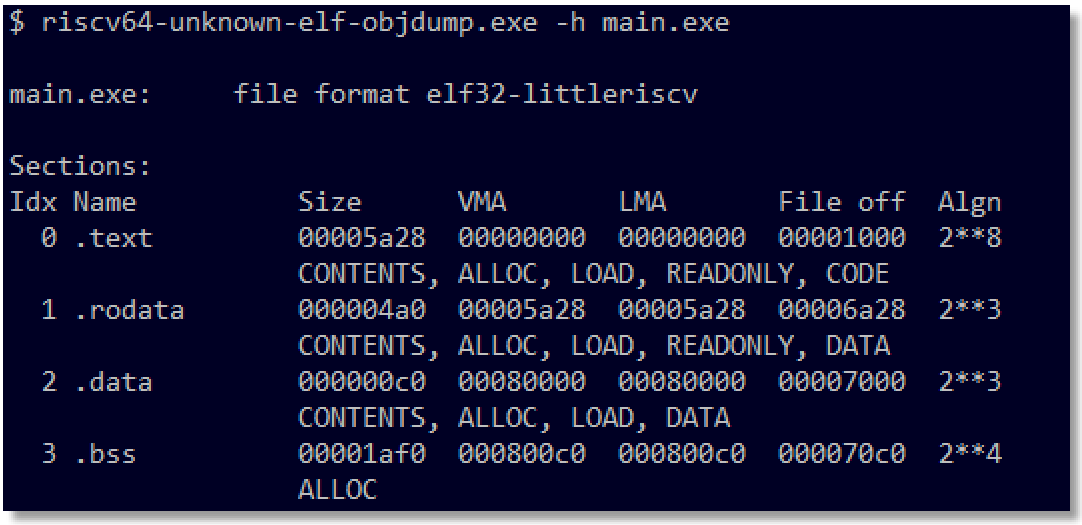
The program code is also known as 'text' and a .text section will be where this code goes in memory. The RISC-V processor will start executing from address 0, and so that is where the text section will be placed. If the program has any read-only data (e.g. constant definitions), this will go in a section `.rodata`, immediately above the `.text` section. These two sections might normally be in some non-volatile memory, such as ROM or flash in an embedded system, and so be separate from the variable data in a RAM. A RAM section will be defined starting from `0x80000`, and initialised data will be compiled into a `.data` section at the bottom of the RAM. Immediately above this a `.bss` section is define for uninitialised data, or data that is initialised to 0 (remember the `crt0.S` code initialised the entire `.bss` section to 0). BSS stands for "block start symbol" for historic reasons and hasn't been changed to be a more meaningful name since. The rest of the RAM is used when the program runs for the stack and heap, as discussed earlier.

The linker script will define these sections and their locations. The `rv32.ld` file first defines some basic architecture parameters, namely little-endian 32-bit compilation and RISC-V architecture. The ENTRY point is set to `_start` (as defined in `crt0.S`). After some definitions are set to define memory sizes and base addresses for ROM and RAM, the memory is defined. We will have a 56K ROM space and 8K RAM space, with ROM at 0 and RAM at `0x80000` addresses. This sets the base address and size of the ROM followed by the RAM, along with the permissions—so that the ROM is read and execute, whilst the RAM also includes writing.

The different sections are then defined. In a linker script, as each section is defined it keep tabs on where the address has got to (separate for each memory block—rom and ram in our case). Thus, the section for a block, unless specified otherwise, will be contiguous. The `.text` section is defined first, with a 4-byte alignment to start. The KEEP specifies that a section is included, even if nothing else references that section. Normally, code that is never referenced elsewhere in the code (from a library of functions, say) will be deleted by the linker to save code space. The very first bit of code, from `crt0.S` (where a `.text.crt0` section is defined), is the startup code and so is never called from anywhere, but we don't want the linker to delete it. The next line just tells the linker to include all the `.text` code sections (using an asterisk as a wild card), as defined in all the compiled source files. The text section is specified to go into the ROM (`> rom`).

The read only data section is then specified in a similar fashion, also to go in the ROM. The read-only data section is followed by the `.data` section, which differs in that it is now directed to RAM (`> ram`), and which not only matches `.data` but also `.sdata` and `.srodata` sections, with the 's' prefix denoting 'small'. These are data collected into small sections that would be useful to be in the proximity of the 'global pointer'. It defines a variable `__global_pointer$` which will be used by the `crt0.S` code to set the `gp` register of the processor. Why is it set equal to "`. + 0x800`"? Well, the "." means the current address—in this case the end of the `.data` section. The hex value is 2048 in decimal, and this is significant in that RISC-V load and store immediate instructions have 12 bits of immediate value and are signed, so have a range of -2048 to 2047. By setting the global pointer somewhere in the following RAM so that it can span from the start to 2047 bytes beyond the `0x800` offset, it is hoped that most memory accesses can all be immediately relative to the global pointer, saving instructions in calculating new addresses. Code will work wherever this is set (in RAM) but will be less efficient (potentially).

The last section is for the `.bss` data, also sent to RAM. This matches two sections, `.bss` and `.sbss`. This section also exports some definitions for the start and end of the section, used by `crt0.S` to zero this area of RAM. Finally, another definition is exported to define where the stack begins, also for use by `crt0.S` to set the stack pointer, which will be the last word of RAM. (Remember, the stack grows downwards.) Once the application is compiled, an object dump of the headers shows the relevant compiled sections (with the others just debug and attribute sections not shown):



```
$ riscv64-unknown-elf-objdump.exe -h main.exe

main.exe:     file format elf32-littleriscv

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00005a28  00000000  00000000  00001000  2**8
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000004a0  00005a28  00005a28  00006a28  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         000000c0  00080000  00080000  00007000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00001af0  000800c0  000800c0  000070c0  2**4
                  ALLOC
```

The four sections discussed above are shown, with their sizes. Both virtual and load (physical) memory are displayed for each section, but are both the same, as this system is not using an MMU with virtual memory support. The file offset shows where in the executable the section starts, and the alignment in memory the start of the section will have. Underneath the section are attributes, defining things like whether code or data, or whether read-only etc. This should tally with our expectations from the linker script.

Running the Code

Having compiled the code and generated a `main.exe` executable, we can now run this on the ISS. From the `freertos/` directory, where `main.exe` is compiled, we can run the ISS (for Linux—use `rv32.exe` for Windows):

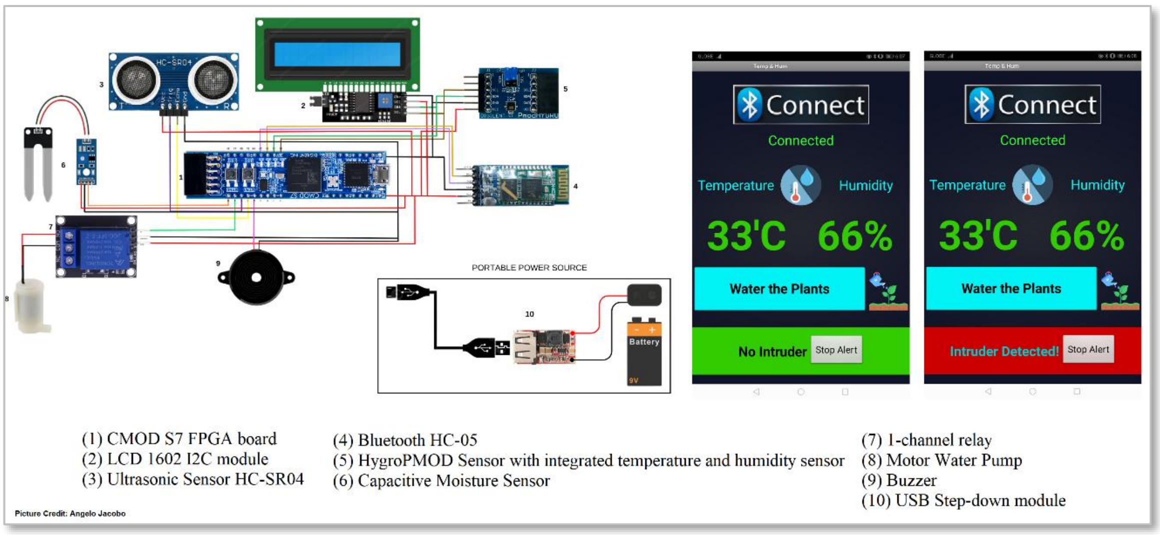
```
../iss/rv32 -t main.exe
```

Actually, the `-t` option is not really needed as there is an `rv32.ini` file that specifies the configurations, including running an executable called `main.exe`. The code will run forever and can be terminated with `<ctrl–C>` (embedded systems are meant to run indefinitely). Using the `rv32.ini` file, or through the command line options, the ISS model can be run for a set number of instructions, or have run-time disassembly enabled, and even dump memory and registers (regular and CSR) at the end (though not when terminated with `<ctrl-C>`). The ISS also supports remote debugging with `gdb` (and hence the Eclipse IDE)—see the ISS manual for details of the configuration options and debugging.

Real-world application

In this part of the document, I have provided a virtual platform in the form of an instruction set simulator based model by way of a usable example (I don't like to talk about these things without actually illustrating that they work). In other documents I might also provide a hardware example using FPGA platforms. For this subject, however, I needn't do so since one of my former Mentees, Angelo Jacobo, has constructed a working hardware example using his own fully pipelined RISC-V softcore processor design, running a real-time embedded software application with FreeRTOS. There is a description and the source code in github repository for his processor, along with a video of the application running.

This is a 'smart garden assistant' that monitors plant temperature and humidity which can be displayed on an LCD, and also be transmitted, via Bluetooth, to a mobile application. It also has an ultrasonic sensor for proximity warnings, setting off an alarm when anything is too close, which can be cancelled from the mobile app. Finally, it monitors for soil moisture content and will activate a water pump when too dry. All this is controlled by the FreeRTOS application, running on the RISC-V softcore, programmed onto an Intel Spartan-7 FPGA.



Take time out to have a look at this application as a real-world working example of a FreeRTOS based RISC-V embedded system which is complex enough to do real control of external components, along with remote access, but also simple enough to study the application and its use of FreeRTOS, with the knowledge gained from this document.

Conclusions

In this last part on real-time operating systems, we have finished up by taking a practical look at configuring and compiling FreeRTOS, along with a demonstration application, specifically to run on a virtual platform provided by the `rv32` RISC-V ISS based system model.

The minimum requirements for a system were discussed and it turned out that not much is required. For a RISC-V based system, the `mtime` memory mapped timer registers and *Zicsr* extension interrupt features are required (know together as the core local interrupt, or CLINT) are also required—and that's it! However, the system is not much use without some external interaction, and the model provides this using an internal model of a UART to allow messages to be displayed from the running code.

We looked at the configuration files settings in some detail, discussing what each of them do and explained some concepts along the way. We looked at what source code was needed from the FreeRTOS kernel code, along with our application code, to compile to an executable, with the choices we had to make along the way. The FreeRTOS code, as was seen, is divided into common code and then target processor and compiler specific code. Defining a linker script was discussed to map the compiled code to the memory layout specifics of the system, and a minimalist script was looked at to show how this is done.

This part of the document finished off by referencing an external project that uses FreeRTOS running on a softcore RISC-V RV32I processor, with an embedded multi-tasking application to control real hardware and interact with the real-world.

It is hoped that the information in this part of the document has covered all the steps required to configure and use FreeRTOS on other RISC-V based system, or even for other processors, and that the document as a whole give some understanding of the basic common features of real-time embedded operating systems and their use, using FreeRTOS as the case study.