FreeRTOS

Christopher Svec

The Architecture of Open Source Applications
Volume II: Structure, Scale, and a Few More Fearless Hacks

FreeRTOS (pronounced "free-arr-toss") is an open source real-time operating system (RTOS) for embedded systems. FreeRTOS supports many different architectures and compiler toolchains, and is designed to be "small, simple, and easy to use".

FreeRTOS is under active development, and has been since Richard Barry started work on it in 2002. As for me, I'm not a developer of or contributor to FreeRTOS, I'm merely a user and a fan. As a result, this chapter will favor the "what" and "how" of FreeRTOS's architecture, with less of the "why" than other chapters in this book.

Like all operating systems, FreeRTOS's main job is to run tasks. Most of FreeRTOS's code involves prioritizing, scheduling, and running user-defined tasks. Unlike all operating systems, FreeRTOS is a real-time operating system which runs on embedded systems.

By the end of this chapter I hope that you'll understand the basic architecture of FreeRTOS. Most of FreeRTOS is dedicated to running tasks, so you'll get a good look at exactly how FreeRTOS does that.

If this is your first look under the hood of an operating system, I also hope that you'll learn the basics about how any OS works. FreeRTOS is relatively simple, especially when compared to Windows, Linux, or OS X, but all operating systems share the same basic concepts and goals, so looking at any OS can be instructive and interesting.

3.1. What is "Embedded" and "Real-Time"?

"Embedded" and "real-time" can mean different things to different people, so let's define them as FreeRTOS uses them.

An embedded system is a computer system that is designed to do only a few things, like the system in a TV remote control, in-car GPS, digital watch, or pacemaker. Embedded systems are typically smaller and slower than general purpose computer systems, and are also usually less expensive. A typical low-end embedded system may have an 8-bit CPU running at 25MHz, a few KB of RAM, and maybe 32KB of flash memory. A higher-end embedded system may have a 32-bit CPU running at 750MHz, a GB of RAM, and multiple GB of flash memory.

Real-time systems are designed to do something within a certain amount of time; they guarantee that stuff happens when it's supposed to.

A pacemaker is an excellent example of a real-time embedded system. A pacemaker must contract the heart muscle at the right time to keep you alive; it can't be too busy to respond in time. Pacemakers and other real-time embedded systems are carefully designed to run their tasks on time, every time.

3.2. Architecture Overview

FreeRTOS is a relatively small application. The minimum core of FreeRTOS is only three source (`.c`) files and a handful of header files, totalling just under 9000 lines of code, including comments and blank lines. A typical binary code image is less than 10KB.

FreeRTOS's code breaks down into three main areas: tasks, communication, and hardware interfacing.

- **Tasks:** Almost half of FreeRTOS's core code deals with the central concern in many operating systems: tasks. A task is a user-defined C function with a given priority. `tasks.c` and `task.h` do all the heavy lifting for creating, scheduling, and maintaining tasks.
- **Communication:** Tasks are good, but tasks that can communicate with each other are even better! Which brings us to the second FreeRTOS job: communication. About 40% of FreeRTOS's core code deals with communication. `queue.c` and `queue.h` handle FreeRTOS communication. Tasks and interrupts use queues to send data to each other and to signal the use of critical resources using semaphores and mutexes.
- **The Hardware Whisperer:** The approximately 9000 lines of code that make up the base of FreeRTOS are hardware-independent; the same code runs whether FreeRTOS is running on the humble 8051 or the newest, shiniest ARM core. About 6% of FreeRTOS's core code acts a shim between the hardware-independent FreeRTOS core and the hardware-dependent code. We'll discuss the hardware-dependent code in the next section.

Hardware Considerations

The hardware-independent FreeRTOS layer sits on top of a hardware-dependent layer. This hardware-dependent layer knows how to talk to whatever chip architecture you choose. Figure 3.1 shows FreeRTOS's layers.

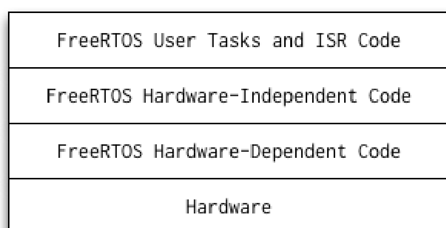

Figure 3.1: FreeRTOS software layers

FreeRTOS ships with all the hardware-independent as well as hardware-dependent code you'll need to get a system up and running. It supports many compilers (CodeWarrior, GCC, IAR, etc.) as well as many processor architectures (ARM7, ARM Cortex-M3, various PICs, Silicon Labs 8051, x86, etc.). See the FreeRTOS website for a list of supported architectures and compilers.

FreeRTOS is highly configurable by design. FreeRTOS can be built as a single CPU, bare-bones RTOS, supporting only a few tasks, or it can be built as a highly functional multicore beast with TCP/IP, a file system, and USB.

Web2PDF
converted by

FreeRTOS

Christopher Svec

The Architecture of Open Source Applications
Volume II: Structure, Scale, and a Few More Fearless Hacks

FreeRTOS (pronounced "free-arr-toss") is an open source real-time operating system (RTOS) for embedded systems. FreeRTOS supports many different architectures and compiler toolchains, and is designed to be "small, simple, and easy to use".

FreeRTOS is under active development, and has been since Richard Barry started work on it in 2002. As for me, I'm not a developer of or contributor to FreeRTOS, I'm merely a user and a fan. As a result, this chapter will favor the "what" and "how" of FreeRTOS's architecture, with less of the "why" than other chapters in this book.

Like all operating systems, FreeRTOS's main job is to run tasks. Most of FreeRTOS's code involves prioritizing, scheduling, and running user-defined tasks. Unlike all operating systems, FreeRTOS is a real-time operating system which runs on embedded systems.

By the end of this chapter I hope that you'll understand the basic architecture of FreeRTOS. Most of FreeRTOS is dedicated to running tasks, so you'll get a good look at exactly how FreeRTOS does that.

If this is your first look under the hood of an operating system, I also hope that you'll learn the basics about how any OS works. FreeRTOS is relatively simple, especially when compared to Windows, Linux, or OS X, but all operating systems share the same basic concepts and goals, so looking at any OS can be instructive and interesting.

3.1. What is "Embedded" and "Real-Time"?

"Embedded" and "real-time" can mean different things to different people, so let's define them as FreeRTOS uses them.

An embedded system is a computer system that is designed to do only a few things, like the system in a TV remote control, in-car GPS, digital watch, or pacemaker. Embedded systems are typically smaller and slower than general purpose computer systems, and are also usually less expensive. A typical low-end embedded system may have an 8-bit CPU running at 25MHz, a few KB of RAM, and maybe 32KB of flash memory. A higher-end embedded system may have a 32-bit CPU running at 750MHz, a GB of RAM, and multiple GB of flash memory.

Real-time systems are designed to do something within a certain amount of time; they guarantee that stuff happens when it's supposed to.

A pacemaker is an excellent example of a real-time embedded system. A pacemaker must contract the heart muscle at the right time to keep you alive; it can't be too busy to respond in time. Pacemakers and other real-time embedded systems are carefully designed to run their tasks on time, every time.

3.2. Architecture Overview

FreeRTOS is a relatively small application. The minimum core of FreeRTOS is only three source (`.c`) files and a handful of header files, totalling just under 9000 lines of code, including comments and blank lines. A typical binary code image is less than 10KB.

FreeRTOS's code breaks down into three main areas: tasks, communication, and hardware interfacing.

- **Tasks:** Almost half of FreeRTOS's core code deals with the central concern in many operating systems: tasks. A task is a user-defined C function with a given priority. `tasks.c` and `task.h` do all the heavy lifting for creating, scheduling, and maintaining tasks.
- **Communication:** Tasks are good, but tasks that can communicate with each other are even better! Which brings us to the second FreeRTOS job: communication. About 40% of FreeRTOS's core code deals with communication. `queue.c` and `queue.h` handle FreeRTOS communication. Tasks and interrupts use queues to send data to each other and to signal the use of critical resources using semaphores and mutexes.
- **The Hardware Whisperer:** The approximately 9000 lines of code that make up the base of FreeRTOS are hardware-independent; the same code runs whether FreeRTOS is running on the humble 8051 or the newest, shiniest ARM core. About 6% of FreeRTOS's core code acts a shim between the hardware-independent FreeRTOS core and the hardware-dependent code. We'll discuss the hardware-dependent code in the next section.

Hardware Considerations

The hardware-independent FreeRTOS layer sits on top of a hardware-dependent layer. This hardware-dependent layer knows how to talk to whatever chip architecture you choose. Figure 3.1 shows FreeRTOS's layers.

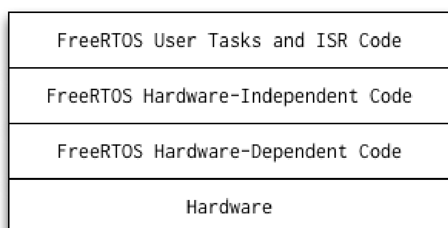

Figure 3.1: FreeRTOS software layers

FreeRTOS ships with all the hardware-independent as well as hardware-dependent code you'll need to get a system up and running. It supports many compilers (CodeWarrior, GCC, IAR, etc.) as well as many processor architectures (ARM7, ARM Cortex-M3, various PICs, Silicon Labs 8051, x86, etc.). See the FreeRTOS website for a list of supported architectures and compilers.

FreeRTOS is highly configurable by design. FreeRTOS can be built as a single CPU, bare-bones RTOS, supporting only a few tasks, or it can be built as a highly functional multicore beast with TCP/IP, a file system, and USB.

Web2PDF
converted by Web2PDFConvert

Configuration options are selected in FreeRTOSConfig.h by setting various #defines. Clock speed, heap size, mutexes, and API subsets are all configurable in this file, along with many other options. Here are a few examples that set the maximum number of task priority levels, the CPU frequency, the system tick frequency, the minimal stack size and the total heap size:

```
#define configMAX_PRIORITIES      ( ( unsigned portBASE_TYPE ) 5 )
#define configCPU_CLOCK_HZ        (            UL )
#define configTICK_RATE_HZ        ( ( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE  ( ( unsigned short ) 100 )
#define configTOTAL_HEAP_SIZE     ( ( size_t ) ( 4 * 1024 ) )
```

Hardware-dependent code lives in separate files for each compiler toolchain and CPU architecture. For example, if you're working with the IAR compiler on an ARM Cortex-M3 chip, the hardware-dependent code lives in the FreeRTOS/Source/portable/IAR/ARM_CM3/ directory. portmacro.h declares all of the hardware-specific functions, while port.c and portasm.s contain all of the actual hardware-dependent code. The hardware-independent header file portable.h #include's the correct portmacro.h file at compile time. FreeRTOS calls the hardware-specific functions using #define'd functions declared in portmacro.h.

Let's look at an example of how FreeRTOS calls a hardware-dependent function. The hardware-independent file tasks.c frequently needs to enter a critical section of code to prevent preemption. Entering a critical section happens differently on different architectures, and the hardware-independent tasks.c does not want to have to understand the hardware-dependent details. So tasks.c calls the global macro portENTER_CRITICAL(), glad to be ignorant of how it actually works. Assuming we're using the IAR compiler on an ARM Cortex-M3 chip, FreeRTOS is built with the file FreeRTOS/Source/portable/IAR/ARM_CM3/portmacro.h which defines portENTER_CRITICAL() like this:

```
#define portENTER_CRITICAL()    vPortEnterCritical()
```

vPortEnterCritical() is actually defined in FreeRTOS/Source/portable/IAR/ARM_CM3/port.c. The port.c file is hardware-dependent, and contains code that understands the IAR compiler and the Cortex-M3 chip. vPortEnterCritical() enters the critical section using this hardware-specific knowledge and returns to the hardware-independent tasks.c.

The portmacro.h file also defines an architecture's basic data types. Data types for basic integer variables, pointers, and the system timer tick data type are defined like this for the IAR compiler on ARM Cortex-M3 chips:

```
#define portBASE_TYPE   long              // Basic integer variable type
#define portSTACK_TYPE  unsigned long     // Pointers to memory locations
typedef unsigned portLONG portTickType;  // The system timer tick type
```

This method of using data types and functions through thin layers of #defines may seem a bit complicated, but it allows FreeRTOS to be recompiled for a completely different system architecture by changing only the hardware-dependent files. And if you want to run FreeRTOS on an architecture it doesn't currently support, you only have to implement the hardware-dependent functionality which is much smaller than the hardware-independent part of FreeRTOS.

As we've seen, FreeRTOS implements hardware-dependent functionality with C preprocessor #define macros. FreeRTOS also uses #define for plenty of hardware-independent code. For non-embedded applications this frequent use of #define is a cardinal sin, but in many smaller embedded systems the overhead for calling a function is not worth the advantages that "real" functions offer.

3.3. Scheduling Tasks: A Quick Overview

Task Priorities and the Ready List

Each task has a user-assigned priority between 0 (the lowest priority) and the compile-time value of configMAX_PRIORITIES-1 (the highest priority). For instance, if configMAX_PRIORITIES is set to 5, then FreeRTOS will use 5 priority levels: 0 (lowest priority), 1, 2, 3, and 4 (highest priority).

FreeRTOS uses a "ready list" to keep track of all tasks that are currently ready to run. It implements the ready list as an array of task lists like this:

```
static xList pxReadyTasksLists[ configMAX_PRIORITIES ];  /* Prioritised ready tasks.  */
```

pxReadyTasksLists[0] is a list of all ready priority 0 tasks, pxReadyTasksLists[1] is a list of all ready priority 1 tasks, and so on, all the way up to pxReadyTasksLists[configMAX_PRIORITIES-1].

The System Tick

The heartbeat of a FreeRTOS system is called the system tick. FreeRTOS configures the system to generate a periodic tick interrupt. The user can configure the tick interrupt frequency, which is typically in the millisecond range. Every time the tick interrupt fires, the vTaskSwitchContext() function is called. vTaskSwitchContext() selects the highest-priority ready task and puts it in the pxCurrentTCB variable like this:

```
/* Find the highest-priority queue that contains ready tasks. */
while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopReadyPriority ] ) ) )
{
    configASSERT(   uxTopReadyPriority   );
    --uxTopReadyPriority;
}

/* listGET_OWNER_OF_NEXT_ENTRY walks through the list, so the tasks of the same
priority get an equal share of the processor time. */
listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB,  &(  pxReadyTasksLists[  uxTopReadyPriority  ]  )  );
```

Before the while loop starts, uxTopReadyPriority is guaranteed to be greater than or equal to the priority of the highest-priority ready task. The while() loop starts at priority level uxTopReadyPriority and walks down through the pxReadyTasksLists[] array to find the highest-priority level with ready tasks. listGET_OWNER_OF_NEXT_ENTRY() then grabs the next ready task from that priority level's ready list.

Now pxCurrentTCB points to the highest-priority task, and when vTaskSwitchContext() returns the hardware-dependent code starts running that task.

Those nine lines of code are the absolute heart of FreeRTOS. The other 8900+ lines of FreeRTOS are there to make sure those nine lines are all that's needed to

Web2PDF
converted by

Configuration options are selected in `FreeRTOSConfig.h` by setting various `#defines`. Clock speed, heap size, mutexes, and API subsets are all configurable in this file, along with many other options. Here are a few examples that set the maximum number of task priority levels, the CPU frequency, the system tick frequency, the minimal stack size and the total heap size:

```
#define configMAX_PRIORITIES      ( ( unsigned portBASE_TYPE ) 5 )
#define configCPU_CLOCK_HZ        (              UL )
#define configTICK_RATE_HZ        ( ( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE  ( ( unsigned short ) 100 )
#define configTOTAL_HEAP_SIZE     ( ( size_t ) ( 4 * 1024 ) )
```

Hardware-dependent code lives in separate files for each compiler toolchain and CPU architecture. For example, if you're working with the IAR compiler on an ARM Cortex-M3 chip, the hardware-dependent code lives in the `FreeRTOS/Source/portable/IAR/ARM_CM3/` directory. `portmacro.h` declares all of the hardware-specific functions, while `port.c` and `portasm.s` contain all of the actual hardware-dependent code. The hardware-independent header file `portable.h` `#include`'s the correct `portmacro.h` file at compile time. FreeRTOS calls the hardware-specific functions using `#define`'d functions declared in `portmacro.h`.

Let's look at an example of how FreeRTOS calls a hardware-dependent function. The hardware-independent file `tasks.c` frequently needs to enter a critical section of code to prevent preemption. Entering a critical section happens differently on different architectures, and the hardware-independent `tasks.c` does not want to have to understand the hardware-dependent details. So `tasks.c` calls the global macro `portENTER_CRITICAL()`, glad to be ignorant of how it actually works. Assuming we're using the IAR compiler on an ARM Cortex-M3 chip, FreeRTOS is built with the file `FreeRTOS/Source/portable/IAR/ARM_CM3/portmacro.h` which defines `portENTER_CRITICAL()` like this:

```
#define portENTER_CRITICAL()   vPortEnterCritical()
```

`vPortEnterCritical()` is actually defined in `FreeRTOS/Source/portable/IAR/ARM_CM3/port.c`. The `port.c` file is hardware-dependent, and contains code that understands the IAR compiler and the Cortex-M3 chip. `vPortEnterCritical()` enters the critical section using this hardware-specific knowledge and returns to the hardware-independent `tasks.c`.

The `portmacro.h` file also defines an architecture's basic data types. Data types for basic integer variables, pointers, and the system timer tick data type are defined like this for the IAR compiler on ARM Cortex-M3 chips:

```
#define portBASE_TYPE  long              // Basic integer variable type
#define portSTACK_TYPE unsigned long     // Pointers to memory locations
typedef unsigned portLONG portTickType;  // The system timer tick type
```

This method of using data types and functions through thin layers of `#defines` may seem a bit complicated, but it allows FreeRTOS to be recompiled for a completely different system architecture by changing only the hardware-dependent files. And if you want to run FreeRTOS on an architecture it doesn't currently support, you only have to implement the hardware-dependent functionality which is much smaller than the hardware-independent part of FreeRTOS.

As we've seen, FreeRTOS implements hardware-dependent functionality with C preprocessor `#define` macros. FreeRTOS also uses `#define` for plenty of hardware-independent code. For non-embedded applications this frequent use of `#define` is a cardinal sin, but in many smaller embedded systems the overhead for calling a function is not worth the advantages that "real" functions offer.

3.3. Scheduling Tasks: A Quick Overview

Task Priorities and the Ready List

Each task has a user-assigned priority between 0 (the lowest priority) and the compile-time value of `configMAX_PRIORITIES-1` (the highest priority). For instance, if `configMAX_PRIORITIES` is set to 5, then FreeRTOS will use 5 priority levels: 0 (lowest priority), 1, 2, 3, and 4 (highest priority).

FreeRTOS uses a "ready list" to keep track of all tasks that are currently ready to run. It implements the ready list as an array of task lists like this:

```
static xList pxReadyTasksLists[ configMAX_PRIORITIES ];  /* Prioritised ready tasks.  */
```

`pxReadyTasksLists[0]` is a list of all ready priority 0 tasks, `pxReadyTasksLists[1]` is a list of all ready priority 1 tasks, and so on, all the way up to `pxReadyTasksLists[configMAX_PRIORITIES-1]`.

The System Tick

The heartbeat of a FreeRTOS system is called the system tick. FreeRTOS configures the system to generate a periodic tick interrupt. The user can configure the tick interrupt frequency, which is typically in the millisecond range. Every time the tick interrupt fires, the `vTaskSwitchContext()` function is called. `vTaskSwitchContext()` selects the highest-priority ready task and puts it in the `pxCurrentTCB` variable like this:

```
/* Find the highest-priority queue that contains ready tasks. */
while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopReadyPriority ] ) ) )
{
    configASSERT( uxTopReadyPriority );
    --uxTopReadyPriority;
}

/* listGET_OWNER_OF_NEXT_ENTRY walks through the list, so the tasks of the same
priority get an equal share of the processor time. */
listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopReadyPriority ] ) );
```

Before the while loop starts, `uxTopReadyPriority` is guaranteed to be greater than or equal to the priority of the highest-priority ready task. The while() loop starts at priority level `uxTopReadyPriority` and walks down through the `pxReadyTasksLists[]` array to find the highest-priority level with ready tasks. `listGET_OWNER_OF_NEXT_ENTRY()` then grabs the next ready task from that priority level's ready list.

Now `pxCurrentTCB` points to the highest-priority task, and when `vTaskSwitchContext()` returns the hardware-dependent code starts running that task.

Those nine lines of code are the absolute heart of FreeRTOS. The other 8900+ lines of FreeRTOS are there to make sure those nine lines are all that's needed to

Web2PDF
converted by Web2PDFConve

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：https://d.book118.com/525341331223011103