

Contents

List of Figures	vi
List of Code Listings	viii
List of Tables	xi
List of Notation.....	xii
Preface	
FreeRTOS and the Cortex-M3	1
Multitasking on a Cortex-M3 Microcontroller.....	2
An Introduction to Multitasking in Small Embedded Systems	2
A Note About Terminology	2
Why Use a Real-time Kernel?	3
The Cortex-M3 Port of FreeRTOS	4
Resources Used By FreeRTOS	5
The FreeRTOS, OpenRTOS, and SafeRTOS Family.....	6
Using the Examples that Accompany this Book.....	8
Required Tools and Hardware	8
Chapter 1 Task Management.....	9
1.1 Chapter Introduction and Scope.....	10
Scope	10
1.2 Task Functions.....	11
1.3 Top Level Task States	12
1.4 Creating Tasks.....	13
The xTaskCreate() API Function.....	13
Example 1. Creating tasks	16
Example 2. Using the task parameter	19
1.5 Task Priorities	22
Example 3. Experimenting with priorities.....	23
1.6 Expanding the 'Not Running' State.....	26
The Blocked State.....	26
The Suspended State	27
The Ready State.....	27
Completing the State Transition Diagram.....	27
Example 4. Using the Blocked state to create a delay.....	28
The vTaskDelayUntil() API Function	31
Example 5. Converting the example tasks to use vTaskDelayUntil()	33
Example 6. Combining blocking and non-blocking tasks.....	34
1.7 The Idle Task and the Idle Task Hook.....	37
Idle Task Hook Functions.....	37
Limitations on the Implementation of Idle Task Hook Functions.....	38

Example 7. Defining an idle task hook function	38
1.8 Changing the Priority of a Task	40
The vTaskPrioritySet() API Function	40
The uxTaskPriorityGet() API Function	40
Example 8. Changing task priorities	41
1.9 Deleting a Task	46
The vTaskDelete() API Function	46
Example 9. Deleting tasks	47
1.10 The Scheduling Algorithm—A Summary	50
Prioritized Pre-emptive Scheduling.....	50
Selecting Task Priorities.....	52
Co-operative Scheduling	52
Chapter 2 Queue Management	55
2.1 Chapter Introduction and Scope	56
Scope.....	56
2.2 Characteristics of a Queue	57
Data Storage	57
Access by Multiple Tasks	57
Blocking on Queue Reads.....	57
Blocking on Queue Writes.....	58
2.3 Using a Queue	60
The xQueueCreate() API Function	60
The xQueueSendToBack() and xQueueSendToFront() API Functions.....	61
The xQueueReceive() and xQueuePeek() API Functions.....	63
The uxQueueMessagesWaiting() API Function	66
Example 10. Blocking when receiving from a queue	67
Using Queues to Transfer Compound Types	71
Example 11. Blocking when sending to a queue or sending structures on a queue.....	73
2.4 Working with Large Data	79
Chapter 3 Interrupt Management.....	81
3.1 Chapter Introduction and Scope	82
Events.....	82
Scope.....	82
3.2 Deferred Interrupt Processing.....	84
Binary Semaphores Used for Synchronization	84
Writing FreeRTOS Interrupt Handlers	85
The vSemaphoreCreateBinary() API Function.....	85
The xSemaphoreTake() API Function	88
The xSemaphoreGiveFromISR() API Function.....	89
Example 12. Using a binary semaphore to synchronize a task with an interrupt.....	91
3.3 Counting Semaphores.....	96
The xSemaphoreCreateCounting() API Function	99

Example 13. Using a counting semaphore to synchronize a task with an interrupt.....	101
3.4 Using Queues within an Interrupt Service Routine	103
The xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() API Functions	103
Efficient Queue Usage	105
Example 14. Sending and receiving on a queue from within an interrupt	105
3.5 Interrupt Nesting	110
Chapter 4 Resource Management	115
4.1 Chapter Introduction and Scope.....	116
Mutual Exclusion.....	118
Scope	119
4.2 Critical Sections and Suspending the Scheduler	120
Basic Critical Sections	120
Suspending (or Locking) the Scheduler	121
The vTaskSuspendAll() API Function.....	122
The xTaskResumeAll() API Function	122
4.3 Mutexes (and Binary Semaphores)	124
The xSemaphoreCreateMutex() API Function.....	126
Example 15. Rewriting vPrintString() to use a semaphore	126
Priority Inversion	129
Priority Inheritance	130
Deadlock (or Deadly Embrace)	131
4.4 Gatekeeper Tasks.....	133
Example 16. Re-writing vPrintString() to use a gatekeeper task.....	133
Chapter 5 Memory Management.....	139
5.1 Chapter Introduction and Scope.....	140
Scope	141
5.2 Example Memory Allocation Schemes	142
Heap_1.c	142
Heap_2.c	143
Heap_3.c	145
The xPortGetFreeHeapSize() API Function	145
Chapter 6 Trouble Shooting	147
6.1 Chapter Introduction and Scope.....	148
printf-stdarg.c.....	148
6.2 Stack Overflow	149
The uxTaskGetStackHighWaterMark() API Function	149
Run Time Stack Checking—Overview	150
Run Time Stack Checking—Method 1	150
Run Time Stack Checking—Method 2	151
6.3 Other Common Sources of Error.....	152
Symptom: Adding a simple task to a demo causes the demo to crash	152

Symptom: Using an API function within an interrupt causes the application to crash....	152
Symptom: Sometimes the application crashes within an interrupt service routine	152
Symptom: Critical sections do not nest correctly	153
Symptom: The application crashes even before the scheduler is started.....	153
Symptom: Calling API functions while the scheduler is suspended causes the application to crash	153
Symptom: The prototype for pxPortInitialiseStack() causes compilation to fail.....	153
Chapter 7 FreeRTOS-MPU	155
7.1 Chapter Introduction and Scope	156
Scope.....	156
7.2 Access Permissions	157
User Mode and Privileged Mode	157
Access Permission Attributes	157
7.3 Defining an MPU Region	159
Overlapping Regions.....	159
Predefined Regions and Task Definable Regions	159
Region Start Address and Size Constraints.....	160
7.4 The FreeRTOS-MPU API	162
The xTaskCreateRestricted() API Function	162
Using xTaskCreate() with FreeRTOS-MPU	167
The vTaskAllocateMPURegions() API Function	168
The portSWITCH_TO_USER_MODE() API Macro.....	170
7.5 Linker Configuration	171
7.6 Practical Usage Tips	174
Accessing Data from a User Mode Task	174
Intertask Communication from User Mode	175
FreeRTOS-MPU Demo Projects	175
Chapter 8 The FreeRTOS Download	177
8.1 Chapter Introduction and Scope	178
Scope.....	178
8.2 Files and Directories.....	179
Removing Unused Source Files	180
8.3 Demo Applications	181
Removing Unused Demo Files.....	182
8.4 Creating a FreeRTOS Project.....	183
Adapting One of the Supplied Demo Projects	183
Creating a New Project from Scratch	184
Header Files.....	185
8.5 Data Types and Coding Style Guide.....	186
Data Types.....	186
Variable Names.....	187
Function Names.....	187

Formatting.....	187
Macro Names	187
Rationale for Excessive Type Casting.....	188
Appendix 1: Licensing Information.....	189
Open Source License Details.....	190
GPL Exception Text	191
INDEX	8-193

List of Figures

Figure 1. Top level task states and transitions.....	12
Figure 2. The output produced when Example 1 is executed	17
Figure 3. The execution pattern of the two Example 1 tasks	18
Figure 4. The execution sequence expanded to show the tick interrupt executing	23
Figure 5. Running both test tasks at different priorities.....	24
Figure 6. The execution pattern when one task has a higher priority than the other	25
Figure 7. Full task state machine	28
Figure 8. The output produced when Example 4 is executed	30
Figure 9. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop.....	30
Figure 10. Bold lines indicate the state transitions performed by the tasks in Example 4	31
Figure 11. The output produced when Example 6 is executed	35
Figure 12. The execution pattern of Example 6.....	36
Figure 13. The output produced when Example 7 is executed	39
Figure 14. The sequence of task execution when running Example 8.....	44
Figure 15. The output produced when Example 8 is executed	45
Figure 16. The output produced when Example 9 is executed	48
Figure 17. The execution sequence for Example 9	49
Figure 18. Execution pattern with pre-emption points highlighted.....	51
Figure 19. An example sequence of writes and reads to and from a queue	59
Figure 20. The output produced when Example 10 is executed	71
Figure 21. The sequence of execution produced by Example 10	71
Figure 22. An example scenario where structures are sent on a queue	72
Figure 23. The output produced by Example 11	76
Figure 24. The sequence of execution produced by Example 11	77
Figure 25. The interrupt interrupts one task but returns to another	84
Figure 26. Using a binary semaphore to synchronize a task with an interrupt	87
Figure 27. The output produced when Example 12 is executed	94
Figure 28. The sequence of execution when Example 12 is executed	95
Figure 29. A binary semaphore can latch at most one event.....	97
Figure 30. Using a counting semaphore to 'count' events	98
Figure 31. The output produced when Example 13 is executed	102
Figure 32. The output produced when Example 14 is executed	109
Figure 33. The sequence of execution produced by Example 14	109
Figure 34. Constants affecting interrupt nesting behavior – this illustration assumes the microcontroller being used implements at least five interrupt priority bits	112
Figure 35. Mutual exclusion implemented using a mutex	125
Figure 36. The output produced when Example 15 is executed	129
Figure 37. A possible sequence of execution for Example 15	129
Figure 38. A worst case priority inversion scenario	130
Figure 39. Priority inheritance minimizing the effect of priority inversion.....	131

Figure 40. The output produced when Example 16 is executed	137
Figure 41. RAM being allocated within the array each time a task is created	142
Figure 42. RAM being allocated from the array as tasks are created and deleted.....	144
Figure 43. The top-level directories—Source and Demo.....	179
Figure 44. The three core files that implement the FreeRTOS kernel.....	180
Figure 45. The source directories required to build a Cortex-M3 microcontroller demo application	180
Figure 46. The demo directories required to build a demo application	182

List of Code Listings

Listing 1. The task function prototype.....	11
Listing 2. The structure of a typical task function	11
Listing 3. The xTaskCreate() API function prototype	13
Listing 4. Implementation of the first task used in Example 1	16
Listing 5. Implementation of the second task used in Example 1	16
Listing 6. Starting the Example 1 tasks	17
Listing 7. Creating a task from within another task after the scheduler has started.....	19
Listing 8. The single task function used to create two tasks in Example 2	20
Listing 9. The main() function for Example 2	21
Listing 10. Creating two tasks at different priorities	24
Listing 11. The vTaskDelay() API function prototype	29
Listing 12. The source code for the example task after the null loop delay has been replaced by a call to vTaskDelay()	29
Listing 13. vTaskDelayUntil() API function prototype	32
Listing 14. The implementation of the example task using vTaskDelayUntil().....	33
Listing 15. The continuous processing task used in Example 6.....	34
Listing 16. The periodic task used in Example 6.....	35
Listing 17. The idle task hook function name and prototype.	38
Listing 18. A very simple Idle hook function.....	38
Listing 19. The source code for the example task prints out the ulIdleCycleCount value	39
Listing 20. The vTaskPrioritySet() API function prototype.....	40
Listing 21. The uxTaskPriorityGet() API function prototype	40
Listing 22. The implementation of Task 1 in Example 8.....	42
Listing 23. The implementation of Task 2 in Example 8.....	43
Listing 24. The implementation of main() for Example 8.....	44
Listing 25. The vTaskDelete() API function prototype.....	46
Listing 26. The implementation of main() for Example 9.....	47
Listing 27. The implementation of Task 1 for Example 9	48
Listing 28. The implementation of Task 2 for Example 9	48
Listing 29. The xQueueCreate() API function prototype	60
Listing 30. The xQueueSendToFront() API function prototype	61
Listing 31. The xQueueSendToBack() API function prototype.....	61
Listing 32. The xQueueReceive() API function prototype	64
Listing 33. The xQueuePeek() API function prototype	64
Listing 34. The uxQueueMessagesWaiting() API function prototype	66
Listing 35. Implementation of the sending task used in Example 10.....	68
Listing 36. Implementation of the receiver task for Example 10.....	69
Listing 37. The implementation of main() for Example 10.....	70
Listing 38. The definition of the structure that is to be passed on a queue, plus the declaration of two variables for use by the example	73
Listing 39. The implementation of the sending task for Example 11.	74

Listing 40. The definition of the receiving task for Example 11	75
Listing 41. The implementation of main() for Example 11	76
Listing 42. The vSemaphoreCreateBinary() API function prototype	86
Listing 43. The xSemaphoreTake() API function prototype	88
Listing 44. The xSemaphoreGiveFromISR() API function prototype.....	89
Listing 45. Implementation of the task that periodically generates a software interrupt in Example 12.....	91
Listing 46. The implementation of the handler task (the task that synchronizes with the interrupt) in Example 12.....	92
Listing 47. The software interrupt handler used in Example 12	93
Listing 48. The implementation of main() for Example 12.....	94
Listing 49. The xSemaphoreCreateCounting() API function prototype	99
Listing 50. Using xSemaphoreCreateCounting() to create a counting semaphore.....	101
Listing 51. The implementation of the interrupt service routine used by Example 13.....	101
Listing 52. The xQueueSendToFrontFromISR() API function prototype	103
Listing 53. The xQueueSendToBackFromISR() API function prototype	103
Listing 54. The implementation of the task that writes to the queue in Example 14	106
Listing 55. The implementation of the interrupt service routine used by Example 14.....	107
Listing 56. The task that prints out the strings received from the interrupt service routine in Example 14.....	108
Listing 57. The main() function for Example 14	108
Listing 58. Using a CMSIS function to set an interrupt priority.....	111
Listing 59. An example read, modify, write sequence	116
Listing 60. An example of a reentrant function	118
Listing 61. An example of a function that is not reentrant	118
Listing 62. Using a critical section to guard access to a variable	120
Listing 63. A possible implementation of vPrintString().....	120
Listing 64. The vTaskSuspendAll() API function prototype.....	122
Listing 65. The xTaskResumeAll() API function prototype.....	122
Listing 66. The implementation of vPrintString().....	123
Listing 67. The xSemaphoreCreateMutex() API function prototype.....	126
Listing 68. The implementation of prvNewPrintString().....	127
Listing 69. The implementation of prvPrintTask() for Example 15	127
Listing 70. The implementation of main() for Example 15.....	128
Listing 71. The name and prototype for a tick hook function	134
Listing 72. The gatekeeper task	134
Listing 73. The print task implementation for Example 16	135
Listing 74. The tick hook implementation	135
Listing 75. The implementation of main() for Example 16.....	136
Listing 76. The heap_3.c implementation.....	145
Listing 77. The xPortGetFreeHeapSize() API function prototype.....	145
Listing 78. The uxTaskGetStackHighWaterMark() API function prototype.....	149
Listing 79. The stack overflow hook function prototype	150

Listing 80. Syntax required by GCC, IAR, and Keil compilers to force a variable onto a particular byte alignment (1024-byte alignment in this example)	160
Listing 81. Defining two arrays that may be placed in adjacent memory.....	160
Listing 82. The xTaskCreateRestricted() API function prototype	162
Listing 83. Definition of the structures required by the xTaskCreateRestricted() API function	163
Listing 84. Using the xTaskParameters structure	166
Listing 85. Using xTaskCreate() to create both User mode and Privileged mode task with FreeRTOS-MPU	168
Listing 86. The vTaskAllocateMPURegions() API function prototype.....	168
Listing 87. Using vTaskAllocateMPURegions() to redefine the MPU regions associated with a task.....	169
Listing 88. Defining the memory map and linker variables using GNU LD syntax.....	172
Listing 89. Defining the privileged_functions named section using GNU LD syntax.....	173
Listing 90. Copying data into a stack variable before setting the task into User mode	174
Listing 91. Copying the value of a global variable into a stack variable using the task parameter	175
Listing 92. The template for a new main() function	184

List of Tables

Table 1. Comparing the FreeRTOS license with the OpenRTOS license	7
Table 2. xTaskCreate() parameters and return value	13
Table 3. vTaskDelay() parameters	29
Table 4. vTaskDelayUntil() parameters	32
Table 5. vTaskPrioritySet() parameters	40
Table 6. uxTaskPriorityGet() parameters and return value	41
Table 7. vTaskDelete() parameters	46
Table 8. xQueueCreate() parameters and return value	60
Table 9. xQueueSendToFront() and xQueueSendToBack() function parameters and return value	61
Table 10. xQueueReceive() and xQueuePeek() function parameters and return values	64
Table 11. uxQueueMessagesWaiting() function parameters and return value	67
Table 12. Key to Figure 24	77
Table 13. vSemaphoreCreateBinary() parameters	86
Table 14. xSemaphoreTake() parameters and return value	88
Table 15. xSemaphoreGiveFromISR() parameters and return value	90
Table 16. xSemaphoreCreateCounting() parameters and return value	100
Table 17. xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() parameters and return values	103
Table 18. Constants that affect interrupt nesting	111
Table 19. xTaskResumeAll() return value	122
Table 20. xSemaphoreCreateMutex() return value	126
Table 21. xPortGetFreeHeapSize() return value	146
Table 22. uxTaskGetStackHighWaterMark() parameters and return value	149
Table 23. MPU region access permissions	158
Table 24. xMemoryRegion structure members	163
Table 25. xTaskParameters structure members	164
Table 26. vTaskAllocateMPURegions() parameters	169
Table 27. Named linker sections required by FreeRTOS-MPU	171
Table 28. Linker variables required by FreeRTOS-MPU	171
Table 29. FreeRTOS source files to include in the project	185
Table 30. Special data types used by FreeRTOS	186
Table 31. Macro prefixes	188
Table 32. Common macro definitions	188
Table 33. Comparing the open source license with the commercial license	190

List of Notation

API	Application Programming Interface
CMSIS	Cortex Microcontroller Software Interface Standard
FAQ	Frequently Asked Question
FIFO	First In First Out
HMI	Human Machine Interface
IDE	Integrated Development Environment
IRQ	Interrupt Request
ISR	Interrupt Service Routine
LCD	Liquid Crystal Display
MCU	Microcontroller
MPU	Memory Protection Unit
RMS	Rate Monotonic Scheduling
RTOS	Real-time Operating System
SIL	Safety Integrity Level
TCB	Task Control Block
UART	Universal Asynchronous Receiver/Transmitter

Preface

FreeRTOS and the Cortex-M3

Multitasking on a Cortex-M3 Microcontroller

An Introduction to Multitasking in Small Embedded Systems

Microcontrollers (MCUs) that contain an ARM Cortex-M3 core are available from many manufacturers and are ideally suited to deeply embedded real-time applications. Typically, applications of this type include a mix of both hard and soft real-time requirements.

Soft real-time requirements are those that state a time deadline—but breaching the deadline would not render the system useless. For example, responding to keystrokes too slowly may make a system seem annoyingly unresponsive without actually making it unusable.

Hard real-time requirements are those that state a time deadline—and breaching the deadline would result in absolute failure of the system. For example, a driver's airbag would be useless if it responded to crash sensor inputs too slowly.

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which Cortex-M3 microcontroller applications can be built to meet their hard real-time requirements. It allows Cortex-M3 microcontroller applications to be organized as a collection of independent threads of execution. As most Cortex-M3 microcontroller have only one core, in reality only a single thread can be executing at any one time. The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer. In the simplest case, the application designer could assign higher priorities to threads that implement hard real-time requirements, and lower priorities to threads that implement soft real-time requirements. This would ensure that hard real-time threads are always executed ahead of soft real-time threads, but priority assignment decisions are not always that simplistic.

Do not be concerned if you do not fully understand the concepts in the previous paragraph yet. The following chapters provide a detailed explanation, with many examples, to help you understand how to use a real-time kernel, and how to use FreeRTOS in particular.

A Note About Terminology

In FreeRTOS, each thread of execution is called a 'task'. There is no consensus on terminology within the embedded community, but I prefer 'task' to 'thread' as 'thread' can have a more specific meaning in some fields of application.

Why Use a Real-time Kernel?

There are many well established techniques for writing good embedded software without the use of a kernel, and, if the system being developed is simple, then these techniques might provide the most appropriate solution. In more complex cases, it is likely that using a kernel would be preferable, but where the crossover point occurs will always be subjective.

As already described, task prioritization can help ensure an application meets its processing deadlines, but a kernel can bring other less obvious benefits, too. Some of these are listed very briefly below:

- Abstracting away timing information

The kernel is responsible for execution timing and provides a time-related API to the application. This allows the structure of the application code to be simpler and the overall code size to be smaller.

- Maintainability/Extensibility

Abstracting away timing details results in fewer interdependencies between modules and allows the software to evolve in a controlled and predictable way. Also, the kernel is responsible for timing, so application performance is less susceptible to changes in the underlying hardware.

- Modularity

Tasks are independent modules, each of which should have a well-defined purpose.

- Team development

Tasks should also have well-defined interfaces, allowing easier development by teams.

- Easier testing

If tasks are well-defined independent modules with clean interfaces, they can be tested in isolation.

- Code reuse

Greater modularity and fewer interdependencies can result in code that can be re-used with less effort.

- Improved efficiency

Using a kernel allows software to be completely event-driven, so no processing time is wasted by polling for events that have not occurred. Code executes only when there is something that must be done.

Counter to the efficiency saving is the need to process the RTOS tick interrupt and to switch execution from one task to another.

- Idle time

The Idle task is created automatically when the kernel is started. It executes whenever there are no application tasks wishing to execute. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.

- Flexible interrupt handling

Interrupt handlers can be kept very short by deferring most of the required processing to handler tasks. Section 3.2 demonstrates this technique.

- Mixed processing requirements

Simple design patterns can achieve a mix of periodic, continuous, and event-driven processing within an application. In addition, hard and soft real-time requirements can be met by selecting appropriate task and interrupt priorities.

- Easier control over peripherals

Gatekeeper tasks can be used to serialize access to peripherals.

The Cortex-M3 Port of FreeRTOS

The Cortex-M3 port includes all the standard FreeRTOS features:

- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- Queues
- Binary semaphores
- Counting semaphores

- Recursive semaphores
- Mutexes
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace hook macros
- Optional commercial licensing and support

FreeRTOS also manages interrupt nesting, and allows interrupts above a user-definable priority level to remain unaffected by the activity of the kernel. Using FreeRTOS will not introduce any additional timing jitter or latency for these interrupts.

There are two separate FreeRTOS ports for the Cortex-M3:

1. FreeRTOS-MPU

FreeRTOS-MPU includes full Memory Protection Unit (MPU) support. In this version, tasks can execute in either User mode or Privileged mode. Also, access to Flash, RAM, and peripheral memory regions can be tightly controlled, on a task-by-task basis.

Not all Cortex-M3 microcontrollers include MPU hardware.

2. FreeRTOS (the original port)

This does not include any MPU support. All tasks execute in the Privileged mode and can access the entire memory map.

The examples that accompany this text use the original FreeRTOS version without MPU support, but a chapter describing FreeRTOS-MPU is included for completeness (see Chapter 7).

Resources Used By FreeRTOS

FreeRTOS makes use of the Cortex-M3 SysTick, PendSV, and SVC interrupts. These interrupts are not available for use by the application.

FreeRTOS has a very small footprint. A typical kernel build will consume approximately 6K bytes of Flash space and a few hundred bytes of RAM. Each task also requires RAM to be allocated for use as the task stack.

The FreeRTOS, OpenRTOS, and SafeRTOS Family

FreeRTOS uses a modified GPL license. The modification is included to ensure:

1. FreeRTOS can be used in commercial applications.
2. FreeRTOS itself remains open source.
3. FreeRTOS users retain ownership of their intellectual property.

When you link FreeRTOS into an application, you are obliged to open source only the kernel, including any additions or modifications you may have made. Components that merely use FreeRTOS through its published API can remain closed source and proprietary. Appendix 1: contains the modification text.

OpenRTOS shares the same code base as FreeRTOS, but is provided under standard commercial license terms. The commercial license removes the requirement to open source any code at all and provides IP infringement protection.

OpenRTOS can be purchased with a professional support contract and a selection of other useful components such as TCP/IP stacks and drivers, USB stacks and drivers, and various different file systems. Evaluation versions can be downloaded from <http://www.OpenRTOS.com>.

Table 1 provides an overview of the differences between the FreeRTOS and OpenRTOS license models.

SafeRTOS has been developed in accordance with the practices, procedures, and processes necessary to claim compliance with various internationally recognized safety related standards.

IEC 61508 is an international standard covering the development and use of electrical, electronic, and programmable electronic safety-related systems. The standard defines the analysis, design, implementation, production, and test requirements for safety-related systems, in accordance with the Safety Integrity Level (SIL) assigned to the system. The SIL is assigned according to the risks associated with the use of the system under development, with a maximum SIL of 4 being assigned to systems with the highest perceived risk. The SafeRTOS development process has been independently certified by TÜV SÜD as being in compliance with that required by IEC 61508 for SIL 3 applications. SafeRTOS is supplied with

complete lifecycle compliance evidence and has itself been certified for use in IEC 61508, IEC 62304 and FDA 510(K) applications.

SafeRTOS was originally derived from FreeRTOS and retains a similar usage model. Visit <http://www.SafeRTOS.com> for additional information.

Table 1. Comparing the FreeRTOS license with the OpenRTOS license

	FreeRTOS License	OpenRTOS License
Is it Free?	Yes	No
Can I use it in a commercial application?	Yes	Yes
Is it royalty free?	Yes	Yes
Do I have to open source my application code that makes use of FreeRTOS services?	No, as long as the code provides functionality that is distinct from that provided by FreeRTOS	No
Do I have to open source my changes to the kernel?	Yes	No
Do I have to document that my product uses FreeRTOS?	Yes	No
Do I have to offer to provide the FreeRTOS code to users of my application?	Yes	No
Can I buy an annual support contract?	No	Yes
Is a warranty provided?	No	Yes
Is legal protection provided?	No	Yes, IP infringement protection is provided