

Using The FreeRTOS Real Time Kernel

NXP LPC17xx Edition

Richard Barry



Using the FreeRTOS™ Real Time Kernel

NXP LPC17xx Edition

Richard Barry

FreeRTOS™ is a portable, open source, royalty free, tiny footprint Real Time Kernel - a free to download and free to deploy RTOS that can be used in commercial applications. With downloads topping 75,000 last year alone, FreeRTOS is now a de facto standard for embedded microcontrollers.

“Using the FreeRTOS Real Time Kernel – a Practical Guide LPC17xx Edition” is a step by step hands on guide to using FreeRTOS on Cortex M3 microcontrollers from NXP. It presents and explains numerous examples that are written using the FreeRTOS API. Full source code for both the kernel and the examples is provided in an accompanying .zip file.

This document provides a summary of the full text. It includes the complete table of contents, the preface and the first section of each chapter. The complete document can be obtained by visiting <http://www.FreeRTOS.org/Documentation>.

Version 1.3.0.

All text, source code and diagrams are the exclusive property of Real Time Engineers Ltd. Distribution or publication in any form is strictly prohibited without prior written authority from Real Time Engineers Ltd.

© Real Time Engineers Ltd. 2010. All rights reserved.

FreeRTOS™, FreeRTOS.org™ and the FreeRTOS logo are trademarks of Real Time Engineers Ltd.

OPENRTOS™, **SAFERTOS™**, and the **OPENRTOS** and **SAFERTOS** logos are trademarks of WITTENSTEIN Aerospace and Simulation Ltd.

ARM™ and Cortex™ are trademarks of ARM Limited. NXP® is a registered trademark of NXP Semiconductors. LPCXpresso™ is a trademark of NXP Semiconductors. All other brands or product names are the property of their respective holders.

<http://www.freertos.org>

ISBN 978-1-4457-4664-7

Contents

List of Figures	vi
List of Code Listings	viii
List of Tables	xi
List of Notation.....	xii
Preface	
FreeRTOS and the LPC17xx	1
Multitasking on an LPC17xx 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 LPC17xx 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
Opening the Example Workspaces	9
Building the Examples	11
Chapter 1 Task Management.....	13
1.1 Chapter Introduction and Scope.....	14
Scope	14
1.2 Task Functions.....	15
1.3 Top Level Task States	16
1.4 Creating Tasks.....	17
The xTaskCreate() API Function.....	17
Example 1. Creating tasks	20
Example 2. Using the task parameter	23
1.5 Task Priorities	26
Example 3. Experimenting with priorities.....	27
1.6 Expanding the 'Not Running' State.....	30
The Blocked State.....	30
The Suspended State	31
The Ready State	31
Completing the State Transition Diagram.....	31
Example 4. Using the Blocked state to create a delay.....	32
The vTaskDelayUntil() API Function	35
Example 5. Converting the example tasks to use vTaskDelayUntil().....	37
Example 6. Combining blocking and non-blocking tasks.....	38
1.7 The Idle Task and the Idle Task Hook.....	41

Idle Task Hook Functions	41
Limitations on the Implementation of Idle Task Hook Functions	42
Example 7. Defining an idle task hook function	42
1.8 Changing the Priority of a Task	44
The vTaskPrioritySet() API Function	44
The uxTaskPriorityGet() API Function	44
Example 8. Changing task priorities	45
1.9 Deleting a Task	50
The vTaskDelete() API Function	50
Example 9. Deleting tasks	51
1.10 The Scheduling Algorithm—A Summary	54
Prioritized Pre-emptive Scheduling.....	54
Selecting Task Priorities.....	56
Co-operative Scheduling.....	56
Chapter 2 Queue Management.....	59
2.1 Chapter Introduction and Scope	60
Scope.....	60
2.2 Characteristics of a Queue	61
Data Storage.....	61
Access by Multiple Tasks	61
Blocking on Queue Reads.....	61
Blocking on Queue Writes.....	62
2.3 Using a Queue	64
The xQueueCreate() API Function	64
The xQueueSendToBack() and xQueueSendToFront() API Functions.....	65
The xQueueReceive() and xQueuePeek() API Functions.....	67
The uxQueueMessagesWaiting() API Function.....	70
Example 10. Blocking when receiving from a queue	71
Using Queues to Transfer Compound Types	75
Example 11. Blocking when sending to a queue or sending structures on a queue.....	77
2.4 Working with Large Data	83
Chapter 3 Interrupt Management.....	85
3.1 Chapter Introduction and Scope	86
Events.....	86
Scope.....	86
3.2 Deferred Interrupt Processing.....	88
Binary Semaphores Used for Synchronization	88
Writing FreeRTOS Interrupt Handlers	89
The vSemaphoreCreateBinary() API Function.....	89
The xSemaphoreTake() API Function	92
The xSemaphoreGiveFromISR() API Function.....	93
Example 12. Using a binary semaphore to synchronize a task with an interrupt.....	95

3.3	Counting Semaphores	100
	The xSemaphoreCreateCounting() API Function	103
	Example 13. Using a counting semaphore to synchronize a task with an interrupt.....	150
3.4	Using Queues within an Interrupt Service Routine	107
	The xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() API Functions	107
	Efficient Queue Usage	109
	Example 14. Sending and receiving on a queue from within an interrupt	109
3.5	Interrupt Nesting	114
Chapter 4	Resource Management	117
4.1	Chapter Introduction and Scope.....	118
	Mutual Exclusion.....	120
	Scope	121
4.2	Critical Sections and Suspending the Scheduler	122
	Basic Critical Sections	122
	Suspending (or Locking) the Scheduler	123
	The vTaskSuspendAll() API Function.....	124
	The xTaskResumeAll() API Function	124
4.3	Mutexes (and Binary Semaphores).....	126
	The xSemaphoreCreateMutex() API Function.....	128
	Example 15. Rewriting vPrintString() to use a semaphore	128
	Priority Inversion	131
	Priority Inheritance	132
	Deadlock (or Deadly Embrace).....	133
4.4	Gatekeeper Tasks.....	135
	Example 16. Re-writing vPrintString() to use a gatekeeper task.....	135
Chapter 5	Memory Management.....	141
5.1	Chapter Introduction and Scope.....	142
	Scope	143
5.2	Example Memory Allocation Schemes	144
	Heap_1.c	144
	Heap_2.c	145
	Heap_3.c	147
	The xPortGetFreeHeapSize() API Function	147
Chapter 6	Trouble Shooting	149
6.1	Chapter Introduction and Scope.....	150
	printf-stdarg.c.....	150
6.2	Stack Overflow	151
	The uxTaskGetStackHighWaterMark() API Function	151
	Run Time Stack Checking—Overview	152
	Run Time Stack Checking—Method 1	152
	Run Time Stack Checking—Method 2	153

6.3	Other Common Sources of Error	154
	Symptom: Adding a simple task to a demo causes the demo to crash	154
	Symptom: Using an API function within an interrupt causes the application to crash....	154
	Symptom: Sometimes the application crashes within an interrupt service routine	154
	Symptom: Critical sections do not nest correctly	155
	Symptom: The application crashes even before the scheduler is started.....	155
	Symptom: Calling API functions while the scheduler is suspended causes the application to crash	155
	Symptom: The prototype for pxPortInitialiseStack() causes compilation to fail.....	155
Chapter 7	FreeRTOS-MPU	157
7.1	Chapter Introduction and Scope	158
	Scope.....	158
7.2	Access Permissions	159
	User Mode and Privileged Mode	159
	Access Permission Attributes	159
7.3	Defining an MPU Region	161
	Overlapping Regions.....	161
	Predefined Regions and Task Definable Regions	161
	Region Start Address and Size Constraints.....	162
7.4	The FreeRTOS-MPU API	164
	The xTaskCreateRestricted() API Function	164
	Using xTaskCreate() with FreeRTOS-MPU	169
	The vTaskAllocateMPURegions() API Function	170
	The portSWITCH_TO_USER_MODE() API Macro.....	172
7.5	Linker Configuration	173
7.6	Practical Usage Tips	176
	Accessing Data from a User Mode Task	176
	Intertask Communication from User Mode	177
	FreeRTOS-MPU Demo Projects	177
Chapter 8	The FreeRTOS Download	179
8.1	Chapter Introduction and Scope	180
	Scope.....	180
8.2	Files and Directories.....	181
	Removing Unused Source Files	182
8.3	Demo Applications	183
	Removing Unused Demo Files.....	184
8.4	Creating a FreeRTOS Project.....	185
	Adapting One of the Supplied Demo Projects	185
	Creating a New Project from Scratch	186
	Header Files.....	187
8.5	Data Types and Coding Style Guide.....	188
	Data Types.....	188

Variable Names	189
Function Names.....	189
Formatting.....	189
Macro Names	189
Rationale for Excessive Type Casting.....	190
Appendix 1: Licensing Information.....	191
Open Source License Details.....	192
GPL Exception Text.....	193
INDEX	195

List of Figures

Figure 1. Locating the 'Change Target MCU' speed button within the LPCXpresso IDE (highlighted by the red square).....	8
Figure 2. Block diagram of the LPC17xx	9
Figure 3. Selecting the option to import existing projects into the workspace	10
Figure 4. Selecting the .zip archive	10
Figure 5. The projects listed in the Project Explorer window of the LPCXpresso IDE	11
Figure 6. Locating the 'Debug' speed button within the LPCXpresso IDE (highlighted by the red square).....	11
Figure 7. Top level task states and transitions.....	16
Figure 8. The output produced when Example 1 is executed	21
Figure 9. The execution pattern of the two Example 1 tasks	22
Figure 10. The execution sequence expanded to show the tick interrupt executing	27
Figure 11. Running both test tasks at different priorities	28
Figure 12. The execution pattern when one task has a higher priority than the other	29
Figure 13. Full task state machine.....	32
Figure 14. The output produced when Example 4 is executed	34
Figure 15. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop.....	34
Figure 16. Bold lines indicate the state transitions performed by the tasks in Example 4	35
Figure 17. The output produced when Example 6 is executed	39
Figure 18. The execution pattern of Example 6.....	40
Figure 19. The output produced when Example 7 is executed	43
Figure 20. The sequence of task execution when running Example 8.....	48
Figure 21. The output produced when Example 8 is executed	49
Figure 22. The output produced when Example 9 is executed	52
Figure 23. The execution sequence for Example 9	53
Figure 24. Execution pattern with pre-emption points highlighted.....	55
Figure 25. An example sequence of writes and reads to and from a queue	63
Figure 26. The output produced when Example 10 is executed	75
Figure 27. The sequence of execution produced by Example 10	75
Figure 28. An example scenario where structures are sent on a queue	76
Figure 29. The output produced by Example 11.....	80
Figure 30. The sequence of execution produced by Example 11	81
Figure 31. The interrupt interrupts one task but returns to another.....	88
Figure 32. Using a binary semaphore to synchronize a task with an interrupt	91
Figure 33. The output produced when Example 12 is executed	98
Figure 34. The sequence of execution when Example 12 is executed	99
Figure 35. A binary semaphore can latch at most one event.....	101
Figure 36. Using a counting semaphore to 'count' events	102
Figure 37. The output produced when Example 13 is executed	106
Figure 38. The output produced when Example 14 is executed	113

Figure 39. The sequence of execution produced by Example 14	113
Figure 40. Constants affecting interrupt nesting behavior	116
Figure 41. Mutual exclusion implemented using a mutex	127
Figure 42. The output produced when Example 15 is executed	131
Figure 43. A possible sequence of execution for Example 15	131
Figure 44. A worst case priority inversion scenario	132
Figure 45. Priority inheritance minimizing the effect of priority inversion	133
Figure 46. The output produced when Example 16 is executed	139
Figure 47. RAM being allocated within the array each time a task is created	144
Figure 48. RAM being allocated from the array as tasks are created and deleted.....	146
Figure 49. The top-level directories—Source and Demo.....	181
Figure 50. The three core files that implement the FreeRTOS kernel.....	182
Figure 51. The source directories required to build an LPC17xx demo application	182
Figure 52. The demo directories required to build an LPC17xx demo application	184

List of Code Listings

Listing 1. The task function prototype.....	15
Listing 2. The structure of a typical task function	15
Listing 3. The xTaskCreate() API function prototype	17
Listing 4. Implementation of the first task used in Example 1	20
Listing 5. Implementation of the second task used in Example 1.....	20
Listing 6. Starting the Example 1 tasks	21
Listing 7. Creating a task from within another task after the scheduler has started.....	23
Listing 8. The single task function used to create two tasks in Example 2	24
Listing 9. The main() function for Example 2	25
Listing 10. Creating two tasks at different priorities	28
Listing 11. The vTaskDelay() API function prototype	33
Listing 12. The source code for the example task after the null loop delay has been replaced by a call to vTaskDelay()	33
Listing 13. vTaskDelayUntil() API function prototype.....	36
Listing 14. The implementation of the example task using vTaskDelayUntil().....	37
Listing 15. The continuous processing task used in Example 6.....	38
Listing 16. The periodic task used in Example 6.....	39
Listing 17. The idle task hook function name and prototype.	42
Listing 18. A very simple Idle hook function.....	42
Listing 19. The source code for the example task prints out the ulIdleCycleCount value	43
Listing 20. The vTaskPrioritySet() API function prototype.....	44
Listing 21. The uxTaskPriorityGet() API function prototype	44
Listing 22. The implementation of Task 1 in Example 8.....	46
Listing 23. The implementation of Task 2 in Example 8.....	47
Listing 24. The implementation of main() for Example 8.....	48
Listing 25. The vTaskDelete() API function prototype.....	50
Listing 26. The implementation of main() for Example 9.....	52
Listing 27. The implementation of Task 1 for Example 9	53
Listing 28. The implementation of Task 2 for Example 9	53
Listing 29. The xQueueCreate() API function prototype	64
Listing 30. The xQueueSendToFront() API function prototype	65
Listing 31. The xQueueSendToBack() API function prototype.....	65
Listing 32. The xQueueReceive() API function prototype	68
Listing 33. The xQueuePeek() API function prototype	68
Listing 34. The uxQueueMessagesWaiting() API function prototype	70
Listing 35. Implementation of the sending task used in Example 10.....	72
Listing 36. Implementation of the receiver task for Example 10.....	73
Listing 37. The implementation of main() for Example 10.....	74
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	77
Listing 39. The implementation of the sending task for Example 11.....	78

Listing 40.	The definition of the receiving task for Example 11	79
Listing 41.	The implementation of main() for Example 11.....	80
Listing 42.	The vSemaphoreCreateBinary() API function prototype	90
Listing 43.	The xSemaphoreTake() API function prototype	92
Listing 44.	The xSemaphoreGiveFromISR() API function prototype.....	93
Listing 45.	Implementation of the task that periodically generates a software interrupt in Example 12.....	95
Listing 46.	The implementation of the handler task (the task that synchronizes with the interrupt) in Example 12.....	96
Listing 47.	The software interrupt handler used in Example 12	97
Listing 48.	The implementation of main() for Example 12.....	98
Listing 49.	The xSemaphoreCreateCounting() API function prototype	102
Listing 50.	Using xSemaphoreCreateCounting() to create a counting semaphore.....	105
Listing 51.	The implementation of the interrupt service routine used by Example 13.....	105
Listing 52.	The xQueueSendToFrontFromISR() API function prototype	107
Listing 53.	The xQueueSendToBackFromISR() API function prototype	107
Listing 54.	The implementation of the task that writes to the queue in Example 14.....	110
Listing 55.	The implementation of the interrupt service routine used by Example 14.....	111
Listing 56.	The task that prints out the strings received from the interrupt service routine in Example 14	112
Listing 57.	The main() function for Example 14	112
Listing 58.	Using a CMSIS function to set an interrupt priority.....	114
Listing 59.	An example read, modify, write sequence	118
Listing 60.	An example of a reentrant function	120
Listing 61.	An example of a function that is not reentrant	120
Listing 62.	Using a critical section to guard access to a variable	122
Listing 63.	A possible implementation of vPrintString().....	122
Listing 64.	The vTaskSuspendAll() API function prototype.....	124
Listing 65.	The xTaskResumeAll() API function prototype.....	124
Listing 66.	The implementation of vPrintString().....	125
Listing 67.	The xSemaphoreCreateMutex() API function prototype.....	128
Listing 68.	The implementation of prvNewPrintString().....	129
Listing 69.	The implementation of prvPrintTask() for Example 15	129
Listing 70.	The implementation of main() for Example 15.....	130
Listing 71.	The name and prototype for a tick hook function	136
Listing 72.	The gatekeeper task	136
Listing 73.	The print task implementation for Example 16	137
Listing 74.	The tick hook implementation	137
Listing 75.	The implementation of main() for Example 16.....	138
Listing 76.	The heap_3.c implementation.....	147
Listing 77.	The xPortGetFreeHeapSize() API function prototype.....	147
Listing 78.	The uxTaskGetStackHighWaterMark() API function prototype.....	151
Listing 79.	The stack overflow hook function prototype	152

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)	162
Listing 81. Defining two arrays that may be placed in adjacent memory.....	162
Listing 82. The xTaskCreateRestricted() API function prototype	164
Listing 83. Definition of the structures required by the xTaskCreateRestricted() API function	165
Listing 84. Using the xTaskParameters structure	168
Listing 85. Using xTaskCreate() to create both User mode and Privileged mode task with FreeRTOS-MPU	170
Listing 86. The vTaskAllocateMPURegions() API function prototype.....	170
Listing 87. Using vTaskAllocateMPURegions() to redefine the MPU regions associated with a task.....	171
Listing 88. Defining the memory map and linker variables using GNU LD syntax.....	174
Listing 89. Defining the privileged_functions named section using GNU LD syntax.....	175
Listing 90. Copying data into a stack variable before setting the task into User mode.....	176
Listing 91. Copying the value of a global variable into a stack variable using the task parameter	177
Listing 92. The template for a new main() function	186

List of Tables

Table 1. Comparing the FreeRTOS license with the OpenRTOS license	7
Table 2. xTaskCreate() parameters and return value	17
Table 3. vTaskDelay() parameters	33
Table 4. vTaskDelayUntil() parameters	36
Table 5. vTaskPrioritySet() parameters	44
Table 6. uxTaskPriorityGet() parameters and return value	45
Table 7. vTaskDelete() parameters	50
Table 8. xQueueCreate() parameters and return value	64
Table 9. xQueueSendToFront() and xQueueSendToBack() function parameters and return value	65
Table 10. xQueueReceive() and xQueuePeek() function parameters and return values	68
Table 11. uxQueueMessagesWaiting() function parameters and return value	71
Table 12. Key to Figure 30	81
Table 13. vSemaphoreCreateBinary() parameters	90
Table 14. xSemaphoreTake() parameters and return value	92
Table 15. xSemaphoreGiveFromISR() parameters and return value	94
Table 16. xSemaphoreCreateCounting() parameters and return value	104
Table 17. xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() parameters and return values	107
Table 18. Constants that affect interrupt nesting	115
Table 19. xTaskResumeAll() return value	124
Table 20. xSemaphoreCreateMutex() return value	128
Table 21. xPortGetFreeHeapSize() return value	148
Table 22. uxTaskGetStackHighWaterMark() parameters and return value	151
Table 23. MPU region access permissions	160
Table 24. xMemoryRegion structure members	165
Table 25. xTaskParameters structure members	166
Table 26. vTaskAllocateMPURegions() parameters	171
Table 27. Named linker sections required by FreeRTOS-MPU	173
Table 28. Linker variables required by FreeRTOS-MPU	173
Table 29. FreeRTOS source files to include in the project	187
Table 30. Special data types used by FreeRTOS	188
Table 31. Macro prefixes	190
Table 32. Common macro definitions	190
Table 33. Comparing the open source license with the commercial license	192

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 LPC17xx

Multitasking on an LPC17xx Cortex M3 Microcontroller

An Introduction to Multitasking in Small Embedded Systems

The LPC17xx Cortex M3 microcontroller from NXP is 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 worse than useless if it responded to crash sensor inputs too slowly.

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which LPC17xx applications can be built to meet their hard real-time requirements. It allows LPC17xx applications to be organized as a collection of independent threads of execution. As the LPC17xx has 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.

- 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 LPC17xx Port of FreeRTOS

The LPC17xx 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 LPC17xx:

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.

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 LPC17xx 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

Using the Examples that Accompany this Book

Required Tools and Hardware

The examples described in this book are included in an accompanying .zip file. You can download the .zip file from <http://www.FreeRTOS.org/Documentation/code> if you did not receive a copy with this book.

To build and execute the examples you will need:

1. The LPCXpresso IDE.

This is a free tool but requires registration to obtain a license. Instructions are provided on the download page: <http://www.code-red-tech.com/lpcxpresso>.

2. LPC17xx or LPC13xx based hardware.

The only output device used by the examples is the console that is accessed through the debugger. This means that the examples should execute on any LPC17xx or LPC13xx based hardware with sufficient RAM. As supplied, the projects are configured to use an LPC1768 (see Figure 2), but can be re-targeted using the 'Change Target MCU' speed button in the LPCXpresso IDE (see Figure 1). Each project within the workspace must be re-targeted individually.

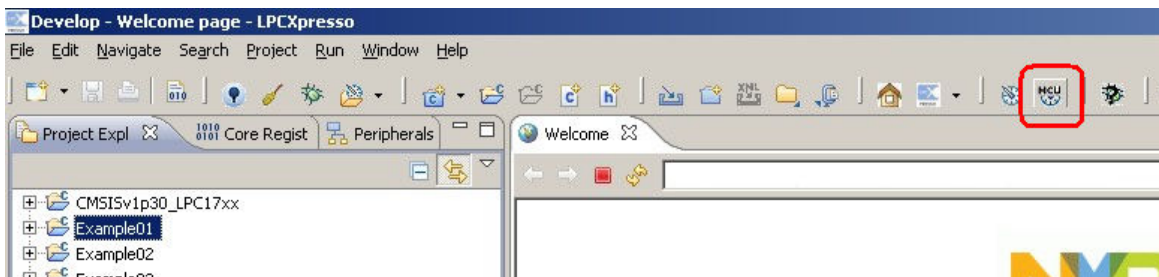


Figure 1. Locating the 'Change Target MCU' speed button within the LPCXpresso IDE (highlighted by the red square)

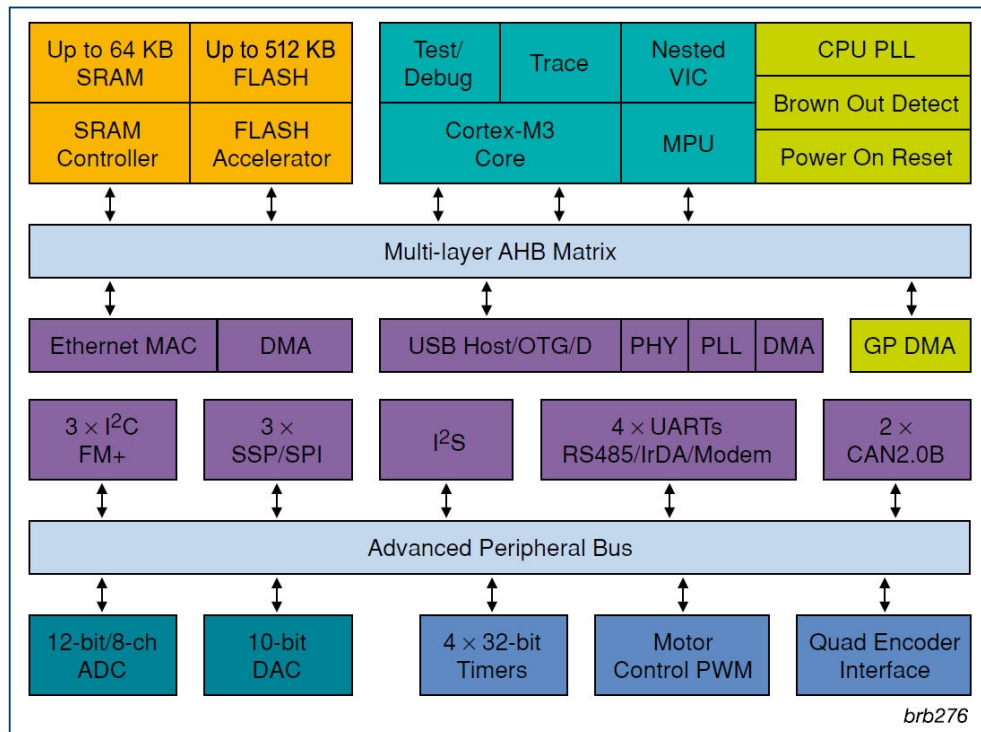


Figure 2. Block diagram of the LPC17xx

Opening the Example Workspaces

All the example projects are provided in a single .zip file archive. Do not extract the files manually from the archive. Instead, follow the procedure defined below.

1. Start the LPCXpresso IDE. You will be prompted to select a workspace.
2. Select an existing workspace (if you have one) or create a new one by entering the path to where you would like the new workspace to be created.
3. Select 'Import' from the 'File' menu. The dialog box shown in Figure 3 will open.

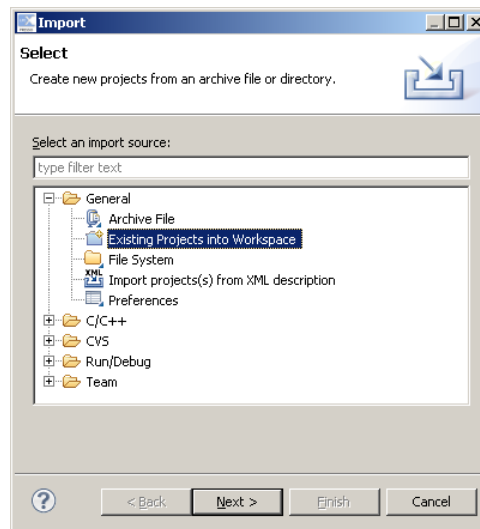


Figure 3. Selecting the option to import existing projects into the workspace

4. Select 'Existing Projects into Workspace' and click 'Next'. The dialog box shown in Figure 4 will open.

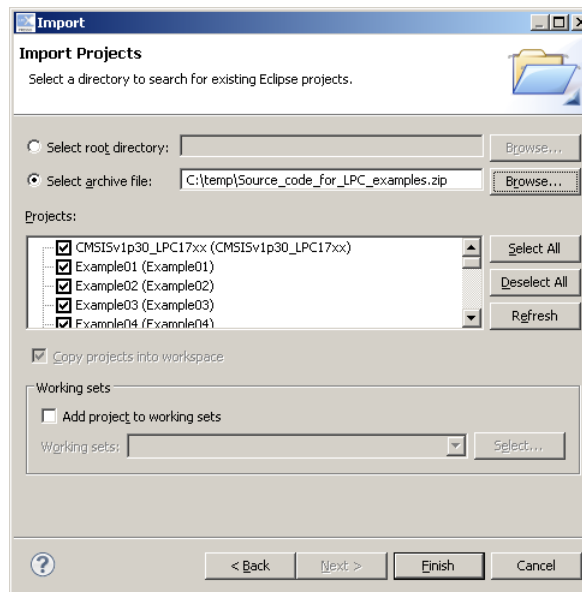


Figure 4. Selecting the .zip archive

5. Choose the 'Select archive file' option, then browse to and select the .zip file that contains the projects.
6. Click 'Finish'.

Building the Examples

The Project Explorer window in the LPCXpresso IDE will list sixteen example projects (named Example01 to Example16) and two library projects (named CMSISv1p30_LPC17xx and FreeRTOS_Library). This arrangement is shown in Figure 5.

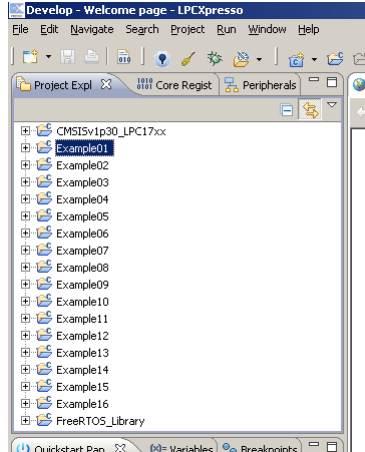


Figure 5. The projects listed in the Project Explorer window of the LPCXpresso IDE

To build an example, highlight the project in the Project Explorer, then select 'Build Project' from the IDE 'Project' menu. Figure 5 shows Example01 highlighted.

The example projects depend on the library projects; therefore, the libraries will build automatically the first time an example is built. The FreeRTOS code itself is contained in the FreeRTOS_Library project. As a consequence of this arrangement, only a single FreeRTOSConfig.h configuration file exists— this is located in the FreeRTOS_Library directory tree. All the example projects reference this single configuration file.

Starting a Debug Session

Highlight the required project in the Project Explorer window, then click the 'Debug' speed button. Figure 6 shows the location of the debug speed button.

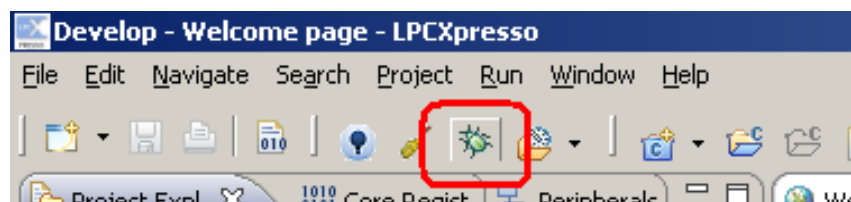


Figure 6. Locating the 'Debug' speed button within the LPCXpresso IDE (highlighted by the red square)

Chapter 1

Task Management

1.1 Chapter Introduction and Scope

Scope

This chapter aims to give readers a good understanding of:

- How FreeRTOS allocates processing time to each task within an application.
- How FreeRTOS chooses which task should execute at any given time.
- How the relative priority of each task affects system behavior.
- The states that a task can exist in.

Readers should also gain a good understanding of:

- How to implement tasks.
- How to create one or more instances of a task.
- How to use the task parameter.
- How to change the priority of a task that has already been created.
- How to delete a task.
- How to implement periodic processing.
- When the idle task will execute and how it can be used.

The concepts presented in this chapter are fundamental to understanding how to use FreeRTOS and how FreeRTOS applications behave. This is, therefore, the most detailed chapter in the book.

Chapter 2

Queue Management

2.1 Chapter Introduction and Scope

Applications that use FreeRTOS are structured as a set of independent tasks—each task is effectively a mini program in its own right. It is likely that these autonomous tasks will have to communicate with each other so that, collectively, they can provide useful system functionality. The ‘queue’ is the underlying primitive used by all FreeRTOS communication and synchronization mechanisms.

Scope

This chapter aims to give readers a good understanding of:

- How to create a queue.
- How a queue manages the data it contains.
- How to send data to a queue.
- How to receive data from a queue.
- What it means to block on a queue.
- The effect of task priorities when writing to and reading from a queue.

Only task-to-task communication is covered in this chapter. Task-to-interrupt and interrupt-to-task communication is covered in Chapter 3.

Chapter 3

Interrupt Management

3.1 Chapter Introduction and Scope

Events

Embedded real-time systems have to take actions in response to events that originate from the environment. For example, a packet arriving on an Ethernet peripheral (the event) might require passing to a TCP/IP stack for processing (the action). Non-trivial systems will have to service events that originate from multiple sources, all of which will have different processing overhead and response time requirements. In each case, a judgment has to be made as to the best event processing implementation strategy:

1. How should the event be detected? Interrupts are normally used, but inputs can also be polled.
2. When interrupts are used, how much processing should be performed inside the interrupt service routine (ISR), and how much outside? It is normally desirable to keep each ISR as short as possible.
3. How can events be communicated to the main (non-ISR) code, and how can this code be structured to best accommodate processing of potentially asynchronous occurrences?

FreeRTOS does not impose any specific event processing strategy on the application designer, but does provide features that allow the chosen strategy to be implemented in a simple and maintainable way.

Note that only API functions and macros ending in 'FromISR' or 'FROM_ISR' should be used within an interrupt service routine.

Scope

This chapter aims to give readers a good understanding of:

- Which FreeRTOS API functions can be used from within an interrupt service routine.
- How a deferred interrupt scheme can be implemented.
- How to create and use binary semaphores and counting semaphores.

- The differences between binary and counting semaphores.
- How to use a queue to pass data into and out of an interrupt service routine.
- The interrupt nesting model of the Cortex M3 FreeRTOS port.

Chapter 4

Resource Management

4.1 Chapter Introduction and Scope

In a multitasking system, there is potential for conflict if one task starts to access a resource, but does not complete its access before being transitioned out of the Running state. If the task leaves the resource in an inconsistent state, then access to the same resource by any other task or interrupt could result in data corruption or other similar error.

Following are some examples:

1. Accessing Peripherals

Consider the following scenario where two tasks attempt to write to an LCD.

1. Task A executes and starts to write the string “Hello world” to the LCD.
2. Task A is pre-empted by Task B after outputting just the beginning of the string—“Hello w”.
3. Task B writes “Abort, Retry, Fail?” to the LCD before entering the Blocked state.
4. Task A continues from the point at which it was pre-empted and completes outputting the remaining characters—“orld”.

The LCD now displays the corrupted string “Hello wAbort, Retry, Fail?orld”.

2. Read, Modify, Write Operations

Listing 1 shows a line of C code and its resultant assembly output. It can be seen that the value of GlobalVar is first read from memory into a register, modified within the register, and then written back to memory. This is called a read, modify, write operation.

```
/* The C code being compiled. */
GlobalVar |= 0x01;

/* The assembly code produced. */
LDR    r4, [pc, #284]
LDR    r0, [r4, #0x08] /* Load the value of GlobalVar into r0. */
ORR    r0, r0, #0x01  /* Set bit 0 of r0. */
STR    r0, [r4, #0x08] /* Write the new r0 value back to GlobalVar. */
```

Listing 1. An example read, modify, write sequence

This is a 'non-atomic' operation because it takes more than one instruction to complete and can be interrupted. Consider the following scenario where two tasks attempt to update a variable called GlobalVar:

1. Task A loads the value of GlobalVar into a register—the read portion of the operation.
2. Task A is pre-empted by Task B before it completes the modify and write portions of the same operation.
3. Task B updates the value of GlobalVar, then enters the Blocked state.
4. Task A continues from the point at which it was pre-empted. It modifies the copy of the GlobalVar value that it already holds in a register before writing the updated value back to GlobalVar.

In this scenario, Task A updates and writes back an out-of-date value for GlobalVar. Task B modifies GlobalVar after Task A takes a copy of the GlobalVar value and before Task A writes its modified value back to the GlobalVar variable. When Task A writes to GlobalVar, it overwrites the modification that has already been performed by Task B, effectively corrupting the GlobalVar variable value.

3. Non-atomic Access to Variables

Updating multiple members of a structure, or updating a variable that is larger than the natural word size of the architecture (for example, updating a 64-bit variable on a 32-bit machine), are examples of non-atomic operations. If they are interrupted, they can result in data loss or corruption.

4. Function Reentrancy

A function is reentrant if it is safe to call the function from more than one task, or from both tasks and interrupts.

Each task maintains its own stack and its own set of core register values. If a function does not access any data other than data stored on the stack or held in a register, then the function is reentrant. Listing 2 is an example of a reentrant function. Listing 3 is an example of a function that is not reentrant.

```
/* A parameter is passed into the function. This will either be
passed on the stack or in a CPU register. Either way is safe as
each task maintains its own stack and its own set of register
values. */
long lAddOneHundreded( long lVar1 )
{
/* This function scope variable will also be allocated to the stack
or a register, depending on the compiler and optimization level. Each
task or interrupt that calls this function will have its own copy
of lVar2. */
long lVar2;

    lVar2 = lVar1 + 100;

    /* Most likely the return value will be placed in a CPU register,
although it too could be placed on the stack. */
    return lVar2;
}
```

Listing 2. An example of a reentrant function

```
/* In this case lVar1 is a global variable so every task that calls
the function will be accessing the same single copy of the variable. */
long lVar1;

long lNonsenseFunction( void )
{
/* This variable is static so is not allocated on the stack. Each task
that calls the function will be accessing the same single copy of the
variable. */
static long lState = 0;
long lReturn;

    switch( lState )
    {
        case 0 : lReturn = lVar1 + 10;
                 lState = 1;
                 break;

        case 1 : lReturn = lVar1 + 20;
                 lState = 0;
                 break;
    }
}
```

Listing 3. An example of a function that is not reentrant

Mutual Exclusion

Access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a 'mutual exclusion' technique, to ensure that data consistency is maintained at all times. The goal is to ensure that, once a task starts to access a shared resource, the same task has exclusive access until the resource has been returned to a consistent state.

FreeRTOS provides several features that can be used to implement mutual exclusion, but the best mutual exclusion method is to (whenever possible) design the application in such a way that resources are not shared and each resource is accessed only from a single task.

Scope

This chapter aims to give readers a good understanding of:

- When and why resource management and control is necessary.
- What a critical section is.
- What mutual exclusion means.
- What it means to suspend the scheduler.
- How to use a mutex.
- How to create and use a gatekeeper task.
- What priority inversion is, and how priority inheritance can reduce (but not remove) its impact.

Chapter 5

Memory Management

5.1 Chapter Introduction and Scope

The kernel has to allocate RAM dynamically each time a task, queue, or semaphore is created. The standard `malloc()` and `free()` library functions can be used, but they may not be suitable or appropriate for one or more of the following reasons:

- They are not always available on small embedded systems.
- Their implementation can be relatively large, taking up valuable code space.
- They are rarely thread-safe.
- They are not deterministic; the amount of time taken to execute the functions will differ from call to call.
- They can suffer from memory fragmentation.
- They can complicate the linker configuration.

Different embedded systems have varying RAM allocation and timing requirements, so a single RAM allocation algorithm will only ever be appropriate for a subset of applications. Therefore, FreeRTOS treats memory allocation as part of the portable layer (as opposed to part of the core code base). This enables individual applications to provide their own specific implementations, when appropriate.

When the kernel requires RAM, instead of calling `malloc()` directly it calls `pvPortMalloc()`. When RAM is being freed, instead of calling `free()` directly, the kernel calls `vPortFree()`. `pvPortMalloc()` has the same prototype as `malloc()`, and `vPortFree()` has the same prototype as `free()`.

FreeRTOS comes with three example implementations of both `pvPortMalloc()` and `vPortFree()`; these examples are all documented in this chapter. Users of FreeRTOS can use one of the example implementations, or provide their own.

The three examples are defined in the files `heap_1.c`, `heap_2.c`, and `heap_3.c`—all of which are located in the `FreeRTOS\Source\portable\MemMang` directory. The original memory pool and block allocation scheme used by very early versions of FreeRTOS have been removed because of the effort and understanding required to dimension the blocks and pools.

It is common for small embedded systems only to create tasks, queues, and semaphores before the scheduler has been started. When this is the case, memory only gets dynamically allocated by the kernel before the application starts to perform any real-time functionality, and the memory remains allocated for the lifetime of the application. This means that the chosen allocation scheme does not have to consider any of the more complex issues such as determinism and fragmentation, and can instead consider only attributes such as code size and simplicity.

Scope

This chapter aims to give readers a good understanding of:

- When FreeRTOS allocates RAM.
- The three example memory allocation schemes supplied with FreeRTOS.

Chapter 6

Trouble Shooting

6.1 Chapter Introduction and Scope

This chapter aims to highlight the most common issues encountered by users who are new to FreeRTOS. It focuses mainly on stack overflow and stack overflow detection, because stack issues have proven to be the most frequent source of support requests over the years. It then briefly, and in an FAQ style, touches on other common errors, their possible cause, and their solutions.

printf-stdarg.c

Stack usage can get particularly high when standard C library functions are used, especially IO and string handling functions such as `sprintf()`. The FreeRTOS download includes a file called `printf-stdarg.c` that contains a minimal and stack-efficient version of `sprintf()`, which can be used in place of the standard library version. In most cases, this will permit a much smaller stack to be allocated to each task that calls `sprintf()` and related functions.

`Printf-stdarg.c` is open source but is owned by a third party. Therefore, it is licensed separately from FreeRTOS. The license terms are contained at the top of the source file.

Chapter 7

FreeRTOS-MPU

7.1 Chapter Introduction and Scope

The LPC17xx includes a Memory Protection Unit (MPU). This allows the entire memory map (including Flash, RAM, and peripherals) to be sub-divided into a number of regions, and access permissions to be assigned to each region, individually. A region is an address range consisting of a start address and a size.

FreeRTOS-MPU is a FreeRTOS Cortex M3 port that includes integrated MPU support. It permits additional functionality and includes a slightly extended API, but is otherwise backward compatible with the standard Cortex M3 port.

Using FreeRTOS-MPU will always:

- Protect the kernel from invalid execution by tasks.
- Protect the data used by the kernel from invalid access by tasks.
- Protect the configuration of Cortex M3 core resources, such as the SysTick timer.
- Guarantee that all task stack overflows are detected as soon as they occur.

Also, at the application level, it is possible to ensure that tasks are isolated in their own memory space and that peripherals are protected from unintended modification.

FreeRTOS-MPU provides a simple interface to the MPU by hiding the register level MPU configuration from the user. However, writing an application for an environment that does not permit free access to all data can be challenging.

Scope

This chapter aims to give readers a good understanding of:

- The constraints the MPU hardware places on how memory regions can be defined.
- The access permissions that can be assigned to each memory region.
- The difference between User Mode tasks and Privileged Mode tasks.
- The FreeRTOS-MPU specific API.

Chapter 8

The FreeRTOS Download

8.1 Chapter Introduction and Scope

FreeRTOS is distributed as a single .zip file archive containing all the official FreeRTOS ports and a large number of pre-configured demo applications. The large number of files can seem overwhelming, but only a subset will actually be required.

Scope

This chapter aims to help users orientate themselves with the FreeRTOS files and directories by:

- Providing a top level view of the FreeRTOS directory structure.
- Describing which files are actually required by LPC17xx projects.
- Introducing the demo applications.
- Providing information on how a new project can be created.

The description here relates only to the main FreeRTOS .zip file distribution. The examples that come with this book use a slightly different organization.

Appendix 1: Licensing Information

FreeRTOS is licensed under a modified version of the GNU General Public License (GPL) and *can* be used in commercial applications under that license. An alternative and optional commercial license is also available if:

- You cannot fulfill the requirements stated in the 'Open source modified GPL license' column of Table 2.
- You wish to receive direct technical support.
- You wish to have assistance with your development.
- You require guarantees and indemnification.

Table 2. Comparing the open source license with the commercial license

	Open source modified GPL license	Commercial 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?	No	No
Do I have to open source my changes to the FreeRTOS kernel?	Yes	No
Do I have to document that my product uses FreeRTOS.	Yes	No
Do I have to offer to provide the FreeRTOS source code to users of my application?	Yes (a WEB link to the FreeRTOS.org site is normally sufficient)	No
Can I receive support on a commercial basis?	No	Yes
Are any legal guarantees provided?	No	Yes

Open Source License Details

The FreeRTOS source code is licensed under version 2 of the GNU General Public License (GPL) *with an exception*.

The full text of the GPL is available at <http://www.freertos.org/license.txt>. The text of the exception is provided below.

The exception permits the source code of applications that use FreeRTOS solely through the API published on the FreeRTOS.org website to remain closed source, thus permitting the use of FreeRTOS in commercial applications without necessitating that the entire application be open sourced. The exception can be used only if you wish to combine FreeRTOS with a proprietary product and you comply with the terms stated in the exception itself.

GPL Exception Text

Note that the exception text is subject to change. Consult the FreeRTOS.org website for the most recent version.

Clause 1

Linking FreeRTOS statically or dynamically with other modules is making a combined work based on FreeRTOS. Thus, the terms and conditions of the GNU General Public License cover the whole combination.

As a special exception, the copyright holder of FreeRTOS gives you permission to link FreeRTOS with independent modules that communicate with FreeRTOS solely through the FreeRTOS API interface, regardless of the license terms of these independent modules, and to copy and distribute the resulting combined work under terms of your choice, provided that:

- 1. Every copy of the combined work is accompanied by a written statement that details to the recipient the version of FreeRTOS used and an offer by yourself to provide the FreeRTOS source code (including any modifications you may have made) should the recipient request it.*
- 2. The combined work is not itself an RTOS, scheduler, kernel or related product.*
- 3. The independent modules add significant and primary functionality to FreeRTOS and do not merely extend the existing functionality already present in FreeRTOS.*

An independent module is a module which is not derived from or based on FreeRTOS.

Clause 2

FreeRTOS may not be used for any competitive or comparative purpose, including the publication of any form of run time or compile time metric, without the express permission of Real Time Engineers Ltd. (this is the norm within the industry and is intended to ensure information accuracy).

INDEX

A

Access Permissions, 159
atomic, 119

B

background
 background processing, 41
best fit, 145
Binary Semaphore, 88
Blocked State, 30
Blocking on Queue Reads, 61
Blocking on Queue Writes, 62

C

C library functions, 150
CMSIS, 114
configCHECK_FOR_STACK_OVERFLOW, 152
configKERNEL_INTERRUPT_PRIORITY, 115
configMAX_PRIORITIES, 19, 26
configMAX_SYSCALL_INTERRUPT_PRIORITY, 115
configMINIMAL_STACK_DEPTH, 18
configTICK_RATE_HZ, 26
configTOTAL_HEAP_SIZE, 144
configUSE_IDLE_HOOK, 43
continuous processing, 38
 continuous processing task, 30
co-operative scheduling, 56
Counting Semaphores, 100
Creating Tasks, 17
critical regions, 122
critical section, 116
Critical sections, 122

D

Data Types, 188
Deadlock, 133
Deadly Embrace, 133
deferred interrupts, 88
Deleting a Task, 50

E

errQUEUE_FULL, 67
event driven, 30
events, 86
Events, 86

F

FDA 510(K), 7
fixed execution period, 36
Fixed Priority Pre-emptive Scheduling, 54
Formatting, 189

free(), 142
FreeRTOS-MPU, 158
FromISR, 86
Function Names, 189
Function Reentrancy, 119

G

Gatekeeper tasks, 135

H

handler tasks, 88
Hard real time, 2
Heap_1, 144
Heap_2, 145
Heap_3, 147
high water mark, 151
highest priority, 19

I

Idle Task, 41
Idle Task Hook, 41
IEC 61508, 6
IEC 62304, 7
Interrupt Nesting, 114
interrupt priority, 114

L

locking the scheduler, 123
low power mode, 41
lowest priority, 19, 26
LPCXpresso IDE, 8

M

Macro Names, 189
malloc(), 142
Measuring the amount of spare processing capacity, 41
Memory Protection Unit, 158
MPU, 158
Mutex, 126
mutual exclusion, 120

N

non-atomic, 119
Not Running state, 16

O

OpenRTOS, 6
Overlapping Regions, 161

P

periodic
 periodic tasks, 32
 periodic interrupt, 26
 portable layer, 182
 portBASE_TYPE, 188
 portEND_SWITCHING_ISR(), 96
 portMAX_DELAY, 66, 69
 portSWITCH_TO_USER_MODE(), 172
 portTICK_RATE_MS, 26, 33
 portTickType, 188
 pre-empted
 pre-emption, 41
 Pre-emptive
 Pre-emptive scheduling, 54
 Prioritized Pre-emptive Scheduling, 54
 priority, 19, 26
 priority inheritance, 132
 priority inversion, 131
 Privileged Mode, 159
 pvParameters, 18
 pvPortMalloc(), 142

Q

queue access by Multiple Tasks, 61
 queue block time, 61
 queue item size, 61
 queue length, 61
 Queues, 59

R

RAM allocation, 142
 Read, Modify, Write Operations, 118
 Ready state, 31
 reentrant, 119
 Removing Unused Files, 182, 184
 Run Time Stack Checking, 152
 Running state, 16, 30

S

SafeRTOS, 6
 Soft real time, 2
 spare processing capacity
 measuring spare processing capacity, 35
 sprintf(), 150
 Stack Overflow, 151
 stack overflow hook, 152
 starvation, 28
 starving
 starvation, 30
 state diagram, 31
 Suspended State, 31
 suspending the scheduler, 123
 swapped in, 16
 swapped out, 16
 switched in, 16
 switched out, 16
 Synchronization, 88
 Synchronization events, 30

T

tabs, 189
 task, 2
 Task Functions, 15
 task handle, 19, 47
 Task Parameter, 23
 Task Priorities, 26
 taskYIELD(), 56, 74
 Temporal
 temporal events, 30
 the xSemaphoreCreateMutex(), 128
 thread, 2
 tick count, 27
 tick hook function, 135
 tick interrupt, 26
 ticks, 26
 time slice, 26
 Type Casting, 190

U

User Mode, 159
 uxQueueMessagesWaiting(), 70
 uxTaskGetStackHighWaterMark(), 151
 uxTaskPriorityGet(), 44

V

vApplicationStackOverflowHook, 152
 Variable Names, 189
 vPortFree(), 142
 vSemaphoreCreateBinary(), 89, 103
 vTaskAllocateMPURegions(), 170
 vTaskDelay(), 32
 vTaskDelayUntil(), 35
 vTaskDelete(), 50
 vTaskPrioritySet(), 44
 vTaskResume(), 31
 vTaskSuspend(), 31
 vTaskSuspendAll(), 124

W

Workspace, 9

X

xMemoryRegion, 164
 xPortGetFreeHeapSize(), 147
 xQueueCreate(), 64
 xQueueHandle, 64
 xQueuePeek(), 67
 xQueueReceive(), 67
 xQueueReceiveFromISR(), 107
 xQueueSend(), 65
 xQueueSendFromISR(), 107
 xQueueSendToBack(), 65
 xQueueSendToBackFromISR(), 107
 xQueueSendToFront(), 65
 xQueueSendToFrontFromISR(), 107
 xSemaphoreCreateCounting(), 103
 xSemaphoreGiveFromISR(), 93
 xSemaphoreHandle, 89, 103, 128

xSemaphoreTake(), 92
xTaskCreate(), 17
xTaskCreateRestricted(), 164
xTaskGetTickCount(), 37
xTaskParameters, 164
xTaskResumeAll(), 124

xTaskResumeFromISR(), 31

Z

zip file, 180