The FreeRTOS™

Reference Manual

The FreeRTOS™

Reference Manual

# API Functions and Configuration Options

Real Time Engineers ltd.

Reference Manual for FreeRTOS version 8.2.1

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

© Real Time Engineers Ltd. 2015. All rights reserved.

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

Engineers Ltd.

http://www.FreeRTOS.org http://www.FreeRTOS.org/plus http://www.FreeRTOS.org/labs

# Contents

Contents ................................................................................................................................... 5

List of Figures ........................................................................................................................... 8

List of Code Listings ................................................................................................................. 9

List of Tables .......................................................................................................................... 14

List of Notation ........................................................................................................................ 14

Chapter 1 About This Manual ............................................................................................... 16

1.1 Scope........................................................................................................................ 17

Chapter 2 Task and Scheduler API ...................................................................................... 20

2.1 portSWITCH\_TO\_USER\_MODE() ............................................................................ 21

2.2 vTaskAllocateMPURegions() .................................................................................... 22

2.3 xTaskCallApplicationHook() ...................................................................................... 25

2.4 xTaskCheckForTimeOut() ......................................................................................... 27

2.5 xTaskCreate() ........................................................................................................... 29

2.6 xTaskCreateRestricted() ........................................................................................... 33

2.7 vTaskDelay() ............................................................................................................. 38

2.8 vTaskDelayUntil() ...................................................................................................... 40

2.9 vTaskDelete() ............................................................................................................ 43

2.10 taskDISABLE\_INTERRUPTS() ................................................................................. 45 2.11 taskENABLE\_INTERRUPTS() .................................................................................. 47

2.12 taskENTER\_CRITICAL() ........................................................................................... 48

2.13 taskEXIT\_CRITICAL() ............................................................................................... 51

2.14 xTaskGetApplicationTaskTag() ................................................................................. 53

2.15 xTaskGetCurrentTaskHandle().................................................................................. 55

2.16 xTaskGetIdleTaskHandle() ........................................................................................ 56

2.17 uxTaskGetNumberOfTasks() .................................................................................... 57

2.18 vTaskGetRunTimeStats() .......................................................................................... 58 2.19 xTaskGetSchedulerState() ........................................................................................ 62

2.20 uxTaskGetStackHighWaterMark() ............................................................................. 63

2.21 eTaskGetState() ........................................................................................................ 65

2.22 uxTaskGetSystemState() .......................................................................................... 67

2.23 pvTaskGetTheadLocalStoragePointer() .................................................................... 71

2.24 pcTaskGetTaskName() ............................................................................................. 73

2.25 xTaskGetTickCount() ................................................................................................ 74

2.26 xTaskGetTickCountFromISR() .................................................................................. 76

2.27 vTaskList() ................................................................................................................ 78

2.28 xTaskNotify() ............................................................................................................. 81

2.29 xTaskNotifyAndQuery() ............................................................................................. 84

2.30 xTaskNotifyFromISR() ............................................................................................... 87

2.31 xTaskNotifyGive() ...................................................................................................... 91

2.32 vTaskNotifyGiveFromISR() ........................................................................................ 93

2.33 ulTaskNotifyTake() .................................................................................................... 96

2.34 xTaskNotifyWait() ...................................................................................................... 99

2.35 uxTaskPriorityGet() ................................................................................................. 102

2.36 vTaskPrioritySet() .................................................................................................... 104

2.37 vTaskResume() ....................................................................................................... 106

2.38 xTaskResumeAll() ................................................................................................... 108

2.39 xTaskResumeFromISR() ......................................................................................... 110

2.40 vTaskSetApplicationTaskTag() ................................................................................ 113

2.41 vTaskSetTheadLocalStoragePointer() ..................................................................... 115

2.42 vTaskSetTimeOutState() ......................................................................................... 117

2.43 vTaskStartScheduler() ............................................................................................. 119

2.44 vTaskStepTick() ...................................................................................................... 121

2.45 vTaskSuspend() ...................................................................................................... 123

2.46 vTaskSuspendAll() .................................................................................................. 125

2.47 taskYIELD() ............................................................................................................. 127

Chapter 3 Queue API ......................................................................................................... 129

3.1 vQueueAddToRegistry() .......................................................................................... 130

3.2 xQueueAddToSet() ................................................................................................. 132

3.3 xQueueCreate()....................................................................................................... 134

3.4 xQueueCreateSet() ................................................................................................. 136

3.5 vQueueDelete() ....................................................................................................... 140

3.6 xQueueIsQueueEmptyFromISR() ............................................................................ 142

3.7 xQueueIsQueueFullFromISR() ................................................................................ 143

3.8 uxQueueMessagesWaiting() ................................................................................... 144

3.9 uxQueueMessagesWaitingFromISR() ..................................................................... 145

3.10 xQueueOverwrite() .................................................................................................. 147

3.11 xQueueOverwriteFromISR() .................................................................................... 149

3.12 xQueuePeek() ......................................................................................................... 151

3.13 xQueuePeekFromISR() ........................................................................................... 154

3.14 xQueueReceive() .................................................................................................... 155

3.15 xQueueReceiveFromISR() ...................................................................................... 158

3.16 xQueueRemoveFromSet() ...................................................................................... 161

3.17 xQueueReset() ........................................................................................................ 163

3.18 xQueueSelectFromSet() .......................................................................................... 164

3.19 xQueueSelectFromSetFromISR() ............................................................................ 166

3.20 xQueueSend(), xQueueSendToFront(), xQueueSendToBack() ............................... 168

3.21 xQueueSendFromISR(), xQueueSendToBackFromISR(),

xQueueSendToFrontFromISR() .............................................................................. 171

3.22 uxQueueSpacesAvailable() ..................................................................................... 175 Chapter 4 Semaphore API ................................................................................................. 177

4.1 vSemaphoreCreateBinary() ..................................................................................... 178 4.2 xSemaphoreCreateBinary() ..................................................................................... 181

4.3 xSemaphoreCreateCounting() ................................................................................ 183

4.4 xSemaphoreCreateMutex() ..................................................................................... 185

4.5 xSemaphoreCreateRecursiveMutex() ..................................................................... 187

4.6 vSemaphoreDelete() ............................................................................................... 189

4.7 xSemaphoreGetMutexHolder()................................................................................ 190

4.8 xSemaphoreGive() .................................................................................................. 191

4.9 xSemaphoreGiveFromISR() .................................................................................... 193

4.10 xSemaphoreGiveRecursive() .................................................................................. 196

4.11 xSemaphoreTake() ................................................................................................. 199

4.12 xSemaphoreTakeFromISR() ................................................................................... 202

4.13 xSemaphoreTakeRecursive() .................................................................................. 204

Chapter 5 Software Timer API ........................................................................................... 207

5.1 xTimerChangePeriod() ............................................................................................ 208

5.2 xTimerChangePeriodFromISR() .............................................................................. 211

5.3 xTimerCreate() ........................................................................................................ 213 5.4 xTimerDelete() ........................................................................................................ 217

5.5 xTimerGetTimerDaemonTaskHandle() .................................................................... 219

5.6 pvTimerGetTimerID() .............................................................................................. 220

5.1 pcTimerGetTimerName() ........................................................................................ 222

5.2 xTimerIsTimerActive() ............................................................................................. 223

5.3 xTimerPendFunctionCall() ....................................................................................... 225

5.4 xTimerPendFunctionCallFromISR() ........................................................................ 227

5.5 xTimerReset() ......................................................................................................... 230

5.6 xTimerResetFromISR() ........................................................................................... 233

5.7 vTimerSetTimerID() ................................................................................................. 235

5.8 xTimerStart() ........................................................................................................... 237

5.9 xTimerStartFromISR() ............................................................................................. 239

5.10 xTimerStop() ........................................................................................................... 241

5.11 xTimerStopFromISR() ............................................................................................. 243

Chapter 6 Event Groups API .............................................................................................. 245

6.1 xEventGroupCreate() .............................................................................................. 246

6.2 xEventGroupClearBits() .......................................................................................... 248

6.3 xEventGroupClearBitsFromISR() ............................................................................ 250

6.1 vEventGroupDelete() .............................................................................................. 252 6.2 xEventGroupGetBits() ............................................................................................. 253

6.1 xEventGroupGetBitsFromISR() ............................................................................... 254

6.2 xEventGroupSetBits() ............................................................................................. 255

6.3 xEventGroupSetBitsFromISR() ............................................................................... 257

6.1 xEventGroupSync() ................................................................................................. 260

6.2 xEventGroupWaitBits() ............................................................................................ 264

Chapter 7 Kernel Configuration .......................................................................................... 267

7.1 FreeRTOSConfig.h .................................................................................................. 268

7.2 Constants that Start “INCLUDE\_” ............................................................................ 269

7.3 Constants that Start “config” .................................................................................... 273

APPENDIX 1: Data Types and Coding Style Guide .......................................................... 292

APPENDIX 2: Licensing Information ................................................................................. 295

INDEX ................................................................................................................................... 298

# List of Figures

Figure 1 An example of the table produced by calling vTaskGetRunTimeStats() .................... 58 Figure 2 An example of the table produced by calling vTaskList() ........................................... 78 Figure 3 Time line showing the execution of 4 tasks, all of which run at the idle priority ........ 278 Figure 4 An example interrupt priority configuration .............................................................. 280

# List of Code Listings

Listing 1 portSWITCH\_TO\_USER\_MODE() macro prototype ................................................. 21 Listing 2 vTaskAllocateMPURegions() function prototype ....................................................... 22 Listing 3 The data structures used by xTaskCreateRestricted() .............................................. 23 Listing 4 Example use of vTaskAllocateMPURegions() ........................................................... 24 Listing 5 xTaskCallApplicationTaskHook() function prototype ................................................. 25 Listing 6 The prototype to which all task hook functions must conform ................................... 25 Listing 7 Example use of xTaskCallApplicationTaskHook() ..................................................... 26 Listing 5 xTaskCheckForTimeOut() function prototype ........................................................... 27 Listing 7 Example use of vTaskSetTimeOutState() and xTaskCheckForTimeOut() ................. 28

Listing 8 xTaskCreate() function prototype .............................................................................. 29 Listing 9 Example use of xTaskCreate() ................................................................................. 32 Listing 10 xTaskCreateRestricted() function prototype ............................................................ 33 Listing 11 The data structures used by xTaskCreateRestricted() ............................................ 34 Listing 12 Statically declaring a correctly aligned stack for use by a restricted task ................. 35 Listing 13 Example use of xTaskCreateRestricted() ................................................................ 37 Listing 14 vTaskDelay() function prototype ............................................................................. 38

Listing 15 Example use of vTaskDelay() ................................................................................. 39 Listing 16 vTaskDelayUntil() function prototype ...................................................................... 40

Listing 17 Example use of vTaskDelayUntil() .......................................................................... 42 Listing 18 vTaskDelete() function prototype ............................................................................ 43 Listing 19 Example use of the vTaskDelete() .......................................................................... 44 Listing 20 taskDISABLE\_INTERRUPTS() macro prototype .................................................... 45

Listing 21 taskENABLE\_INTERRUPTS() macro prototype ..................................................... 47 Listing 22 taskENTER\_CRITICAL macro prototype ................................................................ 48 Listing 23 Example use of taskENTER\_CRITICAL() and taskEXIT\_CRITICAL() ..................... 50 Listing 24 taskEXIT\_CRITICAL() macro prototype .................................................................. 51 Listing 25 xTaskGetApplicationTaskTag() function prototype .................................................. 53 Listing 26 Example use of xTaskGetApplicationTaskTag() ..................................................... 54

Listing 27 xTaskGetCurrentTaskHandle() function prototype .................................................. 55 Listing 28 xTaskGetIdleTaskHandle() function prototype ........................................................ 56 Listing 29 uxTaskGetNumberOfTasks() function prototype ..................................................... 57 Listing 30 vTaskGetRunTimeStats() function prototype .......................................................... 58 Listing 31 Example macro definitions, taken from the LM3Sxxx Eclipse Demo ....................... 60

Listing 32 Example macro definitions, taken from the LPC17xx Eclipse Demo ....................... 61 Listing 33 Example use of vTaskGetRunTimeStats() .............................................................. 61 Listing 34 xTaskGetSchedulerState() function prototype ......................................................... 62 Listing 35 Example use of uxTaskGetStackHighWaterMark() ................................................. 64 Listing 36 eTaskGetState() function prototype ........................................................................ 65 Listing 37 uxTaskGetSystemState() function prototype ........................................................... 67

Listing 38 Example use of uxTaskGetSystemStatus() ............................................................. 69 Listing 39 The TaskStatus\_t definition ..................................................................................... 70

Listing 40 pvTaskGetThreadLocalStoragePointer() function prototype .................................... 71

Listing 41 Example use of pvTaskGetThreadLocalStoragePointer() ........................................ 72 Listing 42 pcTaskGetTaskName() function prototype .............................................................. 73 Listing 43 xTaskGetTickCount() function prototype ................................................................. 74

Listing 44 Example use of xTaskGetTickCount() ..................................................................... 75 Listing 45 xTaskGetTickCountFromISR() function prototype ................................................... 76

Listing 46 Example use of xTaskGetTickCountFromISR() ....................................................... 77 Listing 47 vTaskList() function prototype ................................................................................. 78

Listing 48 Example use of vTaskList() ..................................................................................... 80 Listing 49 xTaskNotify() function prototype .............................................................................. 81 Listing 50 Example use of xTaskNotify() ................................................................................. 83 Listing 51 xTaskNotifyAndQuery() function prototype .............................................................. 84 Listing 52 Example use of xTaskNotify() ................................................................................. 86 Listing 53 xTaskNotifyFrom() function prototype ..................................................................... 87 Listing 54 Example use of xTaskNotifyFromISR() ................................................................... 90 Listing 55 xTaskNotifyGive() function prototype ...................................................................... 91

Listing 56 Example use of xTaskNotifyGive() .......................................................................... 92 Listing 57 vTaskNotifyGiveFromISR() function prototype ........................................................ 93

Listing 58 Example use of vTaskNotifyGiveFromISR() ............................................................ 95 Listing 59 ulTaskNotifyTake() function prototype ..................................................................... 96 Listing 60 Example use of ulTaskNotifyTake()......................................................................... 98 Listing 61 xTaskNotifyWait() function prototype ....................................................................... 99 Listing 62 Example use of xTaskNotifyWait() ........................................................................ 101

Listing 63 uxTaskPriorityGet() function prototype .................................................................. 102

Listing 64 Example use of uxTaskPriorityGet() ...................................................................... 103 Listing 65 vTaskPrioritySet() function prototype .................................................................... 104

Listing 66 Example use of vTaskPrioritySet() ........................................................................ 105 Listing 67 vTaskResume() function prototype ....................................................................... 106

Listing 68 Example use of vTaskResume() ........................................................................... 107

Listing 69 xTaskResumeAll() function prototype .................................................................... 108 Listing 70 Example use of xTaskResumeAll() ....................................................................... 109 Listing 71 xTaskResumeFromISR() function prototype ......................................................... 110

Listing 72 Example use of vTaskResumeFromISR() ............................................................. 112 Listing 73 vTaskSetApplicationTaskTag() function prototype ................................................ 113

Listing 74 Example use of vTaskSetApplicationTaskTag() .................................................... 114 Listing 75 vTaskSetThreadLocalStoragePointer() function prototype .................................... 115

Listing 76 Example use of vTaskSetThreadLocalStoragePointer() ........................................ 116 Listing 5 vTaskSetTimeOutState() function prototype............................................................ 117 Listing 7 Example use of vTaskSetTimeOutState() and xTaskCheckForTimeOut() ............... 118

Listing 77 vTaskStartScheduler() function prototype ............................................................. 119

Listing 78 Example use of vTaskStartScheduler() ................................................................. 120

Listing 79 Example use of vTaskStepTick() ........................................................................... 122 Listing 80 vTaskSuspend() function prototype ...................................................................... 123

Listing 81 Example use of vTaskSuspend() .......................................................................... 124 Listing 82 vTaskSuspendAll() function prototype................................................................... 125

Listing 83 Example use of vTaskSuspendAll() ...................................................................... 126 Listing 84 taskYIELD() macro prototype ................................................................................ 127

Listing 85 Example use of taskYIELD() ................................................................................. 128 Listing 86 vQueueAddToRegistry() function prototype .......................................................... 130

Listing 87 Example use of vQueueAddToRegistry() .............................................................. 131 Listing 88 xQueueAddToSet() function prototype .................................................................. 132 Listing 89 xQueueCreate() function prototype ....................................................................... 134

Listing 90 Example use of xQueueCreate() ........................................................................... 135 Listing 91 xQueueCreateSet() function prototype.................................................................. 136 Listing 92 Example use of xQueueCreateSet() and other queue set API functions ............... 139 Listing 93 vQueueDelete() function prototype ....................................................................... 140

Listing 94 Example use of vQueueDelete() ........................................................................... 141 Listing 95 xQueueIsQueueEmptyFromISR() function prototype ............................................ 142 Listing 96 xQueueIsQueueFullFromISR() function prototype ................................................ 143 Listing 97 uxQueueMessagesWaiting() function prototype .................................................... 144 Listing 98 Example use of uxQueueMessagesWaiting() ....................................................... 144 Listing 99 uxQueueMessagesWaitingFromISR() function prototype...................................... 145 Listing 100 Example use of uxQueueMessagesWaitingFromISR() ....................................... 146 Listing 101 xQueueOverwrite() function prototype ................................................................ 147

Listing 102 Example use of xQueueOverwrite() .................................................................... 148 Listing 103 xQueueOverwriteFromISR() function prototype .................................................. 149

Listing 104 Example use of xQueueOverwriteFromISR() ...................................................... 150 Listing 105 xQueuePeek() function prototype ....................................................................... 151

Listing 106 Example use of xQueuePeek() ........................................................................... 153 Listing 107 xQueuePeekFromISR() function prototype ......................................................... 154 Listing 108 xQueueReceive() function prototype ................................................................... 155 Listing 109 Example use of xQueueReceive() ...................................................................... 157 Listing 110 xQueueReceiveFromISR() function prototype ..................................................... 158 Listing 111 Example use of xQueueReceiveFromISR() ........................................................ 160

Listing 112 xQueueRemoveFromSet() function prototype ..................................................... 161 Listing 113 Example use of xQueueRemoveFromSet() ........................................................ 162 Listing 114 xQueueReset() function prototype ...................................................................... 163 Listing 115 xQueueSelectFromSet() function prototype ........................................................ 164 Listing 116 xQueueSelectFromSetFromISR() function prototype .......................................... 166

Listing 117 Example use of xQueueSelectFromSetFromISR() .............................................. 167

Listing 118 xQueueSend(), xQueueSendToFront() and xQueueSendToBack() function

prototypes ...................................................................................................... 168 Listing 119 Example use of xQueueSendToBack() ............................................................... 170

Listing 120 xQueueSendFromISR(), xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() function prototypes ...................................... 171 Listing 121 Example use of xQueueSendToBackFromISR() ................................................. 174 Listing 122 uxQueueSpacesAvailable() function prototype .................................................... 175

Listing 123 Example use of uxQueueSpacesAvailable() ....................................................... 175

Listing 124 vSemaphoreCreateBinary() macro prototype ...................................................... 178

Listing 125 Example use of vSemaphoreCreateBinary() ....................................................... 180

Listing 126 xSemaphoreCreateBinary() function prototype ................................................... 181

Listing 127 Example use of xSemaphoreCreateBinary() ....................................................... 182 Listing 128 xSemaphoreCreateCounting() function prototype ............................................... 183

Listing 129 Example use of xSemaphoreCreateCounting() ................................................... 184 Listing 130 xSemaphoreCreateMutex() function prototype .................................................... 185

Listing 131 Example use of xSemaphoreCreateMutex() ........................................................ 186 Listing 132 xSemaphoreCreateRecursiveMutex() function prototype .................................... 187

Listing 133 Example use of xSemaphoreCreateRecursiveMutex() ........................................ 188 Listing 134 vSemaphoreDelete() function prototype .............................................................. 189 Listing 135 xSemaphoreGetMutexHolder() function prototype .............................................. 190 Listing 136 xSemaphoreGive() function prototype ................................................................. 191 Listing 137 Example use of xSemaphoreGive()..................................................................... 192 Listing 138 xSemaphoreGiveFromISR() function prototype ................................................... 193 Listing 139 Example use of xSemaphoreGiveFromISR() ...................................................... 195 Listing 140 xSemaphoreGiveRecursive() function prototype ................................................. 196

Listing 141 Example use of xSemaphoreGiveRecursive() ..................................................... 198 Listing 142 xSemaphoreTake() function prototype ................................................................ 199

Listing 143 Example use of xSemaphoreTake() .................................................................... 201 Listing 144 xSemaphoreTakeFromISR() function prototype .................................................. 202

Listing 145 xSemaphoreTakeRecursive() function prototype................................................. 204

Listing 146 Example use of xSemaphoreTakeRecursive() .................................................... 206 Listing 147 xTimerChangePeriod() function prototype ........................................................... 208 Listing 148 Example use of xTimerChangePeriod() .............................................................. 210 Listing 149 xTimerChangePeriodFromISR() function prototype ............................................. 211 Listing 150 Example use of xTimerChangePeriodFromISR() ................................................ 212 Listing 151 xTimerCreate() function prototype ....................................................................... 213 Listing 152 The timer callback function prototype .................................................................. 214

Listing 153 Definition of the callback function used in the calls to xTimerCreate() in

Listing 154 ..................................................................................................... 215 Listing 154 Example use of xTimerCreate() .......................................................................... 216 Listing 155 xTimerDelete() macro prototype .......................................................................... 217 Listing 156 xTimerGetTimerDaemonTaskHandle() function prototype .................................. 219 Listing 157 pvTimerGetTimerID() function prototype ............................................................. 220

Listing 158 Example use of pvTimerGetTimerID() ................................................................. 221 Listing 159 pcTimerGetTimerName() function prototype ....................................................... 222 Listing 160 xTimerIsTimerActive() function prototype ............................................................ 223

Listing 161 Example use of xTimerIsTimerActive() ................................................................ 224 Listing 162 xTimerPendFunctionCall() function prototype ..................................................... 225

Listing 163 The prototype of a function that can be pended using a call to

xTimerPendFunctionCall() .............................................................................. 225

Listing 164 xTimerPendFunctionCallFromISR() function prototype ....................................... 227

Listing 165 The prototype of a function that can be pended using a call to xTimerPendFunctionCallFromISR() ............................................................... 227 Listing 166 Example use of xTimerPendFunctionCallFromISR() ........................................... 229 Listing 167 xTimerReset() function prototype ........................................................................ 230 Listing 168 Example use of xTimerReset()............................................................................ 232 Listing 169 xTimerResetFromISR() function prototype .......................................................... 233 Listing 170 Example use of xTimerResetFromISR() ............................................................. 234 Listing 171 vTimerSetTimerID() function prototype ............................................................... 235

Listing 172 Example use of vTimerSetTimerID() ................................................................... 236 Listing 173 xTimerStart() function prototype .......................................................................... 237 Listing 174 xTimerStartFromISR() macro prototype .............................................................. 239

Listing 175 Example use of xTimerStartFromISR() ............................................................... 240 Listing 176 xTimerStop() function prototype .......................................................................... 241 Listing 177 xTimerStopFromISR() function prototype ............................................................ 243 Listing 178 Example use of xTimerStopFromISR() ............................................................... 244 Listing 179 xEventGroupCreate() function prototype ............................................................. 246 Listing 180 Example use of xEventGroupCreate() ................................................................ 247

Listing 181 xEventGroupClearBits() function prototype ......................................................... 248

Listing 182 Example use of xEventGroupClearBits() ............................................................. 249 Listing 183 xEventGroupClearBitsFromISR() function prototype ........................................... 250 Listing 184 Example use of xEventGroupClearBitsFromISR()............................................... 251 Listing 185 xEventGroupDelete() function prototype ............................................................. 252

Listing 186 xEventGroupGetBits() function prototype ............................................................ 253 Listing 187 xEventGroupGetBitsFromISR() function prototype.............................................. 254 Listing 188 xEventGroupSetBits() function prototype ............................................................ 255

Listing 189 Example use of xEventGroupSetBits() ................................................................ 256 Listing 190 xEventGroupSetBitsFromISR() function prototype .............................................. 257

Listing 191 Example use of xEventGroupSetBitsFromISR() .................................................. 259 Listing 192 xEventGroupSync() function prototype ............................................................... 260

Listing 193 Example use of xEventGroupSync() ................................................................... 263 Listing 194 xEventGroupWaitBits() function prototype .......................................................... 264

Listing 195 Example use of xEventGroupWaitBits() .............................................................. 266 Listing 196 An example configASSERT() definition ............................................................... 273 Listing 197 The stack overflow hook function prototype ........................................................ 274 Listing 198 An example of saving and restoring the processors privilege state ..................... 279 Listing 199 The idle task hook function name and prototype. ................................................ 285 Listing 200 The malloc() failed hook function name and prototype. ....................................... 286

Listing 201 The tick hook function name and prototype......................................................... 289

# List of Tables

Table 1. eTaskGetState() return values.................................................................................. 65 Table 2. Additional macros that are required if ...................................................................... 276 Table 3. Special data types used by FreeRTOS ................................................................... 292 Table 4. Macro prefixes ........................................................................................................ 294 Table 5. Common macro definitions ..................................................................................... 294

Table 6. Comparing the open source license with the commercial license ........................... 296

# List of Notation

API Application Programming Interface

ISR Interrupt Service Routine

MPU Memory Protection Unit

RTOS Real-time Operating System

Chapter 1

About This Manual

# 1.1 Scope

This document provides a technical reference to both the primary FreeRTOS API[[1]](#footnote-1), and the FreeRTOS kernel configuration options. It is assumed the reader is already familiar with the concepts of writing multi tasking applications, and the primitives provided by real time kernels. Readers that are not familiar with these fundamental concepts are recommended to read the book “Using the FreeRTOS Real Time Kernel – A Practical Guide” for a much more descriptive, hands on, and tutorial style text. The book can be obtained from http://www.FreeRTOS.org/Documentation.

The Order in Which Functions Appear in This Manual

Within this document, the API functions have been split into five groups – task and scheduler related functions, queue related functions, semaphore related functions, software timer related functions and event group related functions. Each group is documented in its own chapter, and within each chapter, the API functions are listed in alphabetical order. Note however that the name of each API function is prefixed with one or more letters that specify the function’s return type, and the alphabetical ordering of API functions within each chapter ignores the function return type prefix. APPENDIX 1: describes the prefixes in more detail.

As an example, consider the API function that is used to create a FreeRTOS task. Its name is xTaskCreate(). The ‘x’ prefix specifies that xTaskCreate() returns a non-standard type. The secondary ‘Task’ prefix specifies that the function is a task related function, and, as such, will be documented in the chapter that contains task and scheduler related functions. The ‘x’ is not considered in the alphabetical ordering, so xTaskCreate() will appear in the task and scheduler chapter ordered as if its name was just TaskCreate().

API Usage Restrictions

The following rules apply when using the FreeRTOS API:

1. API functions that do not end in “FromISR” must not be used in an interrupt service routine (ISR). Some FreeRTOS ports make a further restriction that even API functions that do end in “FromISR” cannot be used in an interrupt service routine that has a (hardware) priority above the priority set by the configMAX\_SYSCALL\_INTERRUPT\_PRIORITY (or configMAX\_API\_CALL\_INTERRUPT\_PRIORITY, depending on the port) kernel configuration constant, which is described in section 7.1 of this document. The second restriction is to ensure that the timing, determinism and latency of interrupts that have a priority above that set by configMAX\_SYSCALL\_INTERRUPT\_PRIORITY are not affected by FreeRTOS.
2. API functions that can potentially cause a context switch must not be called while the scheduler is suspended.
3. API functions that can potentially cause a context switch must not be called from within a critical section.

Chapter 2

Task and Scheduler API

# 2.1 portSWITCH\_TO\_USER\_MODE()

#include “FreeRTOS.h” #include “task.h”

void portSWITCH\_TO\_USER\_MODE( void );

Listing 1 portSWITCH\_TO\_USER\_MODE() macro prototype

Summary

This function is intended for advanced users only and is only relevant to FreeRTOS MPU ports (FreeRTOS ports that make use of a Memory Protection Unit).

MPU restricted tasks are created using xTaskCreateRestricted(). The parameters supplied to xTaskCreateRestricted() specify whether the task being created should be a User (unprivileged) mode task, or a Supervisor (privileged) mode task. A Supervisor mode task can call portSWITCH\_TO\_USER\_MODE() to convert itself from a Supervisor mode task into a User mode task.

Parameters None.

Return Values None.

Notes

There is no reciprocal equivalent to portSWITCH\_TO\_USER\_MODE() that permits a task to convert itself from a User mode into a Supervisor mode task.

# 2.2 vTaskAllocateMPURegions()

#include “FreeRTOS.h” #include “task.h”

void vTaskAllocateMPURegions( TaskHandle\_t xTaskToModify, const MemoryRegion\_t \* const xRegions );

Listing 2 vTaskAllocateMPURegions() function prototype

Summary

Define a set of Memory Protection Unit (MPU) regions for use by an MPU restricted task.

This function is intended for advanced users only and is only relevant to FreeRTOS MPU ports (FreeRTOS ports that make use of a Memory Protection Unit).

MPU controlled memory regions can be assigned to an MPU restricted task when the task is created using the xTaskCreateRestricted() function. They can then be redefined (or reassigned) at run time using the vTaskAllocateMPURegions() function.

Parameters

|  |  |
| --- | --- |
| xTaskToModify | The handle of the restricted task being modified (the task that is being given access to the memory regions defined by the xRegions parameter).  The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreateRestricted() API function.  A task can modify its own memory region access definitions by passing NULL in place of a valid task handle. |
| xRegions | An array of MemoryRegion\_t structures. The number of positions in the |

array is defined by the port specific portNUM\_CONFIGURABLE\_REGIONS constant. On a Cortex-M3 portNUM\_CONFIGURABLE\_REGIONS is defined as three.

Each MemoryRegion\_t structure in the array defines a single MPU memory region for use by the task referenced by the xTaskToModify parameter.

Notes

MPU memory regions are defined using the MemoryRegion\_t structure shown in Listing 3.

typedef struct xMEMORY\_REGION

{

void \*pvBaseAddress; unsigned long ulLengthInBytes; unsigned long ulParameters;

} MemoryRegion\_t;

Listing 3 The data structures used by xTaskCreateRestricted()

The pvBaseAddress and ulLengthInBytes members are self explanatory as the start of the memory region and the length of the memory region respectively. These must comply with the size and alignment restrictions imposed by the MPU. In particular, the size and alignment of each region must both be equal to the same power of two value.

ulParameters defines how the task is permitted to access the memory region being defined, and can take the bitwise OR of the following values:

* portMPU\_REGION\_READ\_WRITE
* portMPU\_REGION\_PRIVILEGED\_READ\_ONLY
* portMPU\_REGION\_READ\_ONLY
* portMPU\_REGION\_PRIVILEGED\_READ\_WRITE
* portMPU\_REGION\_CACHEABLE\_BUFFERABLE
* portMPU\_REGION\_EXECUTE\_NEVER

Example

/\* Define an array that the task will both read from and write to. Make sure the size and alignment are appropriate for an MPU region (note this uses GCC syntax). \*/ static unsigned char ucOneKByte[ 1024 ] \_\_attribute\_\_((align( 1024 )));

/\* Define an array of MemoryRegion\_t structures that configures an MPU region allowing read/write access for 1024 bytes starting at the beginning of the ucOneKByte array. The other two of the maximum three definable regions are unused, so set to zero. \*/ static const MemoryRegion\_t xAltRegions[ portNUM\_CONFIGURABLE\_REGIONS ] =

{

/\* Base address Length Parameters \*/

{ ucOneKByte, 1024, portMPU\_REGION\_READ\_WRITE },

{ 0, 0, 0 },

{ 0, 0, 0 }

};

void vATask( void \*pvParameters )

{

/\* This task was created using xTaskCreateRestricted() to have access to a maximum of three MPU controlled memory regions. At some point it is required that these MPU regions are replaced with those defined in the xAltRegions const structure defined above. Use a call to vTaskAllocateMPURegions() for this purpose. NULL is used as the task handle to indicate that the change should be applied to the calling task. \*/

vTaskAllocateMPURegions( NULL, xAltRegions );

/\* Now the task can continue its function, but from this point on can only access its stack and the ucOneKByte array (unless any other statically defined or shared regions have been declared elsewhere). \*/

}

Listing 4 Example use of vTaskAllocateMPURegions()

# 2.3 xTaskCallApplicationTaskHook()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskCallApplicationTaskHook( TaskHandle\_t xTask, void \*pvParameters );

Listing 5 xTaskCallApplicationTaskHook() function prototype

Summary

This function is intended for advanced users only.

The vTaskSetApplicationTaskTag() function can be used to assign a ‘tag’ value to a task. The meaning and use of the tag value is defined by the application writer. The kernel itself will not normally access the tag value.

As a special case, the tag value can be used to associate a ‘task hook’ (or callback) function to a task. When this is done, the hook function is called using xTaskCallApplicationTaskHook().

Task hook functions can be used for any purpose. The example shown in this section demonstrates a task hook being used to output debug trace information.

Task hook functions must have the prototype demonstrated by Listing 6.

BaseType\_t xAnExampleTaskHookFunction( void \*pvParameters );

Listing 6 The prototype to which all task hook functions must conform

xTaskCallApplicationTaskHook() is only available when configUSE\_APPLICATION\_TASK\_TAG is set to 1 in FreeRTOSConfig.h.

Parameters xTask The handle of the task whose associated hook function is being called. See

the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks.

A task can call its own hook function by passing NULL in place of a valid task handle.

pvParameters The value used as the parameter to the task hook function itself.

This parameter has the type ‘pointer to void’ to allow the task hook function parameter to effectively, and indirectly by means of casting, receive a parameter of any type. For example, integer types can be passed into a hook function by casting the integer to a void pointer at the point the hook function is called, then by casting the void pointer parameter back to an integer within the hook function itself.

Example

/\* Define a hook (callback) function – using the required prototype as demonstrated by Listing 6 \*/

static BaseType\_t prvExampleTaskHook( void \* pvParameter )

{

/\* Perform an action - this could be anything. In this example the hook is used to output debug trace information. pxCurrentTCB is the handle of the currently executing task. (vWriteTrace() is not an API function, its just used as an example.) \*/ vWriteTrace( pxCurrentTCB );

/\* This example does not make use of the hook return value so just returns

0 in every case. \*/ return 0;

}

/\* Define an example task that makes use of its tag value. \*/ void vAnotherTask( void \*pvParameters )

{

/\* vTaskSetApplicationTaskTag() sets the ‘tag’ value associated with a task. NULL is used in place of a valid task handle to indicate that it should be the tag value of the calling task that gets set. In this example the ‘value’ being set is the hook function. \*/

vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );

for( ;; )

{

/\* The rest of the task code goes here. \*/ }

}

/\* Define the traceTASK\_SWITCHED\_OUT() macro to call the hook function of each task that is switched out. pxCurrentTCB points to the handle of the currently running task. \*/

#define traceTASK\_SWITCHED\_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, 0 )

Listing 7 Example use of xTaskCallApplicationTaskHook()

# 2.4 xTaskCheckForTimeOut()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskCheckForTimeOut( TimeOut\_t \* const pxTimeOut,

TickType\_t \* const pxTicksToWait );

Listing 8 xTaskCheckForTimeOut() function prototype

Summary

This function is intended for advanced users only.

A task can enter the Blocked state to wait for an event. Typically, the task will not wait in the Blocked state indefinitely, but instead a timeout period will be specified. The task will be removed from the Blocked state if the timeout period expires before the event the task is waiting for occurs.

If a task enters and exits the Blocked state more than once while it is waiting for the event to occur then the timeout used each time the task enters the Blocked state must be adjusted to ensure the total of all the time spent in the Blocked state does not exceed the originally specified timeout period. xTaskCheckForTimeOut() performs the adjustment, taking into account occasional occurrences such as tick count overflows, which would otherwise make a manual adjustment prone to error.

xTaskCheckForTimeOut() is used with vTaskSetTimeOutState(). vTaskSetTimeOutState() is called to set the initial condition, after which xTaskCheckForTimeOut() can be called to check for a timeout condition, and adjust the remaining block time if a timeout has not occurred.

Parameters pxTimeOut A pointer to a structure that holds information necessary to determine if a

timeout has occurred. pxTimeOut is initialized using vTaskSetTimeOutState().

pxTicksToWait Used to pass out an adjusted block time, which is the block time that remains

after taking into account the time already spent in the Blocked state.

Returned If pdTRUE is returned then no block time remains, and a timeout has

value occurred.

If pdFALSE is returned then some block time remains, so a timeout has not occurred.

Example

/\* Driver library function used to receive uxWantedBytes from an Rx buffer that is filled by a UART interrupt. If there are not enough bytes in the Rx buffer then the task enters the Blocked state until it is notified that more data has been placed into the buffer. If there is still not enough data then the task re-enters the Blocked state, and xTaskCheckForTimeOut() is used to re-calculate the Block time to ensure the total amount of time spent in the Blocked state does not exceed MAX\_TIME\_TO\_WAIT. This continues until either the buffer contains at least uxWantedBytes bytes, or the total amount of time spent in the Blocked state reaches MAX\_TIME\_TO\_WAIT – at which point the task reads however many bytes are available up to a maximum of uxWantedBytes. \*/ size\_t xUART\_Receive( uint8\_t \*pucBuffer, size\_t uxWantedBytes )

{

size\_t uxReceived = 0;

TickType\_t xTicksToWait = MAX\_TIME\_TO\_WAIT;

TimeOut\_t xTimeOut;

/\* Initialize xTimeOut. This records the time at which this function was entered. \*/ vTaskSetTimeOutState( &xTimeOut );

/\* Loop until the buffer contains the wanted number of bytes, or a timeout occurs. \*/ while( UART\_bytes\_in\_rx\_buffer( pxUARTInstance ) < uxWantedBytes )

{

/\* The buffer didn’t contain enough data so this task is going to enter the Blocked state. Adjusting xTicksToWait to account for any time that has been spent in the Blocked state within this function so far to ensure the total amount of time spent in the Blocked state does not exceed MAX\_TIME\_TO\_WAIT. \*/ if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE ) {

/\* Timed out before the wanted number of bytes were available, exit the loop. \*/ break;

}

/\* Wait for a maximum of xTicksToWait ticks to be notified that the receive interrupt has placed more data into the buffer. \*/ ulTaskNotifyTake( pdTRUE, xTicksToWait ); }

/\* Attempt to read uxWantedBytes from the receive buffer into pucBuffer. The actual number of bytes read (which might be less than uxWantedBytes) is returned. \*/ uxReceived = UART\_read\_from\_receive\_buffer( pxUARTInstance, pucBuffer, uxWantedBytes );

return uxReceived;

}

Listing 9 Example use of vTaskSetTimeOutState() and xTaskCheckForTimeOut()

# 2.5 xTaskCreate()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskCreate( TaskFunction\_t pvTaskCode, const char \* const pcName, unsigned short usStackDepth, void \*pvParameters,

UBaseType\_t uxPriority,

TaskHandle\_t \*pvCreatedTask );

Listing 10 xTaskCreate() function prototype

Summary

Creates a new instance of a task.

Newly created tasks are initially placed in the Ready state, but will immediately become the Running state task if there are no higher priority tasks that are able to run.

Tasks can be created both before and after the scheduler has been started.

Parameters

|  |  |
| --- | --- |
| pvTaskCode | Tasks are simply C functions that never exit and, as such, are normally implemented as an infinite loop. The pvTaskCode parameter is simply a pointer to the function (in effect, just the function name) that implements the task. |
| pcName | A descriptive name for the task. This is not used by FreeRTOS in any way. It is included purely as a debugging aid. Identifying a task by a human readable name is much simpler than attempting to identify it by its handle.  The application-defined constant configMAX\_TASK\_NAME\_LEN defines the maximum length a task name can take – including the NULL terminator. Supplying a string longer than this maximum will result in the string being silently truncated. |
| usStackDepth | Each task has its own unique stack that is allocated by the kernel to the task |

when the task is created. The usStackDepth value tells the kernel how large to make the stack.

The value specifies the number of words the stack can hold, not the number of bytes. For example, on an architecture with a 4 byte stack width, if usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 \* 4 bytes). The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size\_t.

The size of the stack used by the idle task is defined by the applicationdefined constant configMINIMAL\_STACK\_SIZE. The value assigned to this constant in the demo application provided for the chosen microcontroller architecture is the minimum recommended for any task on that architecture.

If your task uses a lot of stack space, then you must assign a larger value.

|  |  |
| --- | --- |
| pvParameters | Task functions accept a parameter of type ‘pointer to void’ ( void\* ). The value assigned to pvParameters will be the value passed into the task.  This parameter has the type ‘pointer to void’ to allow the task parameter to effectively, and indirectly by means of casting, receive a parameter of any type. For example, integer types can be passed into a task function by casting the integer to a void pointer at the point the task is created, then by casting the void pointer parameter back to an integer in the task function definition itself. |
| uxPriority | Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX\_PRIORITIES – 1), which is the highest priority.  configMAX\_PRIORITIES is a user defined constant. There is no upper limit to the number of priorities that can be available (other than the limit of the data types used and the RAM available in your microcontroller), but you should use the lowest number of priorities required, to avoid wasting RAM.  Passing a uxPriority value above (configMAX\_PRIORITIES – 1) will result in the priority assigned to the task being capped silently to the maximum legitimate value. |
| pxCreatedTask | pxCreatedTask can be used to pass out a handle to the task being created. |

This handle can then be used to reference the task in API calls that, for

example, change the task priority or delete the task.

If your application has no use for the task handle, then pxCreatedTask can be set to NULL.

Return Values

pdPASS Indicates that the task has been

created successfully.

errCOULD\_NOT\_ALLOCATE\_REQUIRED\_MEMORY Indicates that the task could not be

created because there was insufficient heap memory available for FreeRTOS to allocate the task data structures and stack.

If heap\_1.c, heap\_2.c or heap\_4.c are included in the project then the total amount of heap available is defined by configTOTAL\_HEAP\_SIZE in FreeRTOSConfig.h, and failure to allocate memory can be trapped using the

vApplicationMallocFailedHook() callback (or ‘hook’) function, and the amount of free heap memory remaining can be queried using the xPortGetFreeHeapSize() API function.

If heap\_3.c is included in the project then the total heap size is defined by the linker configuration.

Example

/\* Define a structure called xStruct and a variable of type xStruct. These are just used to demonstrate a parameter being passed into a task function. \*/ typedef struct A\_STRUCT

{ char cStructMember1; char cStructMember2;

} xStruct;

/\* Define a variable of the type xStruct to pass as the task parameter. \*/ xStruct xParameter = { 1, 2 };

/\* Define the task that will be created. Note the name of the function that implements the task is used as the first parameter in the call to xTaskCreate() below. \*/ void vTaskCode( void \* pvParameters )

{

xStruct \*pxParameters;

/\* Cast the void \* parameter back to the required type. \*/ pxParameters = ( xStruct \* ) pvParameters;

/\* The parameter can now be accessed as expected. \*/ if( pxParameters->cStructMember1 != 1 )

{

/\* Etc. \*/

}

/\* Enter an infinite loop to perform the task processing. \*/ for( ;; )

{

/\* Task code goes here. \*/

}

}

/\* Define a function that creates a task. This could be called either before or after the scheduler has been started. \*/ void vAnotherFunction( void )

{

TaskHandle\_t xHandle;

/\* Create the task. \*/ if( xTaskCreate(

vTaskCode, /\* Pointer to the function that implements the task. \*/

"Demo task", /\* Text name given to the task. \*/

STACK\_SIZE, /\* The size of the stack that should be created for the task.

This is defined in words, not bytes. \*/

(void\*) &xParameter,/\* A reference to xParameters is used as the task parameter. This is cast to a void \* to prevent compiler warnings. \*/

TASK\_PRIORITY, /\* The priority to assign to the newly created task. \*/ &xHandle /\* The handle to the task being created will be placed in xHandle. \*/

) != pdPASS )

{

/\* The task could not be created as there was insufficient heap memory remaining. If heap\_1.c, heap\_2.c or heap\_4.c are included in the project then this situation can be trapped using the vApplicationMallocFailedHook() callback (or ‘hook’) function, and the amount of FreeRTOS heap memory that remains unallocated can be queried using the xPortGetFreeHeapSize() API function.\*/

} else

{

/\* The task was created successfully. The handle can now be used in other API functions, for example to change the priority of the task.\*/ vTaskPrioritySet( xHandle, 2 );

}

}

Listing 11 Example use of xTaskCreate()

# 2.6 xTaskCreateRestricted()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskCreateRestricted( TaskParameters\_t \*pxTaskDefinition,

TaskHandle\_t \*pxCreatedTask );

Listing 12 xTaskCreateRestricted() function prototype

Summary

This function is intended for advanced users only and is only relevant to FreeRTOS MPU ports (FreeRTOS ports that make use of a Memory Protection Unit).

Create a new Memory Protection Unit (MPU) restricted task.

Newly created tasks are initially placed in the Ready state, but will immediately become the Running state task if there are no higher priority tasks that are able to run.

Tasks can be created both before and after the scheduler has been started.

Parameters

|  |  |
| --- | --- |
| pxTaskDefinition | Pointer to a structure that defines the task. The structure is described under the notes heading in this section of the reference manual. |
| pxCreatedTask  Return Values | pxCreatedTask can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task.  If your application has no use for the task handle, then pxCreatedTask can be set to NULL. |
| pdPASS | Indicates that the task has been created successfully. |
| Any other value | Indicates that the task could not be created as specified, probably because |

there is insufficient FreeRTOS heap memory available to allocate the task data structures.

If heap\_1.c, heap\_2.c or heap\_4.c are included in the project then the total amount of heap available is defined by configTOTAL\_HEAP\_SIZE in FreeRTOSConfig.h, and failure to allocate memory can be trapped using the vApplicationMallocFailedHook() callback (or ‘hook’) function, and the amount of free heap memory remaining can be queried using the xPortGetFreeHeapSize() API function.

If heap\_3.c is included in the project then the total heap size is defined by the linker configuration.

Notes

xTaskCreateRestricted() makes use of the two data structures shown in Listing 13.

typedef struct xTASK\_PARAMTERS {

TaskFunction\_t pvTaskCode; const signed char \* const pcName; unsigned short usStackDepth; void \*pvParameters;

UBaseType\_t uxPriority; portSTACK\_TYPE \*puxStackBuffer;

MemoryRegion\_t xRegions[ portNUM\_CONFIGURABLE\_REGIONS ];

} TaskParameters\_t;

/\* ....where MemoryRegion\_t is defined as: \*/

typedef struct xMEMORY\_REGION

{

void \*pvBaseAddress; unsigned long ulLengthInBytes; unsigned long ulParameters;

} MemoryRegion\_t;

Listing 13 The data structures used by xTaskCreateRestricted()

A description of the Listing 13 structure members is given below.

|  |  |
| --- | --- |
| pvTaskCode  to uxPriority | These structure members are equivalent to the xTaskCreate() API function parameters that have the same names. |

Unlike standard FreeRTOS tasks, protected tasks can be created in either User (un-privileged) or Supervisor (privileged) modes, and the uxPriority structure member is used to control this option. To create a task in User mode, uxPriority is set to equal the priority at which the task is to be created. To create a task in Supervisor mode, uxPriority is set to equal the priority at which the task is to be created and to have its most significant bit set. The macro portPRIVILEGE\_BIT is provided for this purpose.

For example, to create a User mode task at priority three, set uxPriority to equal 3. To create a Supervisor mode task at priority three, set uxPriority to equal ( 3 | portPRIVILEGE\_BIT ).

puxStackBuffer The xTaskCreate() API function will automatically allocate a stack for use

by the task being created. The restrictions imposed by using an MPU means that the xTaskCreateRestricted() function cannot do the same, and instead, the stack used by the task being created must be statically allocated and passed into the xTaskCreateRestricted() function using the puxStackBuffer parameter.

Each time a restricted task is switched in (transitioned to the Running state) the MPU is dynamically re-configured to define an MPU region that provides the task read and write access to its own stack. Therefore, the statically allocated task stack must comply with the size and alignment restrictions imposed by the MPU. In particular, the size and alignment of each region must both be equal to the same power of two value.

Statically declaring a stack buffer allows the alignment to be managed using compiler extensions, and allows the linker to take care of stack placement, which it will do as efficiently as possible. For example, if using GCC, a stack can be declared and correctly aligned using the following syntax:

char cTaskStack[ 1024 ] \_\_attribute\_\_((align(1024));

Listing 14 Statically declaring a correctly aligned stack for use by a restricted task

MemoryRegion\_t An array of MemoryRegion\_t structures. Each MemoryRegion\_t structure

defines a single MPU memory region for use by the task being created.

The Cortex-M3 FreeRTOS-MPU port defines

portNUM\_CONFIGURABLE\_REGIONS to be 3. Three regions can be defined when the task is created. The regions can be redefined at run time using the vTaskAllocateMPURegions() function.

The pvBaseAddress and ulLengthInBytes members are self explanatory as the start of the memory region and the length of the memory region respectively. ulParameters defines how the task is permitted to access the memory region being defined, and can take the bitwise OR of the following values:

* portMPU\_REGION\_READ\_WRITE
* portMPU\_REGION\_PRIVILEGED\_READ\_ONLY
* portMPU\_REGION\_READ\_ONLY
* portMPU\_REGION\_PRIVILEGED\_READ\_WRITE
* portMPU\_REGION\_CACHEABLE\_BUFFERABLE
* portMPU\_REGION\_EXECUTE\_NEVER

Example

/\* Declare the stack that will be used by the protected task being created. The stack alignment must match its size, and be a power of 2. So, if 128 words are reserved for the stack then it must be aligned on a ( 128 \* 4 ) byte boundary. This example uses GCC syntax. \*/ static portSTACK\_TYPE xTaskStack[ 128 ] \_\_attribute\_\_((aligned(128\*4)));

/\* Declare an array that will be accessed by the protected task being created. The task should only be able to read from the array, and not write to it. \*/ char cReadOnlyArray[ 512 ] \_\_attribute\_\_((aligned(512)));

/\* Fill in a TaskParameters\_t structure to define the task - this is the structure passed to the xTaskCreateRestricted() function. \*/

static const TaskParameters\_t xTaskDefinition =

{ vTaskFunction, /\* pvTaskCode \*/

"A task", /\* pcName \*/

128, /\* usStackDepth - defined in words, not bytes. \*/

NULL, /\* pvParameters \*/

1, /\* uxPriority - priority 1, start in User mode. \*/ xTaskStack, /\* puxStackBuffer - the array to use as the task stack. \*/

/\* xRegions - In this case only one of the three user definable regions is actually used.

The parameters are used to set the region to read only. \*/

{

/\* Base address Length Parameters \*/

{ cReadOnlyArray, 512, portMPU\_REGION\_READ\_ONLY }, { 0, 0, 0 },

{ 0, 0, 0 },

}

};

void main( void ) {

/\* Create the task defined by xTaskDefinition. NULL is used as the second parameter as a task handle is not required. \*/

xTaskCreateRestricted( &xTaskDefinition, NULL );

/\* Start the scheduler. \*/ vTaskStartScheduler();

/\* Should not reach here! \*/

}

Listing 15 Example use of xTaskCreateRestricted()

# 2.7 vTaskDelay()

#include “FreeRTOS.h” #include “task.h”

void vTaskDelay( TickType\_t xTicksToDelay );

Listing 16 vTaskDelay() function prototype

Summary

Places the task that calls vTaskDelay() into the Blocked state for a fixed number of tick interrupts.

Specifying a delay period of zero ticks will not result in the calling task being placed into the Blocked state, but will result in the calling task yielding to any Ready state tasks that share its priority. Calling vTaskDelay( 0 ) is equivalent to calling taskYIELD().

Parameters

xTicksToDelay The number of tick interrupts that the calling task will remain in the Blocked

state before being transitioned back into the Ready state. For example, if a task called vTaskDelay( 100 ) when the tick count was 10,000, then it would immediately enter the Blocked state and remain in the Blocked state until the tick count reached 10,100.

Any time that remains between vTaskDelay() being called, and the next tick interrupt occurring, counts as one complete tick period. Therefore, the highest time resolution that can be achieved when specifying a delay period is, in the worst case, equal to one complete tick interrupt period.

The macro pdMS\_TO\_TICKS() can be used to convert milliseconds into ticks.

This is demonstrated in the example in this section.

Return Values

None.

Notes

INCLUDE\_vTaskDelay must be set to 1 in FreeRTOSConfig.h for the vTaskDelay() API function to be available.

Example

void vAnotherTask( void \* pvParameters )

{

for( ;; )

{

/\* Perform some processing here. \*/

…

/\* Enter the Blocked state for 20 tick interrupts – the actual time spent in the Blocked state is dependent on the tick frequency. \*/ vTaskDelay( 20 );

/\* 20 ticks will have passed since the first call to vTaskDelay() was executed. \*/

/\* Enter the Blocked state for 20 milliseconds. Using the pdMS\_TO\_TICKS() macro means the tick frequency can change without effecting the time spent in the blocked state (other than due to the resolution of the tick frequency). \*/ vTaskDelay( pdMS\_TO\_TICKS( 20 ) );

}

}

Listing 17 Example use of vTaskDelay()

# 2.8 vTaskDelayUntil()

#include “FreeRTOS.h” #include “task.h”

void vTaskDelayUntil( TickType\_t \*pxPreviousWakeTime, TickType\_t xTimeIncrement );

Listing 18 vTaskDelayUntil() function prototype

Summary

Places the task that calls vTaskDelayUntil() into the Blocked state until an absolute time is reached.

Periodic tasks can use vTaskDelayUntil() to achieve a constant execution frequency.

Differences Between vTaskDelay() and vTaskDelayUntil()

vTaskDelay() results in the calling task entering into the Blocked state, and then remaining in the Blocked state, for the specified number of ticks from the time vTaskDelay() was called. The time at which the task that called vTaskDelay() exits the Blocked state is relative to when vTaskDelay() was called.

vTaskDelayUntil() results in the calling task entering into the Blocked state, and then remaining in the Blocked state, until an absolute time has been reached. The task that called vTaskDelayUntil() exits the Blocked state exactly at the specified time, not at a time that is relative to when vTaskDelayUntil() was called.

Parameters pxPreviousWakeTime This parameter is named on the assumption that vTaskDelayUntil() is

being used to implement a task that executes periodically and with a fixed frequency. In this case pxPreviousWakeTime holds the time at which the task last left the Blocked state (was ‘woken’ up). This time is used as a reference point to calculate the time at which the task should next leave the Blocked state.

The variable pointed to by pxPreviousWakeTime is updated automatically within the vTaskDelayUntil() function; it would not normally be modified by the application code, other than when the variable is first initialized. The example in this section demonstrates how the initialization is performed.

xTimeIncrement This parameter is also named on the assumption that

vTaskDelayUntil() is being used to implement a task that executes periodically and with a fixed frequency – the frequency being set by the xTimeIncrement value.

xTimeIncrement is specified in ‘ticks’. The pdMS\_TO\_TICKS() macro can be used to convert milliseconds to ticks.

Return Values None.

Notes

INCLUDE\_vTaskDelayUntil must be set to 1 in FreeRTOSConfig.h for the vTaskDelay() API function to be available.

Example

/\* Define a task that performs an action every 50 milliseconds. \*/ void vCyclicTaskFunction( void \* pvParameters )

{

TickType\_t xLastWakeTime;

const TickType\_t xPeriod = pdMS\_TO\_TICKS( 50 );

/\* The xLastWakeTime variable needs to be initialized with the current tick count. Note that this is the only time the variable is written to explicitly. After this assignment, xLastWakeTime is updated automatically internally within vTaskDelayUntil(). \*/

xLastWakeTime = xTaskGetTickCount();

/\* Enter the loop that defines the task behavior. \*/ for( ;; )

{

/\* This task should execute every 50 milliseconds. Time is measured in ticks. The pdMS\_TO\_TICKS macro is used to convert milliseconds into ticks. xLastWakeTime is automatically updated within vTaskDelayUntil() so is not explicitly updated by the task. \*/ vTaskDelayUntil( &xLastWakeTime, xPeriod );

/\* Perform the periodic actions here. \*/

}

}

Listing 19 Example use of vTaskDelayUntil()

# 2.9 vTaskDelete()

#include “FreeRTOS.h” #include “task.h”

void vTaskDelete( TaskHandle\_t pxTask );

Listing 20 vTaskDelete() function prototype

Summary

Deletes an instance of a task that was previously created using a call to xTaskCreate().

Deleted tasks no longer exist so cannot enter the Running state.

Do not attempt to use a task handle to reference a task that has been deleted.

When a task is deleted, it is the responsibility of the idle task to free the memory that had been used to hold the deleted task’s stack and data structures (task control block). Therefore, if an application makes use of the vTaskDelete() API function, it is vital that the application also ensures the idle task is not starved of processing time (the idle task must be allocated time in the Running state).

Only memory that is allocated to a task by the kernel itself is automatically freed when a task is deleted. Memory, or any other resource, that the application (rather than the kernel) allocates to a task must be explicitly freed by the application when the task is deleted.

Parameters

pxTask The handle of the task being deleted (the subject task) – see the pxCreatedTask

parameter of the xTaskCreate() API function for information on obtaining handles to tasks.

A task can delete itself by passing NULL in place of a valid task handle.

Return Values

None.

Example

void vAnotherFunction( void )

{

TaskHandle\_t xHandle;

/\* Create a task, storing the handle to the created task in xHandle. \*/ if(

xTaskCreate( vTaskCode, "Demo task",

STACK\_SIZE,

NULL,

PRIORITY,

&xHandle /\* The address of xHandle is passed in as the last parameter to xTaskCreate() to obtain a handle to the task being created. \*/

)

!= pdPASS )

{

/\* The task could not be created because there was not enough FreeRTOS heap memory available for the task data structures and stack to be allocated. \*/

} else

{

/\* Delete the task just created. Use the handle passed out of xTaskCreate() to reference the subject task. \*/ vTaskDelete( xHandle );

}

/\* Delete the task that called this function by passing NULL in as the vTaskDelete() parameter. The same task (this task) could also be deleted by passing in a valid handle to itself. \*/ vTaskDelete( NULL );

}

Listing 21 Example use of the vTaskDelete()

# 2.10 taskDISABLE\_INTERRUPTS()

#include “FreeRTOS.h” #include “task.h”

void taskDISABLE\_INTERRUPTS( void );

Listing 22 taskDISABLE\_INTERRUPTS() macro prototype

Summary

If the FreeRTOS port being used does not make use of the configMAX\_SYSCALL\_INTERRUPT\_PRIORITY (or configMAX\_API\_CALL\_INTERRUPT\_PRIORITY, depending on the port) kernel configuration constant, then calling taskDISABLE\_INTERRUPTS() will leave interrupts globally disabled.

If the FreeRTOS port being used does make use of the configMAX\_SYSCALL\_INTERRUPT\_PRIORITY kernel configuration constant, then calling taskDISABLE\_INTERRUPTS() will leave interrupts at and below the interrupt priority set by configMAX\_SYSCALL\_INTERRUPT\_PRIORITY disabled, and all higher priority interrupt enabled. configMAX\_SYSCALL\_INTERRUPT\_PRIORITY is normally defined in FreeRTOSConfig.h.

Calls to taskDISABLE\_INTERRUPTS() and taskENABLE\_INTERRUPTS() are not designed to nest. For example, if taskDISABLE\_INTERRUPTS() is called twice, a single call to taskENABLE\_INTERRUPTS() will still result in interrupts becoming enabled. If nesting is required then use taskENTER\_CRITICAL() and taskEXIT\_CRITICAL() in place of taskDISABLE\_INTERRUPTS() and taskENABLE\_INTERRUPTS() respectively.

Some FreeRTOS API functions use critical sections that will re-enable interrupts if the critical section nesting count is zero – even if interrupts were disabled by a call to taskDISABLE\_INTERRUPTS() before the API function was called. It is not recommended to call FreeRTOS API functions when interrupts have already been disabled.

Parameters

None.

Return Values None.

# 2.11 taskENABLE\_INTERRUPTS()

#include “FreeRTOS.h” #include “task.h”

void taskENABLE\_INTERRUPTS( void );

Listing 23 taskENABLE\_INTERRUPTS() macro prototype

Summary

Calling taskENABLE\_INTERRUPTS() will result in all interrupt priorities being enabled.

Calls to taskDISABLE\_INTERRUPTS() and taskENABLE\_INTERRUPTS() are not designed to nest. For example, if taskDISABLE\_INTERRUPTS() is called twice a single call to taskENABLE\_INTERRUPTS() will still result in interrupts becoming enabled. If nesting is required then use taskENTER\_CRITICAL() and taskEXIT\_CRITICAL() in place of taskDISABLE\_INTERRUPTS() and taskENABLE\_INTERRUPTS() respectively.

Some FreeRTOS API functions use critical sections that will re-enable interrupts if the critical section nesting count is zero – even if interrupts were disabled by a call to taskDISABLE\_INTERRUPTS() before the API function was called. It is not recommended to call FreeRTOS API functions when interrupts have already been disabled.

Parameters None.

Return Values None.

# 2.12 taskENTER\_CRITICAL()

#include “FreeRTOS.h” #include “task.h”

void taskENTER\_CRITICAL( void );

Listing 24 taskENTER\_CRITICAL macro prototype

Summary

Critical sections are entered by calling taskENTER\_CRITICAL(), and subsequently exited by calling taskEXIT\_CRITICAL().

The taskENTER\_CRITICAL() and taskEXIT\_CRITICAL() macros provide a basic critical section implementation that works by simply disabling interrupts, either globally or up to a specific interrupt priority level.

If the FreeRTOS port being used does not make use of the configMAX\_SYSCALL\_INTERRUPT\_PRIORITY kernel configuration constant, then calling taskENTER\_CRITICAL() will leave interrupts globally disabled.

If the FreeRTOS port being used does make use of the configMAX\_SYSCALL\_INTERRUPT\_PRIORITY (or configMAX\_API\_CALL\_INTERRUPT\_PRIORITY, depending on the port) kernel configuration constant, then calling taskENTER\_CRITICAL() will leave interrupts at and below the interrupt priority set by configMAX\_SYSCALL\_INTERRUPT\_PRIORITY disabled, and all higher priority interrupt enabled.

Preemptive context switches only occur inside an interrupt, so will not occur when interrupts are disabled. Therefore, the task that called taskENTER\_CRITICAL() is guaranteed to remain in the Running state until the critical section is exited, unless the task explicitly attempts to block or yield (which it should not do from inside a critical section).

Calls to taskENTER\_CRITICAL() and taskEXIT\_CRITICAL() are designed to nest. Therefore, a critical section will only be exited when one call to taskEXIT\_CRITICAL() has been executed for every preceding call to taskENTER\_CRITICAL().

Critical sections must be kept very short, otherwise they will adversely affect interrupt response times. Every call to taskENTER\_CRITICAL() must be closely paired with a call to taskEXIT\_CRITICAL().

FreeRTOS API functions must not be called from within a critical section.

Parameters None.

Return Values

None.

Example

/\* A function that makes use of a critical section. \*/ void vDemoFunction( void )

{

/\* Enter the critical section – in this example, this function is itself called from within a critical section, so entering this critical section will result in a nesting depth of 2. \*/ taskENTER\_CRITICAL();

/\* Perform the action that is being protected by the critical section here. \*/

/\* Exit the critical section – in this example, this function is itself called from a critical section, so this call to taskEXIT\_CRITICAL() will decrement the nesting count by one, but not result in interrupts becoming enabled. \*/ taskEXIT\_CRITICAL();

}

/\* A task that calls vDemoFunction() from within a critical section. \*/ void vTask1( void \* pvParameters )

{

for( ;; )

{

/\* Perform some functionality here. \*/

/\* Call taskENTER\_CRITICAL() to create a critical section. \*/ taskENTER\_CRITICAL();

/\* Execute the code that requires the critical section here. \*/

/\* Calls to taskENTER\_CRITICAL() can be nested so it is safe to call a function that includes its own calls to taskENTER\_CRITICAL() and taskEXIT\_CRITICAL(). \*/ vDemoFunction();

/\* The operation that required the critical section is complete so exit the critical section. After this call to taskEXIT\_CRITICAL(), the nesting depth will be zero, so interrupts will have been re-enabled. \*/ taskEXIT\_CRITICAL();

}

}

Listing 25 Example use of taskENTER\_CRITICAL() and taskEXIT\_CRITICAL()

# 2.13 taskEXIT\_CRITICAL()

#include “FreeRTOS.h” #include “task.h”

void taskEXIT\_CRITICAL( void );

Listing 26 taskEXIT\_CRITICAL() macro prototype

Summary

Critical sections are entered by calling taskENTER\_CRITICAL(), and subsequently exited by calling taskEXIT\_CRITICAL().

The taskENTER\_CRITICAL() and taskEXIT\_CRITICAL() macros provide a basic critical section implementation that works by simply disabling interrupts, either globally or up to a specific interrupt priority level.

If the FreeRTOS port being used does not make use of the configMAX\_SYSCALL\_INTERRUPT\_PRIORITY kernel configuration constant, then calling taskENTER\_CRITICAL() will leave interrupts globally disabled.

If the FreeRTOS port being used does make use of the configMAX\_SYSCALL\_INTERRUPT\_PRIORITY kernel configuration constant, then calling taskENTER\_CRITICAL() will leave interrupts at and below the interrupt priority set by configMAX\_SYSCALL\_INTERRUPT\_PRIORITY disabled, and all higher priority interrupt enabled.

Preemptive context switches only occur inside an interrupt, so will not occur when interrupts are disabled. Therefore, the task that called taskENTER\_CRITICAL() is guaranteed to remain in the Running state until the critical section is exited, unless the task explicitly attempts to block or yield (which it should not do from inside a critical section).

Calls to taskENTER\_CRITICAL() and taskEXIT\_CRITICAL() are designed to nest. Therefore, a critical section will only be exited when one call to taskEXIT\_CRITICAL() has been executed for every preceding call to taskENTER\_CRITICAL().

Critical sections must be kept very short otherwise they will adversely affect interrupt response times. Every call to taskENTER\_CRITICAL() must be closely paired with a call to taskEXIT\_CRITICAL().

FreeRTOS API functions must not be called from within a critical section.

Parameters None.

Return Values None.

Example

See Listing 25.

# 2.14 xTaskGetApplicationTaskTag()

#include “FreeRTOS.h” #include “task.h”

TaskHookFunction\_t xTaskGetApplicationTaskTag( TaskHandle\_t xTask );

Listing 27 xTaskGetApplicationTaskTag() function prototype

Summary

Returns the ‘tag’ value associated with a task. The meaning and use of the tag value is defined by the application writer. The kernel itself will not normally access the tag value.

This function is intended for advanced users only.

Parameters

xTask The handle of the task being queried. This is the subject task.

A task can obtain its own tag value by either using its own task handle, or by using NULL in place of a valid task handle.

Return Values

The ‘tag’ value of the task being queried.

Notes

The tag value can be used to hold a function pointer. When this is done the function assigned to the tag value can be called using the xTaskCallApplicationTaskHook() API function. This technique is in effect assigning a callback function to the task. It is common for such a callback to be used in combination with the traceTASK\_SWITCHED\_IN() macro to implement an execution trace feature.

configUSE\_APPLICATION\_TASK\_TAG must be set to 1 in FreeRTOSConfig.h for xTaskGetApplicationTaskTag() to be available.

Example

/\* In this example, an integer is set as the task tag value. \*/ void vATask( void \*pvParameters )

{

/\* Assign a tag value of 1 to the currently executing task. The (void \*) cast is used to prevent compiler warnings. \*/ vTaskSetApplicationTaskTag( NULL, ( void \* ) 1 );

for( ;; )

{

/\* Rest of task code goes here. \*/

}

}

void vAFunction( void )

{

TaskHandle\_t xHandle; long lReturnedTaskHandle;

/\* Create a task from the vATask() function, storing the handle to the created task in the xTask variable. \*/

/\* Create the task. \*/ if( xTaskCreate(

vATask, /\* Pointer to the function that implements the task. \*/

"Demo task", /\* Text name given to the task. \*/

STACK\_SIZE, /\* The size of the stack that should be created for the task.

This is defined in words, not bytes. \*/

NULL, /\* The task does not use the parameter. \*/

TASK\_PRIORITY, /\* The priority to assign to the newly created task. \*/ &xHandle /\* The handle to the task being created will be placed in xHandle. \*/

) == pdPASS )

{

/\* The task was created successfully. Delay for a short period to allow the task to run. \*/ vTaskDelay( 100 );

/\* What tag value is assigned to the task? The returned tag value is stored in an integer, so cast to an integer to prevent compiler warnings. \*/ lReturnedTaskHandle = ( long ) xTaskGetApplicationTaskTag( xHandle ); }

}

Listing 28 Example use of xTaskGetApplicationTaskTag()

# 2.15 xTaskGetCurrentTaskHandle()

#include “FreeRTOS.h” #include “task.h”

TaskHandle\_t xTaskGetCurrentTaskHandle( void );

Listing 29 xTaskGetCurrentTaskHandle() function prototype

Summary

Returns the handle of the task that is in the Running state – which will be the handle of the task that called xTaskGetCurrentTaskHandle().

Parameters None.

Return Values

The handle of the task that called xTaskGetCurrentTaskHandle().

Notes

INCLUDE\_xTaskGetCurrentTaskHandle must be set to 1 in FreeRTOSConfig.h for xTaskGetCurrentTaskHandle() to be available.

# 2.16 xTaskGetIdleTaskHandle()

#include “FreeRTOS.h” #include “task.h”

TaskHandle\_t xTaskGetIdleTaskHandle( void );

Listing 30 xTaskGetIdleTaskHandle() function prototype

Summary

Returns the task handle associated with the Idle task. The Idle task is created automatically when the scheduler is started.

Parameters None.

Return Values

The handle of the Idle task.

Notes

INCLUDE\_xTaskGetIdleTaskHandle must be set to 1 in FreeRTOSConfig.h for xTaskGetIdleTaskHandle() to be available.

# 2.17 uxTaskGetNumberOfTasks()

#include “FreeRTOS.h” #include “task.h”

UBaseType\_t uxTaskGetNumberOfTasks( void );

Listing 31 uxTaskGetNumberOfTasks() function prototype

Summary

Returns the total number of tasks that exist at the time uxTaskGetNumberOfTasks() is called.

Parameters None.

Return Values

The value returned is the total number of tasks that are under the control of the FreeRTOS kernel at the time uxTaskGetNumberOfTasks() is called. This is the number of Suspended state tasks, plus the number of Blocked state tasks, plus the number of Ready state tasks, plus the idle task, plus the Running state task.

# 2.18 vTaskGetRunTimeStats()

#include “FreeRTOS.h” #include “task.h”

void vTaskGetRunTimeStats( char \*pcWriteBuffer );

Listing 32 vTaskGetRunTimeStats() function prototype

Summary

FreeRTOS can be configured to collect task run time statistics. Task run time statistics provide information on the amount of processing time each task has received. Figures are provided as both an absolute time and a percentage of the total application run time. The vTaskGetRunTimeStats() API function formats the collected run time statistics into a human readable table. Columns are generated for the task name, the absolute time allocated to that task, and the percentage of the total application run time allocated to that task. A row is generated for each task in the system, including the Idle task. An example output is shown in Figure 1.



Figure 1 An example of the table produced by calling vTaskGetRunTimeStats()

Parameters

pcWriteBuffer A pointer to a character buffer into which the formatted and human readable

table is written. The buffer must be large enough to hold the entire table, as no boundary checking is performed.

Return Values None.

Notes

vTaskGetRunTimeStats() is a utility function that is provided for convenience only. It is not considered part of the kernel. vTaskGetRunTimeStats() obtains its raw data using the xTaskGetSystemState() API function.

configGENERATE\_RUN\_TIME\_STATS and configUSE\_STATS\_FORMATTING\_FUNCTIONS must both be set to 1 in FreeRTOSConfig.h for vTaskGetRunTimeStats() to be available. Setting configGENERATE\_RUN\_TIME\_STATS will also require the application to define the following macros:

portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() This macro must be provided to

initialize whichever peripheral is used to generate the time base. The time base used by the run time stats must have a higher resolution than the tick interrupt, otherwise the gathered statistics may be too inaccurate to be truly useful. It is recommended to make the time base between 10 and 20 times faster than the tick interrupt portGET\_RUN\_TIME\_COUNTER\_VALUE(), or One of these two macros must be

provided to return the current time portALT\_GET\_RUN\_TIME\_COUNTER\_VALUE(Time)

base value – which is the total time that the application has been running in the chosen time base units. If the first macro is used it must be defined to evaluate to the current time base value. If the second macro is used it must be defined to set its ‘Time’ parameter to the current time base value.

These macros can be defined in FreeRTOSConfig.h.

Example

/\* The LM3Sxxxx Eclipse demo application already includes a 20KHz timer interrupt.

The interrupt handler was updated to simply increment a variable called ulHighFrequencyTimerTicks each time it executed.

portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() then sets this variable to 0 and portGET\_RUN\_TIME\_COUNTER\_VALUE() returns its value. To implement this the following few lines are added to FreeRTOSConfig.h. \*/

extern volatile unsigned long ulHighFrequencyTimerTicks;

/\* ulHighFrequencyTimerTicks is already being incremented at 20KHz. Just set its value back to 0. \*/

#define portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() ( ulHighFrequencyTimerTicks = 0UL )

/\* Simply return the high frequency counter value. \*/

#define portGET\_RUN\_TIME\_COUNTER\_VALUE() ulHighFrequencyTimerTicks

Listing 33 Example macro definitions, taken from the LM3Sxxx Eclipse Demo

/\* The LPC17xx demo application does not include the high frequency interrupt test, so portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() is used to configure the timer 0 peripheral to generate the time base. portGET\_RUN\_TIME\_COUNTER\_VALUE() simply returns the current timer 0 counter value. This was implemented using the following functions and macros. \*/

/\* Defined in main.c. \*/

void vConfigureTimerForRunTimeStats( void )

{

const unsigned long TCR\_COUNT\_RESET = 2, CTCR\_CTM\_TIMER = 0x00,

TCR\_COUNT\_ENABLE = 0x01;

/\* Power up and feed the timer with a clock. \*/

PCONP |= 0x02UL;

PCLKSEL0 = (PCLKSEL0 & (~(0x3<<2))) | (0x01 << 2);

/\* Reset Timer 0 \*/

T0TCR = TCR\_COUNT\_RESET;

/\* Just count up. \*/

T0CTCR = CTCR\_CTM\_TIMER;

/\* Prescale to a frequency that is good enough to get a decent resolution, but not too fast so as to overflow all the time. \*/

T0PR = ( configCPU\_CLOCK\_HZ / 10000UL ) - 1UL;

/\* Start the counter. \*/

T0TCR = TCR\_COUNT\_ENABLE;

}

/\* Defined in FreeRTOSConfig.h. \*/

extern void vConfigureTimerForRunTimeStats( void );

#define portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() vConfigureTimerForRunTimeStats()

#define portGET\_RUN\_TIME\_COUNTER\_VALUE() T0TC

Listing 34 Example macro definitions, taken from the LPC17xx Eclipse Demo

void vAFunction( void )

{

/\* Define a buffer that is large enough to hold the generated table. In most cases the buffer will be too large to allocate on the stack, hence in this example it is declared static. \*/

static char cBuffer[ BUFFER\_SIZE ];

/\* Pass the buffer into vTaskGetRunTimeStats() to generate the table of data. \*/ vTaskGetRunTimeStats( cBuffer );

/\* The generated information can be saved or viewed here. \*/

}

Listing 35 Example use of vTaskGetRunTimeStats()

# 2.19 xTaskGetSchedulerState()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskGetSchedulerState( void );

Listing 36 xTaskGetSchedulerState() function prototype

Summary

Returns a value that indicates the state the scheduler is in at the time xTaskGetSchedulerState() is called.

Parameters None.

Return Values

|  |  |
| --- | --- |
| taskSCHEDULER\_NOT\_STARTED This value will only be returned when | |
|  | xTaskGetSchedulerState() is called before vTaskStartScheduler() has been called. |
| taskSCHEDULER\_RUNNING | Returned if vTaskStartScheduler() has already been called, provided the scheduler is not in the Suspended state. |
| taskSCHEDULER\_SUSPENDED | Returned when the scheduler is in the Suspended sate |

because vTaskSuspendAll() was called.

Notes

INCLUDE\_xTaskGetSchedulerState must be set to 1 in FreeRTOSConfig.h for xTaskGetSchedulerState() to be available.

# 2.20 uxTaskGetStackHighWaterMark()

#include “FreeRTOS.h” #include “task.h”

UBaseType\_t uxTaskGetStackHighWaterMark( TaskHandle\_t xTask );

Summary

Each task maintains its own stack, the total size of which is specified when the task is created. uxTaskGetStackHighWaterMark() is used to query how close a task has come to overflowing the stack space allocated to it. This value is called the stack 'high water mark'.

Parameters

xTask The handle of the task whose stack high water mark is being queried (the subject task). See the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks.

A task can query its own stack high water mark by passing NULL in place of a valid task handle.

Return Values

The amount of stack used by a task grows and shrinks as the task executes and interrupts are processed. uxTaskGetStackHighWaterMark() returns the minimum amount of remaining stack space that has been available since the task started executing. This is the amount of stack that remained unused when stack usage was at its greatest (or deepest) value. The closer the high water mark is to zero, the closer the task has come to overflowing its stack.

Notes uxTaskGetStackHighWaterMark() can take a relatively long time to execute. It is therefore recommended that its use is limited to test and debug builds.

INCLUDE\_uxTaskGetStackHighWaterMark must be set to 1 in FreeRTOSConfig.h for uxTaskGetStackHighWaterMark() to be available.

Example

void vTask1( void \* pvParameters )

{

UBaseType\_t uxHighWaterMark;

/\* Inspect the high water mark of the calling task when the task starts to execute. \*/

uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );

for( ;; )

{

/\* Call any function. \*/ vTaskDelay( 1000 );

/\* Calling a function will have used some stack space, so it will be expected that uxTaskGetStackHighWaterMark() will return a lower value at this point than when it was called on entry to the task function. \*/ uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );

}

}

Listing 37 Example use of uxTaskGetStackHighWaterMark()

# 2.21 eTaskGetState()

#include “FreeRTOS.h” #include “task.h”

eTaskState eTaskGetState( TaskHandle\_t pxTask );

Listing 38 eTaskGetState() function prototype

Summary

Returns as an enumerated type the state in which a task existed at the time eTaskGetState() was executed.

Parameters

pxTask The handle of the subject task.

The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function.

Return Values

Table 1 lists the value that eTaskGetState() will return for each possible state that the task referenced by the pxTask parameter can exist in.

Table 1. eTaskGetState() return values

|  |  |  |  |
| --- | --- | --- | --- |
| State | | Return Value | |
| Running | eRunning (the task is querying its own state) |
| Ready | eReady |
| Blocked | eBlocked |
| Suspended | eSuspended |
| Deleted | eDeleted (the task’s structures are waiting to be cleaned up) |

Notes

INCLUDE\_eTaskGetState must be set to 1 in FreeRTOSConfig.h for the eTaskGetState() API function to be available.

# 2.22 uxTaskGetSystemState()

#include “FreeRTOS.h” #include “task.h”

UBaseType\_t uxTaskGetSystemState( TaskStatus\_t \* const pxTaskStatusArray, const UBaseType\_t uxArraySize, unsigned long \* const pulTotalRunTime );

Listing 39 uxTaskGetSystemState() function prototype

Summary

uxTaskGetSystemState() populates an TaskStatus\_t structure for each task in the system. TaskStatus\_t structures contain, among other things, members for the task handle, task name, task priority, task state, and total amount of run time consumed by the task.

The TaskStatus\_t structure is defined in Listing 41.

Parameters

|  |  |
| --- | --- |
| pxTaskStatusArray A pointer to an array of TaskStatus\_t structures. The array must contain | |
|  | at least one TaskStatus\_t structure for each task that is under the control of the RTOS. The number of tasks under the control of the RTOS can be determined using the uxTaskGetNumberOfTasks() API function. |
| uxArraySize | The size of the array pointed to by the pxTaskStatusArray parameter. The size is specified as the number of indexes in the array (the number of TaskStatus\_t structures contained in the array), not by the number of bytes in the array. |
| pulTotalRunTime | If configGENERATE\_RUN\_TIME\_STATS is set to 1 in |

FreeRTOSConfig.h then \*pulTotalRunTime is set by

uxTaskGetSystemState() to the total run time (as defined by the run time stats clock) since the target booted. pulTotalRunTime can be set to NULL to omit the total run time value.

Return Values

The number of TaskStatus\_t structures that were populated by uxTaskGetSystemState(). This should equal the number returned by the uxTaskGetNumberOfTasks() API function, but will be zero if the value passed in the uxArraySize parameter was too small.

Notes

This function is intended for debugging use only as its use results in the scheduler remaining suspended for an extended period.

configUSE\_TRACE\_FACILITY must bet defined as 1 in FreeRTOSConfig.h for uxTaskGetSystemState() to be available.

Example

/\* This example demonstrates how a human readable table of run time stats information is generated from raw data provided by uxTaskGetSystemState(). The human readable table is written to pcWriteBuffer. (see the vTaskList() API function which actually does just this). \*/ void vTaskGetRunTimeStats( signed char \*pcWriteBuffer )

{

TaskStatus\_t \*pxTaskStatusArray; volatile UBaseType\_t uxArraySize, x;

unsigned long ulTotalRunTime, ulStatsAsPercentage;

/\* Make sure the write buffer does not contain a string. \*/

\*pcWriteBuffer = 0x00;

/\* Take a snapshot of the number of tasks in case it changes while this function is executing. \*/

uxArraySize = uxTaskGetNumberOfTasks();

/\* Allocate a TaskStatus\_t structure for each task. An array could be allocated statically at compile time. \*/

pxTaskStatusArray = pvPortMalloc( uxArraySize \* sizeof( TaskStatus\_t ) );

if( pxTaskStatusArray != NULL )

{

/\* Generate raw status information about each task. \*/

uxArraySize = uxTaskGetSystemState( pxTaskStatusArray, uxArraySize, &ulTotalRunTime );

/\* For percentage calculations. \*/ ulTotalRunTime /= 100UL;

/\* Avoid divide by zero errors. \*/ if( ulTotalRunTime > 0 )

{

/\* For each populated position in the pxTaskStatusArray array, format the raw data as human readable ASCII data. \*/ for( x = 0; x < uxArraySize; x++ )

{

/\* What percentage of the total run time has the task used? This will always be rounded down to the nearest integer. ulTotalRunTimeDiv100 has already been divided by 100. \*/

ulStatsAsPercentage = pxTaskStatusArray[ x ].ulRunTimeCounter / ulTotalRunTime;

if( ulStatsAsPercentage > 0UL )

{

sprintf( pcWriteBuffer, "%s\t\t%lu\t\t%lu%%\r\n", pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter, ulStatsAsPercentage );

} else

{

/\* If the percentage is zero here then the task has consumed less than 1% of the total run time. \*/ sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%%\r\n", pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter ); }

pcWriteBuffer += strlen( ( char \* ) pcWriteBuffer ); }

}

/\* The array is no longer needed, free the memory it consumes. \*/ vPortFree( pxTaskStatusArray );

}

}

Listing 40 Example use of uxTaskGetSystemState()

typedef struct xTASK\_STATUS

{

/\* The handle of the task to which the rest of the information in the structure relates. \*/

TaskHandle\_t xHandle;

/\* A pointer to the task's name. This value will be invalid if the task was deleted since the structure was populated! \*/ const signed char \*pcTaskName;

/\* A number unique to the task. \*/

UBaseType\_t xTaskNumber;

/\* The state in which the task existed when the structure was populated. \*/ eTaskState eCurrentState;

/\* The priority at which the task was running (may be inherited) when the structure was populated. \*/

UBaseType\_t uxCurrentPriority;

/\* The priority to which the task will return if the task's current priority has been inherited to avoid unbounded priority inversion when obtaining a mutex. Only valid

if configUSE\_MUTEXES is defined as 1 in FreeRTOSConfig.h. \*/

UBaseType\_t uxBasePriority;

/\* The total run time allocated to the task so far, as defined by the run time stats

clock. Only valid when configGENERATE\_RUN\_TIME\_STATS is defined as 1 in FreeRTOSConfig.h. \*/

unsigned long ulRunTimeCounter;

/\* The minimum amount of stack space that has remained for the task since the task was created. The closer this value is to zero the closer the task has come to overflowing its stack. \*/

unsigned short usStackHighWaterMark;

} TaskStatus\_t;

Listing 41 The TaskStatus\_t definition

# 2.23 pvTaskGetThreadLocalStoragePointer()

#include “FreeRTOS.h” #include “task.h”

void \*pvTaskGetThreadLocalStoragePointer( TaskHandle\_t xTaskToQuery,

BaseType\_t xIndex );

Listing 42 pvTaskGetThreadLocalStoragePointer() function prototype

Summary

Thread local storage (or TLS) allows the application writer to store values inside a task's control block, making the value specific to (local to) the task itself, and allowing each task to have its own unique value.

Each task has its own array of pointers that can be used as thread local storage. The number of indexes in the array is set by the configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS compile time configuration constant in FreeRTOSConfig.h.

pvTaskGetThreadLocalStoragePointer() reads a value from an index in the array, effectively retrieving a thread local value.

Parameters

xTaskToQuery The handle of the task from which the thread local data is being read.

A task can read its own thread local data by using NULL as the parameter value..

xIndex The index into the thread local storage array from which data is being read.

Return Values

The value read from the task’s thread local storage array at index xIndex.

Example

uint32\_t ulVariable;

/\* Read the value stored in index 5 of the calling task's thread local storage array into ulVariable. \*/

ulVariable = ( uint32\_t ) pvTaskGetThreadLocalStoragePointer( NULL, 5 );

Listing 43 Example use of pvTaskGetThreadLocalStoragePointer()

# 2.24 pcTaskGetTaskName()

#include “FreeRTOS.h” #include “task.h”

char \* pcTaskGetTaskName( TaskHandle\_t xTaskToQuery );

Listing 44 pcTaskGetTaskName() function prototype

Summary

Queries the human readable text name of a task. A text name is assigned to a task using the pcName parameter of the xTaskCreate() API function call used to create the task.

Parameters

xTaskToQuery The handle of the task being queried (the subject task).

The handle to a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function call used to create the task.

A task may query its own name by passing NULL in place of a valid task handle.

Return Values

Task names are standard NULL terminated C strings. The value returned is a pointer to the subject tasks name.

Notes

INCLUDE\_pcTaskGetTaskName must be set to 1 in FreeRTOSConfig.h for pcTaskGetTaskName() to be available.

# 2.25 xTaskGetTickCount()

#include “FreeRTOS.h” #include “task.h”

TickType\_t xTaskGetTickCount( void );

Listing 45 xTaskGetTickCount() function prototype

Summary

The tick count is the total number of tick interrupts that have occurred since the scheduler was started. xTaskGetTickCount() returns the current tick count value.

Parameters None.

Return Values xTaskGetTickCount() always returns the tick count value at the time that xTaskGetTickCount() was called.

Notes

The actual time one tick period represents depends on the value assigned to configTICK\_RATE\_HZ within FreeRTOSConfig.h. The pdMS\_TO\_TICKS() macro can be used to convert a time in milliseconds to a time in ‘ticks’.

The tick count will eventually overflow and return to zero. This will not affect the internal operation of the kernel – for example, tasks will always block for the specified period even if the tick count overflows while the task is in the Blocked state. Overflows must however be considered by host applications if the application makes direct use of the tick count value.

The frequency at which the tick count overflows depends on both the tick frequency and the data type used to hold the count value. If configUSE\_16\_BIT\_TICKS is set to 1, then the tick count will be held in a 16-bit variable. If configUSE\_16\_BIT\_TICKS is set to 0, then the tick count will be held in a 32-bit variable.

Example

void vAFunction( void )

{

TickType\_t xTime1, xTime2, xExecutionTime;

/\* Get the time the function started. \*/ xTime1 = xTaskGetTickCount();

/\* Perform some operation. \*/

/\* Get the time following the execution of the operation. \*/ xTime2 = xTaskGetTickCount();

/\* Approximately how long did the operation take? \*/ xExecutionTime = xTime2 – xTime1;

}

Listing 46 Example use of xTaskGetTickCount()

# 2.26 xTaskGetTickCountFromISR()

#include “FreeRTOS.h” #include “task.h”

TickType\_t xTaskGetTickCountFromISR( void );

Listing 47 xTaskGetTickCountFromISR() function prototype

Summary

A version of xTaskGetTickCount() that can be called from an ISR.

The tick count is the total number of tick interrupts that have occurred since the scheduler was started.

Parameters None.

Return Values xTaskGetTickCountFromISR() always returns the tick count value at the time xTaskGetTickCountFromISR() is called.

Notes

The actual time one tick period represents depends on the value assigned to configTICK\_RATE\_HZ within FreeRTOSConfig.h. The pdMS\_TO\_TICKS() macro can be used to convert a time in milliseconds to a time in ‘ticks’.

The tick count will eventually overflow and return to zero. This will not affect the internal operation of the kernel – for example, tasks will always block for the specified period even if the tick count overflows while the task is in the Blocked state. Overflows must however be considered by host applications if the application makes direct use of the tick count value.

The frequency at which the tick count overflows depends on both the tick frequency and the data type used to hold the count value. If configUSE\_16\_BIT\_TICKS is set to 1, then the tick count will be held in a 16-bit variable. If configUSE\_16\_BIT\_TICKS is set to 0, then the tick count will be held in a 32-bit variable.

Example

void vAnISR( void )

{

static TickType\_t xTimeISRLastExecuted = 0; TickType\_t xTimeNow, xTimeBetweenInterrupts;

/\* Store the time at which this interrupt was entered. \*/ xTimeNow = xTaskGetTickCountFromISR();

/\* Perform some operation. \*/

/\* How many ticks occurred between this and the previous interrupt? \*/ xTimeBetweenInterrupts = xTimeISRLastExecuted – xTimeNow;

/\* If more than 200 ticks occurred between this and the previous interrupt then do something. \*/

if( xTimeBetweenInterrupts > 200 )

{

/\* Take appropriate action here. \*/

}

/\* Remember the time at which this interrupt was entered. \*/ xTimeISRLastExecuted = xTimeNow;

}

Listing 48 Example use of xTaskGetTickCountFromISR()

# 2.27 vTaskList()

#include “FreeRTOS.h” #include “task.h”

void vTaskList( char \*pcWriteBuffer );

Listing 49 vTaskList() function prototype

Summary

Creates a human readable table in a character buffer that describes the state of each task at the time vTaskList() was called. An example is shown in Figure 2.

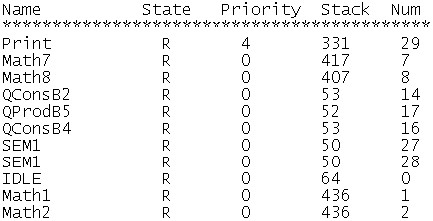


Figure 2 An example of the table produced by calling vTaskList()

The table includes the following information:

* Name – This is the name given to the task when the task was created.
* State – The state of the task at the time vTaskList() was called, as follows:
  + ‘B’ if the task is in the Blocked state. o ‘R’ if the task is in the Ready state.
  + ‘S’ if the task is in the Suspended state, or in the Blocked state without a timeout.
  + ‘D’ if the task has been deleted, but the idle task has not yet freed the memory that was being used by the task to hold its data structures and stack.
* Priority – The priority assigned to the task at the time vTaskList() was called.
* Stack – Shows the ‘high water mark’ of the task’s stack. This is the minimum amount of free stack that has been available during the lifetime of the task. The closer this value is to zero, the closer the task has come to overflowing its stack.
* Num – This is a unique number that is assigned to each task. It has no purpose other than to help identify tasks when more than one task has been assigned the same name.

Parameters

pcWriteBuffer The buffer into which the table text is written. This must be large enough to

hold the entire table as no boundary checking is performed.

Return Values None.

Notes

vTaskList() is a utility function that is provided for convenience only. It is not considered part of the kernel. vTaskList() obtains its raw data using the xTaskGetSystemState() API function.

vTaskList() will disable interrupts for the duration of its execution. This might not be acceptable for applications that include hard real time functionality.

configUSE\_TRACE\_FACILITY and configUSE\_STATS\_FORMATTING\_FUNCTIONS must both be set to 1 within FreeRTOSConfig.h for vTaskList() to be available.

By default, vTaskList() makes use of the standard library sprintf() function. This can result in a marked increase in the compiled image size, and in stack usage. The FreeRTOS download includes an open source cut down version of sprintf() in a file called printf-stdarg.c. This can be used in place of the standard library sprintf() to help minimise the code size impact. Note that printf-stdarg.c is licensed separately to FreeRTOS. Its license terms are contained in the file itself.

Example

void vAFunction( void )

{

/\* Define a buffer that is large enough to hold the generated table. In most cases the buffer will be too large to allocate on the stack, hence in this example it is declared static. \*/

static char cBuffer[ BUFFER\_SIZE ];

/\* Pass the buffer into vTaskList() to generate the table of information. \*/ vTaskList( cBuffer );

/\* The generated information can be saved or viewed here. \*/

}

Listing 50 Example use of vTaskList()

# 2.28 xTaskNotify()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskNotify( TaskHandle\_t xTaskToNotify, uint32\_t ulValue, eNotifyAction eAction );

Listing 51 xTaskNotify() function prototype

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. xTaskNotify() is used to send an event directly to and potentially unblock a task, and optionally update the receiving task's notification value in one of the following ways:

* Write a 32-bit number to the notification value
* Add one (increment) the notification value
* Set one or more bits in the notification value
* Leave the notification value unchanged

Parameters

|  |  |
| --- | --- |
| xTaskToNotify | The handle of the RTOS task being notified.  Task handles are obtained using the pxCreatedTask parameter of the xTaskCreate() call used to create the task. |
| ulValue | Used to update the notification value of the task being notified. How ulValue is interpreted depends on the value of the eAction parameter. |
| eAction | The action to perform when notifying the task. |

eAction is an enumerated type and can take one of the following values:

* eNoAction – The task is notified but its notification value is not changed. In this case ulValue is not used.
* eSetBits – The task’s notification value is bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will be set within the task's notification value. If ulValue is 0x04 then bit 2 will be set in the task's notification value. Using eSetBits allows task notifications to be used as a faster and light weight alternative to an event group.
* eIncrement – The task’s notification value is incremented by one. In this case ulValue is not used.
* eSetValueWithOverwrite – The task’s notification value is unconditionally set to the value of ulValue, even if the task already had a notification pending when xTaskNotify() was called.
* eSetValueWithoutOverwrite – If the task already has a notification pending then its notification value is not changed and xTaskNotify() returns pdFAIL. If the task did not already have a notification pending then its notification value is set to ulValue.

Return Values

If eAction is set to eSetValueWithoutOverwrite and the task’s notification value is not updated then pdFAIL is returned. In all other cases pdPASS is returned.

Notes

If the task’s notification value is being used as a light weight and faster alternative to a binary or counting semaphore then use the simpler xTaskNotifyGive() API function instead of xTaskNotify().

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE\_TASK\_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

Listing 52 demonstrates xTaskNotify() being used to perform various different actions.

/\* Set bit 8 in the notification value of the task referenced by xTask1Handle. \*/ xTaskNotify( xTask1Handle, ( 1UL << 8UL ), eSetBits );

/\* Send a notification to the task referenced by xTask2Handle, potentially removing the task from the Blocked state, but without updating the task's notification value. \*/

xTaskNotify( xTask2Handle, 0, eNoAction );

/\* Set the notification value of the task referenced by xTask3Handle to 0x50, even if the task had not read its previous notification value. \*/ xTaskNotify( xTask3Handle, 0x50, eSetValueWithOverwrite );

/\* Set the notification value of the task referenced by xTask4Handle to 0xfff, but only if to do so would not overwrite the task's existing notification value before the task had obtained it (by a call to xTaskNotifyWait() or ulTaskNotifyTake()). \*/

if( xTaskNotify( xTask4Handle, 0xfff, eSetValueWithoutOverwrite ) == pdPASS ) {

/\* The task's notification value was updated. \*/

} else

{

/\* The task's notification value was not updated. \*/

}

Listing 52 Example use of xTaskNotify()

# 2.29 xTaskNotifyAndQuery()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskNotifyAndQuery( TaskHandle\_t xTaskToNotify, uint32\_t ulValue, eNotifyAction eAction,

uint32\_t \*pulPreviousNotifyValue );

Listing 53 xTaskNotifyAndQuery() function prototype

Summary

xTaskNotifyAndQuery() is similar to xTaskNotify(), but includes an additional parameter in which the subject task’s previous notification value is returned.

Each task has a 32-bit notification value that is initialized to zero when the task is created. xTaskNotify() is used to send an event directly to and potentially unblock a task, and optionally update the receiving task's notification value in one of the following ways:

* Write a 32-bit number to the notification value
* Add one (increment) the notification value
* Set one or more bits in the notification value
* Leave the notification value unchanged

Parameters

|  |  |
| --- | --- |
| xTaskToNotify | The handle of the RTOS task being notified.  Task handles are obtained using the pxCreatedTask parameter of the xTaskCreate() call used to create the task. |
| ulValue | Used to update the notification value of the task being notified. How ulValue is interpreted depends on the value of the eAction parameter. |
| eAction | The action to perform when notifying the task. |

eAction is an enumerated type and can take one of the following

values:

* eNoAction – The task is notified but its notification value is not changed. In this case ulValue is not used.
* eSetBits – The task’s notification value is bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will be set within the task's notification value. If ulValue is 0x04 then bit 2 will be set in the task's notification value. Using eSetBits allows task notifications to be used as a faster and light weight alternative to an event group.
* eIncrement – The task’s notification value is incremented by one. In this case ulValue is not used.
* eSetValueWithOverwrite – The task’s notification value is unconditionally set to the value of ulValue, even if the task already had a notification pending when xTaskNotify() was called.
* eSetValueWithoutOverwrite – If the task already has a notification pending then its notification value is not changed and xTaskNotify() returns pdFAIL. If the task did not already have a notification pending then its notification value is set to ulValue.

pulPreviousNotifyValue Used to pass out the subject task's notification value before any bits

are modified by the action of xTaskNotifyAndQuery().

pulPreviousNotifyValue is an optional parameter, and can be set to NULL if it is not required. If pulPreviousNotifyValue is not used then consider using xTaskNotify() in place of xTaskNotifyAndQuery().

Return Values

If eAction is set to eSetValueWithoutOverwrite and the task’s notification value is not updated then pdFAIL is returned. In all other cases pdPASS is returned.

Notes

If the task’s notification value is being used as a light weight and faster alternative to a binary or counting semaphore then use the simpler xTaskNotifyGive() API function instead of xTaskNotify().

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE\_TASK\_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

Listing 52 demonstrates xTaskNotify() being used to perform various different actions.

uint32\_t ulPreviousValue;

/\* Set bit 8 in the notification value of the task referenced by xTask1Handle. The task’s previous notification value is not needed, so the last pulPreviousNotifyValue parameter is set to NULL. \*/

xTaskNotifyAndQuery( xTask1Handle, ( 1UL << 8UL ), eSetBits, NULL );

/\* Send a notification to the task referenced by xTask2Handle, potentially removing the task from the Blocked state, but without updating the task's notification value.

The task’s current notification value is saved in ulPreviousValue. \*/ xTaskNotifyAndQuery( xTask2Handle, 0, eNoAction, &ulPreviousValue );

/\* Set the notification value of the task referenced by xTask3Handle to 0x50, even if the task had not read its previous notification value. Save the task’s previous notification value (before it was set to 0x50) in ulPreviousValue. \*/

xTaskNotifyAndQuery( xTask3Handle, 0x50, eSetValueWithOverwrite, &ulPreviousValue );

/\* Set the notification value of the task referenced by xTask4Handle to 0xfff, but only if to do so would not overwrite the task's existing notification value before the task had obtained it (by a call to xTaskNotifyWait() or ulTaskNotifyTake()).

Save the task’s previous notification value (before it was set to 0xfff) in ulPreviousValue. \*/

if( xTaskNotifyAndQuery( xTask4Handle,

0xfff,

eSetValueWithoutOverwrite,

&ulPreviousValue ) == pdPASS ) {

/\* The task's notification value was updated. \*/

} else

{

/\* The task's notification value was not updated. \*/

}

Listing 54 Example use of xTaskNotify()

# 2.30 xTaskNotifyFromISR()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskNotifyFromISR( TaskHandle\_t xTaskToNotify, uint32\_t ulValue, eNotifyAction eAction

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 55 xTaskNotifyFromISR() function prototype

Summary

A version of xTaskNotify() that can be called from an interrupt service routine (ISR).

Each task has a 32-bit notification value that is initialized to zero when the task is created. xTaskNotifyFromISR() is used to send an event directly to and potentially unblock a task, and optionally update the receiving task's notification value in one of the following ways:

* Write a 32-bit number to the notification value
* Add one (increment) the notification value
* Set one or more bits in the notification value
* Leave the notification value unchanged Parameters

|  |  |
| --- | --- |
| xTaskToNotify | The handle of the RTOS task being notified.  Task handles are obtained using the pxCreatedTask parameter of the xTaskCreate() call used to create the task. |
| ulValue | Used to update the notification value of the task being notified. How ulValue is interpreted depends on the value of the eAction parameter. |
| eAction | The action to perform when notifying the task. |

eAction is an enumerated type and can take one of the following values:

* eNoAction – The task is notified but its notification value is not changed. In this case ulValue is not used.
* eSetBits – The task’s notification value is bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will be set within the task's notification value. If ulValue is 0x04 then bit 2 will be set in the task's notification value. Using eSetBits allows task notifications to be used as a faster and light weight alternative to an event group.
* eIncrement – The task’s notification value is incremented by one. In this case ulValue is not used.
* eSetValueWithOverwrite – The task’s notification value is unconditionally set to the value of ulValue, even if the task already had a notification pending when xTaskNotify() was called.
* eSetValueWithoutOverwrite – If the task already has a notification pending then its notification value is not changed and xTaskNotify() returns pdFAIL. If the task did not already have a notification pending then its notification value is set to ulValue.

pxHigherPriorityTaskWoken \*pxHigherPriorityTaskWoken must be initialized to pdFALSE.

xTaskNotifyFromISR() will then set \*pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused the task being notified to leave the Blocked state, and the unblocked task has a priority above that of the currently running task.

If xTaskNotifyFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. See Listing 56 for an example.

pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values

If eAction is set to eSetValueWithoutOverwrite and the task’s notification value is not updated then pdFAIL is returned. In all other cases pdPASS is returned.

Notes

If the task’s notification value is being used as a light weight and faster alternative to a binary or counting semaphore then use the simpler vTaskNotifyGiveFromISR() API function instead of xTaskNotifyFromISR().

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE\_TASK\_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

This example demonstrates a single RTOS task being used to process events that originate from two separate interrupt service routines - a transmit interrupt and a receive interrupt. Many peripherals will use the same handler for both, in which case the peripheral's interrupt status register can simply be bitwise ORed with the receiving task's notification value.

/\* First bits are defined to represent each interrupt source. \*/

#define TX\_BIT 0x01

#define RX\_BIT 0x02

/\* The handle of the task that will receive notifications from the interrupts. The handle was obtained when the task was created. \*/ static TaskHandle\_t xHandlingTask;

/\* The implementation of the transmit interrupt service routine. \*/ void vTxISR( void )

{

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

/\* Clear the interrupt source. \*/ prvClearInterrupt();

/\* Notify the task that the transmission is complete by setting the TX\_BIT in the task's notification value. \*/

xTaskNotifyFromISR( xHandlingTask, TX\_BIT, eSetBits, &xHigherPriorityTaskWoken );

/\* If xHigherPriorityTaskWoken is now set to pdTRUE then a context switch should be performed to ensure the interrupt returns directly to the highest priority task. The macro used for this purpose is dependent on the port in use and may be called portEND\_SWITCHING\_ISR(). \*/

portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken ); }

/\*-----------------------------------------------------------\*/

/\* The implementation of the receive interrupt service routine is identical except for the bit that gets set in the receiving task's notification value. \*/ void vRxISR( void )

{

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

/\* Clear the interrupt source. \*/ prvClearInterrupt();

/\* Notify the task that the reception is complete by setting the RX\_BIT in the task's notification value. \*/

xTaskNotifyFromISR( xHandlingTask, RX\_BIT, eSetBits, &xHigherPriorityTaskWoken );

/\* If xHigherPriorityTaskWoken is now set to pdTRUE then a context switch should be performed to ensure the interrupt returns directly to the highest priority task. The macro used for this purpose is dependent on the port in use and may be called portEND\_SWITCHING\_ISR(). \*/

portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );

}

/\*-----------------------------------------------------------\*/

/\* The implementation of the task that is notified by the interrupt service routines. \*/

static void prvHandlingTask( void \*pvParameter )

{

const TickType\_t xMaxBlockTime = pdMS\_TO\_TICKS( 500 );

BaseType\_t xResult;

for( ;; )

{

/\* Wait to be notified of an interrupt. \*/

xResult = xTaskNotifyWait( pdFALSE, /\* Don't clear bits on entry. \*/ ULONG\_MAX, /\* Clear all bits on exit. \*/ &ulNotifiedValue, /\* Stores the notified value. \*/ xMaxBlockTime );

if( xResult == pdPASS )

{

/\* A notification was received. See which bits were set. \*/ if( ( ulNotifiedValue & TX\_BIT ) != 0 )

{

/\* The TX ISR has set a bit. \*/ prvProcessTx();

}

if( ( ulNotifiedValue & RX\_BIT ) != 0 )

{

/\* The RX ISR has set a bit. \*/ prvProcessRx();

} } else

{

/\* Did not receive a notification within the expected time. \*/ prvCheckForErrors();

}

}

}

Listing 56 Example use of xTaskNotifyFromISR()

# 2.31 xTaskNotifyGive()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskNotifyGive( TaskHandle\_t xTaskToNotify );

Listing 57 xTaskNotifyGive() function prototype

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value.

xTaskNotifyGive() is a macro intended for use when a task notification value is being used as a lighter weight and faster alternative to a binary semaphore or a counting semaphore. FreeRTOS semaphores are given using the xSemaphoreGive() API function, and xTaskNotifyGive() is the equivalent that uses the receiving task's notification value instead of a separate semaphore object.

Parameters

xTaskToNotify The handle of the task being notified and having its notification value

incremented.

Task handles are obtained using the pxCreatedTask parameter of the xTaskCreate() call used to create the task.

Return Values xTaskNotifyGive() is a macro that calls xTaskNotify() with the eAction parameter set to eIncrement. Therefore pdPASS is always returned.

Notes

When a task notification value is being used as a binary or counting semaphore then the task being notified should wait for the notification using the simpler ulTaskNotifyTake() API function rather than the xTaskNotifyWait() API function.

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE\_TASK\_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

/\* Prototypes of the two tasks created by main(). \*/ static void prvTask1( void \*pvParameters ); static void prvTask2( void \*pvParameters );

/\* Handles for the tasks create by main(). \*/ static TaskHandle\_t xTask1 = NULL, xTask2 = NULL;

/\* Create two tasks that send notifications back and forth to each other, then start the RTOS scheduler. \*/ void main( void )

{

xTaskCreate( prvTask1, "Task1", 200, NULL, tskIDLE\_PRIORITY, &xTask1 ); xTaskCreate( prvTask2, "Task2", 200, NULL, tskIDLE\_PRIORITY, &xTask2 ); vTaskStartScheduler();

}

/\*-----------------------------------------------------------\*/

static void prvTask1( void \*pvParameters )

{

for( ;; )

{

/\* Send a notification to prvTask2(), bringing it out of the Blocked state. \*/

xTaskNotifyGive( xTask2 );

/\* Block to wait for prvTask2() to notify this task. \*/ ulTaskNotifyTake( pdTRUE, portMAX\_DELAY );

}

}

/\*-----------------------------------------------------------\*/

static void prvTask2( void \*pvParameters )

{

for( ;; )

{

/\* Block to wait for prvTask1() to notify this task. \*/ ulTaskNotifyTake( pdTRUE, portMAX\_DELAY );

/\* Send a notification to prvTask1(), bringing it out of the Blocked state. \*/

xTaskNotifyGive( xTask1 );

}

}

Listing 58 Example use of xTaskNotifyGive()

# 2.32 vTaskNotifyGiveFromISR()

#include “FreeRTOS.h” #include “task.h”

void vTaskNotifyGiveFromISR( TaskHandle\_t xTaskToNotify,

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 59 vTaskNotifyGiveFromISR() function prototype

Summary

A version of xTaskNotifyGive() that can be called from an interrupt service routine (ISR).

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value.

vTaskNotifyGiveFromISR() is intended for use when a task notification value is being used as a lighter weight and faster alternative to a binary semaphore or a counting semaphore. FreeRTOS semaphores are given using the xSemaphoreGiveFromISR() API function, and vTaskNotifyGiveFromISR() is the equivalent that uses the receiving task's notification value instead of a separate semaphore object.

Parameters xTaskToNotify The handle of the RTOS task being notified and having its

notification value incremented.

Task handles are obtained using the pxCreatedTask parameter of the xTaskCreate() call used to create the task.

pxHigherPriorityTaskWoken \*pxHigherPriorityTaskWoken must be initialized to pdFALSE.

vTaskNotifyGiveFromISR() will then set

\*pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused the task being notified to leave the Blocked state, and the unblocked task has a priority above that of the currently running task.

If vTaskNotifyGiveFromISR() sets this value to pdTRUE then a

context switch should be requested before the interrupt is exited. See Listing 60 for an example.

pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Notes

When a task notification value is being used as a binary or counting semaphore then the task being notified should wait for the notification using the ulTaskNotifyTake() API function rather than the xTaskNotifyWait() API function.

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE\_TASK\_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

This is an example of a transmit function in a generic peripheral driver. An task calls the transmit function, then waits in the Blocked state (so not using an CPU time) until it is notified that the transmission is complete. The transmission is performed by a DMA, and the DMA end interrupt is used to notify the task.

static TaskHandle\_t xTaskToNotify = NULL;

/\* The peripheral driver's transmit function. \*/

void StartTransmission( uint8\_t \*pcData, size\_t xDataLength )

{

/\* At this point xTaskToNotify should be NULL as no transmission is in progress.

A mutex can be used to guard access to the peripheral if necessary. \*/ configASSERT( xTaskToNotify == NULL );

/\* Store the handle of the calling task. \*/ xTaskToNotify = xTaskGetCurrentTaskHandle();

/\* Start the transmission - an interrupt is generated when the transmission is complete. \*/

vStartTransmit( pcData, xDataLength );

}

/\*-----------------------------------------------------------\*/

/\* The transmit end interrupt. \*/ void vTransmitEndISR( void )

{

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

/\* At this point xTaskToNotify should not be NULL as a transmission was in progress. \*/

configASSERT( xTaskToNotify != NULL );

/\* Notify the task that the transmission is complete. \*/

vTaskNotifyGiveFromISR( xTaskToNotify, &xHigherPriorityTaskWoken );

/\* There are no transmissions in progress, so no tasks to notify. \*/ xTaskToNotify = NULL;

/\* If xHigherPriorityTaskWoken is now set to pdTRUE then a context switch should be performed to ensure the interrupt returns directly to the highest priority task. The macro used for this purpose is dependent on the port in use and may be called portEND\_SWITCHING\_ISR(). \*/ portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );

}

/\*-----------------------------------------------------------\*/

/\* The task that initiates the transmission, then enters the Blocked state (so not consuming any CPU time) to wait for it to complete. \*/

void vAFunctionCalledFromATask( uint8\_t ucDataToTransmit, size\_t xDataLength )

{

uint32\_t ulNotificationValue;

const TickType\_t xMaxBlockTime = pdMS\_TO\_TICKS( 200 );

/\* Start the transmission by calling the function shown above. \*/

StartTransmission( ucDataToTransmit, xDataLength );

/\* Wait for the transmission to complete. \*/

ulNotificationValue = ulTaskNotifyTake( pdFALSE, xMaxBlockTime );

if( ulNotificationValue == 1 )

{

/\* The transmission ended as expected. \*/

} else

{

/\* The call to ulTaskNotifyTake() timed out. \*/

}

}

Listing 60 Example use of vTaskNotifyGiveFromISR()

# 2.33 ulTaskNotifyTake()

#include “FreeRTOS.h” #include “task.h”

uint32\_t ulTaskNotifyTake( BaseType\_t xClearCountOnExit, TickType\_t xTicksToWait );

Listing 61 ulTaskNotifyTake() function prototype

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value.

ulTaskNotifyTake() is intended for use when a task notification is used as a faster and lighter weight alternative to a binary semaphore or a counting semaphore. FreeRTOS semaphores are taken using the xSemaphoreTake() API function, ulTaskNotifyTake() is the equivalent that uses a task notification value instead of a separate semaphore object.

Where as xTaskNotifyWait() will return when a notification is pending, ulTaskNotifyTake() will return when the task's notification value is not zero, decrementing the task's notification value before it returns.

A task can use ulTaskNotifyTake() to optionally block to wait for a the task's notification value to be non-zero. The task does not consume any CPU time while it is in the Blocked state.

ulTaskNotifyTake() can either clear the task's notification value to zero on exit, in which case the notification value acts like a binary semaphore, or decrement the task's notification value on exit, in which case the notification value acts more like a counting semaphore.

Parameters

xClearCountOnExit If xClearCountOnExit is set to pdFALSE then the task's notification value

is decremented before ulTaskNotifyTake() exits. This is equivalent to the value of a counting semaphore being decremented by a successful call to xSemaphoreTake().

If xClearCountOnExit is set to pdTRUE then the task's notification value is reset to 0 before ulTaskNotifyTake() exits. This is equivalent to the value of a binary semaphore being left at zero (or empty, or 'not available') after a successful call to xSemaphoreTake().

xTicksToWait The maximum time to wait in the Blocked state for a notification to be

received if a notification is not already pending when ulTaskNotifyTake() is called.

The RTOS task does not consume any CPU time when it is in the Blocked state.

The time is specified in RTOS tick periods. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in ticks.

Return Values

The value of the task's notification value before it is decremented or cleared (see the description of xClearCountOnExit).

Notes

When a task is using its notification value as a binary or counting semaphore other tasks and interrupts should send notifications to it using either the xTaskNotifyGive() macro, or the xTaskNotify() function with the function's eAction parameter set to eIncrement (the two are equivalent).

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE\_TASK\_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

/\* An interrupt handler that unblocks a high priority task in which the event that generated the interrupt is processed. If the priority of the task is high enough then the interrupt will return directly to the task (so it will interrupt one task then return to a different task), so the processing will occur contiguously in time - just as if all the processing had been done in the interrupt handler itself. \*/ void vANInterruptHandler( void )

{

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

/\* Clear the interrupt. \*/ prvClearInterruptSource();

/\* Unblock the handling task so the task can perform any processing necessitated by the interrupt. xHandlingTask is the task's handle, which was obtained when the task was created. \*/

vTaskNotifyGiveFromISR( xHandlingTask, &xHigherPriorityTaskWoken );

/\* Force a context switch if xHigherPriorityTaskWoken is now set to pdTRUE.

The macro used to do this is dependent on the port and may be called portEND\_SWITCHING\_ISR(). \*/

portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );

}

/\*-----------------------------------------------------------\*/

/\* Task that blocks waiting to be notified that the peripheral needs servicing. \*/ void vHandlingTask( void \*pvParameters )

{

BaseType\_t xEvent;

for( ;; )

{

/\* Block indefinitely (without a timeout, so no need to check the function's return value) to wait for a notification. Here the RTOS task notification is being used as a binary semaphore, so the notification value is cleared to zero on exit. NOTE! Real applications should not block indefinitely, but instead time out occasionally in order to handle error conditions that may prevent the interrupt from sending any more notifications. \*/ ulTaskNotifyTake( pdTRUE, /\* Clear the notification value on exit. \*/ portMAX\_DELAY );/\* Block indefinitely. \*/

/\* The RTOS task notification is used as a binary (as opposed to a counting) semaphore, so only go back to wait for further notifications when all events pending in the peripheral have been processed. \*/ do {

xEvent = xQueryPeripheral();

if( xEvent != NO\_MORE\_EVENTS )

{

vProcessPeripheralEvent( xEvent );

}

} while( xEvent != NO\_MORE\_EVENTS );

}

}

Listing 62 Example use of ulTaskNotifyTake()

# 2.34 xTaskNotifyWait()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskNotifyWait( uint32\_t ulBitsToClearOnEntry, uint32\_t ulBitsToClearOnExit, uint32\_t \*pulNotificationValue,

TickType\_t xTicksToWait );

Listing 63 xTaskNotifyWait() function prototype

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value in a number of different ways. For example, a notification may overwrite the receiving task's notification value, or just set one or more bits in the receiving task's notification value. See the xTaskNotify() API documentation for examples. xTaskNotifyWait() waits, with an optional timeout, for the calling task to receive a notification.

If the receiving task was already Blocked waiting for a notification when one arrives the receiving task will be removed from the Blocked state and the notification cleared.

Parameters

|  |  |
| --- | --- |
| ulBitsToClearOnEntry | Any bits set in ulBitsToClearOnEntry will be cleared in the calling task's notification value on entry to the xTaskNotifyWait() function (before the task waits for a new notification) provided a notification is not already pending when xTaskNotifyWait() is called.  For example, if ulBitsToClearOnEntry is 0x01, then bit 0 of the task's notification value will be cleared on entry to the function.  Setting ulBitsToClearOnEntry to 0xffffffff (ULONG\_MAX) will clear all the bits in the task's notification value, effectively clearing the value to  0. |
| ulBitsToClearOnExit | Any bits set in ulBitsToClearOnExit will be cleared in the calling task's |

notification value before xTaskNotifyWait() function exits if a notification was received.

The bits are cleared after the task's notification value has been saved in \*pulNotificationValue (see the description of pulNotificationValue below).

For example, if ulBitsToClearOnExit is 0x03, then bit 0 and bit 1 of the task's notification value will be cleared before the function exits.

Setting ulBitsToClearOnExit to 0xffffffff (ULONG\_MAX) will clear all the bits in the task's notification value, effectively clearing the value to

0.

|  |  |
| --- | --- |
| pulNotificationValue | Used to pass out the task's notification value. The value copied to \*pulNotificationValue is the task's notification value as it was before any bits were cleared due to the ulBitsToClearOnExit setting.  pulNotificationValue is an optional parameter and can be set to NULL if it is not required. |
| xTicksToWait | The maximum time to wait in the Blocked state for a notification to be |

received if a notification is not already pending when xTaskNotifyWait() is called.

The task does not consume any CPU time when it is in the Blocked state.

The time is specified in RTOS tick periods. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in ticks.

Return Values

pdTRUE is returned if a notification was received, or if a notification was already pending when xTaskNotifyWait() was called.

pdFALSE is returned if the call to xTaskNotifyWait() timed out before a notification was received.

Notes

If you are using task notifications to implement binary or counting semaphore type behavior then use the simpler ulTaskNotifyTake() API function instead of xTaskNotifyWait().

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE\_TASK\_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

/\* This task shows bits within the RTOS task notification value being used to pass different events to the task in the same way that flags in an event group might be used for the same purpose. \*/

void vAnEventProcessingTask( void \*pvParameters )

{

uint32\_t ulNotifiedValue;

for( ;; )

{

/\* Block indefinitely (without a timeout, so no need to check the function's return value) to wait for a notification.

Bits in this RTOS task's notification value are set by the notifying tasks and interrupts to indicate which events have occurred. \*/ xTaskNotifyWait( 0x00, /\* Don't clear any notification bits on entry. \*/

ULONG\_MAX, /\* Reset the notification value to 0 on exit. \*/ &ulNotifiedValue, /\* Notified value pass out in ulNotifiedValue. \*/ portMAX\_DELAY ); /\* Block indefinitely. \*/

/\* Process any events that have been latched in the notified value. \*/

if( ( ulNotifiedValue & 0x01 ) != 0 )

{

/\* Bit 0 was set - process whichever event is represented by bit 0. \*/ prvProcessBit0Event();

}

if( ( ulNotifiedValue & 0x02 ) != 0 )

{

/\* Bit 1 was set - process whichever event is represented by bit 1. \*/ prvProcessBit1Event();

}

if( ( ulNotifiedValue & 0x04 ) != 0 )

{

/\* Bit 2 was set - process whichever event is represented by bit 2. \*/ prvProcessBit2Event();

}

/\* Etc. \*/

}

}

Listing 64 Example use of xTaskNotifyWait()

# 2.35 uxTaskPriorityGet()

#include “FreeRTOS.h” #include “task.h”

UBaseType\_t uxTaskPriorityGet( TaskHandle\_t pxTask );

Listing 65 uxTaskPriorityGet() function prototype

Summary

Queries the priority assigned to a task at the time uxTaskPriorityGet() is called.

Parameters

pxTask The handle of the task being queried (the subject task).

The handle to a task is obtained using the pxCreatedTask parameter to the xTaskCreate() API function when the task is created.

A task may query its own priority by passing NULL in place of a valid task handle.

Return Values

The value returned is the priority of the task being queried at the time uxTaskPriorityGet() is called.

Example

void vAFunction( void )

{

TaskHandle\_t xHandle;

UBaseType\_t uxCreatedPriority, uxOurPriority;

/\* Create a task, storing the handle of the created task in xHandle. \*/ if( xTaskCreate( vTaskCode, "Demo task",

STACK\_SIZE, NULL, PRIORITY,

&xHandle

) != pdPASS )

{

/\* The task was not created successfully. \*/

} else

{

/\* Use the handle to query the priority of the created task. \*/ uxCreatedPriority = uxTaskPriorityGet( xHandle );

/\* Query the priority of the calling task by using NULL in place of a valid task handle. \*/

uxOurPriority = uxTaskPriorityGet( NULL );

/\* Is the priority of this task higher than the priority of the task just created? \*/

if( uxOurPriority > uxCreatedPriority )

{

/\* Yes. \*/

}

}

}

Listing 66 Example use of uxTaskPriorityGet()

# 2.36 vTaskPrioritySet()

#include “FreeRTOS.h” #include “task.h”

void vTaskPrioritySet( TaskHandle\_t pxTask, UBaseType\_t uxNewPriority );

Listing 67 vTaskPrioritySet() function prototype

Summary

Changes the priority of a task.

Parameters

|  |  |
| --- | --- |
| pxTask | The handle of the task being modified (the subject task).  The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function.  A task can change its own priority by passing NULL in place of a valid task handle. |
| uxNewPriority | The priority to which the subject task will be set. Priorities can be assigned |

from 0, which is the lowest priority, to (configMAX\_PRIORITIES – 1), which is the highest priority.

configMAX\_PRIORITIES is defined in FreeRTOSConfig.h. Passing a value above (configMAX\_PRIORITIES – 1) will result in the priority assigned to the task being capped to the maximum legitimate value.

Return Values None.

Notes vTaskPrioritySet() must only be called from an executing task, and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

It is possible to have a set of tasks that are all blocked waiting for the same queue or semaphore event. These tasks will be ordered according to their priority – for example, the first event will unblock the highest priority task that was waiting for the event, the second event will unblock the second highest priority task that was originally waiting for the event, etc. Using vTaskPrioritySet() to change the priority of such a blocked task will not cause the order in which the blocked tasks are assessed to be re-evaluated.

Example

void vAFunction( void )

{

TaskHandle\_t xHandle;

/\* Create a task, storing the handle of the created task in xHandle. \*/ if( xTaskCreate( vTaskCode, "Demo task",

STACK\_SIZE,

NULL,

PRIORITY,

&xHandle

) != pdPASS )

{

/\* The task was not created successfully. \*/

} else

{

/\* Use the handle to raise the priority of the created task. \*/ vTaskPrioritySet( xHandle, PRIORITY + 1 );

/\* Use NULL in place of a valid task handle to set the priority of the calling task to 1. \*/ vTaskPrioritySet( NULL, 1 );

}

}

Listing 68 Example use of vTaskPrioritySet()

# 2.37 vTaskResume()

#include “FreeRTOS.h” #include “task.h”

void vTaskResume( TaskHandle\_t pxTaskToResume );

Listing 69 vTaskResume() function prototype

Summary

Transition a task from the Suspended state to the Ready state. The task must have previously been placed into the Suspended state using a call to vTaskSuspend().

Parameters pxTaskToResume The handle of the task being resumed (transitioned out of the Suspended

state). This is the subject task.

The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function.

Return Values None.

Notes

A task can be blocked to wait for a queue event, specifying a timeout period. It is legitimate to move such a Blocked task into the Suspended state using a call to vTaskSuspend(), then out of the Suspended state and into the Ready state using a call to vTaskResume(). Following this scenario, the next time the task enters the Running state it will check whether or not its timeout period has (in the meantime) expired. If the timeout period has not expired, the task will once again enter the Blocked state to wait for the queue event for the remainder of the originally specified timeout period.

A task can also be blocked to wait for a temporal event using the vTaskDelay() or vTaskDelayUntil() API functions. It is legitimate to move such a Blocked task into the Suspended state using a call to vTaskSuspend(), then out of the Suspended state and into the

Ready state using a call to vTaskResume(). Following this scenario, the next time the task enters the Running state it will exit the vTaskDelay() or vTaskDelayUntil() function as if the specified delay period had expired, even if this is not actually the case.

vTaskResume() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

Example

void vAFunction( void )

{

TaskHandle\_t xHandle;

/\* Create a task, storing the handle to the created task in xHandle. \*/ if( xTaskCreate( vTaskCode, "Demo task",

STACK\_SIZE,

NULL,

PRIORITY,

&xHandle

) != pdPASS )

{

/\* The task was not created successfully. \*/

} else {

/\* Use the handle to suspend the created task. \*/ vTaskSuspend( xHandle );

/\* The suspended task will not run during this period, unless another task calls vTaskResume( xHandle ). \*/

/\* Resume the suspended task again. \*/ vTaskResume( xHandle );

/\* The created task is again available to the scheduler and can enter

The Running state. \*/

}

}

Listing 70 Example use of vTaskResume()

# 2.38 xTaskResumeAll()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskResumeAll( void );

Listing 71 xTaskResumeAll() function prototype

Summary

Resumes scheduler activity, following a previous call to vTaskSuspendAll(), by transitioning the scheduler into the Active state from the Suspended state.

Parameters None.

Return Values pdTRUE The scheduler was transitioned into the Active state. The transition caused a

pending context switch to occur.

pdFALSE Either the scheduler was transitioned into the Active state and the transition did not

cause a context switch to occur, or the scheduler was left in the Suspended state due to nested calls to vTaskSuspendAll().

Notes

The scheduler can be suspended by calling vTaskSuspendAll(). When the scheduler is suspended, interrupts remain enabled, but a context switch will not occur. If a context switch is requested while the scheduler is suspended, then the request will be held pending until such time that the scheduler is resumed (un-suspended).

Calls to vTaskSuspendAll() can be nested. The same number of calls must be made to xTaskResumeAll() as have previously been made to vTaskSuspendAll() before the scheduler will leave the Suspended state and re-enter the Active state.

xTaskResumeAll() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

Other FreeRTOS API functions should not be called while the scheduler is suspended.

Example

/\* A function that suspends then resumes the scheduler. \*/ void vDemoFunction( void )

{

/\* This function suspends the scheduler. When it is called from vTask1 the scheduler is already suspended, so this call creates a nesting depth of 2. \*/ vTaskSuspendAll();

/\* Perform an action here. \*/

/\* As calls to vTaskSuspendAll() are now nested, resuming the scheduler here does not cause the scheduler to re-enter the active state. \*/ xTaskResumeAll();

}

void vTask1( void \* pvParameters )

{

for( ;; )

{

/\* Perform some actions here. \*/

/\* At some point the task wants to perform an operation during which it does not want to get swapped out, or it wants to access data which is also accessed from another task (but not from an interrupt). It cannot use taskENTER\_CRITICAL()/taskEXIT\_CRITICAL() as the length of the operation may cause interrupts to be missed. \*/

/\* Prevent the scheduler from performing a context switch. \*/ vTaskSuspendAll();

/\* Perform the operation here. There is no need to use critical sections as the task has all the processing time other than that utilized by interrupt service routines.\*/

/\* Calls to vTaskSuspendAll() can be nested, so it is safe to call a (non

API) function that also calls vTaskSuspendAll(). API functions should not be called while the scheduler is suspended. \*/ vDemoFunction();

/\* The operation is complete. Set the scheduler back into the Active state. \*/

if( xTaskResumeAll() == pdTRUE )

{

/\* A context switch occurred within xTaskResumeAll(). \*/

} else

{

/\* A context switch did not occur within xTaskResumeAll(). \*/

}

}

}

Listing 72 Example use of xTaskResumeAll()

# 2.39 xTaskResumeFromISR()

#include “FreeRTOS.h” #include “task.h”

BaseType\_t xTaskResumeFromISR( TaskHandle\_t pxTaskToResume );

Listing 73 xTaskResumeFromISR() function prototype

Summary

A version of vTaskResume() that can be called from an interrupt service routine.

Parameters pxTaskToResume The handle of the task being resumed (transitioned out of the Suspended

state). This is the subject task.

The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function.

Return Values pdTRUE Returned if the task being resumed (unblocked) has a priority equal to or higher

than the currently executing task (the task that was interrupted) – meaning a context switch should be performed before exiting the interrupt.

pdFALSE Returned if the task being resumed has a priority lower that the currently executing

task (the task that was interrupted) – meaning it is not necessary to perform a context switch before exiting the interrupt.

Notes

A task can be suspended by calling vTaskSuspend(). While in the Suspended state the task will not be selected to enter the Running state. vTaskResume() and xTaskResumeFromISR() can be used to resume (un-suspend) a suspended task. xTaskResumeFromISR() can be called from an interrupt, but vTaskResume() cannot.

Calls to vTaskSuspend() do not maintain a nesting count. A task that has been suspended by one of more calls to vTaskSuspend() will always be un-suspended by a single call to vTaskResume() or xTaskResumeFromISR().

xTaskResumeFromISR() must not be used to synchronize a task with an interrupt. Doing so will result in interrupt events being missed if the interrupt events occur faster than the execution of its associated task level handling functions. Task and interrupt synchronization can be achieved safely using a binary or counting semaphore because the semaphore will latch events.

Example

TaskHandle\_t xHandle;

void vAFunction( void )

{

/\* Create a task, storing the handle of the created task in xHandle. \*/ xTaskCreate( vTaskCode, "NAME", STACK\_SIZE, NULL, tskIDLE\_PRIORITY, &xHandle );

/\* ... Rest of code. \*/

}

void vTaskCode( void \*pvParameters )

{

/\* The task being suspended and resumed. \*/ for( ;; )

{

/\* ... Perform some function here. \*/

/\* The task suspends itself by using NULL as the parameter to vTaskSuspend() in place of a valid task handle. \*/ vTaskSuspend( NULL );

/\* The task is now suspended, so will not reach here until the ISR resumes

(un-suspends) it. \*/

}

}

void vAnExampleISR( void ) {

BaseType\_t xYieldRequired;

/\* Resume the suspended task. \*/

xYieldRequired = xTaskResumeFromISR( xHandle );

if( xYieldRequired == pdTRUE )

{

/\* A context switch should now be performed so the ISR returns directly to the resumed task. This is because the resumed task had a priority that was equal to or higher than the task that is currently in the Running state. NOTE: The syntax required to perform a context switch from an ISR varies from port to port, and from compiler to compiler. Check the documentation and examples for the port being used to find the syntax required by your application. It is likely that this if() statement can be replaced by a single call to portYIELD\_FROM\_ISR() [or portEND\_SWITCHING\_ISR()] using xYieldRequired as the macro parameter: portYIELD\_FROM\_ISR( xYieldRequired );\*/ portYIELD\_FROM\_ISR();

}

}

Listing 74 Example use of xTaskResumeFromISR()

# 2.40 vTaskSetApplicationTaskTag()

#include “FreeRTOS.h” #include “task.h”

void vTaskSetApplicationTaskTag( TaskHandle\_t xTask, TaskHookFunction\_t pxTagValue );

Listing 75 vTaskSetApplicationTaskTag() function prototype

Summary

This function is intended for advanced users only.

The vTaskSetApplicationTaskTag() API function can be used to assign a ‘tag’ value to a task. The meaning and use of the tag value is defined by the application writer. The kernel itself will not normally access the tag value.

Parameters xTask The handle of the task to which a tag value is being assigned. This is the

subject task.

A task can assign a tag value to itself by either using its own task handle or by using NULL in place of a valid task handle.

pxTagValue The value being assigned as the tag value of the subject task. This is of type TaskHookFunction\_t to permit a function pointer to be assigned to the tag, although, indirectly by casting, tag values can be of any type.

Return Values None.

Notes

The tag value can be used to hold a function pointer. When this is done the function assigned to the tag value can be called using the xTaskCallApplicationTaskHook() API function. This technique is in effect assigning a callback function to the task. It is common for such a callback to be used in combination with the traceTASK\_SWITCHED\_IN() macro to implement an execution trace feature.

configUSE\_APPLICATION\_TASK\_TAG must be set to 1 in FreeRTOSConfig.h for vTaskSetApplicationTaskTag() to be available.

Example

/\* In this example, an integer is set as the task tag value. \*/ void vATask( void \*pvParameters )

{

/\* Assign a tag value of 1 to the currently executing task. The (void \*) cast is used to prevent compiler warnings. \*/ vTaskSetApplicationTaskTag( NULL, ( void \* ) 1 );

for( ;; )

{

/\* Rest of task code goes here. \*/

}

}

/\* In this example a callback function is assigned as the task tag. First define the callback function - this must have type TaskHookFunction\_t, as per this example. \*/ static BaseType\_t prvExampleTaskHook( void \* pvParameter )

{

/\* Perform some action - this could be anything from logging a value, updating the task state, outputting a value, etc. \*/

return 0;

}

/\* Now define the task that sets prvExampleTaskHook() as its hook/tag value. This is in effect registering the task callback function. \*/ void vAnotherTask( void \*pvParameters )

{

/\* Register a callback function for the currently running (calling) task. \*/ vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );

for( ;; )

{

/\* Rest of task code goes here. \*/

}

}

/\* [As an example use of the hook (callback)] Define the traceTASK\_SWITCHED\_OUT() macro to call the hook function. The kernel will then automatically call the task hook each time the task is switched out. This technique can be used to generate an execution trace. pxCurrentTCB references the currently executing task. \*/

#define traceTASK\_SWITCHED\_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, 0 )

Listing 76 Example use of vTaskSetApplicationTaskTag()

# 2.41 vTaskSetThreadLocalStoragePointer()

#include “FreeRTOS.h” #include “task.h”

void vTaskSetThreadLocalStoragePointer( TaskHandle\_t xTaskToSet, BaseType\_t xIndex, void \*pvValue );

Listing 77 vTaskSetThreadLocalStoragePointer() function prototype

Summary

Thread local storage (or TLS) allows the application writer to store values inside a task's control block, making the value specific to (local to) the task itself, and allowing each task to have its own unique value.

Each task has its own array of pointers that can be used as thread local storage. The number of indexes in the array is set by the configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS compile time configuration constant in FreeRTOSConfig.h.

vTaskSetThreadLocalStoragePointer() sets the value of an index in the array, effectively storing a thread local value.

Parameters

xTaskToSet The handle of the task to which the thread local data is being written.

A task can write to its own thread local data by using NULL as the parameter value..

|  |  |
| --- | --- |
| xIndex | The index into the thread local storage array to which data is being written. |
| pvValue | The value to write into the into the index specified by xIndex. |

Return Values

None.

Example

uint32\_t ulVariable;

/\* Write the 32-bit 0x12345678 value directly into index 1 of the thread local storage array. Passing NULL as the task handle has the effect of writing to the calling task's thread local storage array. \*/

vTaskSetThreadLocalStoragePointer( NULL, /\* Task handle. \*/ 1, /\* Index into the array. \*/

( void \* ) 0x12345678 );

/\* Store the value of the 32-bit variable ulVariable to index 0 of the calling task's thread local storage array. \*/ ulVariable = ERROR\_CODE;

vTaskSetThreadLocalStoragePointer( NULL, /\* Task handle. \*/ 0, /\* Index into the array. \*/

( void \* ) &ulVariable );

Listing 78 Example use of vTaskSetThreadLocalStoragePointer()

# 2.42 vTaskSetTimeOutState()

#include “FreeRTOS.h” #include “task.h”

void vTaskSetTimeOutState( TimeOut\_t \* const pxTimeOut );

Listing 79 vTaskSetTimeOutState() function prototype

Summary

This function is intended for advanced users only.

A task can enter the Blocked state to wait for an event. Typically, the task will not wait in the Blocked state indefinitely, but instead a timeout period will be specified. The task will be removed from the Blocked state if the timeout period expires before the event the task is waiting for occurs.

If a task enters and exits the Blocked state more than once while it is waiting for the event to occur then the timeout used each time the task enters the Blocked state must be adjusted to ensure the total of all the time spent in the Blocked state does not exceed the originally specified timeout period. xTaskCheckForTimeOut() performs the adjustment, taking into account occasional occurrences such as tick count overflows, which would otherwise make a manual adjustment prone to error.

vTaskSetTimeOutState() is used with xTaskCheckForTimeOut(). vTaskSetTimeOutState() is called to set the initial condition, after which xTaskCheckForTimeOut() can be called to check for a timeout condition, and adjust the remaining block time if a timeout has not occurred.

Parameters

pxTimeOut A pointer to a structure that will be initialized to hold information necessary to

determine if a timeout has occurred.

Example

/\* Driver library function used to receive uxWantedBytes from an Rx buffer that is filled by a UART interrupt. If there are not enough bytes in the Rx buffer then the task enters the Blocked state until it is notified that more data has been placed into the buffer. If there is still not enough data then the task re-enters the Blocked state, and xTaskCheckForTimeOut() is used to re-calculate the Block time to ensure the total amount of time spent in the Blocked state does not exceed MAX\_TIME\_TO\_WAIT. This continues until either the buffer contains at least uxWantedBytes bytes, or the total amount of time spent in the Blocked state reaches MAX\_TIME\_TO\_WAIT – at which point the task reads however many bytes are available up to a maximum of uxWantedBytes. \*/ size\_t xUART\_Receive( uint8\_t \*pucBuffer, size\_t uxWantedBytes )

{

size\_t uxReceived = 0;

TickType\_t xTicksToWait = MAX\_TIME\_TO\_WAIT;

TimeOut\_t xTimeOut;

/\* Initialize xTimeOut. This records the time at which this function was entered. \*/ vTaskSetTimeOutState( &xTimeOut );

/\* Loop until the buffer contains the wanted number of bytes, or a timeout occurs. \*/ while( UART\_bytes\_in\_rx\_buffer( pxUARTInstance ) < uxWantedBytes )

{

/\* The buffer didn’t contain enough data so this task is going to enter the Blocked state. Adjusting xTicksToWait to account for any time that has been spent in the Blocked state within this function so far to ensure the total amount of time spent in the Blocked state does not exceed MAX\_TIME\_TO\_WAIT. \*/ if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )

{

/\* Timed out before the wanted number of bytes were available, exit the loop. \*/ break;

}

/\* Wait for a maximum of xTicksToWait ticks to be notified that the receive interrupt has placed more data into the buffer. \*/ ulTaskNotifyTake( pdTRUE, xTicksToWait );

}

/\* Attempt to read uxWantedBytes from the receive buffer into pucBuffer. The actual number of bytes read (which might be less than uxWantedBytes) is returned. \*/ uxReceived = UART\_read\_from\_receive\_buffer( pxUARTInstance, pucBuffer, uxWantedBytes );

return uxReceived;

}

Listing 80 Example use of vTaskSetTimeOutState() and xTaskCheckForTimeOut()

# 2.43 vTaskStartScheduler()

#include “FreeRTOS.h” #include “task.h”

void vTaskStartScheduler( void );

Listing 81 vTaskStartScheduler() function prototype

Summary

Starts the FreeRTOS scheduler running.

Typically, before the scheduler has been started, main() (or a function called by main()) will be executing. After the scheduler has been started, only tasks and interrupts will ever execute.

Starting the scheduler causes the highest priority task that was created while the scheduler was in the Initialization state to enter the Running state.

Parameters None.

Return Values

The Idle task is created automatically when the scheduler is started. vTaskStartScheduler() will only return if there is not enough FreeRTOS heap memory available for the Idle task to be created.

Notes

Ports that execute on ARM7 and ARM9 microcontrollers require the processor to be in Supervisor mode before vTaskStartScheduler() is called.

Example

TaskHandle\_t xHandle;

/\* Define a task function. \*/ void vATask( void )

{

for( ;; )

{

/\* Task code goes here. \*/

}

}

void main( void )

{

/\* Create at least one task, in this case the task function defined above is created. Calling vTaskStartScheduler() before any tasks have been created will cause the idle task to enter the Running state. \*/

xTaskCreate( vTaskCode, "task name", STACK\_SIZE, NULL, TASK\_PRIORITY, NULL );

/\* Start the scheduler. \*/ vTaskStartScheduler();

/\* This code will only be reached if the idle task could not be created inside vTaskStartScheduler(). An infinite loop is used to assist debugging by ensuring this scenario does not result in main() exiting. \*/ for( ;; );

}

Listing 82 Example use of vTaskStartScheduler()

# 2.44 vTaskStepTick()

#include “FreeRTOS.h” #include “task.h”

void vTaskStepTick( TickType\_t xTicksToJump );

Summary

If the RTOS is configured to use tickless idle functionality then the tick interrupt will be stopped, and the microcontroller placed into a low power state, whenever the Idle task is the only task able to execute. Upon exiting the low power state the tick count value must be corrected to account for the time that passed while it was stopped.

If a FreeRTOS port includes a default portSUPPRESS\_TICKS\_AND\_SLEEP() implementation, then vTaskStepTick() is used internally to ensure the correct tick count value is maintained. vTaskStepTick() is a public API function to allow the default portSUPPRESS\_TICKS\_AND\_SLEEP() implementation to be overridden, and for a portSUPPRESS\_TICKS\_AND\_SLEEP() to be provided if the port being used does not provide a default.

Parameters xTicksToJump The number of RTOS tick periods that passed between the tick interrupt

being stopped and restarted (how long the tick interrupt was suppressed for). For correct operation the parameter must be less than or equal to the portSUPPRESS\_TICKS\_AND\_SLEEP() parameter.

Return Values None.

Notes configUSE\_TICKLESS\_IDLE must be set to 1 in FreeRTOSConfig.h for vTaskStepTick() to be available.

/\* This is an example of how portSUPPRESS\_TICKS\_AND\_SLEEP() might be implemented by an application writer. This basic implementation will introduce inaccuracies in the tracking of the time maintained by the kernel in relation to calendar time. Official

FreeRTOS implementations account for these inaccuracies as much as possible.

Only vTaskStepTick() is part of the FreeRTOS API. The other function calls are for demonstration only. \*/

/\* First define the portSUPPRESS\_TICKS\_AND\_SLEEP() macro. The parameter is the time, in ticks, until the kernel next needs to execute. \*/

#define portSUPPRESS\_TICKS\_AND\_SLEEP( xIdleTime ) vApplicationSleep( xIdleTime )

/\* Define the function that is called by portSUPPRESS\_TICKS\_AND\_SLEEP(). \*/ void vApplicationSleep( TickType\_t xExpectedIdleTime )

{

unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;

/\* Read the current time from a time source that will remain operational when the microcontroller is in a low power state. \*/ ulLowPowerTimeBeforeSleep = ulGetExternalTime();

/\* Stop the timer that is generating the tick interrupt. \*/ prvStopTickInterruptTimer();

/\* Configure an interrupt to bring the microcontroller out of its low power state at the time the kernel next needs to execute. The interrupt must be generated from a source that remains operational when the microcontroller is in a low power state. \*/

vSetWakeTimeInterrupt( xExpectedIdleTime );

/\* Enter the low power state. \*/ prvSleep();

/\* Determine how long the microcontroller was actually in a low power state for, which will be less than xExpectedIdleTime if the microcontroller was brought out of low power mode by an interrupt other than that configured by the vSetWakeTimeInterrupt() call. Note that the scheduler is suspended before portSUPPRESS\_TICKS\_AND\_SLEEP() is called, and resumed when

portSUPPRESS\_TICKS\_AND\_SLEEP() returns. Therefore no other tasks will execute until this function completes. \*/

ulLowPowerTimeAfterSleep = ulGetExternalTime();

/\* Correct the kernels tick count to account for the time the microcontroller spent in its low power state. \*/

vTaskStepTick( ulLowPowerTimeAfterSleep – ulLowPowerTimeBeforeSleep );

/\* Restart the timer that is generating the tick interrupt. \*/ prvStartTickInterruptTimer();

}

Listing 83 Example use of vTaskStepTick()

# 2.45 vTaskSuspend()

#include “FreeRTOS.h” #include “task.h”

void vTaskSuspend( TaskHandle\_t pxTaskToSuspend );

Listing 84 vTaskSuspend() function prototype

Summary

Places a task into the Suspended state. A task that is in the Suspended state will never be selected to enter the Running state.

The only way of removing a task from the Suspended state is to make it the subject of a call to vTaskResume().

Parameters pxTaskToSuspend The handle of the task being suspended.

The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function.

A task may suspend itself by passing NULL in place of a valid task handle.

Return Values None.

Notes

If FreeRTOS version 6.1.0 or later is being used, then vTaskSuspend() can be called to place a task into the Suspended state before the scheduler has been started (before vTaskStartScheduler() has been called). This will result in the task (effectively) starting in the Suspended state.

void vAFunction( void )

{

TaskHandle\_t xHandle;

/\* Create a task, storing the handle of the created task in xHandle. \*/ if( xTaskCreate( vTaskCode, "Demo task",

STACK\_SIZE,

NULL,

PRIORITY,

&xHandle

) != pdPASS )

{

/\* The task was not created successfully. \*/

} else {

/\* Use the handle of the created task to place the task in the Suspended state. From FreeRTOS version 6.1.0, this can be done before the Scheduler has been started. \*/ vTaskSuspend( xHandle );

/\* The created task will not run during this period, unless another task calls vTaskResume( xHandle ). \*/

/\* Use a NULL parameter to suspend the calling task. \*/ vTaskSuspend( NULL );

/\* This task can only execute past the call to vTaskSuspend( NULL ) if another task has resumed (un-suspended) it using a call to vTaskResume(). \*/

}

}

Listing 85 Example use of vTaskSuspend()

# 2.46 vTaskSuspendAll()

#include “FreeRTOS.h” #include “task.h”

void vTaskSuspendAll( void );

Listing 86 vTaskSuspendAll() function prototype

Summary

Suspends the scheduler. Suspending the scheduler prevents a context switch from occurring but leaves interrupts enabled. If an interrupt requests a context switch while the scheduler is suspended, then the request is held pending and is performed only when the scheduler is resumed (un-suspended).

Parameters None.

Return Values None.

Notes

Calls to xTaskResumeAll() transition the scheduler out of the Suspended state following a previous call to vTaskSuspendAll().

Calls to vTaskSuspendAll() can be nested. The same number of calls must be made to xTaskResumeAll() as have previously been made to vTaskSuspendAll() before the scheduler will leave the Suspended state and re-enter the Active state.

xTaskResumeAll() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

Other FreeRTOS API functions must not be called while the scheduler is suspended.

/\* A function that suspends then resumes the scheduler. \*/ void vDemoFunction( void )

{

/\* This function suspends the scheduler. When it is called from vTask1 the scheduler is already suspended, so this call creates a nesting depth of 2. \*/ vTaskSuspendAll();

/\* Perform an action here. \*/

/\* As calls to vTaskSuspendAll() are nested, resuming the scheduler here will not cause the scheduler to re-enter the active state. \*/ xTaskResumeAll();

}

void vTask1( void \* pvParameters )

{

for( ;; )

{

/\* Perform some actions here. \*/

/\* At some point the task wants to perform an operation during which it does not want to get swapped out, or it wants to access data which is also accessed from another task (but not from an interrupt). It cannot use taskENTER\_CRITICAL()/taskEXIT\_CRITICAL() as the length of the operation may cause interrupts to be missed. \*/

/\* Prevent the scheduler from performing a context switch. \*/ vTaskSuspendAll();

/\* Perform the operation here. There is no need to use critical sections as the task has all the processing time other than that utilized by interrupt service routines.\*/

/\* Calls to vTaskSuspendAll() can be nested so it is safe to call a (non API) function which also contains calls to vTaskSuspendAll(). API functions should not be called while the scheduler is suspended. \*/ vDemoFunction();

/\* The operation is complete. Set the scheduler back into the Active state. \*/

if( xTaskResumeAll() == pdTRUE )

{

/\* A context switch occurred within xTaskResumeAll(). \*/

} else

{

/\* A context switch did not occur within xTaskResumeAll(). \*/ }

}

}

Listing 87 Example use of vTaskSuspendAll()

# 2.47 taskYIELD()

#include “FreeRTOS.h” #include “task.h”

void taskYIELD( void );

Listing 88 taskYIELD() macro prototype

Summary

Yield to another task of equal priority.

Yielding is where a task volunteers to leave the Running state, without being pre-empted, and before its time slice has been fully utilized.

Parameters None.

Return Values None.

Notes

taskYIELD() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

When a task calls taskYIELD(), the scheduler will select another Ready state task of equal priority to enter the Running state in its place. If there are no other Ready state tasks of equal priority then the task that called taskYIELD() will itself be transitioned straight back into the Running state.

The scheduler will only ever select a task of equal priority to the task that called taskYIELD() because, if there were any tasks of higher priority that were in the Ready state, the task that called taskYIELD() would not have been executing in the first place.

void vATask( void \* pvParameters)

{

for( ;; )

{

/\* Perform some actions. \*/

/\* If there are any tasks of equal priority to this task that are in the

Ready state then let them execute now - even though this task has not used all of its time slice. \*/ taskYIELD();

/\* If there were any tasks of equal priority to this task in the Ready state, then they will have executed before this task reaches here. \*/

}

}

Listing 89 Example use of taskYIELD()

Chapter 3

Queue API

# 3.1 vQueueAddToRegistry()

#include “FreeRTOS.h”

#include “queue.h”

void vQueueAddToRegistry( QueueHandle\_t xQueue, char \*pcQueueName );

Listing 90 vQueueAddToRegistry() function prototype

Summary

Assigns a human readable name to a queue, and adds the queue to the queue registry.

Parameters xQueue The handle of the queue that will be added to the registry. Semaphore

handles can also be used.

pcQueueName A descriptive name for the queue or semaphore. This is not used by FreeRTOS in any way. It is included purely as a debugging aid. Identifying a queue or semaphore by a human readable name is much simpler than attempting to identify it by its handle.

Return Values None.

Notes

The queue registry is used by kernel aware debuggers:

1. It allows a text name to be associated with a queue or semaphore for easy queue and semaphore identification in a debugging interface.
2. It provides a means for a debugger to locate queue and semaphore structures.

The configQUEUE\_REGISTRY\_SIZE kernel configuration constant defines the maximum number of queues and semaphores that can be registered at any one time. Only the queues and semaphores that need to be viewed in a kernel aware debugging interface need to be registered.

The queue registry is only required when a kernel aware debugger is being used. At all other times it has no purpose and can be omitted by setting configQUEUE\_REGISTRY\_SIZE to 0, or by omitting the configQUEUE\_REGISTRY\_SIZE configuration constant definition altogether.

Deleting a registered queue will automatically remove it from the registry.

Example

void vAFunction( void )

{

QueueHandle\_t xQueue;

/\* Create a queue big enough to hold 10 chars. \*/ xQueue = xQueueCreate( 10, sizeof( char ) );

/\* The created queue needs to be viewable in a kernel aware debugger, so add it to the registry. \*/

vQueueAddToRegistry( xQueue, "AMeaningfulName" );

}

Listing 91 Example use of vQueueAddToRegistry()

# 3.2 xQueueAddToSet()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueAddToSet( QueueSetMemberHandle\_t xQueueOrSemaphore,

QueueSetHandle\_t xQueueSet );

Listing 92 xQueueAddToSet() function prototype

Summary

Adds a queue or semaphore to a queue set that was previously created by a call to xQueueCreateSet().

A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Parameters xQueueOrSemaphore The handle of the queue or semaphore being added to the queue set (cast to an QueueSetMemberHandle\_t type).

xQueueSet The handle of the queue set to which the queue or semaphore is

being added.

Return Values

|  |  |
| --- | --- |
| pdPASS | The queue or semaphore was successfully added to the queue set. |
| pdFAIL | The queue or semaphore could not be added to the queue set because it is |

already a member of a different set.

Notes configUSE\_QUEUE\_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueAddToSet() API function to be available.

Example

See the example provided for the xQueueCreateSet() function in this manual.

# 3.3 xQueueCreate()

#include “FreeRTOS.h”

#include “queue.h”

QueueHandle\_t xQueueCreate( UBaseType\_t uxQueueLength,

UBaseType\_t uxItemSize );

Listing 93 xQueueCreate() function prototype

Summary

Creates a queue. A queue must be explicitly created before it can be used.

Parameters uxQueueLength The maximum number of items that the queue being created can hold at

any one time.

uxItemSize The size, in bytes, of each data item that can be stored in the queue.

Return Values

NULL The queue cannot be created because there is insufficient heap memory

available for FreeRTOS to allocate the queue data structures and storage area.

Any other value The queue was created successfully. The returned value should be stored

as the handle to the created queue.

Notes

Queues are used to pass data between tasks, and between tasks and interrupts.

Queues can be created before or after the scheduler has been started.

Example

/\* Define the data type that will be queued. \*/ typedef struct A\_Message

{

char ucMessageID; char ucData[ 20 ];

} AMessage;

/\* Define the queue parameters. \*/

#define QUEUE\_LENGTH 5 #define QUEUE\_ITEM\_SIZE sizeof( AMessage )

int main( void )

{

QueueHandle\_t xQueue;

/\* Create the queue, storing the returned handle in the xQueue variable. \*/ xQueue = xQueueCreate( QUEUE\_LENGTH, QUEUE\_ITEM\_SIZE ); if( xQueue == NULL )

{

/\* The queue could not be created. \*/

}

/\* Rest of code goes here. \*/

}

Listing 94 Example use of xQueueCreate()

# 3.4 xQueueCreateSet()

#include “FreeRTOS.h”

#include “queue.h”

QueueSetHandle\_t xQueueCreateSet( const UBaseType\_t uxEventQueueLength );

Listing 95 xQueueCreateSet() function prototype

Summary

Queue sets provide a mechanism to allow an RTOS task to block (pend) on a read operation from multiple RTOS queues or semaphores simultaneously. Note that there are simpler alternatives to using queue sets. See the Blocking on Multiple Objects page of the FreeRTOS.org website for more information.

A queue set must be explicitly created using a call to xQueueCreateSet() before it can be used. Once created, standard FreeRTOS queues and semaphores can be added to the set using calls to xQueueAddToSet(). xQueueSelectFromSet() is then used to determine which, if any, of the queues or semaphores contained in the set is in a state where a queue read or semaphore take operation would be successful.

Parameters uxEventQueueLength Queue sets store events that occur on the queues and semaphores

contained in the set. uxEventQueueLength specifies the maximum number of events that can be queued at once.

To be absolutely certain that events are not lost uxEventQueueLength must be set to the sum of the lengths of the queues added to the set, where binary semaphores and mutexes have a length of 1, and counting semaphores have a length set by their maximum count value. For example:

* If a queue set is to hold a queue of length 5, another queue of length 12, and a binary semaphore, then uxEventQueueLength should be set to (5 + 12 + 1), or 18.
* If a queue set is to hold three binary semaphores then uxEventQueueLength should be set to (1 + 1 + 1 ), or 3.
* If a queue set is to hold a counting semaphore that has a maximum count of 5, and a counting semaphore that has a maximum count of 3, then uxEventQueueLength should be set to (5 + 3), or 8.

Return Values

NULL The queue set could not be created.

Any other value The queue set was created successfully. The returned value should be

stored as the handle to the created queue set.

Notes

Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

An additional 4 bytes of RAM are required for each space in every queue added to a queue set. Therefore a counting semaphore that has a high maximum count value should not be added to a queue set.

A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

configUSE\_QUEUE\_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueCreateSet() API function to be available.

Example

/\* Define the lengths of the queues that will be added to the queue set. \*/

#define QUEUE\_LENGTH\_1 10

#define QUEUE\_LENGTH\_2 10

/\* Binary semaphores have an effective length of 1. \*/

#define BINARY\_SEMAPHORE\_LENGTH 1

/\* Define the size of the item to be held by queue 1 and queue 2 respectively. The values used here are just for demonstration purposes. \*/ #define ITEM\_SIZE\_QUEUE\_1 sizeof( uint32\_t )

#define ITEM\_SIZE\_QUEUE\_2 sizeof( something\_else\_t )

/\* The combined length of the two queues and binary semaphore that will be added to the queue set. \*/

#define COMBINED\_LENGTH ( QUEUE\_LENGTH\_1 + QUEUE\_LENGTH\_2 + BINARY\_SEMAPHORE\_LENGTH )

void vAFunction( void )

{

static QueueSetHandle\_t xQueueSet;

QueueHandle\_t xQueue1, xQueue2, xSemaphore;

QueueSetMemberHandle\_t xActivatedMember; uint32\_t xReceivedFromQueue1; something\_else\_t xReceivedFromQueue2;

/\* Create a queue set large enough to hold an event for every space in every queue and semaphore that is to be added to the set. \*/ xQueueSet = xQueueCreateSet( COMBINED\_LENGTH );

/\* Create the queues and semaphores that will be contained in the set. \*/ xQueue1 = xQueueCreate( QUEUE\_LENGTH\_1, ITEM\_SIZE\_QUEUE\_1 ); xQueue2 = xQueueCreate( QUEUE\_LENGTH\_2, ITEM\_SIZE\_QUEUE\_2 );

/\* Create the semaphore that is being added to the set. \*/ xSemaphore = xSemaphoreCreateBinary();

/\* Take the semaphore, so it starts empty. A block time of zero can be used as the semaphore is guaranteed to be available - it has just been created. \*/ xSemaphoreTake( xSemaphore, 0 );

/\* Add the queues and semaphores to the set. Reading from these queues and semaphore can only be performed after a call to xQueueSelectFromSet() has returned the queue or semaphore handle from this point on. \*/ xQueueAddToSet( xQueue1, xQueueSet ); xQueueAddToSet( xQueue2, xQueueSet ); xQueueAddToSet( xSemaphore, xQueueSet );

/\* CONTINUED ON NEXT PAGE \*/

/\* CONTINUED FROM PREVIOUS PAGE \*/

for( ;; )

{

/\* Block to wait for something to be available from the queues or semaphore that have been added to the set. Don't block longer than 200ms. \*/ xActivatedMember = xQueueSelectFromSet( xQueueSet, pdMS\_TO\_TICKS( 200 ) );

/\* Which set member was selected? Receives/takes can use a block time of zero as they are guaranteed to pass because xQueueSelectFromSet() would not have returned the handle unless something was available. \*/ if( xActivatedMember == xQueue1 )

{

xQueueReceive( xActivatedMember, &xReceivedFromQueue1, 0 ); vProcessValueFromQueue1( xReceivedFromQueue1 );

}

else if( xActivatedQueue == xQueue2 )

{

xQueueReceive( xActivatedMember, &xReceivedFromQueue2, 0 ); vProcessValueFromQueue2( &xReceivedFromQueue2 );

}

else if( xActivatedQueue == xSemaphore )

{

/\* Take the semaphore to make sure it can be "given" again. \*/ xSemaphoreTake( xActivatedMember, 0 ); vProcessEventNotifiedBySemaphore(); break; } else

{

/\* The 200ms block time expired without an RTOS queue or semaphore being ready to process. \*/

}

}

}

Listing 96 Example use of xQueueCreateSet() and other queue set API functions

# 3.5 vQueueDelete()

#include “FreeRTOS.h”

#include “queue.h”

void vQueueDelete( TaskHandle\_t pxQueueToDelete );

Listing 97 vQueueDelete() function prototype

Summary

Deletes a queue that was previously created using a call to xQueueCreate(). vQueueDelete() can also be used to delete a semaphore.

Parameters pxQueueToDelete The handle of the queue being deleted. Semaphore handles can also be

used.

Return Values

None

Notes

Queues are used to pass data between tasks and between tasks and interrupts.

Tasks can opt to block on a queue/semaphore (with an optional timeout) if they attempt to send data to the queue/semaphore and the queue/semaphore is already full, or they attempt to receive data from a queue/semaphore and the queue/semaphore is already empty. A queue/semaphore must not be deleted if there are any tasks currently blocked on it.

Example

/\* Define the data type that will be queued. \*/ typedef struct A\_Message

{

char ucMessageID; char ucData[ 20 ];

} AMessage;

/\* Define the queue parameters. \*/

#define QUEUE\_LENGTH 5 #define QUEUE\_ITEM\_SIZE sizeof( AMessage )

int main( void )

{

QueueHandle\_t xQueue;

/\* Create the queue, storing the returned handle in the xQueue variable. \*/ xQueue = xQueueCreate( QUEUE\_LENGTH, QUEUE\_ITEM\_SIZE ); if( xQueue == NULL )

{

/\* The queue could not be created. \*/

} else

{

/\* Delete the queue again by passing xQueue to vQueueDelete(). \*/ vQueueDelete( xQueue );

}

}

Listing 98 Example use of vQueueDelete()

# 3.6 xQueueIsQueueEmptyFromISR()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueIsQueueEmptyFromISR( const QueueHandle\_t pxQueue );

Listing 99 xQueueIsQueueEmptyFromISR() function prototype

Summary

Queries a queue to see if it contains items, or if it is already empty. Items cannot be received from a queue if the queue is empty.

This function should only be used from an ISR.

Parameters pxQueue The queue being queried.

Return Values

|  |  |
| --- | --- |
| pdFALSE | The queue being queried is empty (does not contain any data items) at the time xQueueIsQueueEmptyFromISR() was called. |
| Any other value | The queue being queried was not empty (contained data items) at the |

time xQueueIsQueueEmptyFromISR() was called.

Notes

None.

# 3.7 xQueueIsQueueFullFromISR()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueIsQueueFullFromISR( const QueueHandle\_t pxQueue );

Listing 100 xQueueIsQueueFullFromISR() function prototype

Summary

Queries a queue to see if it is already full, or if it has space to receive a new item. A queue can only successfully receive new items when it is not full.

This function should only be used from an ISR.

Parameters pxQueue The queue being queried.

Return Values

|  |  |
| --- | --- |
| pdFALSE | The queue being queried is not full at the time xQueueIsQueueFullFromISR() was called. |
| Any other value | The queue being queried was full at the time |

xQueueIsQueueFullFromISR() was called.

Notes None.

# 3.8 uxQueueMessagesWaiting()

#include “FreeRTOS.h”

#include “queue.h”

UBaseType\_t uxQueueMessagesWaiting( const QueueHandle\_t xQueue );

Listing 101 uxQueueMessagesWaiting() function prototype

Summary

Returns the number of items that are currently held in a queue.

Parameters xQueue The handle of the queue being queried.

Returned Value

The number of items that are held in the queue being queried at the time that uxQueueMessagesWaiting() is called.

Example

void vAFunction( QueueHandle\_t xQueue )

{

UBaseType\_t uxNumberOfItems;

/\* How many items are currently in the queue referenced by the xQueue handle? \*/ uxNumberOfItems = uxQueueMessagesWaiting( xQueue );

}

Listing 102 Example use of uxQueueMessagesWaiting()

# 3.9 uxQueueMessagesWaitingFromISR()

#include “FreeRTOS.h”

#include “queue.h”

UBaseType\_t uxQueueMessagesWaitingFromISR( const QueueHandle\_t xQueue );

Listing 103 uxQueueMessagesWaitingFromISR() function prototype

Summary

A version of uxQueueMessagesWaiting() that can be used from inside an interrupt service routine.

Parameters xQueue The handle of the queue being queried.

Returned Value

The number of items that are contained in the queue being queried at the time that uxQueueMessagesWaitingFromISR() is called.

Example

void vAnInterruptHandler( void )

{

UBaseType\_t uxNumberOfItems; BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

/\* Check the status of the queue, if it contains more than 10 items then wake the task that will drain the queue. \*/

/\* How many items are currently in the queue referenced by the xQueue handle? \*/ uxNumberOfItems = uxQueueMessagesWaitingFromISR( xQueue );

if( uxNumberOfItems > 10 )

{

/\* The task being woken is currently blocked on xSemaphore. Giving the semaphore will unblock the task. \*/

xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

}

/\* If xHigherPriorityTaskWoken is equal to pdTRUE at this point then the task that was unblocked by the call to xSemaphoreGiveFromISR() had a priority either equal to or greater than the currently executing task (the task that was in the Running state when this interrupt occurred). In that case a context switch should be performed before leaving this interrupt service routine to ensure the interrupt returns to the highest priority ready state task (the task that was unblocked). The syntax required to perform a context switch from inside an interrupt varies from port to port, and from compiler to compiler. Check the web documentation and examples for the port in use to find the correct syntax for your application. \*/

}

Listing 104 Example use of uxQueueMessagesWaitingFromISR()

# 3.10 xQueueOverwrite()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueOverwrite( QueueHandle\_t xQueue, const void \*pvItemToQueue );

Listing 105 xQueueOverwrite() function prototype

Summary

A version of xQueueSendToBack() that will write to the queue even if the queue is full, overwriting data that is already held in the queue.

xQueueOverwrite() is intended for use with queues that have a length of one, meaning the queue is either empty or full.

This function must not be called from an interrupt service routine. See xQueueOverwriteFromISR() for an alternative which may be used in an interrupt service routine.

Parameters xQueue The handle of the queue to which the data is to be sent.

pvItemToQueue A pointer to the item that is to be placed in the queue. The size of each item

the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.

Returned Value xQueueOverwrite() is a macro that calls xQueueGenericSend(), and therefore has the same return values as xQueueSendToFront(). However, pdPASS is the only value that can be returned because xQueueOverwrite() will write to the queue even when the queue is already full.

Example

void vFunction( void \*pvParameters )

{

QueueHandle\_t xQueue;

unsigned long ulVarToSend, ulValReceived;

/\* Create a queue to hold one unsigned long value. It is strongly recommended \*not\* to use xQueueOverwrite() on queues that can contain more than one value, and doing so will trigger an assertion if configASSERT() is defined. \*/

xQueue = xQueueCreate( 1, sizeof( unsigned long ) );

/\* Write the value 10 to the queue using xQueueOverwrite(). \*/ ulVarToSend = 10;

xQueueOverwrite( xQueue, &ulVarToSend );

/\* Peeking the queue should now return 10, but leave the value 10 in the queue. A block time of zero is used as it is known that the queue holds a value. \*/ ulValReceived = 0;

xQueuePeek( xQueue, &ulValReceived, 0 );

if( ulValReceived != 10 )

{

/\* Error, unless another task removed the value. \*/

}

/\* The queue is still full. Use xQueueOverwrite() to overwrite the value held in the queue with 100. \*/ ulVarToSend = 100;

xQueueOverwrite( xQueue, &ulVarToSend );

/\* This time read from the queue, leaving the queue empty once more.

A block time of 0 is used again. \*/ xQueueReceive( xQueue, &ulValReceived, 0 );

/\* The value read should be the last value written, even though the queue was already full when the value was written. \*/ if( ulValReceived != 100 )

{

/\* Error unless another task is using the same queue. \*/ }

/\* ... \*/

}

Listing 106 Example use of xQueueOverwrite()

# 3.11 xQueueOverwriteFromISR()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueOverwriteFromISR( QueueHandle\_t xQueue, const void \*pvItemToQueue,

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 107 xQueueOverwriteFromISR() function prototype

Summary

A version of xQueueOverwrite() that can be used in an ISR. xQueueOverwriteFromISR() is similar to xQueueSendToBackFromISR(), but will write to the queue even if the queue is full, overwriting data that is already held in the queue.

xQueueOverwriteFromISR() is intended for use with queues that have a length of one, meaning the queue is either empty or full.

Parameters xQueue The handle of the queue to which the data is to be sent.

pvItemToQueue A pointer to the item that is to be placed in the queue. The size

of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.

pxHigherPriorityTaskWoken xQueueOverwriteFromISR() will set

\*pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If

xQueueOverwriteFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. Refer to the Interrupt Service Routines section of the documentation for the port being used to see how that is done.

Returned Value xQueueOverwriteFromISR() is a macro that calls xQueueGenericSendFromISR(), and therefore has the same return values as xQueueSendToFrontFromISR(). However, pdPASS is the only value that can be returned because xQueueOverwriteFromISR() will write to the queue even when the queue is already full.

Example

QueueHandle\_t xQueue;

void vFunction( void \*pvParameters )

{

/\* Create a queue to hold one unsigned long value. It is strongly recommended not to use xQueueOverwriteFromISR() on queues that can contain more than one value, and doing so will trigger an assertion if configASSERT() is defined. \*/

xQueue = xQueueCreate( 1, sizeof( unsigned long ) );

}

void vAnInterruptHandler( void )

{

/\* xHigherPriorityTaskWoken must be set to pdFALSE before it is used. \*/

BaseType\_t xHigherPriorityTaskWoken = pdFALSE; unsigned long ulVarToSend, ulValReceived;

/\* Write the value 10 to the queue using xQueueOverwriteFromISR(). \*/ ulVarToSend = 10;

xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

/\* The queue is full, but calling xQueueOverwriteFromISR() again will still pass because the value held in the queue will be overwritten with the new value. \*/ ulVarToSend = 100;

xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

/\* Reading from the queue will now return 100. \*/

/\* ... \*/

if( xHigherPriorityTaskWoken == pdTRUE )

{

/\* Writing to the queue caused a task to unblock and the unblocked task has a priority higher than or equal to the priority of the currently executing task (the task this interrupt interrupted). Perform a context switch so this interrupt returns directly to the unblocked task. \*/ portYIELD\_FROM\_ISR(); /\* or portEND\_SWITCHING\_ISR() depending on the port.\*/

}

}

Listing 108 Example use of xQueueOverwriteFromISR()

# 3.12 xQueuePeek()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueuePeek( QueueHandle\_t xQueue, void \*pvBuffer, TickType\_t xTicksToWait );

Listing 109 xQueuePeek() function prototype

Summary

Reads an item from a queue, but without removing the item from the queue. The same item will be returned the next time xQueueReceive() or xQueuePeek() is used to obtain an item from the same queue.

Parameters

|  |  |
| --- | --- |
| xQueue | The handle of the queue from which data is to be read. |
| pvBuffer | A pointer to the memory into which the data read from the queue will be copied.  The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() used to create the queue. |
| xTicksToWait | The maximum amount of time the task should remain in the Blocked state to |

wait for data to become available on the queue, should the queue already be empty.

If xTicksToWait is zero, then xQueuePeek() will return immediately if the queue is already empty.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Return Values

|  |  |
| --- | --- |
| pdPASS | Returned if data was successfully read from the queue.  If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for data to become available on the queue, but data was successfully read from the queue before the block time expired. |
| errQUEUE\_EMPTY | Returned if data cannot be read from the queue because the queue is |

already empty.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for another task or interrupt to send data to the queue, but the block time expired before this happened.

Notes None.

Example

struct AMessage

{ char ucMessageID; char ucData[ 20 ];

} xMessage;

QueueHandle\_t xQueue;

/\* Task that creates a queue and posts a value. \*/ void vATask( void \*pvParameters )

{

struct AMessage \*pxMessage;

/\* Create a queue capable of containing 10 pointers to AMessage structures. Store the handle to the created queue in the xQueue variable. \*/ xQueue = xQueueCreate( 10, sizeof( struct AMessage \* ) ); if( xQueue == 0 )

{

/\* The queue was not created because there was not enough FreeRTOS heap memory available to allocate the queues data structures or storage area. \*/

} else {

/\* ... \*/

/\* Send a pointer to a struct AMessage object to the queue referenced by the xQueue variable. Don't block if the queue is already full (the third parameter to xQueueSend() is zero, so not block time is specified). \*/ pxMessage = &xMessage;

xQueueSend( xQueue, ( void \* ) &pxMessage, 0 );

}

/\* ... Rest of the task code. \*/ for( ;; )

{

}

}

/\* Task to peek the data from the queue. \*/ void vADifferentTask( void \*pvParameters )

{

struct AMessage \*pxRxedMessage;

if( xQueue != 0 )

{

/\* Peek a message on the created queue. Block for 10 ticks if a message is not available immediately. \*/

if( xQueuePeek( xQueue, &( pxRxedMessage ), 10 ) == pdPASS ) {

/\* pxRxedMessage now points to the struct AMessage variable posted by vATask, but the item still remains on the queue. \*/

} } else

{

/\* The queue could not or has not been created. \*/ }

/\* ... Rest of the task code. \*/ for( ;; )

{

}

}

Listing 110 Example use of xQueuePeek()

# 3.13 xQueuePeekFromISR()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueuePeekFromISR( QueueHandle\_t xQueue, void \*pvBuffer );

Listing 111 xQueuePeekFromISR() function prototype

Summary

A version of xQueuePeek() that can be used from an interrupt service routine (ISR).

Reads an item from a queue, but without removing the item from the queue. The same item will be returned the next time xQueueReceive() or xQueuePeek() is used to obtain an item from the same queue.

Parameters xQueue The handle of the queue from which data is to be read.

pvBuffer A pointer to the memory into which the data read from the queue will be copied.

The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() used to create the queue.

Return Values pdPASS Returned if data was successfully read from the queue.

errQUEUE\_EMPTY Returned if data cannot be read from the queue because the queue is

already empty.

Notes None.

# 3.14 xQueueReceive()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueReceive( QueueHandle\_t xQueue, void \*pvBuffer,

TickType\_t xTicksToWait );

Listing 112 xQueueReceive() function prototype

Summary

Receive (read) an item from a queue.

Parameters

|  |  |
| --- | --- |
| xQueue | The handle of the queue from which the data is being received (read). The queue handle will have been returned from the call to xQueueCreate() used to create the queue. |
| pvBuffer | A pointer to the memory into which the received data will be copied.  The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() used to create the queue. |
| xTicksToWait | The maximum amount of time the task should remain in the Blocked state to |

wait for data to become available on the queue, should the queue already be empty.

If xTicksToWait is zero, then xQueueReceive() will return immediately if the queue is already empty.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Return Values

|  |  |
| --- | --- |
| pdPASS | Returned if data was successfully read from the queue.  If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for data to become available on the queue, but data was successfully read from the queue before the block time expired. |
| errQUEUE\_EMPTY | Returned if data cannot be read from the queue because the queue is |

already empty.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for another task or interrupt to send data to the queue, but the block time expired before this happened.

Notes None.

Example

/\* Define the data type that will be queued. \*/ typedef struct A\_Message

{ char ucMessageID; char ucData[ 20 ];

} AMessage;

/\* Define the queue parameters. \*/

#define QUEUE\_LENGTH 5

#define QUEUE\_ITEM\_SIZE sizeof( AMessage )

int main( void )

{

QueueHandle\_t xQueue;

/\* Create the queue, storing the returned handle in the xQueue variable. \*/ xQueue = xQueueCreate( QUEUE\_LENGTH, QUEUE\_ITEM\_SIZE ); if( xQueue == NULL )

{

/\* The queue could not be created – do something. \*/

}

/\* Create a task, passing in the queue handle as the task parameter. \*/ xTaskCreate( vAnotherTask, “Task”,

STACK\_SIZE,

( void \* ) xQueue, /\* The queue handle is used as the task parameter. \*/ TASK\_PRIORITY,

NULL );

/\* Start the task executing. \*/ vTaskStartScheduler();

/\* Execution will only reach here if there was not enough FreeRTOS heap memory remaining for the idle task to be created. \*/ for( ;; );

}

void vAnotherTask( void \*pvParameters )

{

QueueHandle\_t xQueue;

AMessage xMessage;

/\* The queue handle is passed into this task as the task parameter. Cast the void \* parameter back to a queue handle. \*/ xQueue = ( QueueHandle\_t ) pvParameters;

for( ;; )

{

/\* Wait for the maximum period for data to become available on the queue.

The period will be indefinite if INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h. \*/

if( xQueueReceive( xQueue, &xMessage, portMAX\_DELAY ) != pdPASS )

{

/\* Nothing was received from the queue – even after blocking to wait for data to arrive. \*/

} else

{

/\* xMessage now contains the received data. \*/ }

}

}

Listing 113 Example use of xQueueReceive()

# 3.15 xQueueReceiveFromISR()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueReceiveFromISR( QueueHandle\_t xQueue, void \*pvBuffer,

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 114 xQueueReceiveFromISR() function prototype

Summary

A version of xQueueReceive() that can be called from an ISR. Unlike xQueueReceive(), xQueueReceiveFromISR() does not permit a block time to be specified.

Parameters xQueue The handle of the queue from which the data is being received

(read). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.

pvBuffer A pointer to the memory into which the received data will be

copied.

The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() used to create the queue.

pxHigherPriorityTaskWoken It is possible that a single queue will have one or more tasks

blocked on it waiting for space to become available on the queue. Calling xQueueReceiveFromISR() can make space available, and so cause such a task to leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set

\*pxHigherPriorityTaskWoken to pdTRUE.

If xQueueReceiveFromISR() sets this value to pdTRUE, then a

context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task.

From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values pdPASS Data was successfully received from the queue. pdFAIL Data was not received from the queue because the queue was already empty.

Notes

Calling xQueueReceiveFromISR() within an interrupt service routine can potentially cause a task that was blocked on a queue to leave the Blocked state. A context switch should be performed if such an unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted). The context switch will ensure that the interrupt returns directly to the highest priority Ready state task. Unlike the xQueueReceive() API function, xQueueReceiveFromISR() will not itself perform a context switch. It will instead just indicate whether or not a context switch is required.

xQueueReceiveFromISR() must not be called prior to the scheduler being started. Therefore an interrupt that calls xQueueReceiveFromISR() must not be allowed to execute prior to the scheduler being started.

Example

For clarity of demonstration, the example in this section makes multiple calls to xQueueReceiveFromISR() to receive multiple small data items. This is inefficient and therefore not recommended for most applications. A preferable approach would be to send the multiple data items in a structure to the queue in a single post, allowing xQueueReceiveFromISR() to be called only once. Alternatively, and preferably, processing can be deferred to the task level.

/\* vISR is an interrupt service routine that empties a queue of values, sending each to a peripheral. It might be that there are multiple tasks blocked on the queue waiting for space to write more data to the queue. \*/ void vISR( void )

{ char cByte;

BaseType\_t xHigherPriorityTaskWoken;

/\* No tasks have yet been unblocked. \*/ xHigherPriorityTaskWoken = pdFALSE;

/\* Loop until the queue is empty.

xHigherPriorityTaskWoken will get set to pdTRUE internally within xQueueReceiveFromISR() if calling xQueueReceiveFromISR()caused a task to leave the Blocked state, and the unblocked task has a priority equal to or greater than the task currently in the Running state (the task this ISR interrupted). \*/ while( xQueueReceiveFromISR( xQueue, &cByte, &xHigherPriorityTaskWoken ) == pdPASS )

{

/\* Write the received byte to the peripheral. \*/

OUTPUT\_BYTE( TX\_REGISTER\_ADDRESS, cByte ); }

/\* Clear the interrupt source. \*/

/\* Now the queue is empty and we have cleared the interrupt we can perform a context switch if one is required (if xHigherPriorityTaskWoken has been set to pdTRUE. NOTE: The syntax required to perform a context switch from an ISR varies from port to port, and from compiler to compiler. Check the web documentation and examples for the port being used to find the correct syntax required for your application. \*/

portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );

}

Listing 115 Example use of xQueueReceiveFromISR()

# 3.16 xQueueRemoveFromSet()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueRemoveFromSet( QueueSetMemberHandle\_t xQueueOrSemaphore,

QueueSetHandle\_t xQueueSet );

Listing 116 xQueueRemoveFromSet() function prototype

Summary

Summary

Remove a queue or semaphore from a queue set.

A queue or semaphore can only be removed from a queue set if the queue or semaphore is empty.

Parameters xQueueOrSemaphore The handle of the queue or semaphore being removed from the

queue set (cast to an QueueSetMemberHandle\_t type).

xQueueSet The handle of the queue set in which the queue or semaphore is

included.

Return Values pdPASS The queue or semaphore was successfully removed from the queue set.

pdFAIL The queue or semaphore was not removed from the queue set because either the

queue or semaphore was not in the queue set, or the queue or semaphore was not empty.

Notes configUSE\_QUEUE\_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueRemoveFromSet() API function to be available.

Example

This example assumes xQueueSet is a queue set that has already been created, and xQueue is a queue that has already been created and added to xQueueSet.

if( xQueueRemoveFromSet( xQueue, xQueueSet ) != pdPASS )

{

/\* Either xQueue was not a member of the xQueueSet set, or xQueue is not empty and therefore cannot be removed from the set. \*/

} else

{

/\* The queue was successfully removed from the set. \*/

}

Listing 117 Example use of xQueueRemoveFromSet()

# 3.17 xQueueReset()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueReset( QueueHandle\_t xQueue );

Listing 118 xQueueReset() function prototype

Summary

Resets a queue to its original empty state. Any data contained in the queue at the time it is reset is discarded.

Parameters

xQueue The handle of the queue that is being reset. The queue handle will have been

returned from the call to xQueueCreate() used to create the queue.

Return Values

Original versions of xQueueReset() returned pdPASS or pdFAIL. Since FreeRTOS V7.2.0 xQueueReset() always returns pdPASS.

# 3.18 xQueueSelectFromSet()

#include “FreeRTOS.h”

#include “queue.h”

QueueSetMemberHandle\_t xQueueSelectFromSet( QueueSetHandle\_t xQueueSet, const TickType\_t xTicksToWait );

Listing 119 xQueueSelectFromSet() function prototype

Summary

xQueueSelectFromSet() selects from the members of a queue set a queue or semaphore that either contains data (in the case of a queue) or is available to take (in the case of a semaphore). xQueueSelectFromSet() effectively allows a task to block (pend) on a read operation on all the queues and semaphores in a queue set simultaneously.

Parameters xQueueSet The queue set on which the task will (potentially) block.

xTicksToWait The maximum time, in ticks, that the calling task will remain in the Blocked

state (with other tasks executing) to wait for a member of the queue set to be ready for a successful queue read or semaphore take operation.

Return Values

NULL A queue or semaphore contained in the set did not become available before the

block time specified by the xTicksToWait parameter expired.

Any The handle of a queue (cast to a QueueSetMemberHandle\_t type) contained in the other queue set that contains data, or the handle of a semaphore (cast to a value QueueSetMemberHandle\_t type) contained in the queue set that is available.

Notes configUSE\_QUEUE\_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueSelectFromSet() API function to be available.

There are simpler alternatives to using queue sets. See the Blocking on Multiple Objects page on the FreeRTOS.org website for more information.

Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Example

See the example provided for the xQueueCreateSet() function in this manual.

# 3.19 xQueueSelectFromSetFromISR()

#include “FreeRTOS.h”

#include “queue.h”

QueueSetMemberHandle\_t xQueueSelectFromSetFromISR( QueueSetHandle\_t xQueueSet );

Listing 120 xQueueSelectFromSetFromISR() function prototype

Summary

A version of xQueueSelectFromSet() that can be used from an interrupt service routine.

Parameters

xQueueSet The queue set being queried. It is not possible to block on a read as this function

is designed to be used from an interrupt.

Return Values

NULL No members of the queue set were available.

Any The handle of a queue (cast to a QueueSetMemberHandle\_t type) contained in the other queue set that contains data, or the handle of a semaphore (cast to a value QueueSetMemberHandle\_t type) contained in the queue set that is available.

Notes configUSE\_QUEUE\_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueSelectFromSetFromISR() API function to be available.

Example

void vReceiveFromQueueInSetFromISR( void )

{

QueueSetMemberHandle\_t xActivatedQueue; unsigned long ulReceived;

/\* See if any of the queues in the set contain data. \*/ xActivatedQueue = xQueueSelectFromSetFromISR( xQueueSet );

if( xActivatedQueue != NULL )

{

/\* Reading from the queue returned by xQueueSelectFromSetFormISR(). \*/ if( xQueueReceiveFromISR( xActivatedQueue, &ulReceived, NULL ) != pdPASS )

{

/\* Data should have been available as the handle was returned from xQueueSelectFromSetFromISR(). \*/

}

}

}

Listing 121 Example use of xQueueSelectFromSetFromISR()

# 3.20 xQueueSend(), xQueueSendToFront(), xQueueSendToBack()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueSend( QueueHandle\_t xQueue, const void \* pvItemToQueue, TickType\_t xTicksToWait );

BaseType\_t xQueueSendToFront( QueueHandle\_t xQueue, const void \* pvItemToQueue, TickType\_t xTicksToWait );

BaseType\_t xQueueSendToBack( QueueHandle\_t xQueue, const void \* pvItemToQueue, TickType\_t xTicksToWait );

Listing 122 xQueueSend(), xQueueSendToFront() and xQueueSendToBack() function prototypes

Summary

Sends (writes) an item to the front or the back of a queue.

xQueueSend() and xQueueSendToBack() perform the same operation so are equivalent. Both send data to the back of a queue. xQueueSend() was the original version, and it is now recommended to use xQueueSendToBack() in its place.

Parameters xQueue The handle of the queue to which the data is being sent (written). The

queue handle will have been returned from the call to xQueueCreate() used to create the queue

pvItemToQueue A pointer to the data to be copied into the queue.

The size of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.

xTicksToWait The maximum amount of time the task should remain in the Blocked state to

wait for space to become available on the queue, should the queue already be full.

xQueueSend(), xQueueSendToFront() and xQueueSendToBack() will return immediately if xTicksToWait is zero and the queue is already full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Return Values

|  |  |
| --- | --- |
| pdPASS | Returned if data was successfully sent to the queue.  If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for space to become available in the queue before the function returned, but data was successfully written to the queue before the block time expired. |
| errQUEUE\_FULL | Returned if data could not be written to the queue because the queue was |

already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for another task or interrupt to make room in the queue, but the specified block time expired before that happened.

Notes None.

Example

/\* Define the data type that will be queued. \*/ typedef struct A\_Message

char ucMessageID; char ucData[ 20 ];

} AMessage;

/\* Define the queue parameters. \*/

#define QUEUE\_LENGTH 5 #define QUEUE\_ITEM\_SIZE sizeof( AMessage )

int main( void )

{

QueueHandle\_t xQueue;

/\* Create the queue, storing the returned handle in the xQueue variable. \*/ xQueue = xQueueCreate( QUEUE\_LENGTH, QUEUE\_ITEM\_SIZE ); if( xQueue == NULL )

{

/\* The queue could not be created – do something. \*/

}

/\* Create a task, passing in the queue handle as the task parameter. \*/ xTaskCreate( vAnotherTask, “Task”,

STACK\_SIZE,

( void \* ) xQueue, /\* xQueue is used as the task parameter. \*/

TASK\_PRIORITY, NULL );

/\* Start the task executing. \*/ vTaskStartScheduler();

/\* Execution will only reach here if there was not enough FreeRTOS heap memory remaining for the idle task to be created. \*/ for( ;; );

}

void vATask( void \*pvParameters )

{

QueueHandle\_t xQueue;

AMessage xMessage;

/\* The queue handle is passed into this task as the task parameter. Cast the parameter back to a queue handle. \*/ xQueue = ( QueueHandle\_t ) pvParameters;

for( ;; )

{

/\* Create a message to send on the queue. \*/ xMessage.ucMessageID = SEND\_EXAMPLE;

/\* Send the message to the queue, waiting for 10 ticks for space to become available if the queue is already full. \*/

if( xQueueSendToBack( xQueue, &xMessage, 10 ) != pdPASS ) {

/\* Data could not be sent to the queue even after waiting 10 ticks. \*/

}

}

}

Listing 123 Example use of xQueueSendToBack()

3.21 xQueueSendFromISR(), xQueueSendToBackFromISR(),

# xQueueSendToFrontFromISR()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xQueueSendFromISR( QueueHandle\_t xQueue, const void \*pvItemToQueue,

BaseType\_t \*pxHigherPriorityTaskWoken );

BaseType\_t xQueueSendToBackFromISR( QueueHandle\_t xQueue, const void \*pvItemToQueue,

BaseType\_t \*pxHigherPriorityTaskWoken );

BaseType\_t xQueueSendToFrontFromISR( QueueHandle\_t xQueue, const void \*pvItemToQueue,

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 124 xQueueSendFromISR(), xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() function prototypes

Summary

Versions of the xQueueSend(), xQueueSendToFront() and xQueueSendToBack() API functions that can be called from an ISR. Unlike xQueueSend(), xQueueSendToFront() and xQueueSendToBack(), the ISR safe versions do not permit a block time to be specified.

xQueueSendFromISR() and xQueueSendToBackFromISR() perform the same operation so are equivalent. Both send data to the back of a queue. xQueueSendFromISR() was the original version and it is now recommended to use xQueueSendToBackFromISR() in its place.

Parameters xQueue The handle of the queue to which the data is being sent

(written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.

pvItemToQueue A pointer to the data to be copied into the queue.

The size of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.

pxHigherPriorityTaskWoken It is possible that a single queue will have one or more tasks blocked on it waiting for data to become available. Calling xQueueSendFromISR(), xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() can make data available, and so cause such a task to leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set \*pxHigherPriorityTaskWoken to pdTRUE. If xQueueSendFromISR(), xQueueSendToFrontFromISR() or

xQueueSendToBackFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task.

From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values pdTRUE Data was successfully sent to the queue. errQUEUE\_FULL Data could not be sent to the queue because the queue was already full.

Notes

Calling xQueueSendFromISR(), xQueueSendToBackFromISR() or xQueueSendToFrontFromISR() within an interrupt service routine can potentially cause a task that was blocked on a queue to leave the Blocked state. A context switch should be performed if such an unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted). The context switch will ensure that the interrupt returns directly to the highest priority Ready state task. Unlike the xQueueSend(), xQueueSendToBack() and xQueueSendToFront() API functions, xQueueSendFromISR(), xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() will not themselves perform a context switch. They will instead just indicate whether or not a context switch is required.

xQueueSendFromISR(), xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() must not be called prior to the scheduler being started. Therefore an interrupt that calls any of these functions must not be allowed to execute prior to the scheduler being started.

Example

For clarity of demonstration, the following example makes multiple calls to xQueueSendToBackFromISR() to send multiple small data items. This is inefficient and therefore not recommended. Preferable approaches include:

1. Packing the multiple data items into a structure, then using a single call to xQueueSendToBackFromISR() to send the entire structure to the queue. This approach is only appropriate if the number of data items is small.
2. Writing the data items into a circular RAM buffer, then using a single call to xQueueSendToBackFromISR() to let a task know how many new data items the buffer contains.

/\* vBufferISR() is an interrupt service routine that empties a buffer of values, writing each value to a queue. It might be that there are multiple tasks blocked on the queue waiting for the data. \*/ void vBufferISR( void )

{ char cIn;

BaseType\_t xHigherPriorityTaskWoken;

/\* No tasks have yet been unblocked. \*/ xHigherPriorityTaskWoken = pdFALSE;

/\* Loop until the buffer is empty. \*/ do

{

/\* Obtain a byte from the buffer. \*/ cIn = INPUT\_BYTE( RX\_REGISTER\_ADDRESS );

/\* Write the byte to the queue. xHigherPriorityTaskWoken will get set to pdTRUE if writing to the queue causes a task to leave the Blocked state, and the task leaving the Blocked state has a priority higher than the currently executing task (the task that was interrupted). \*/ xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

} while( INPUT\_BYTE( BUFFER\_COUNT ) );

/\* Clear the interrupt source here. \*/

/\* Now the buffer is empty, and the interrupt source has been cleared, a context switch should be performed if xHigherPriorityTaskWoken is equal to pdTRUE.

NOTE: The syntax required to perform a context switch from an ISR varies from port to port, and from compiler to compiler. Check the web documentation and examples for the port being used to find the syntax required for your application. \*/

portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );

}

Listing 125 Example use of xQueueSendToBackFromISR()

# 3.22 uxQueueSpacesAvailable()

#include “FreeRTOS.h”

#include “queue.h”

UBaseType\_t uxQueueSpacesAvailable( const QueueHandle\_t xQueue );

Listing 126 uxQueueSpacesAvailable() function prototype

Summary

Returns the number of free spaces that are available in a queue. That is, the number of items that can be posted to the queue before the queue becomes full.

Parameters xQueue The handle of the queue being queried.

Returned Value

The number of free spaces that are available in the queue being queried at the time uxQueueSpacesAvailable() is called.

Example

void vAFunction( QueueHandle\_t xQueue )

{

UBaseType\_t uxNumberOfFreeSpaces;

/\* How many free spaces are currently available in the queue referenced by the xQueue handle? \*/

uxNumberOfFreeSpaces = uxQueueSpacesAvailable( xQueue );

}

Listing 127 Example use of uxQueueSpacesAvailable()

Chapter 4

Semaphore API

# 4.1 vSemaphoreCreateBinary()

#include “FreeRTOS.h”

#include “semphr.h”

void vSemaphoreCreateBinary( SemaphoreHandle\_t xSemaphore );

Listing 128 vSemaphoreCreateBinary() macro prototype

Summary

NOTE: The vSemaphoreCreateBinary() macro remains in the source code to ensure backward compatibility, but it should not be used in new designs. Use the xSemaphoreCreateBinary() function instead.

A macro that creates a binary semaphore. A semaphore must be explicitly created before it can be used.

Parameters

xSemaphore Variable of type SemaphoreHandle\_t that will store the handle of the

semaphore being created.

Return Values None.

If, following a call to vSemaphoreCreateBinary(), xSemaphore is equal to NULL, then the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures. In all other cases, xSemaphore will hold the handle of the created semaphore.

Notes

Binary semaphores and mutexes are very similar, but do have some subtle differences. Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

Binary Semaphores – A binary semaphore used for synchronization does not need to be ‘given’ back after it has been successfully ‘taken’ (obtained). Task synchronization is implemented by having one task or interrupt ‘give’ the semaphore, and another task ‘take’ the semaphore (see the xSemaphoreGiveFromISR() documentation).

Mutexes – The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to ‘inherit’ the priority of the task that is attempting to ‘take’ the same mutex. The inherited priority will be ‘disinherited’ when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back – otherwise no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section of this manual. A fast, small, and simple priority inheritance mechanism is implemented that assumes a task will never hold more than one mutex at a time.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle\_t type, and can be used in any API function that takes a parameter of that type.

Mutexes and binary semaphores that were created using the vSemaphoreCreateBinary() macro (as opposed to the preferred xSemaphoreCreateBinary() function) are both created such that the first call to xSemaphoreTake() on the semaphore or mutex will pass. Binary semaphores created using the xSemaphoreCreateBinary() function are created ‘empty’, so the semaphore must first be given before the semaphore can be taken (obtained) using a call to xSemaphoreTake().

Example

SemaphoreHandle\_t xSemaphore;

void vATask( void \* pvParameters )

{

/\* Attempt to create a semaphore.

NOTE: New designs should use the xSemaphoreCreateBinary() function, not the vSemaphoreCreateBinary() macro. \*/ vSemaphoreCreateBinary( xSemaphore );

if( xSemaphore == NULL )

{

/\* There was insufficient FreeRTOS heap available for the semaphore to be created successfully. \*/

} else

{

/\* The semaphore can now be used. Its handle is stored in the xSemaphore variable. \*/

}

}

Listing 129 Example use of vSemaphoreCreateBinary()

# 4.2 xSemaphoreCreateBinary()

#include “FreeRTOS.h”

#include “semphr.h”

SemaphoreHandle\_t xSemaphoreCreateBinary( void );

Listing 130 xSemaphoreCreateBinary() function prototype

Summary

A function that creates a binary semaphore. A semaphore must be explicitly created before it can be used.

The semaphore is created in the ‘empty’ state, meaning the semaphore must first be given before it can be taken (obtained) using the xSemaphoreTake() function.

Parameters None.

Return Values

NULL Returned if the semaphore cannot be created because there is insufficient

heap memory available for FreeRTOS to allocate the semaphore data structures.

Any other value The semaphore was created successfully. The returned value should be

stored as the handle to the created semaphore.

Notes

Binary semaphores and mutexes are very similar, but do have some subtle differences. Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

Binary Semaphores – A binary semaphore used for synchronization does not need to be

‘given’ back after it has been successfully ‘taken’ (obtained). Task synchronization is implemented by having one task or interrupt ‘give’ the semaphore, and another task ‘take’ the semaphore (see the xSemaphoreGiveFromISR() documentation).

Mutexes – The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to ‘inherit’ the priority of the task that is attempting to ‘take’ the same mutex. The inherited priority will be ‘disinherited’ when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back – otherwise no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section of this manual. A fast, small, and simple priority inheritance mechanism is implemented that assumes a task will never hold more than one mutex at a time.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle\_t type, and can be used in any API function that takes a parameter of that type.

Example

SemaphoreHandle\_t xSemaphore;

void vATask( void \* pvParameters )

{

/\* Attempt to create a semaphore. \*/ xSemaphore = xSemaphoreCreateBinary();

if( xSemaphore == NULL )

{

/\* There was insufficient FreeRTOS heap available for the semaphore to be created successfully. \*/

} else

{

/\* The semaphore can now be used. Its handle is stored in the xSemaphore variable. Calling xSemaphoreTake() on the semaphore here will fail until the semaphore has first been given. \*/

}

}

Listing 131 Example use of xSemaphoreCreateBinary()

# 4.3 xSemaphoreCreateCounting()

#include “FreeRTOS.h”

#include “semphr.h”

SemaphoreHandle\_t xSemaphoreCreateCounting( UBaseType\_t uxMaxCount, UBaseType\_t uxInitialCount );

Listing 132 xSemaphoreCreateCounting() function prototype

Summary

Creates a counting semaphore. A semaphore must be explicitly created before it can be used.

Parameters uxMaxCount The maximum count value that can be reached. When the semaphore

reaches this value it can no longer be ‘given’.

uxInitialCount The count value assigned to the semaphore when it is created.

Return Values

NULL Returned if the semaphore cannot be created because there is insufficient

heap memory available for FreeRTOS to allocate the semaphore data structures.

Any other value The semaphore was created successfully. The returned value should be

stored as the handle to the created semaphore.

Notes

Counting semaphores are typically used for two things:

1. Counting events.

In this usage scenario, an event handler will ‘give’ the semaphore each time an event occurs, and a handler task will ‘take’ the semaphore each time it processes an event.

The semaphore’s count value will be incremented each time it is ‘given’ and decremented each time it is ‘taken’. The count value is therefore the difference between the number of events that have occurred and the number of events that have been processed.

Semaphores created to count events should be created with an initial count value of zero, because no events will have been counted prior to the semaphore being created.

1. Resource management.

In this usage scenario, the count value of the semaphore represents the number of resources that are available.

To obtain control of a resource, a task must first successfully ‘take’ the semaphore. The action of ‘taking’ the semaphore will decrement the semaphore’s count value. When the count value reaches zero, no more resources are available, and further attempts to ‘take’ the semaphore will fail.

When a task finishes with a resource, it must ‘give’ the semaphore. The action of ‘giving’ the semaphore will increment the semaphore’s count value, indicating that a resource is available, and allowing future attempts to ‘take’ the semaphore to be successful.

Semaphores created to manage resources should be created with an initial count value equal to the number of resource that are available.

Example

void vATask( void \* pvParameters )

{

SemaphoreHandle\_t xSemaphore;

/\* The semaphore cannot be used before it is created using a call to xSemaphoreCreateCounting(). The maximum value to which the semaphore can count in this example case is set to 10, and the initial value assigned to the count is set to 0. \*/

xSemaphore = xSemaphoreCreateCounting( 10, 0 );

if( xSemaphore != NULL )

{

/\* The semaphore was created successfully. The semaphore can now be used. \*/

}

}

Listing 133 Example use of xSemaphoreCreateCounting()

# 4.4 xSemaphoreCreateMutex()

#include “FreeRTOS.h”

#include “semphr.h”

SemaphoreHandle\_t xSemaphoreCreateMutex( void );

Listing 134 xSemaphoreCreateMutex() function prototype

Summary

Creates a mutex type semaphore. A semaphore must be explicitly created before it can be used.

Parameters

None

Return Values

NULL Returned if the semaphore cannot be created because there is insufficient

heap memory available for FreeRTOS to allocate the semaphore data structures.

Any other value The semaphore was created successfully. The returned value should be

stored as the handle to the created semaphore.

Notes

Binary semaphores and mutexes are very similar, but do have some subtle differences. Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

Binary Semaphores – A binary semaphore used for synchronization does not need to be ‘given’ back after it has been successfully ‘taken’ (obtained). Task synchronization is implemented by having one task or interrupt ‘give’ the semaphore, and another task ‘take’ the semaphore (see the xSemaphoreGiveFromISR() documentation).

Mutexes – The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to ‘inherit’ the priority of the task that is attempting to ‘take’ the same mutex. The inherited priority will be ‘disinherited’ when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back – otherwise no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section of this manual. A fast, small, and simple priority inheritance mechanism is implemented that assumes a task will never hold more than one mutex at a time.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle\_t type, and can be used in any API function that takes a parameter of that type.

Mutexes and binary semaphores are both created such that the first call to xSemaphoreTake() on the semaphore or mutex will pass.

Example

SemaphoreHandle\_t xSemaphore;

void vATask( void \* pvParameters )

{

/\* Attempt to create a mutex type semaphore. \*/ xSemaphore = xSemaphoreCreateMutex();

if( xSemaphore == NULL )

{

/\* There was insufficient heap memory available for the mutex to be created. \*/

} else

{

/\* The mutex can now be used. The handle of the created mutex will be stored in the xSemaphore variable. \*/

}

}

Listing 135 Example use of xSemaphoreCreateMutex()

# 4.5 xSemaphoreCreateRecursiveMutex()

#include “FreeRTOS.h”

#include “semphr.h”

SemaphoreHandle\_t xSemaphoreCreateRecursiveMutex( void );

Listing 136 xSemaphoreCreateRecursiveMutex() function prototype

Summary

Creates a recursive mutex type semaphore. A semaphore must be explicitly created before it can be used.

Parameters None.

Return Values

NULL Returned if the semaphore cannot be created because there is insufficient

heap memory available for FreeRTOS to allocate the semaphore data structures.

Any other value The semaphore was created successfully. The returned value should be

stored as the handle to the created semaphore.

Notes

A recursive mutex is ‘taken’ using the xSemaphoreTakeRecursive() function, and ‘given’ using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, once a recursive mutex has been successfully ‘taken’ by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively ‘takes’ the same mutex five times, then the mutex will not be available to any other task until the task that successfully obtained the mutex has also ‘given’ the mutex back exactly five times.

As with standard mutexes, a recursive mutex can only be held/obtained by a single task at any one time.

The priority of a task that holds a recursive mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the recursive mutex is said to ‘inherit’ the priority of the task that is attempting to ‘take’ the same mutex. The inherited priority will be ‘disinherited’ when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

Example

void vATask( void \* pvParameters )

{

SemaphoreHandle\_t xSemaphore;

/\* Recursive semaphores cannot be used before being explicitly created using a call to xSemaphoreCreateRecursiveMutex(). \*/ xSemaphore = xSemaphoreCreateRecursiveMutex();

if( xSemaphore != NULL )

{

/\* The recursive mutex semaphore was created successfully and its handle will be stored in xSemaphore variable. The recursive mutex can now be used. \*/

}

}

Listing 137 Example use of xSemaphoreCreateRecursiveMutex()

# 4.6 vSemaphoreDelete()

#include “FreeRTOS.h”

#include “semphr.h”

void vSemaphoreDelete( SemaphoreHandle\_t xSemaphore );

Listing 138 vSemaphoreDelete() function prototype

Summary

Deletes a semaphore that was previously created using a call to vSemaphoreCreateBinary(), xSemaphoreCreateCounting(), xSemaphoreCreateRecursiveMutex(), or xSemaphoreCreateMutex().

Parameters xSemaphore The handle of the semaphore being deleted.

Return Values

None

Notes

Tasks can opt to block on a semaphore (with an optional timeout) if they attempt to obtain a semaphore that is not available. A semaphore must not be deleted if there are any tasks currently blocked on it.

# 4.7 xSemaphoreGetMutexHolder()

#include “FreeRTOS.h”

#include “semphr.h”

TaskHandle\_t xSemaphoreGetMutexHolder( SemaphoreHandle\_t xMutex );

Listing 139 xSemaphoreGetMutexHolder() function prototype

Summary

Return the handle of the task that holds the mutex specified by the function parameter, if any.

Parameters xMutex The handle of the mutex being queried.

Return Values

NULL Either:

* The semaphore specified by the xMutex parameter is not a mutex type semaphore, or
* The semaphore is available, and not held by any task.

Any other value The handle of the task that holds the semaphore specified by the xMutex

parameter.

Notes xSemaphoreGetMutexHolder() can be used reliably to determine if the calling task is the mutex holder, but cannot be used reliably if the mutex is held by any task other than the calling task. This is because the mutex holder might change between the calling task calling the function, and the calling task testing the function’s return value.

configUSE\_MUTEXES and INCLUDE\_xSemaphoreGetMutexHolder must both be set to 1 in FreeRTOSConfig.h for xSemaphoreGetMutexHolder() to be available.

# 4.8 xSemaphoreGive()

#include “FreeRTOS.h”

#include “semphr.h”

BaseType\_t xSemaphoreGive( SemaphoreHandle\_t xSemaphore );

Listing 140 xSemaphoreGive() function prototype

Summary

‘Gives’ (or releases) a semaphore that has previously been created using a call to vSemaphoreCreateBinary(), xSemaphoreCreateCounting() or xSemaphoreCreateMutex() – and has also been successfully ‘taken’.

Parameters xSemaphore The Semaphore being ‘given’. A semaphore is referenced by a variable of type SemaphoreHandle\_t and must be explicitly created before being used.

Return Values pdPASS The semaphore ‘give’ operation was successful.

pdFAIL The semaphore ‘give’ operation was not successful because the task calling

xSemaphoreGive() is not the semaphore holder. A task must successfully ‘take’ a semaphore before it can successfully ‘give’ it back.

Notes None.

Example

SemaphoreHandle\_t xSemaphore = NULL;

void vATask( void \* pvParameters )

{

/\* A semaphore is going to be used to guard a shared resource. In this case a mutex type semaphore is created because it includes priority inheritance functionality. \*/

xSemaphore = xSemaphoreCreateMutex();

for( ;; ) {

if( xSemaphore != NULL )

{

if( xSemaphoreGive( xSemaphore ) != pdTRUE )

{

/\* This call should fail because the semaphore has not yet been

‘taken’. \*/

}

/\* Obtain the semaphore – don’t block if the semaphore is not immediately available (the specified block time is zero). \*/ if( xSemaphoreTake( xSemaphore, 0 ) == pdPASS )

{

/\* The semaphore was ‘taken’ successfully, so the resource it is guarding can be accessed safely. \*/

/\* ... \*/

/\* Access to the resource the semaphore is guarding is complete, so the semaphore must be ‘given’ back. \*/ if( xSemaphoreGive( xSemaphore ) != pdPASS )

{

/\* This call should not fail because the calling task has already successfully ‘taken’ the semaphore. \*/ }

} } else

{

/\* The semaphore was not created successfully because there is not enough FreeRTOS heap remaining for the semaphore data structures to be allocated. \*/

}

}

}

Listing 141 Example use of xSemaphoreGive()

# 4.9 xSemaphoreGiveFromISR()

#include “FreeRTOS.h”

#include “semphr.h”

BaseType\_t xSemaphoreGiveFromISR( SemaphoreHandle\_t xSemaphore,

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 142 xSemaphoreGiveFromISR() function prototype

Summary

A version of xSemaphoreGive() that can be used in an ISR. Unlike xSemaphoreGive(), xSemaphoreGiveFromISR() does not permit a block time to be specified.

Parameters xSemaphore The semaphore being ‘given’.

A semaphore is referenced by a variable of type SemaphoreHandle\_t and must be explicitly created before being used.

\*pxHigherPriorityTaskWoken It is possible that a single semaphore will have one or more

tasks blocked on it waiting for the semaphore to become available. Calling xSemaphoreGiveFromISR() can make the semaphore available, and so cause such a task to leave the Blocked state. If calling xSemaphoreGiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted), then, internally, xSemaphoreGiveFromISR() will set

\*pxHigherPriorityTaskWoken to pdTRUE.

If xSemaphoreGiveFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task.

From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an

optional parameter and can be set to NULL.

Return Values pdTRUE The call to xSemaphoreGiveFromISR() was successful.

errQUEUE\_FULL If a semaphore is already available, it cannot be given, and

xSemaphoreGiveFromISR() will return errQUEUE\_FULL.

Notes

Calling xSemaphoreGiveFromISR() within an interrupt service routine can potentially cause a task that was blocked waiting to take the semaphore to leave the Blocked state. A context switch should be performed if such an unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted). The context switch will ensure that the interrupt returns directly to the highest priority Ready state task.

Unlike the xSemaphoreGive() API function, xSemaphoreGiveFromISR() will not itself perform a context switch. It will instead just indicate whether or not a context switch is required.

xSemaphoreGiveFromISR() must not be called prior to the scheduler being started. Therefore an interrupt that calls xSemaphoreGiveFromISR() must not be allowed to execute prior to the scheduler being started.

Example

#define LONG\_TIME 0xffff

#define TICKS\_TO\_WAIT 10

SemaphoreHandle\_t xSemaphore = NULL;

/\* Define a task that performs an action each time an interrupt occurs. The

Interrupt processing is deferred to this task. The task is synchronized with the interrupt using a semaphore. \*/ void vATask( void \* pvParameters )

{

/\* It is assumed the semaphore has already been created outside of this task. \*/

for( ;; )

{

/\* Wait for the next event. \*/

if( xSemaphoreTake( xSemaphore, portMAX\_DELAY ) == pdTRUE )

{

/\* The event has occurred, process it here. \*/

...

/\* Processing is complete, return to wait for the next event. \*/

}

}

}

/\* An ISR that defers its processing to a task by using a semaphore to indicate when events that require processing have occurred. \*/ void vISR( void \* pvParameters )

{

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

/\* The event has occurred, use the semaphore to unblock the task so the task can process the event. \*/

xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

/\* Clear the interrupt here. \*/

/\* Now the task has been unblocked a context switch should be performed if xHigherPriorityTaskWoken is equal to pdTRUE. NOTE: The syntax required to perform a context switch from an ISR varies from port to port, and from compiler to compiler. Check the web documentation and examples for the port being used to find the syntax required for your application. \*/ portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );

}

Listing 143 Example use of xSemaphoreGiveFromISR()

# 4.10 xSemaphoreGiveRecursive()

#include “FreeRTOS.h”

#include “semphr.h”

BaseType\_t xSemaphoreGiveRecursive( SemaphoreHandle\_t xMutex );

Listing 144 xSemaphoreGiveRecursive() function prototype

Summary

‘Gives’ (or releases) a recursive mutex type semaphore that has previously been created using xSemaphoreCreateRecursiveMutex().

Parameters xMutex The semaphore being ‘given’. A semaphore is referenced by a variable of type SemaphoreHandle\_t and must be explicitly created before being used.

Return Values pdPASS The call to xSemaphoreGiveRecursive() was successful.

pdFAIL The call to xSemaphoreGiveRecursive() failed because the calling task is not the

mutex holder.

Notes

A recursive mutex is ‘taken’ using the xSemaphoreTakeRecursive() function, and ‘given’ using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, once a recursive mutex has been successfully ‘taken’ by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively ‘takes’ the same mutex five times, then the mutex will not be available to any other task until the task that successfully obtained the mutex has also ‘given’ the mutex back exactly five times.

xSemaphoreGiveRecursive() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

xSemaphoreGiveRecursive() must not be called from within a critical section or while the scheduler is suspended.

Example

/\* A task that creates a recursive mutex. \*/ void vATask( void \* pvParameters )

{

/\* Recursive mutexes cannot be used before being explicitly created using a call to xSemaphoreCreateRecursiveMutex(). \*/ xMutex = xSemaphoreCreateRecursiveMutex();

/\* Rest of task code goes here. \*/ for( ;; )

{

}

}

/\* A function (called by a task) that uses the mutex. \*/ void vAFunction( void )

{

/\* ... Do other things. \*/

if( xMutex != NULL )

{

/\* See if the mutex can be obtained. If the mutex is not available wait 10 ticks to see if it becomes free. \*/

if( xSemaphoreTakeRecursive( xMutex, 10 ) == pdTRUE )

{

/\* The mutex was successfully ‘taken’. \*/

...

/\* For some reason, due to the nature of the code, further calls to xSemaphoreTakeRecursive() are made on the same mutex. In real code these would not be just sequential calls, as that would serve no purpose. Instead, the calls are likely to be buried inside a more complex call structure, for example in a TCP/IP stack.\*/ xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 ); xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 );

/\* The mutex has now been ‘taken’ three times, so will not be available to another task until it has also been given back three times. Again it is unlikely that real code would have these calls sequentially, but instead buried in a more complex call structure. This is just for illustrative purposes. \*/ xSemaphoreGiveRecursive( xMutex ); xSemaphoreGiveRecursive( xMutex ); xSemaphoreGiveRecursive( xMutex );

/\* Now the mutex can be taken by other tasks. \*/

} else

{

/\* The mutex was not successfully ‘taken’. \*/ }

}

}

Listing 145 Example use of xSemaphoreGiveRecursive()

# 4.11 xSemaphoreTake()

#include “FreeRTOS.h”

#include “semphr.h”

BaseType\_t xSemaphoreTake( SemaphoreHandle\_t xSemaphore, TickType\_t xTicksToWait );

Listing 146 xSemaphoreTake() function prototype

Summary

‘Takes’ (or obtains) a semaphore that has previously been created using a call to vSemaphoreCreateBinary(), xSemaphoreCreateCounting() or xSemaphoreCreateMutex().

Parameters

|  |  |
| --- | --- |
| xSemaphore | The semaphore being ‘taken’. A semaphore is referenced by a variable of type SemaphoreHandle\_t and must be explicitly created before being used. |
| xTicksToWait | The maximum amount of time the task should remain in the Blocked state to |

wait for the semaphore to become available, if the semaphore is not available immediately.

If xTicksToWait is zero, then xSemaphoreTake() will return immediately if the semaphore is not available.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Return Values pdPASS Returned only if the call to xSemaphoreTake() was successful in obtaining the

semaphore.

If a block time was specified (xTicksToWait was not zero), then it is possible that

the calling task was placed into the Blocked state to wait for the semaphore if it was not immediately available, but the semaphore became available before the block time expired.

pdFAIL Returned if the call to xSemaphoreTake() did not successfully obtain the

semaphore.

If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the semaphore to become available, but the block time expired before this happened.

Notes xSemaphoreTake() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

xSemaphoreTake() must not be called from within a critical section or while the scheduler is suspended.

Example

SemaphoreHandle\_t xSemaphore = NULL;

/\* A task that creates a mutex type semaphore. \*/ void vATask( void \* pvParameters )

{

/\* A semaphore is going to be used to guard a shared resource. In this case a mutex type semaphore is created because it includes priority inheritance functionality. \*/

xSemaphore = xSemaphoreCreateMutex();

/\* The rest of the task code goes here. \*/ for( ;; )

{

/\* ... \*/

}

}

/\* A task that uses the mutex. \*/ void vAnotherTask( void \* pvParameters )

{

for( ;; )

{

/\* ... Do other things. \*/

if( xSemaphore != NULL )

{

/\* See if the mutex can be obtained. If the mutex is not available wait 10 ticks to see if it becomes free. \*/ if( xSemaphoreTake( xSemaphore, 10 ) == pdTRUE ) {

/\* The mutex was successfully obtained so the shared resource can be accessed safely. \*/

/\* ... \*/

/\* Access to the shared resource is complete, so the mutex is returned. \*/

xSemaphoreGive( xSemaphore );

} else

{

/\* The mutex could not be obtained even after waiting 10 ticks, so the shared resource cannot be accessed. \*/ }

}

}

}

Listing 147 Example use of xSemaphoreTake()

# 4.12 xSemaphoreTakeFromISR()

#include “FreeRTOS.h”

#include “queue.h”

BaseType\_t xSemaphoreTakeFromISR( SemaphoreHandle\_t xSemaphore,

signed BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 148 xSemaphoreTakeFromISR() function prototype

Summary

A version of xSemaphoreTake() that can be called from an ISR. Unlike xSemaphoreTake(), xSemaphoreTakeFromISR() does not permit a block time to be specified.

Parameters xSemaphore The semaphore being ‘taken’. A semaphore is referenced by a

variable of type SemaphoreHandle\_t and must be explicitly created before being used.

pxHigherPriorityTaskWoken It is possible (although unlikely, and dependent on the

semaphore type) that a semaphore will have one or more tasks blocked on it waiting to give the semaphore. Calling xSemaphoreTakeFromISR() will make a task that was blocked waiting to give the semaphore leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set \*pxHigherPriorityTaskWoken to pdTRUE.

If xSemaphoreTakeFromISR() sets

\*pxHigherPriorityTaskWoken to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. The mechanism is identical to that used in the xQueueReceiveFromISR() function, and readers are referred to the xQueueReceiveFromISR() documentation for further explanation.

From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values pdPASS The semaphore was successfully taken (acquired). pdFAIL The semaphore was not successfully taken because it was not available.

# 4.13 xSemaphoreTakeRecursive()

#include “FreeRTOS.h”

#include “semphr.h”

BaseType\_t xSemaphoreTakeRecursive( SemaphoreHandle\_t xMutex, TickType\_t xTicksToWait );

Listing 149 xSemaphoreTakeRecursive() function prototype

Summary

‘Takes’ (or obtains) a recursive mutex type semaphore that has previously been created using xSemaphoreCreateRecursiveMutex().

Parameters

|  |  |
| --- | --- |
| xMutex | The semaphore being ‘taken’. A semaphore is referenced by a variable of type SemaphoreHandle\_t and must be explicitly created before being used. |
| xTicksToWait | The maximum amount of time the task should remain in the Blocked state to |

wait for the semaphore to become available, if the semaphore is not available immediately.

If xTicksToWait is zero, then xSemaphoreTakeRecursive() will return immediately if the semaphore is not available.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Return Values pdPASS Returned only if the call to xSemaphoreTakeRecursive() was successful in

obtaining the semaphore.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for the semaphore if it was not immediately available, but the semaphore became available before the block time expired.

pdFAIL Returned if the call to xSemaphoreTakeRecursive() did not successfully obtain the

semaphore.

If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the semaphore to become available, but the block time expired before this happened.

Notes

A recursive mutex is ‘taken’ using the xSemaphoreTakeRecursive() function, and ‘given’ using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, once a recursive mutex has been successfully ‘taken’ by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively ‘takes’ the same mutex five times, then the mutex will not be available to any other task until the task that successfully obtained the mutex has also ‘given’ the mutex back exactly five times.

xSemaphoreTakeRecursive() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

xSemaphoreTakeRecursive() must not be called from within a critical section or while the scheduler is suspended.

Example

/\* A task that creates a recursive mutex. \*/ void vATask( void \* pvParameters )

{

/\* Recursive mutexes cannot be used before being explicitly created using a call to xSemaphoreCreateRecursiveMutex(). \*/ xMutex = xSemaphoreCreateRecursiveMutex();

/\* Rest of task code goes here. \*/ for( ;; )

{

}

}

/\* A function (called by a task) that uses the mutex. \*/ void vAFunction( void )

{

/\* ... Do other things. \*/

if( xMutex != NULL )

{

/\* See if the mutex can be obtained. If the mutex is not available wait 10 ticks to see if it becomes free. \*/

if( xSemaphoreTakeRecursive( xMutex, 10 ) == pdTRUE )

{

/\* The mutex was successfully ‘taken’. \*/

...

/\* For some reason, due to the nature of the code, further calls to xSemaphoreTakeRecursive() are made on the same mutex. In real code these would not be just sequential calls, as that would serve no purpose. Instead, the calls are likely to be buried inside a more complex call structure, for example in a TCP/IP stack.\*/ xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 ); xSemaphoreTakeRecursive( xMutex, ( TickType\_t ) 10 );

/\* The mutex has now been ‘taken’ three times, so will not be available to another task until it has also been given back three times. Again it is unlikely that real code would have these calls sequentially, but instead buried in a more complex call structure. This is just for illustrative purposes. \*/ xSemaphoreGiveRecursive( xMutex ); xSemaphoreGiveRecursive( xMutex ); xSemaphoreGiveRecursive( xMutex );

/\* Now the mutex can be taken by other tasks. \*/

} else

{

/\* The mutex was not successfully ‘taken’. \*/ }

}

}

Listing 150 Example use of xSemaphoreTakeRecursive()

Chapter 5

Software Timer API

# 5.1 xTimerChangePeriod()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerChangePeriod( TimerHandle\_t xTimer,

TickType\_t xNewPeriod,

TickType\_t xTicksToWait );

Listing 151 xTimerChangePeriod() function prototype

Summary

Changes the period of a timer. xTimerChangePeriodFormISR() is an equivalent function that can be called from an interrupt service routine.

If xTimerChangePeriod() is used to change the period of a timer that is already running, then the timer will use the new period value to recalculate its expiry time. The recalculated expiry time will then be relative to when xTimerChangePeriod() was called, and not relative to when the timer was originally started.

If xTimerChangePeriod() is used to change the period of a timer that is not already running, then the timer will use the new period value to calculate an expiry time, and the timer will start running.

Parameters

|  |  |
| --- | --- |
| xTimer | The timer to which the new period is being assigned. |
| xNewPeriod | The new period for the timer referenced by the xTimer parameter.  Timer periods are specified in multiples of tick periods. The  pdMS\_TO\_TICKS() macro can be used to convert a time in milliseconds to a time in ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to pdMS\_TO\_TICKS( 500 ), provided configTICK\_RATE\_HZ is less than or equal to 1000. |
| xTicksToWait | Timer functionality is not provided by the core FreeRTOS code, but by a timer |

service (or daemon) task. The FreeRTOS timer API sends commands to the

timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. As with the xNewPeriod parameter, The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xTicksToWait is ignored if xTimerChangePeriod() is called before the scheduler is started.

Return Values pdPASS The change period command was successfully sent to the timer command queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired.

When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer’s expiry time is relative to when xTimerChangePeriod() is actually called. The priority of the timer service task is set by the configTIMER\_TASK\_PRIORITY configuration constant.

pdFAIL The change period command was not sent to the timer command queue because

the queue was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerChangePeriod() to be available.

Example

/\* This function assumes xTimer has already been created. If the timer referenced by xTimer is already active when it is called, then the timer is deleted. If the timer referenced by xTimer is not active when it is called, then the period of the timer is set to 500ms, and the timer is started. \*/ void vAFunction( TimerHandle\_t xTimer )

{

if( xTimerIsTimerActive( xTimer ) != pdFALSE )

{

/\* xTimer is already active - delete it. \*/ xTimerDelete( xTimer );

} else

{

/\* xTimer is not active, change its period to 500ms. This will also cause the timer to start. Block for a maximum of 100 ticks if the change period command cannot immediately be sent to the timer command queue. \*/ if( xTimerChangePeriod( xTimer, pdMS\_TO\_TICKS( 500 ), 100 ) == pdPASS ) {

/\* The command was successfully sent. \*/

} else

{

/\* The command could not be sent, even after waiting for 100 ticks to pass. Take appropriate action here. \*/

}

}

}

Listing 152 Example use of xTimerChangePeriod()

# 5.2 xTimerChangePeriodFromISR()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerChangePeriodFromISR( TimerHandle\_t xTimer,

TickType\_t xNewPeriod,

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 153 xTimerChangePeriodFromISR() function prototype

Summary

A version of xTimerChangePeriod() that can be called from an interrupt service routine.

Parameters xTimer The timer to which the new period is being assigned.

xNewPeriod The new period for the timer referenced by the xTimer

parameter.

Timer periods are specified in multiples of tick periods. The pdMS\_TO\_TICKS() macro can be used to convert a time in milliseconds to a time in ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to pdMS\_TO\_TICKS( 500 ), provided configTICK\_RATE\_HZ is less than or equal to 1000.

pxHigherPriorityTaskWoken xTimerChangePeriodFromISR() writes a command to the timer

command queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then

\*pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerChangePeriodFromISR() function. If xTimerChangePeriodFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values pdPASS The change period command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer’s expiry time is relative to when xTimerChangePeriodFromISR() is actually called. The priority of the timer service task is set by the configTIMER\_TASK\_PRIORITY configuration constant.

pdFAIL The change period command was not sent to the timer command queue because

the queue was already full.

Notes

configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerChangePeriodFromISR() to be available.

Example

/\* This scenario assumes xTimer has already been created and started. When an interrupt occurs, the period of xTimer should be changed to 500ms. \*/

/\* The interrupt service routine that changes the period of xTimer. \*/ void vAnExampleInterruptServiceRoutine( void ) {

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

/\* The interrupt has occurred - change the period of xTimer to 500ms.

xHigherPriorityTaskWoken was set to pdFALSE where it was defined (within this function). As this is an interrupt service routine, only FreeRTOS API functions that end in "FromISR" can be used. \*/

if( xTimerChangePeriodFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS ) {

/\* The command to change the timer’s period was not executed successfully.

Take appropriate action here. \*/

}

/\* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be performed. The syntax required to perform a context switch from inside an ISR varies from port to port, and from compiler to compiler. Inspect the demos for the port you are using to find the actual syntax required. \*/ if( xHigherPriorityTaskWoken != pdFALSE )

{

/\* Call the interrupt safe yield function here (actual function depends on the FreeRTOS port being used). \*/

}

}

Listing 154 Example use of xTimerChangePeriodFromISR()

# 5.3 xTimerCreate()

#include “FreeRTOS.h”

#include “timers.h”

TimerHandle\_t xTimerCreate( const char \*pcTimerName, const TickType\_t xTimerPeriod, const UBaseType\_t uxAutoReload, void \* const pvTimerID,

TimerCallbackFunction\_t pxCallbackFunction );

Listing 155 xTimerCreate() function prototype

Summary

Creates and initializes a new instance of a software timer.

Creating a timer does not start the timer running. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to start the timer running.

Parameters

|  |  |
| --- | --- |
| pcTimerName | A plain text name that is assigned to the timer, purely to assist debugging. |
| xTimerPeriod | The timer period.  Timer periods are specified in multiples of tick periods. The  pdMS\_TO\_TICKS() macro can be used to convert a time in milliseconds to a time in ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to  pdMS\_TO\_TICKS( 500 ), provided configTICK\_RATE\_HZ is less than or equal to 1000. |
| uxAutoReload | Set to pdTRUE to create an autoreload timer. Set to pdFALSE to create |

a one-shot timer.

Once started, an autoreload timer will expire repeatedly with a frequency set by the xTimerPeriod parameter.

Once started, a one-shot timer will expire only once. A one-shot timer can be manually restarted after it has expired.

pvTimerID An identifier that is assigned to the timer being created. The identifier

can later be updated using the vTimerSetTimerID() API function.

If the same callback function is assigned to multiple timers, then the timer identifier can be inspected inside the callback function to determine which timer actually expired. In addition, the timer identifier can be used to store a value in between calls to the timer’s callback function.

pxCallbackFunction The function to call when the timer expires. Callback functions must

have the prototype defined by the TimerCallbackFunction\_t typedef.

The required prototype is shown in Listing 156.

void vCallbackFunctionExample( TimerHandle\_t xTimer );

Listing 156 The timer callback function prototype

Return Values

NULL The timer could not be created because there was insufficient FreeRTOS heap

memory available to successfully allocate the timer data structures.

Non-NULL A new timer was created, and the handle of the timer was returned.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerCreate() to be available.

Example

#define NUM\_TIMERS 5

/\* An array to hold handles to the created timers. \*/

TimerHandle\_t xTimers[ NUM\_TIMERS ];

/\* An array to hold a count of the number of times each timer expires. \*/ long lExpireCounters[ NUM\_TIMERS ] = { 0 };

/\* Define a callback function that will be used by multiple timer instances. The callback function does nothing but count the number of times the associated timer expires, and stop the timer once the timer has expired 10 times. \*/ void vTimerCallback( TimerHandle\_t pxTimer )

{

long lArrayIndex;

const long xMaxExpiryCountBeforeStopping = 10;

/\* Optionally do something if the pxTimer parameter is NULL. \*/ configASSERT( pxTimer );

/\* Which timer expired? \*/

lArrayIndex = ( long ) pvTimerGetTimerID( pxTimer );

/\* Increment the number of times the timer has expired. \*/ lExpireCounters[ lArrayIndex ] += 1;

/\* If the timer has expired 10 times, then stop it from running. \*/ if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping )

{

/\* Do not use a block time if calling a timer API function from a timer callback function, as doing so could cause a deadlock! \*/ xTimerStop( pxTimer, 0 );

}

}

Listing 157 Definition of the callback function used in the calls to xTimerCreate() in Listing 158

void main( void )

{ long x;

/\* Create then start some timers. Starting the timers before the scheduler has been started means the timers will start running immediately that the scheduler starts. Care must be taken to ensure the timer command queue is not filled up because the timer service task will not start draining the timer command queue until after the scheduler has been started. \*/ for( x = 0; x < NUM\_TIMERS; x++ )

{

xTimers[ x ] = xTimerCreate( "Timer", /\* Just a text name, not used by the kernel. \*/

( 100 \* x ), /\* The timer period in ticks. \*/ pdTRUE, /\* The timers will auto-reload themselves when they expire. \*/

( void \* ) x, /\* Assign each timer a unique id equal to its array index. \*/

vTimerCallback /\* Each timer calls the same callback when it expires. \*/

); if( xTimers[ x ] == NULL )

{

/\* The timer was not created. \*/

} else

{

/\* Start the timer. No block time is specified, and even if one was it would be ignored because the scheduler has not yet been started. \*/ if( xTimerStart( xTimers[ x ], 0 ) != pdPASS ) {

/\* The start command could not be sent to the timer command queue. \*/

}

}

}

/\* ...

Create tasks here.

... \*/

/\* Starting the scheduler will start the timers running as they have already been set into the active state. \*/ xTaskStartScheduler();

/\* Should not reach here. \*/ for( ;; );

}

Listing 158 Example use of xTimerCreate()

# 5.4 xTimerDelete()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerDelete( TimerHandle\_t xTimer, TickType\_t xTicksToWait );

Listing 159 xTimerDelete() macro prototype

Summary

Deletes a timer. The timer must first have been created using the xTimerCreate() API function.

Parameters

|  |  |
| --- | --- |
| xTimer | The handle of the timer being deleted. |
| xTicksToWait | Timer functionality is not provided by the core FreeRTOS code, but by a timer |

service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xTicksToWait is ignored if xTimerDelete() is called before the scheduler is started

Return Values pdPASS The delete command was successfully sent to the timer command queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired.

When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system. The priority of the timer service task is set by the configTIMER\_TASK\_PRIORITY configuration constant.

pdFAIL The delete command was not sent to the timer command queue because the queue

was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerDelete() to be available.

Example

See the example provided for the xTimerChangePeriod() API function.

# 5.5 xTimerGetTimerDaemonTaskHandle()

#include “FreeRTOS.h”

#include “timers.h”

TaskHandle\_t xTimerGetTimerDaemonTaskHandle( void );

Listing 160 xTimerGetTimerDaemonTaskHandle() function prototype

Summary

Returns the task handle associated with the software timer daemon (or service) task. If configUSE\_TIMERS is set to 1 in FreeRTOSConfig.h, then the timer daemon task is created automatically when the scheduler is started. All FreeRTOS software timer callback functions run in the context of the timer daemon task.

Parameters None.

Return Values

The handle of the timer daemon task. FreeRTOS software timer callback functions run in the context of the software daemon task.

Notes

INCLUDE\_xTimerGetTimerDaemonTaskHandle and configUSE\_TIMERS must both be set to 1 in FreeRTOSConfig.h for xTimerGetTimerDaemonTaskHandle() to be available.

# 5.6 pvTimerGetTimerID()

#include “FreeRTOS.h”

#include “timers.h”

void \*pvTimerGetTimerID( TimerHandle\_t xTimer );

Listing 161 pvTimerGetTimerID() function prototype

Summary

Returns the identifier (ID) assigned to the timer. An identifier is assigned to the timer when the timer is created, and can be updated using the vTimerSetTimerID() API function. See the xTimerCreate() API function for more information.

If the same callback function is assigned to multiple timers, the timer identifier can be inspected inside the callback function to determine which timer actually expired. This is demonstrated in the example code provided for the xTimerCreate() API function.

In addition the timer’s identifier can be used to store values in between calls to the timer’s callback function.

Parameters xTimer The timer being queried.

Return Values

The identifier assigned to the timer being queried.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for pvTimerGetTimerID() to be available.

Example

/\* A callback function assigned to a timer. \*/

void TimerCallbackFunction( TimerHandle\_t pxExpiredTimer )

{

uint32\_t ulCallCount;

/\* A count of the number of times this timer has expired and executed its callback function is stored in the timer's ID. Retrieve the count, increment it, then save it back into the timer's ID. \*/

ulCallCount = ( uint32\_t ) pvTimerGetTimerID( pxExpiredTimer ); ulCallCount++;

vTimerSetTimerID( pxExpiredTimer, ( void \* ) ulCallCount );

}

Listing 162 Example use of pvTimerGetTimerID()

# 5.1 pcTimerGetTimerName()

#include “FreeRTOS.h”

#include “timers.h”

const char \* pcTimerGetTimerName( TimerHandle\_t xTimer );

Listing 163 pcTimerGetTimerName() function prototype

Summary

Returns the human readable text name assigned to the timer when the timer was created.

See the xTimerCreate() API function for more information.

Parameters xTimer The timer being queried.

Return Values

Timer names are standard NULL terminated C strings. The value returned is a pointer to the subject timer’s name.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for pcTimerGetTimerName() to be available.

# 5.2 xTimerIsTimerActive()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerIsTimerActive( TimerHandle\_t xTimer );

Listing 164 xTimerIsTimerActive() function prototype

Summary

Queries a timer to determine if the timer is running.

A timer will not be running if:

1. The timer has been created, but not started.
2. The timer is a one shot timer that has not been restarted since it expired.

The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to start a timer running.

Parameters xTimer The timer being queried.

Return Values pdFALSE The timer is not running.

Any other value The timer is running.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerIsTimerActive() to be available.

Example

/\* This function assumes xTimer has already been created. \*/ void vAFunction( TimerHandle\_t xTimer )

{

/\* The following line could equivalently be written as:

"if( xTimerIsTimerActive( xTimer ) )" \*/ if( xTimerIsTimerActive( xTimer ) != pdFALSE )

{

/\* xTimer is active, do something. \*/

} else

{

/\* xTimer is not active, do something else. \*/

}

}

Listing 165 Example use of xTimerIsTimerActive()

# 5.3 xTimerPendFunctionCall()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerPendFunctionCall( PendedFunction\_t xFunctionToPend, void \*pvParameter1, uint32\_t ulParameter2,

TickType\_t xTicksToWait );

Listing 166 xTimerPendFunctionCall() function prototype

Summary

Used to defer the execution of a function to the RTOS daemon task (also known as the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

This function must not be called from an interrupt service routine. See xTimerPendFunctionCallFromISR() for a version that can be called from an interrupt service routine.

Functions that can be deferred to the RTOS daemon task must have the prototype demonstrated by Listing 167.

void vPendableFunction( void \*pvParameter1, uint32\_t ulParameter2 );

Listing 167 The prototype of a function that can be pended using a call to xTimerPendFunctionCall()

The pvParameter1 and ulParameter2 parameters are provided for use by the application code.

Parameters

|  |  |
| --- | --- |
| xFunctionToPend | The function to execute from the timer service/daemon task. The function must conform to the PendedFunction\_t prototype shown in Listing 167. |
| pvParameter1 | The value to pass into the callback function as the function's first parameter. The parameter has a void \* type to allow it to be used to pass any type. For example, integer types can be cast to a void \*, or the void \* can be used to point to a structure. |
| ulParameter2 | The value to pass into the callback function as the function’s second |

parameter.

xTicksToWait Calling xTimerPendFunctionCall() will result in a message being sent on a

queue to the timer daemon task (also known as the timer service task). xTicksToWait specifies the amount of time the calling task should wait in the Blocked state (so not consuming any processing time) for space to come available on the queue if the queue is full.

Return Values pdPASS The message was successfully sent to the RTOS daemon task.

Any other The message was not sent to the RTOS daemon task because the message value queue was already full. The length of the queue is set by the value of

configTIMER\_QUEUE\_LENGTH in FreeRTOSConfig.h.

Notes

INCLUDE\_xTimerPendFunctionCall() and configUSE\_TIMERS must both be set to 1 in FreeRTOSConfig.h for xTimerPendFunctionCall() to be available.

# 5.4 xTimerPendFunctionCallFromISR()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerPendFunctionCallFromISR( PendedFunction\_t xFunctionToPend, void \*pvParameter1, uint32\_t ulParameter2,

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 168 xTimerPendFunctionCallFromISR() function prototype

Summary

Used from application interrupt service routines to defer the execution of a function to the RTOS daemon task (also known as the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

Ideally an interrupt service routine (ISR) is kept as short as possible, but sometimes an ISR either has a lot of processing to do, or needs to perform processing that is not deterministic. In these cases xTimerPendFunctionCallFromISR() can be used to defer processing of a function to the RTOS daemon task.

A mechanism is provided that allows the interrupt to return directly to the task that will subsequently execute the pended function. This allows the callback function to execute contiguously in time with the interrupt - just as if the callback had executed in the interrupt itself.

Functions that can be deferred to the RTOS daemon task must have the prototype demonstrated by Listing 169.

void vPendableFunction( void \*pvParameter1, uint32\_t ulParameter2 );

Listing 169 The prototype of a function that can be pended using a call to xTimerPendFunctionCallFromISR()

The pvParameter1 and ulParameter2 parameters are provided for use by the application code.

Parameters xFunctionToPend The function to execute from the timer service/daemon task. The function must conform to the PendedFunction\_t prototype shown in Listing 169 .

pvParameter1 The value that will be passed into the callback function as the

function’s first parameter. The parameter has a void \* type to allow it to be used to pass any type. For example, integer types can be cast to a void \*, or the void \* can be used to point to a structure.

ulParameter2 The value that will be passed into the callback function as the

function’s second parameter.

pxHigherPriorityTaskWoken Calling xTimerPendFunctionCallFromISR() will result in a

message being sent on a queue to the RTOS timer daemon task. If the priority of the daemon task (which is set by the value of configTIMER\_TASK\_PRIORITY in FreeRTOSConfig.h) is

higher than the priority of the currently running task (the task the interrupt interrupted) then \*pxHigherPriorityTaskWoken will be set to pdTRUE within xTimerPendFunctionCallFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason \*pxHigherPriorityTaskWoken must be initialized to pdFALSE.

Return Values pdPASS The message was successfully sent to the RTOS daemon task.

Any other The message was not sent to the RTOS daemon task because the message value queue was already full. The length of the queue is set by the value of

configTIMER\_QUEUE\_LENGTH in FreeRTOSConfig.h.

Notes

INCLUDE\_xTimerPendFunctionCall() and configUSE\_TIMERS must both be set to 1 in FreeRTOSConfig.h for xTimerPendFunctionCallFromISR() to be available.

Example

/\* The callback function that will execute in the context of the daemon task. Note callback functions must all use this same prototype. \*/ void vProcessInterface( void \*pvParameter1, uint32\_t ulParameter2 )

{

BaseType\_t xInterfaceToService;

/\* The interface that requires servicing is passed in the second parameter.

The first parameter is not used in this case. \*/ xInterfaceToService = ( BaseType\_t ) ulParameter2;

/\* ...Perform the processing here... \*/

}

/\* An ISR that receives data packets from multiple interfaces \*/ void vAnISR( void )

{

BaseType\_t xInterfaceToService, xHigherPriorityTaskWoken;

/\* Query the hardware to determine which interface needs processing. \*/ xInterfaceToService = prvCheckInterfaces();

/\* The actual processing is to be deferred to a task. Request the vProcessInterface() callback function is executed, passing in the number of the interface that needs processing. The interface to service is passed in the second parameter. The first parameter is not used in this case. \*/ xHigherPriorityTaskWoken = pdFALSE;

xTimerPendFunctionCallFromISR( vProcessInterface,

NULL,

( uint32\_t ) xInterfaceToService,

&xHigherPriorityTaskWoken );

/\* If xHigherPriorityTaskWoken is now set to pdTRUE then a context switch should be requested. The macro used is port specific and will be either portYIELD\_FROM\_ISR() or portEND\_SWITCHING\_ISR() - refer to the documentation page for the port being used. \*/

portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );

}

Listing 170 Example use of xTimerPendFunctionCallFromISR()

# 5.5 xTimerReset()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerReset( TimerHandle\_t xTimer, TickType\_t xTicksToWait );

Listing 171 xTimerReset() function prototype

Summary

Re-starts a timer. xTimerResetFromISR() is an equivalent function that can be called from an interrupt service routine.

If the timer is already running, then the timer will recalculate its expiry time to be relative to when xTimerReset() was called.

If the timer was not running, then the timer will calculate an expiry time relative to when xTimerReset() was called, and the timer will start running. In this case, xTimerReset() is functionally equivalent to xTimerStart().

Resetting a timer ensures the timer is running. If the timer is not stopped, deleted, or reset in the meantime, the callback function associated with the timer will get called ‘n’ ticks after xTimerReset() was called, where ‘n’ is the timer’s defined period.

If xTimerReset() is called before the scheduler is started, then the timer will not start running until the scheduler has been started, and the timer’s expiry time will be relative to when the scheduler started.

Parameters xTimer The timer being reset, started, or restarted.

xTicksToWait Timer functionality is not provided by the core FreeRTOS code, but by a timer

service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xTicksToWait is ignored if xTimerReset() is called before the scheduler is started.

Return Values pdPASS The reset command was successfully sent to the timer command queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired.

When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer’s expiry time is relative to when xTimerReset() is actually called. The priority of the timer service task is set by the configTIMER\_TASK\_PRIORITY configuration constant.

pdFAIL The reset command was not sent to the timer command queue because the queue

was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerReset() to be available.

Example

/\* In this example, when a key is pressed, an LCD back-light is switched on. If 5 seconds pass without a key being pressed, then the LCD back-light is switched off by a one-shot timer. \*/

TimerHandle\_t xBacklightTimer = NULL;

/\* The callback function assigned to the one-shot timer. In this case the parameter is not used. \*/

void vBacklightTimerCallback( TimerHandle\_t pxTimer )

{

/\* The timer expired, therefore 5 seconds must have passed since a key was pressed. Switch off the LCD back-light.

vSetBacklightState( BACKLIGHT\_OFF );

}

/\* The key press event handler. \*/ void vKeyPressEventHandler( char cKey )

{

/\* Ensure the LCD back-light is on, then reset the timer that is responsible for turning the back-light off after 5 seconds of key inactivity. Wait 10 ticks for the reset command to be successfully sent if it cannot be sent immediately. \*/ vSetBacklightState( BACKLIGHT\_ON );

if( xTimerReset( xBacklightTimer, 10 ) != pdPASS )

{

/\* The reset command was not executed successfully. Take appropriate action here. \*/

}

/\* Perform the rest of the key processing here. \*/

}

void main( void ) {

/\* Create then start the one-shot timer that is responsible for turning the back-light off if no keys are pressed within a 5 second period. \*/

xBacklightTimer = xTimerCreate( "BcklghtTmr" /\* Just a text name, not used by the kernel. \*/ pdMS\_TO\_TICKS( 5000 ), /\* The timer period in ticks. \*/ pdFALSE, /\* It is a one-shot timer. \*/ 0, /\* ID not used by the callback so can take any value. \*/ vBacklightTimerCallback /\* The callback function that switches the LCD back-light off. \*/ );

if( xBacklightTimer == NULL )

{

/\* The timer was not created. \*/

} else

{

/\* Start the timer. No block time is specified, and even if one was it would be ignored because the scheduler has not yet been started. \*/ if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )

{

/\* The timer could not be set into the Active state. \*/ }

}

/\* Create tasks here. \*/

/\* Starting the scheduler will start the timer running as xTimerStart has already been called. \*/

xTaskStartScheduler();

}

Listing 172 Example use of xTimerReset()

# 5.6 xTimerResetFromISR()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerResetFromISR( TimerHandle\_t xTimer,

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 173 xTimerResetFromISR() function prototype

Summary

A version of xTimerReset() that can be called from an interrupt service routine.

Parameters xTimer The handle of the timer that is being started, reset, or restarted.

pxHigherPriorityTaskWoken xTimerResetFromISR() writes a command to the timer

command queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then

\*pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerResetFromISR() function. If xTimerResetFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values pdPASS The reset command was successfully sent to the timer command queue. When the

command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer’s expiry time is relative to when xTimerResetFromISR() is actually called. The priority of the timer service task is set by the configTIMER\_TASK\_PRIORITY configuration constant.

pdFAIL The reset command was not sent to the timer command queue because the queue

was already full.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerResetFromISR() to be available.

Example

/\* This scenario assumes xBacklightTimer has already been created. When a key is pressed, an LCD back-light is switched on. If 5 seconds pass without a key being pressed, then the LCD back-light is switched off by a one-shot timer. Unlike the example given for the xTimerReset() function, the key press event handler is an interrupt service routine. \*/

/\* The callback function assigned to the one-shot timer. In this case the parameter is not used. \*/

void vBacklightTimerCallback( TimerHandle\_t pxTimer ) {

/\* The timer expired, therefore 5 seconds must have passed since a key was pressed. Switch off the LCD back-light. \*/ vSetBacklightState( BACKLIGHT\_OFF );

}

/\* The key press interrupt service routine. \*/ void vKeyPressEventInterruptHandler( void )

{

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

/\* Ensure the LCD back-light is on, then reset the timer that is responsible for turning the back-light off after 5 seconds of key inactivity. This is an interrupt service routine so can only call FreeRTOS API functions that end in

"FromISR". \*/

vSetBacklightState( BACKLIGHT\_ON );

/\* xTimerStartFromISR() or xTimerResetFromISR() could be called here as both cause the timer to re-calculate its expiry time. xHigherPriorityTaskWoken was initialized to pdFALSE when it was declared (in this function). \*/

if( xTimerResetFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS ) {

/\* The reset command was not executed successfully. Take appropriate action here. \*/

}

/\* Perform the rest of the key processing here. \*/

/\* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be performed. The syntax required to perform a context switch from inside an ISR varies from port to port, and from compiler to compiler. Inspect the demos for the port you are using to find the actual syntax required. \*/ if( xHigherPriorityTaskWoken != pdFALSE )

{

/\* Call the interrupt safe yield function here (actual function depends on the FreeRTOS port being used). \*/

}

}

Listing 174 Example use of xTimerResetFromISR()

# 5.7 vTimerSetTimerID()

#include “FreeRTOS.h”

#include “timers.h”

void vTimerSetTimerID( TimerHandle\_t xTimer, void \*pvNewID );

Listing 175 vTimerSetTimerID() function prototype

Summary

An identifier (ID) is assigned to a timer when the timer is created, and can be changed at any time using the vTimerSetTimerID() API function.

If the same callback function is assigned to multiple timers, the timer identifier can be inspected inside the callback function to determine which timer actually expired.

The timer identifier can also be used to store data in the timer between calls to the timer’s callback function.

Parameters

xTimer The handle of the timer being updated with a new identifier. pvNewID The value to which the timer’s identifier will be set.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerSetTimerID() to be available.

Example

/\* A callback function assigned to a timer. \*/

void TimerCallbackFunction( TimerHandle\_t pxExpiredTimer )

{

uint32\_t ulCallCount;

/\* A count of the number of times this timer has expired and executed its callback function is stored in the timer's ID. Retrieve the count, increment it, then save it back into the timer's ID. \*/

ulCallCount = ( uint32\_t ) pvTimerGetTimerID( pxExpiredTimer ); ulCallCount++;

vTimerSetTimerID( pxExpiredTimer, ( void \* ) ulCallCount );

}

Listing 176 Example use of vTimerSetTimerID()

# 5.8 xTimerStart()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerStart( TimerHandle\_t xTimer, TickType\_t xTicksToWait );

Listing 177 xTimerStart() function prototype

Summary

Starts a timer running. xTimerStartFromISR() is an equivalent function that can be called from an interrupt service routine.

If the timer was not already running, then the timer will calculate an expiry time relative to when xTimerStart() was called.

If the timer was already running, then xTimerStart() is functionally equivalent to xTimerReset().

If the timer is not stopped, deleted, or reset in the meantime, the callback function associated with the timer will get called ‘n’ ticks after xTimerStart() was called, where ‘n’ is the timer’s defined period.

Parameters

|  |  |
| --- | --- |
| xTimer | The timer to be reset, started, or restarted. |
| xTicksToWait | Timer functionality is not provided by the core FreeRTOS code, but by a timer |

service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks. Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xTicksToWait is ignored if xTimerStart() is called before the scheduler is started.

Return Values pdPASS The start command was successfully sent to the timer command queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired.

When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer’s expiry time is relative to when xTimerStart() is actually called. The priority of the timer service task is set by the configTIMER\_TASK\_PRIORITY configuration constant.

pdFAIL The start command was not sent to the timer command queue because the queue

was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStart() to be available.

Example

See the example provided for the xTimerCreate() API function.

# 5.9 xTimerStartFromISR()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerStartFromISR( TimerHandle\_t xTimer,

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 178 xTimerStartFromISR() macro prototype

Summary

A version of xTimerStart() that can be called from an interrupt service routine.

Parameters xTimer The handle of the timer that is being started, reset, or restarted.

pxHigherPriorityTaskWoken xTimerStartFromISR() writes a command to the timer command

queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then

\*pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerStartFromISR() function. If

xTimerStartFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values pdPASS The start command was successfully sent to the timer command queue. When the

command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer’s expiry time is relative to when xTimerStartFromISR() is actually called. The priority of the timer service task is set by the configTIMER\_TASK\_PRIORITY configuration constant.

pdFAIL The start command was not sent to the timer command queue because the queue

was already full.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStartFromISR() to be available.

Example

/\* This scenario assumes xBacklightTimer has already been created. When a key is pressed, an LCD back-light is switched on. If 5 seconds pass without a key being pressed, then the LCD back-light is switched off by a one-shot timer. Unlike the example given for the xTimerReset() function, the key press event handler is an interrupt service routine. \*/

/\* The callback function assigned to the one-shot timer. In this case the parameter is not used. \*/

void vBacklightTimerCallback( TimerHandle\_t pxTimer ) {

/\* The timer expired, therefore 5 seconds must have passed since a key was pressed. Switch off the LCD back-light. \*/ vSetBacklightState( BACKLIGHT\_OFF );

}

/\* The key press interrupt service routine. \*/ void vKeyPressEventInterruptHandler( void )

{

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

/\* Ensure the LCD back-light is on, then restart the timer that is responsible for turning the back-light off after 5 seconds of key inactivity. This is an interrupt service routine so can only call FreeRTOS API functions that end in

"FromISR". \*/

vSetBacklightState( BACKLIGHT\_ON );

/\* xTimerStartFromISR() or xTimerResetFromISR() could be called here as both cause the timer to re-calculate its expiry time. xHigherPriorityTaskWoken was initialized to pdFALSE when it was declared (in this function). \*/

if( xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS ) {

/\* The start command was not executed successfully. Take appropriate action here. \*/

}

/\* Perform the rest of the key processing here. \*/

/\* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be performed. The syntax required to perform a context switch from inside an ISR varies from port to port, and from compiler to compiler. Inspect the demos for the port you are using to find the actual syntax required. \*/ if( xHigherPriorityTaskWoken != pdFALSE )

{

/\* Call the interrupt safe yield function here (actual function depends on the FreeRTOS port being used). \*/

}

}

Listing 179 Example use of xTimerStartFromISR()

# 5.10 xTimerStop()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerStop( TimerHandle\_t xTimer, TickType\_t xTicksToWait );

Listing 180 xTimerStop() function prototype

Summary

Stops a timer running. xTimerStopFromISR() is an equivalent function that can be called from an interrupt service routine.

Parameters

|  |  |
| --- | --- |
| xTimer | The timer to be stopped. |
| xTicksToWait | Timer functionality is not provided by the core FreeRTOS code, but by a timer |

service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xTicksToWait is ignored if xTimerStop() is called before the scheduler is started.

Return Values pdPASS The stop command was successfully sent to the timer command queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired.

When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system. The priority of the timer service task is set by the configTIMER\_TASK\_PRIORITY configuration constant.

pdFAIL The stop command was not sent to the timer command queue because the queue

was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStop() to be available.

Example

See the example provided for the xTimerCreate() API function.

# 5.11 xTimerStopFromISR()

#include “FreeRTOS.h”

#include “timers.h”

BaseType\_t xTimerStopFromISR( TimerHandle\_t xTimer,

BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 181 xTimerStopFromISR() function prototype

Summary

A version of xTimerStop() that can be called from an interrupt service routine.

Parameters xTimer The handle of the timer that is being stopped.

pxHigherPriorityTaskWoken xTimerStopFromISR() writes a command to the timer command

queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then

\*pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerStopFromISR() function. If

xTimerStopFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values pdPASS The stop command was successfully sent to the timer command queue. When the

command is actually processed will depend on the priority of the timer service task relative to other tasks in the system. The priority of the timer service task is set by the configTIMER\_TASK\_PRIORITY configuration constant.

pdFAIL The stop command was not sent to the timer command queue because the queue

was already full.

Notes configUSE\_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStopFromISR() to be available.

Example

/\* This scenario assumes xTimer has already been created and started. When an interrupt occurs, the timer should be simply stopped. \*/

/\* The interrupt service routine that stops the timer. \*/ void vAnExampleInterruptServiceRoutine( void ) {

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

/\* The interrupt has occurred - simply stop the timer. xHigherPriorityTaskWoken was set to pdFALSE where it was defined (within this function). As this is an interrupt service routine, only FreeRTOS API functions that end in "FromISR" can be used. \*/

if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )

{

/\* The stop command was not executed successfully. Take appropriate action here. \*/

}

/\* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be performed. The syntax required to perform a context switch from inside an ISR varies from port to port, and from compiler to compiler. Inspect the demos for the port you are using to find the actual syntax required. \*/ if( xHigherPriorityTaskWoken != pdFALSE )

{

/\* Call the interrupt safe yield function here (actual function depends on the FreeRTOS port being used). \*/

}

}

Listing 182 Example use of xTimerStopFromISR()

Chapter 6

Event Groups API

# 6.1 xEventGroupCreate()

#include “FreeRTOS.h”

#include “event\_groups.h”

EventGroupHandle\_t xEventGroupCreate( void );

Listing 183 xEventGroupCreate() function prototype

Summary

Create a new event group.

Event groups are stored in variables of type EventGroupHandle\_t. The number of bits (or flags) implemented within an event group is 8 if configUSE\_16\_BIT\_TICKS is set to 1, or 24 if configUSE\_16\_BIT\_TICKS is set to 0. The dependency on configUSE\_16\_BIT\_TICKS results from the data type used for thread local storage in the internal implementation of RTOS tasks.

This function cannot be called from an interrupt.

Parameters

None

Return Values

NULL The event group could not be created because there was insufficient FreeRTOS heap available.

Any other value The event group was created successfully. The returned value should be

stored as the handle to the created event group.

Notes

The RTOS source file FreeRTOS/source/event\_groups.c must be included in the build for the xEventGroupCreate() function to be available.

/\* Declare a variable to hold the created event group. \*/

EventGroupHandle\_t xCreatedEventGroup;

/\* Attempt to create the event group. \*/ xCreatedEventGroup = xEventGroupCreate();

/\* Was the event group created successfully? \*/ if( xCreatedEventGroup == NULL )

{

/\* The event group was not created because there was insufficient

FreeRTOS heap available. \*/

} else

{

/\* The event group was created. \*/

}

Listing 184 Example use of xEventGroupCreate()

# 6.2 xEventGroupClearBits()

#include “FreeRTOS.h”

#include “event\_groups.h”

EventBits\_t xEventGroupClearBits( EventGroupHandle\_t xEventGroup, const EventBits\_t uxBitsToClear );

Listing 185 xEventGroupClearBits() function prototype

Summary

Clear bits (flags) within an RTOS event group. This function cannot be called from an interrupt. See xEventGroupClearBitsFromISR() for a version that can be called from an interrupt.

Parameters

|  |  |
| --- | --- |
| xEventGroup | The event group in which the bits are to be cleared. The event group must have previously been created using a call to xEventGroupCreate(). |
| uxBitsToClear | A bitwise value that indicates the bit or bits to clear in the event group. For |

example set uxBitsToClear to 0x08 to clear just bit 3. Set uxBitsToClear to 0x09 to clear bit 3 and bit 0.

Return Values

All values The value of the bits in the event group before any bits were cleared.

Notes

The RTOS source file FreeRTOS/source/event\_groups.c must be included in the build for the xEventGroupClearBits() function to be available.

#define BIT\_0 ( 1 << 0 )

#define BIT\_4 ( 1 << 4 )

void aFunction( EventGroupHandle\_t xEventGroup )

{

EventBits\_t uxBits;

/\* Clear bit 0 and bit 4 in xEventGroup. \*/ uxBits = xEventGroupClearBits(

xEventGroup, /\* The event group being updated. \*/

BIT\_0 | BIT\_4 );/\* The bits being cleared. \*/

if( ( uxBits & ( BIT\_0 | BIT\_4 ) ) == ( BIT\_0 | BIT\_4 ) )

{

/\* Both bit 0 and bit 4 were set before xEventGroupClearBits() was called. Both will now be clear (not set). \*/

}

else if( ( uxBits & BIT\_0 ) != 0 )

{

/\* Bit 0 was set before xEventGroupClearBits() was called. It will now be clear. \*/

}

else if( ( uxBits & BIT\_4 ) != 0 )

{

/\* Bit 4 was set before xEventGroupClearBits() was called. It will now be clear. \*/

} else

{

/\* Neither bit 0 nor bit 4 were set in the first place. \*/

}

}

Listing 186 Example use of xEventGroupClearBits()

# 6.3 xEventGroupClearBitsFromISR()

#include “FreeRTOS.h”

#include “event\_groups.h”

EventBits\_t xEventGroupClearBitsFromISR( EventGroupHandle\_t xEventGroup, const EventBits\_t uxBitsToClear );

Listing 187 xEventGroupClearBitsFromISR() function prototype

Summary

A version of xEventGroupClearBits() that can be called from an interrupt.

Parameters xEventGroup The event group in which the bits are to be cleared. The event group must

have previously been created using a call to xEventGroupCreate().

uxBitsToClear A bitwise value that indicates the bit or bits to clear in the event group. For

example set uxBitsToClear to 0x08 to clear just bit 3. Set uxBitsToClear to 0x09 to clear bit 3 and bit 0.

Return Values

All values The value of the bits in the event group before any bits were cleared.

Notes

The RTOS source file FreeRTOS/source/event\_groups.c must be included in the build for the xEventGroupClearBitsFromISR() function to be available.

#define BIT\_0 ( 1 << 0 )

#define BIT\_4 ( 1 << 4 )

/\* This code assumes the event group referenced by the xEventGroup variable has

Already been created using a call to xEventGroupCreate(). \*/ void anInterruptHandler( void )

{

EventBits\_t uxBits;

/\* Clear bit 0 and bit 4 in xEventGroup. \*/ uxBits = xEventGroupClearBitsFromISR(

xEventGroup, /\* The event group being updated. \*/

BIT\_0 | BIT\_4 );/\* The bits being cleared. \*/

if( ( uxBits & ( BIT\_0 | BIT\_4 ) ) == ( BIT\_0 | BIT\_4 ) )

{

/\* Both bit 0 and bit 4 were set before xEventGroupClearBitsFromISR() was called. Both will now be clear (not set). \*/

}

else if( ( uxBits & BIT\_0 ) != 0 )

{

/\* Bit 0 was set before xEventGroupClearBitsFromISR() was called. It will now be clear. \*/

}

else if( ( uxBits & BIT\_4 ) != 0 )

{

/\* Bit 4 was set before xEventGroupClearBitsFromISR() was called. It will now be clear. \*/

} else

{

/\* Neither bit 0 nor bit 4 were set in the first place. \*/

}

}

Listing 188 Example use of xEventGroupClearBitsFromISR()

# 6.1 vEventGroupDelete()

#include “FreeRTOS.h”

#include “event\_groups.h”

void vEventGroupDelete( EventGroupHandle\_t xEventGroup );

Listing 189 vEventGroupDelete() function prototype

Summary

Delete an event group that was previously created using a call to xEventGroupCreate().

Tasks that are blocked on the event group being deleted will be unblocked and report an event group value of 0.

This function must not be called from an interrupt.

Parameters xEventGroup The event group to delete.

Return Values

None

Notes

The RTOS source file FreeRTOS/source/event\_groups.c must be included in the build for the vEventGroupDelete() function to be available.

# 6.2 xEventGroupGetBits()

#include “FreeRTOS.h”

#include “event\_groups.h”

EventBits\_t xEventGroupGetBits( EventGroupHandle\_t xEventGroup );

Listing 190 xEventGroupGetBits() function prototype

Summary

Returns the current value of the event bits (event flags) in an event group. This function cannot be used from an interrupt. See xEventGroupGetBitsFromISR() for a version that can be used in an interrupt.

Parameters xEventGroup The event group being queried. The event group must have previously been

created using a call to xEventGroupCreate().

Return Values

All values The value of the event bits in the event group at the time

xEventGroupGetBits() was called.

Notes

The RTOS source file FreeRTOS/source/event\_groups.c must be included in the build for the xEventGroupGetBits() function to be available.

# 6.1 xEventGroupGetBitsFromISR()

#include “FreeRTOS.h”

#include “event\_groups.h”

EventBits\_t xEventGroupGetBitsFromISR( EventGroupHandle\_t xEventGroup );

Listing 191 xEventGroupGetBitsFromISR() function prototype

Summary

A version of xEventGroupGetBits() that can be called from an interrupt.

Parameters xEventGroup The event group being queried. The event group must have previously been

created using a call to xEventGroupCreate().

Return Values

All values The value of the event bits in the event group at the time

xEventGroupGetBitsFromISR() was called.

Notes

The RTOS source file FreeRTOS/source/event\_groups.c must be included in the build for the xEventGroupGetBitsFromISR() function to be available.

# 6.2 xEventGroupSetBits()

#include “FreeRTOS.h”

#include “event\_groups.h”

EventBits\_t xEventGroupSetBits( EventGroupHandle\_t xEventGroup, const EventBits\_t uxBitsToSet );

Listing 192 xEventGroupSetBits() function prototype

Summary

Sets bits (flags) within an RTOS event group. This function cannot be called from an interrupt. See xEventGroupSetBitsFromISR() for a version that can be called from an interrupt.

Setting bits in an event group will automatically unblock any tasks that were blocked waiting for the bits to be set.

Parameters

xEventGroup The event group in which the bits are to be set. The event group must have

previously been created using a call to xEventGroupCreate().

uxBitsToSet A bitwise value that indicates the bit or bits to set in the event group. For

example, set uxBitsToSet to 0x08 to set only bit 3. Set uxBitsToSet to 0x09 to set bit 3 and bit 0.

Return Values

Any Value The value of the bits in the event group at the time the call to

xEventGroupSetBits() returned.

There are two reasons why the returned value might have the bits specified by the uxBitsToSet parameter cleared:

1. If setting a bit results in a task that was waiting for the bit leaving the blocked state then it is possible the bit will have been cleared automatically (see the xClearBitsOnExit parameter of xEventGroupWaitBits()).
2. Any task that leaves the blocked state as a result of the bits being set (or otherwise any Ready state task) that has a priority above that of the task that called xEventGroupSetBits() will execute and may change the event group value before the call to xEventGroupSetBits() returns.

Notes

The RTOS source file FreeRTOS/source/event\_groups.c must be included in the build for the xEventGroupSetBits() function to be available.

Example

#define BIT\_0 ( 1 << 0 )

#define BIT\_4 ( 1 << 4 )

void aFunction( EventGroupHandle\_t xEventGroup )

{

EventBits\_t uxBits;

/\* Set bit 0 and bit 4 in xEventGroup. \*/ uxBits = xEventGroupSetBits(

xEventGroup, /\* The event group being updated. \*/

BIT\_0 | BIT\_4 );/\* The bits being set. \*/

if( ( uxBits & ( BIT\_0 | BIT\_4 ) ) == ( BIT\_0 | BIT\_4 ) ) {

/\* Both bit 0 and bit 4 remained set when the function returned. \*/

}

else if( ( uxBits & BIT\_0 ) != 0 )

{

/\* Bit 0 remained set when the function returned, but bit 4 was cleared. It might be that bit 4 was cleared automatically as a task that was waiting for bit 4 was removed from the Blocked state. \*/

}

else if( ( uxBits & BIT\_4 ) != 0 )

{

/\* Bit 4 remained set when the function returned, but bit 0 was cleared. It might be that bit 0 was cleared automatically as a task that was waiting for bit 0 was removed from the Blocked state. \*/

} else

{

/\* Neither bit 0 nor bit 4 remained set. It might be that a task was waiting for both of the bits to be set, and the bits were cleared as the task left the Blocked state. \*/

}

}

Listing 193 Example use of xEventGroupSetBits()

# 6.3 xEventGroupSetBitsFromISR()

#include “FreeRTOS.h”

#include “event\_groups.h”

EventBits\_t xEventGroupSetBitsFromISR( EventGroupHandle\_t xEventGroup, const EventBits\_t uxBitsToSet, BaseType\_t \*pxHigherPriorityTaskWoken );

Listing 194 xEventGroupSetBitsFromISR() function prototype

Summary

Set bits (flags) within an event group. A version of xEventGroupSetBits() that can be called from an interrupt service routine (ISR).

Setting bits in an event group will automatically unblock any tasks that were blocked waiting for the bits to be set.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow non-deterministic operations to be performed in interrupts or from critical sections. Therefore xEventGroupSetBitsFromISR() sends a message to the RTOS daemon task to have the set operation performed in the context of the daemon task - where a scheduler lock is used in place of a critical section. The priority of the daemon task is set by configTIMER\_TASK\_PRIORITY in FreeRTOSConfig.h.

Parameters xEventGroup The event group in which the bits are to be set. The event

group must have previously been created using a call to xEventGroupCreate().

uxBitsToSet A bitwise value that indicates the bit or bits to set in the event

group. For example, set uxBitsToSet to 0x08 to set only bit 3.

Set uxBitsToSet to 0x09 to set bit 3 and bit 0.

pxHigherPriorityTaskWoken Calling xEventGroupSetBitsFromISR() results in a message

being sent to the RTOS daemon task. If the priority of the daemon task is higher than the priority of the currently running task (the task the interrupt interrupted) then \*pxHigherPriorityTaskWoken will be set to pdTRUE by xEventGroupSetBitsFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason \*pxHigherPriorityTaskWoken must be initialized to pdFALSE.

See the example code below.

Return Values

|  |  |
| --- | --- |
| pdPASS | The message was sent to the RTOS daemon task. |
| pdFAIL | The message could not be sent to the RTOS daemon task (also known as |

the timer service task) because the timer command queue was full. The length of the queue is set by the configTIMER\_QUEUE\_LENGTH setting in FreeRTOSConfig.h.

Notes

The RTOS source file FreeRTOS/source/event\_groups.c must be included in the build for the xEventGroupSetBitsFromISR() function to be available.

INCLUDE\_xEventGroupSetBitsFromISR, configUSE\_TIMERS and

INCLUDE\_xTimerPendFunctionCall must all be set to 1 in FreeRTOSConfig.h for the xEventGroupSetBitsFromISR() function to be available.

Example

#define BIT\_0 ( 1 << 0 )

#define BIT\_4 ( 1 << 4 )

/\* An event group which it is assumed has already been created by a call to xEventGroupCreate(). \*/

EventGroupHandle\_t xEventGroup;

void anInterruptHandler( void )

{

BaseType\_t xHigherPriorityTaskWoken, xResult;

/\* xHigherPriorityTaskWoken must be initialized to pdFALSE. \*/ xHigherPriorityTaskWoken = pdFALSE;

/\* Set bit 0 and bit 4 in xEventGroup. \*/ xResult = xEventGroupSetBitsFromISR(

xEventGroup, /\* The event group being updated. \*/

BIT\_0 | BIT\_4 /\* The bits being set. \*/

&xHigherPriorityTaskWoken );

/\* Was the message posted successfully? \*/ if( xResult != pdFAIL )

{

/\* If xHigherPriorityTaskWoken is now set to pdTRUE then a context switch should be requested. The macro used is port specific and will be either portYIELD\_FROM\_ISR() or portEND\_SWITCHING\_ISR() - refer to the documentation page for the port being used. \*/ portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );

}

}

Listing 195 Example use of xEventGroupSetBitsFromISR()

# 6.1 xEventGroupSync()

#include “FreeRTOS.h”

#include “event\_groups.h”

EventBits\_t xEventGroupSync( EventGroupHandle\_t xEventGroup, const EventBits\_t uxBitsToSet, const EventBits\_t uxBitsToWaitFor, TickType\_t xTicksToWait );

Listing 196 xEventGroupSync() function prototype

Summary

Atomically set bits (flags) within an event group, then wait for a combination of bits to be set within the same event group. This functionality is typically used to synchronize multiple tasks (often called a task rendezvous), where each task has to wait for the other tasks to reach a synchronization point before proceeding.

The function will return before its block time expires if the bits specified by the uxBitsToWaitFor parameter are set, or become set within that time. In this case all the bits specified by uxBitsToWaitFor will be automatically cleared before the function returns.

This function cannot be used from an interrupt.

Parameters

|  |  |
| --- | --- |
| xEventGroup | The event group in which the bits are being set and tested. The event group must have previously been created using a call to xEventGroupCreate(). |
| uxBitsToSet | A bitwise value that indicates the bit or bits to set in the event group before determining if (and possibly waiting for) all the bits specified by the uxBitsToWaitFor parameter are set. For example, set uxBitsToSet to 0x04 to set bit 2 within the event group. |
| uxBitsToWaitFor | A bitwise value that indicates the bit or bits to test inside the event group. For example, set uxBitsToWaitFor to 0x05 to wait for bit 0 and bit 2. Set uxBitsToWaitFor to 0x07 to wait for bit 0 and bit 1 and bit 2. Etc. |
| xTicksToWait | The maximum amount of time (specified in 'ticks') to wait for all the bits |

specified by the uxBitsToWaitFor parameter value to become set.

Return Values

All values The value of the event group at the time either the bits being waited for

became set, or the block time expired. Test the return value to know which bits were set.

If xEventGroupSync() returned because its timeout expired then not all the bits being waited for will be set in the returned value.

If xEventGroupSync() returned because all the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared.

Notes

The RTOS source file FreeRTOS/source/event\_groups.c must be included in the build for the xEventGroupSync() function to be available.

Example

/\* Bits used by the three tasks. \*/

#define TASK\_0\_BIT ( 1 << 0 )

#define TASK\_1\_BIT ( 1 << 1 )

#define TASK\_2\_BIT ( 1 << 2 )

#define ALL\_SYNC\_BITS ( TASK\_0\_BIT | TASK\_1\_BIT | TASK\_2\_BIT )

/\* Use an event group to synchronize three tasks. It is assumed this event group has already been created elsewhere. \*/

EventGroupHandle\_t xEventBits;

void vTask0( void \*pvParameters )

{

EventBits\_t uxReturn;

TickType\_t xTicksToWait = pdMS\_TO\_TICKS( 100 );

for( ;; )

{

/\* Perform task functionality here. \*/

. . .

/\* Set bit 0 in the event group to note this task has reached the sync point. The other two tasks will set the other two bits defined by ALL\_SYNC\_BITS. All three tasks have reached the synchronization point when all the ALL\_SYNC\_BITS bits are set. Wait a maximum of 100ms for this to happen. \*/

uxReturn = xEventGroupSync( xEventBits,

TASK\_0\_BIT, /\* The bit to set. \*/

ALL\_SYNC\_BITS, /\* The bits to wait for. \*/ xTicksToWait );/\* Timeout value. \*/

if( ( uxReturn & ALL\_SYNC\_BITS ) == ALL\_SYNC\_BITS )

{

/\* All three tasks reached the synchronization point before the call to xEventGroupSync() timed out. \*/

}

}

}

void vTask1( void \*pvParameters )

{

for( ;; )

{

/\* Perform task functionality here. \*/

. . .

/\* Set bit 1 in the event group to note this task has reached the synchronization point. The other two tasks will set the other two bits defined by ALL\_SYNC\_BITS. All three tasks have reached the synchronization point when all the ALL\_SYNC\_BITS are set. Wait indefinitely for this to happen. \*/

xEventGroupSync( xEventBits, TASK\_1\_BIT, ALL\_SYNC\_BITS, portMAX\_DELAY );

/\* xEventGroupSync() was called with an indefinite block time, so this task will only reach here if the synchronization was made by all three tasks, so there is no need to test the return value. \*/

}

}

void vTask2( void \*pvParameters )

{

for( ;; )

{

/\* Perform task functionality here. \*/

. . .

/\* Set bit 2 in the event group to note this task has reached the synchronization point. The other two tasks will set the other two bits defined by ALL\_SYNC\_BITS. All three tasks have reached the synchronization point when all the ALL\_SYNC\_BITS are set. Wait indefinitely for this to happen. \*/

xEventGroupSync( xEventBits, TASK\_2\_BIT, ALL\_SYNC\_BITS, portMAX\_DELAY );

/\* xEventGroupSync() was called with an indefinite block time, so this task will only reach here if the synchronization was made by all three tasks, so there is no need to test the return value. \*/

}

}

Listing 197 Example use of xEventGroupSync()

# 6.2 xEventGroupWaitBits()

#include “FreeRTOS.h”

#include “event\_groups.h”

EventBits\_t xEventGroupWaitBits( const EventGroupHandle\_t xEventGroup, const EventBits\_t uxBitsToWaitFor, const BaseType\_t xClearOnExit, const BaseType\_t xWaitForAllBits,

TickType\_t xTicksToWait );

Listing 198 xEventGroupWaitBits() function prototype

Summary

Read bits within an RTOS event group, optionally entering the Blocked state (with a timeout) to wait for a bit or group of bits to become set.

This function cannot be called from an interrupt.

Parameters

|  |  |
| --- | --- |
| xEventGroup | The event group in which the bits are being tested. The event group must have previously been created using a call to xEventGroupCreate(). |
| uxBitsToWaitFor | A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and/or bit 2 set uxBitsToWaitFor to 0x05. To wait for bits 0 and/or bit 1 and/or bit 2 set uxBitsToWaitFor to 0x07. Etc. uxBitsToWaitFor must not be set to 0. |
| xClearOnExit | If xClearOnExit is set to pdTRUE then any bits set in the value passed as the uxBitsToWaitFor parameter will be cleared in the event group before xEventGroupWaitBits() returns if xEventGroupWaitBits() returns for any reason other than a timeout. The timeout value is set by the xTicksToWait parameter.  If xClearOnExit is set to pdFALSE then the bits set in the event group are not altered when the call to xEventGroupWaitBits() returns. |
| xWaitAllBits | xWaitForAllBits is used to create either a logical AND test (where all bits |

must be set) or a logical OR test (where one or more bits must be set) as

follows:

If xWaitForAllBits is set to pdTRUE then xEventGroupWaitBits() will return when either all the bits set in the value passed as the uxBitsToWaitFor parameter are set in the event group or the specified block time expires.

If xWaitForAllBits is set to pdFALSE then xEventGroupWaitBits() will return when any of the bits set in the value passed as the uxBitsToWaitFor parameter are set in the event group or the specified block time expires.

xTicksToWait The maximum amount of time (specified in 'ticks') to wait for one/all (depending on the xWaitForAllBits value) of the bits specified by uxBitsToWaitFor to become set.

Return Values

Any Value The value of the event group at the time either the event bits being waited

for became set, or the block time expired. The current value of the event bits in an event group will be different to the returned value if a higher priority task or interrupt changed the value of an event bit between the calling task leaving the Blocked state and exiting the xEventGroupWaitBits() function.

Test the return value to know which bits were set. If xEventGroupWaitBits() returned because its timeout expired then not all the bits being waited for will be set. If xEventGroupWaitBits() returned because the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared in the case that xClearOnExit parameter was set to pdTRUE.

Notes

The RTOS source file FreeRTOS/source/event\_groups.c must be included in the build for the xEventGroupWaitBits() function to be available.

Example

#define BIT\_0 ( 1 << 0 )

#define BIT\_4 ( 1 << 4 )

void aFunction( EventGroupHandle\_t xEventGroup )

{

EventBits\_t uxBits;

const TickType\_t xTicksToWait = pdMS\_TO\_TICKS( 100 );

/\* Wait a maximum of 100ms for either bit 0 or bit 4 to be set within the event group. Clear the bits before exiting. \*/ uxBits = xEventGroupWaitBits(

xEventGroup, /\* The event group being tested. \*/

BIT\_0 | BIT\_4, /\* The bits within the event group to wait for. \*/ pdTRUE, /\* BIT\_0 and BIT\_4 should be cleared before returning. \*/ pdFALSE, /\* Don't wait for both bits, either bit will do. \*/ xTicksToWait );/\* Wait a maximum of 100ms for either bit to be set. \*/

if( ( uxBits & ( BIT\_0 | BIT\_4 ) ) == ( BIT\_0 | BIT\_4 ) )

{

/\* xEventGroupWaitBits() returned because both bits were set. \*/

}

else if( ( uxBits & BIT\_0 ) != 0 )

{

/\* xEventGroupWaitBits() returned because just BIT\_0 was set. \*/

}

else if( ( uxBits & BIT\_4 ) != 0 )

{

/\* xEventGroupWaitBits() returned because just BIT\_4 was set. \*/

} else

{

/\* xEventGroupWaitBits() returned because xTicksToWait ticks passed without either BIT\_0 or BIT\_4 becoming set. \*/

}

}

Listing 199 Example use of xEventGroupWaitBits()

Chapter 7

Kernel Configuration

# 7.1 FreeRTOSConfig.h

Kernel configuration is achieved by setting #define constants in FreeRTOSConfig.h. Each application that uses FreeRTOS must provide a FreeRTOSConfig.h header file.

All the demo application projects included in the FreeRTOS download contains a pre-defined FreeRTOSConfig.h that can be used as a reference or simply copied. Note, however, that some of the demo projects were generated before all the options documented in this chapter were available, so the FreeRTOSConfig.h header files they contain will not include all the constants and options that are documented in the following sub-sections.

# 7.2 Constants that Start “INCLUDE\_”

Constants that start with the text “INCLUDE\_” are used to included or excluded FreeRTOS API functions from the application. For example, setting INCLUDE\_vTaskPrioritySet to 0 will exclude the vTaskPrioritySet() API function from the build, meaning the application cannot call vTaskPrioritySet(). Setting INCLUDE\_vTaskPrioritySet to 1 will include the vTaskPrioritySet() API function in the build, so the application can call vTaskPrioritySet().

In some cases, a single INCLUDE\_ configuration constant will include or exclude multiple API functions.

The “INCLUDE\_” constants are provided to permit the code size to be reduced by removing FreeRTOS functions and features that are not required. However, most linkers will, by default, automatically remove unreferenced code unless optimization is turned completely off. Linkers that do not have this default behavior can normally be configured to remove unreferenced code. Therefore, in most practical cases, the INCLUDE\_ configuration constants will have little if any impact on the executable code size.

It is possible that excluding an API function from an application will also reduce the amount of RAM used by the FreeRTOS kernel. For example, removing the vTaskSuspend() API function will also prevent the structures that would otherwise reference Suspended tasks from ever being allocated.

INCLUDE\_xEventGroupSetBitsFromISR

configUSE\_TIMERS, INCLUDE\_xTimerPendFunctionCall and INCLUDE\_xEventGroupSetBitsFromISR must all be set to 1 for the xEventGroupSetBitsFromISR () API function to be available.

INCLUDE\_xSemaphoreGetMutexHolder

INCLUDE\_xSemaphoreGetMutexHolder must be set to 1 for the xSemaphoreGetMutexHolder() API function to be available.

INCLUDE\_vTaskDelay

INCLUDE\_vTaskDelay must be set to 1 for the vTaskDelay() API function to be available.

INCLUDE\_vTaskDelayUntil

INCLUDE\_vTaskDelayUntil must be set to 1 for the vTaskDelayUntil() API function to be available.

INCLUDE\_vTaskDelete

INCLUDE\_vTaskDelete must be set to 1 for the vTaskDelete() API function to be available.

INCLUDE\_xTaskGetCurrentTaskHandle

INCLUDE\_xTaskGetCurrentTaskHandle must be set to 1 for the xTaskGetCurrentTaskHandle() API function to be available.

INCLUDE\_xTaskGetIdleTaskHandle

INCLUDE\_xTaskGetIdleTaskHandle must be set to 1 for the xTaskGetIdleTaskHandle() API function to be available.

INCLUDE\_xTaskGetSchedulerState

INCLUDE\_xTaskGetSchedulerState must be set to 1 for the xTaskGetSchedulerState() API function to be available.

INCLUDE\_uxTaskGetStackHighWaterMark

INCLUDE\_uxTaskGetStackHighWaterMark must be set to 1 for the uxTaskGetStackHighWaterMark() API function to be available.

INCLUDE\_pcTaskGetTaskName

INCLUDE\_pcTaskGetTaskName must be set to 1 for the pcTaskGetTaskName() API function to be available.

INCLUDE\_uxTaskPriorityGet

INCLUDE\_uxTaskPriorityGet must be set to 1 for the uxTaskPriorityGet() API function to be available.

INCLUDE\_vTaskPrioritySet

INCLUDE\_vTaskPrioritySet must be set to 1 for the vTaskPrioritySet() API function to be available.

INCLUDE\_xTaskResumeFromISR

INCLUDE\_xTaskResumeFromISR and INCLUDE\_vTaskSuspend must both be set to 1 for the xTaskResumeFromISR() API function to be available.

INCLUDE\_eTaskGetState

INCLUDE\_eTaskGetState must be set to 1 for the eTaskGetState() API function to be available.

INCLUDE\_vTaskSuspend

INCLUDE\_vTaskSuspend must be set to 1 for the vTaskSuspend(), vTaskResume(), and xTaskIsTaskSuspended() API functions to be available.

INCLUDE\_vTaskSuspend and INCLUDE\_xTaskResumeFromISR must both be set to 1 for the xTaskResumeFromISR() API function to be available.

Some queue and semaphore API functions allow the calling task to opt to be placed into the Blocked state to wait for a queue or semaphore event to occur. These API functions require that a maximum block period, or time out, is specified. The calling task will then be held in the

Blocked state until either the queue or semaphore event occurs, or the block period expires.

The maximum block period that can be specified is defined by portMAX\_DELAY. If INCLUDE\_vTaskSuspend is set to 0, then specifying a block period of portMAX\_DELAY will result in the calling task being placed into the Blocked state for a maximum of portMAX\_DELAY ticks. If INCLUDE\_vTaskSuspend is set to 1, then specifying a block period of portMAX\_DELAY will result in the calling task being placed into the Blocked state indefinitely (without a time out). In the second case, the block period is indefinite, so the only way out of the Blocked state is for the queue or semaphore event to occur.

INCLUDE\_xTimerGetTimerDaemonTaskHandle configUSE\_TIMERS and INCLUDE\_xTimerGetTimerDaemonTaskHandle must both be set to 1 for the xTimerGetTimerDaemonTaskHandle() API function to be available.

INCLUDE\_ xTimerPendFunctionCall configUSE\_TIMERS and INCLUDE\_xTimerPendFunctionCall must both be set to 1 for the xTimerPendFunctionCall() and xTimerPendFunctionCallFromISR () API functions to be available.

# 7.3 Constants that Start “config”

Constants that start with the text “config” define attributes of the kernel, or include or exclude features of the kernel.

configASSERT

Calls to configASSERT( x ) exist at key points in the FreeRTOS kernel code.

If FreeRTOS is functioning correctly, and is being used correctly, then the configASSERT() parameter will be non-zero. If the parameter is found to equal zero, then an error has occurred.

It is likely that most errors trapped by configASSERT() will be a result of an invalid parameter being passed into a FreeRTOS API function. configASSERT() can therefore assist in run time debugging. However, defining configASSERT() will also increase the application code size, and slow down its execution.

configASSERT() is equivalent to the standard C assert() macro. It is used in place of the standard C assert() macro because not all the compilers that can be used to build FreeRTOS provide an assert.h header file.

configASSERT() should be defined in FreeRTOSConfig.h. Listing 200 shows an example configASSERT() definition that assumed vAssertCalled() is defined elsewhere by the application.

#define configASSERT( ( x ) ) if( ( x ) == 0 ) vAssertCalled( \_\_FILE\_\_, \_\_LINE\_\_ )

Listing 200 An example configASSERT() definition

configCHECK\_FOR\_STACK\_OVERFLOW

Each task has a unique stack that is automatically allocated from the FreeRTOS heap when the task is created. The size of the stack is set by the usStackDepth parameter of the xTaskCreate() call used to create the task.

Stack overflow is a very common cause of application instability. FreeRTOS provides two optional mechanisms that can be used to assist in stack overflow detection and debugging.

Which (if any) option is used is configured by the configCHECK\_FOR\_STACK\_OVERFLOW configuration constant.

If configCHECK\_FOR\_STACK\_OVERFLOW is not set to 0 then the application must also provide a stack overflow hook (or callback) function. The kernel will call the stack overflow hook whenever a stack overflow is detected.

The stack overflow hook function must be called vApplicationStackOverflowHook(), and have the prototype shown in Listing 201.

void vApplicationStackOverflowHook( TaskHandle\_t \*pxTask, signed char \*pcTaskName );

Listing 201 The stack overflow hook function prototype

The name and handle of the task that has exceeded its stack space are passed into the stack overflow hook function using the pcTaskName and pxTask parameters respectively. It should be noted that a stack overflow can potentially corrupted these parameters, in which case the pxCurrentTCB variable can be inspected to determine which task caused the stack overflow hook function to be called.

Stack overflow checking can only be used on architectures that have a linear (rather than segmented) memory map.

Some processors will generate a fault exception in response to a stack corruption before the stack overflow callback function can be called.

Stack overflow checking increases the time taken to perform a context switch.

Stack overflow Method one is selected by setting detection method one configCHECK\_FOR\_STACK\_OVERFLOW to 1.

It is likely that task stack utilization will reach its maximum when the task’s context is saved to the stack during a context switch. Stack overflow detection method one checks the stack utilization at that time to ensure the task stack pointer remains within the valid stack area. The stack overflow hook function will be called if the stack pointer contains an invalid value (a value that references memory outside of the valid stack area).

Method one is quick, but will not necessarily catch all stack overflow occurrences.

Stack overflow Method two is selected by setting detection method two configCHECK\_FOR\_STACK\_OVERFLOW to 2.

Method two includes the checks performed by method one. In addition, method two will also verify that the limit of the valid stack region has not been overwritten.

The stack allocated to a task is filled with a known pattern at the time the task is created. Method two checks the last n bytes within the valid stack range to ensure this pattern remains unmodified (has not been overwritten). The stack overflow hook function is called if any of these n bytes have changed from their original values.

Method two is less efficient than method one, but still fast. It will catch most stack overflow occurrences, although it is conceivable that some could be missed (for example, where a stack overflow occurs without the last n bytes being written to).

configCPU\_CLOCK\_HZ

This must be set to the frequency of the clock that drives the peripheral used to generate the kernels periodic tick interrupt. This is very often, but not always, equal to the main system clock frequency.

configENABLE\_BACKWARD\_COMPATIBILITY

The FreeRTOS.h header file includes a set of #define macros that map the names of data types used in versions of FreeRTOS prior to version 8.0.0 to the names used in FreeRTOS version 8.0.0. The macros allow application code to update the version of FreeRTOS they are built against from a pre 8.0.0 version to a post 8.0.0 version without modification. Setting configENABLE\_BACKWARD\_COMPATIBILITY to 0 in FreeRTOSConfig.h excludes the macros from the build, and in so doing allowing validation that no pre version 8.0.0 names are being used.

configGENERATE\_RUN\_TIME\_STATS

The task run time statistics feature collects information on the amount of processing time each task is receiving. The feature requires the application to configure a run time statistics time base. The frequency of the run time statistics time base must be at least ten times greater than the frequency of the tick interrupt.

Setting configGENERATE\_RUN\_TIME\_STATS to 1 will include the run time statistics gathering functionality and associated API in the build. Setting configGENERATE\_RUN\_TIME\_STATS to 0 will exclude the run time statistics gathering functionality and associated API from the build.

If configGENERATE\_RUN\_TIME\_STATS is set to 1, then the application must also provide definitions for the macros described in Table 2. If configGENERATE\_RUN\_TIME\_STATS is set to 0 then the application must not define any of the macros described in Table 2, otherwise there is a risk that the application will not compiler and/or link.

Table 2. Additional macros that are required if configGENERATE\_RUN\_TIME\_STATS is set to 1

|  |  |
| --- | --- |
| Macro | Description |

portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() This macro must be provided to

initialize whichever peripheral is used to generate the run time statistics time base.

Table 2. Additional macros that are required if configGENERATE\_RUN\_TIME\_STATS is set to 1

|  |  |
| --- | --- |
| Macro | Description |

portGET\_RUN\_TIME\_COUNTER\_VALUE(), or One of these two macros must be

provided to return the current time portALT\_GET\_RUN\_TIME\_COUNTER\_VALUE(Time)

base value—this is the total time that the application has been running in the chosen time base units. If the first macro is used it must be defined to evaluate to the current time base value. If the second macro is used it must be defined to set its ‘Time’ parameter to the current time base value. (‘ALT’ in the macro name is an abbreviation of ‘ALTernative’).

configIDLE\_SHOULD\_YIELD

configIDLE\_SHOULD\_YIELD controls the behavior of the idle task if there are application tasks that also run at the idle priority. It only has an effect if the preemptive scheduler is being used.

Tasks that share a priority are scheduled using a round robin, time sliced, algorithm. Each task will be selected in turn to enter the running state, but may not remain in the running state for an entire tick period. For example, a task may be preempted, choose to yield, or choose to enter the Blocked state before the next tick interrupt.

If configIDLE\_SHOULD\_YIELD is set to 0, then the idle task will never yield to another task, and will only leave the Running state when it is pre-empted.

If configIDLE\_SHOULD\_YIELD is set to 1, then idle task will never perform more than one iteration of its defined functionality without yielding to another task if there is another Idle priority task that is in the Ready state. This ensures a minimum amount of time is spent in the idle task when application tasks are available to run.

The Idle task consistently yielding to another Idle priority Ready state tasks has the side effect shown in Figure 3.





Figure 3 Time line showing the execution of 4 tasks, all of which run at the idle priority

Figure 3 shows the execution pattern of four tasks that all run at the idle priority. Tasks A, B and C are application tasks. Task I is the idle task. The tick interrupt initiates a context switch at regular intervals, shown at times T0, T1, T2, etc. It can be seen that the Idle task starts to execute at time T2. It executes for part of a time slice, then yields to Task A. Task A executes for the remainder of the same time slice, then gets pre-empted at time T3. Task I and task A effectively share a single time slice, resulting in task B and task C consistently utilizing more processing time than task A.

Setting configIDLE\_SHOULD\_YIELD to 0 prevents this behavior by ensuring the Idle task remains in the Running state for an entire tick period (unless pre-empted by an interrupt other than the tick interrupt). When this is the case, averaged over time, the other tasks that share the idle priority will get an equal share of the processing time, but more time will also be spent executing the idle task. Using an Idle task hook function can ensure the time spent executing the Idle task is used productively.

configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS is only used by FreeRTOS MPU.

If configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS is set to 1 then the application writer must provide a header file called

"application\_defined\_privileged\_functions.h", in which functions the application writer needs to execute in privileged mode can be implemented. Note that, despite having a .h extension, the header file should contain the implementation of the C functions, not just the functions' prototypes.

Functions implemented in "application\_defined\_privileged\_functions.h" must save and restore the processor's privilege state using the prvRaisePrivilege() function and portRESET\_PRIVILEGE() macro respectively. For example, if a library provided print function accesses RAM that is outside of the control of the application writer, and therefore cannot be allocated to a memory protected user mode task, then the print function can be encapsulated in a privileged function using the following code:

void MPU\_debug\_printf( const char \*pcMessage )

{

State the privilege level of the processor when the function was called. \*/

BaseType\_t xRunningPrivileged = prvRaisePrivilege();

/\* Call the library function, which now has access to all RAM. \*/ debug\_printf( pcMessage );

/\* Reset the processor privilege level to its original value. \*/ portRESET\_PRIVILEGE( xRunningPrivileged );

}

Listing 202 An example of saving and restoring the processors privilege state

This technique should only be use during development, and not deployment, as it circumvents the memory protection.

configKERNEL\_INTERRUPT\_PRIORITY, configMAX\_SYSCALL\_INTERRUPT\_PRIORITY, configMAX\_API\_CALL\_INTERRUPT\_PRIORITY configMAX\_API\_CALL\_INTERRUPT\_PRIORITY is a new name for configMAX\_SYSCALL\_INTERRUPT\_PRIORITY that is used by newer ports only. The two are equivalent.

configKERNEL\_INTERRUPT\_PRIORITY and configMAX\_SYSCALL\_INTERRUPT\_PRIORITY are only relevant to ports that implement interrupt nesting.

If a port only implements the configKERNEL\_INTERRUPT\_PRIORITY configuration constant, then configKERNEL\_INTERRUPT\_PRIORITY sets the priority of interrupts that are used by the kernel itself. In this case, ISR safe FreeRTOS API functions (those that end in “FromISR”) must not be called from any interrupt that has been assigned a priority above that set by configKERNEL\_INTERRUPT\_PRIORITY. Interrupts that do not call API functions can execute at higher priorities to ensure the interrupt timing, determinism and latency is not adversely affected by anything the kernel is executing.

If a port implements both the configKERNEL\_INTERRUPT\_PRIORITY and the configMAX\_SYSCALL\_INTERRUPT\_PRIORITY configuration constants, then configKERNEL\_INTERRUPT\_PRIORITY sets the interrupt priority of interrupts that are used by the kernel itself, and configMAX\_SYSCALL\_INTERRUPT\_PRIORITY sets the maximum priority of interrupts from which ISR safe FreeRTOS API functions (those that end in “FromISR”) can be called. A full interrupt nesting model is achieved by setting configMAX\_SYSCALL\_INTERRUPT\_PRIORITY above (that is, at a higher priority level) than configKERNEL\_INTERRUPT\_PRIORITY. Interrupts that do not call API functions can execute at priorities above configMAX\_SYSCALL\_INTERRUPT\_PRIORITY to ensure the interrupt timing, determinism and latency is not adversely affected by anything the kernel is executing.

As an example – imagine a hypothetical microcontroller that has seven interrupt priority levels. In this hypothetical case, one is the lowest interrupt priority and seven is the highest interrupt priority[[2]](#footnote-2). Figure 4 describes what can and cannot be done at each priority level when configKERNEL\_INTERRUPT\_PRIORITY and configMAX\_SYSCALL\_INTERRUPT\_PRIORITY are set to one and three respectively.

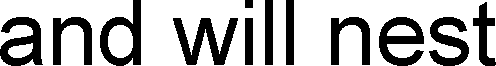
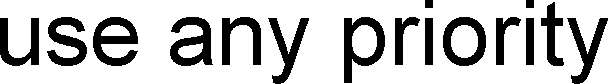
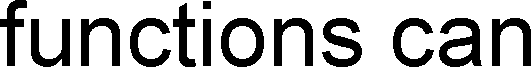
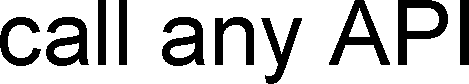
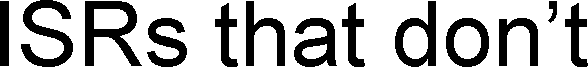
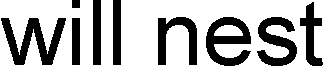
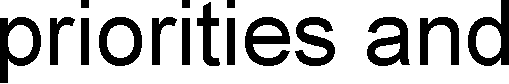
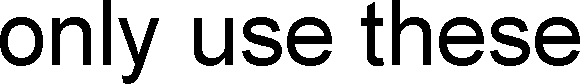
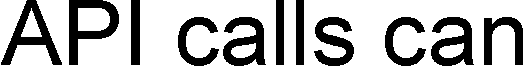
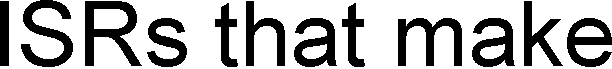
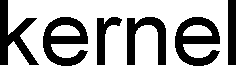
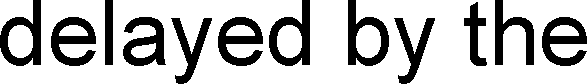
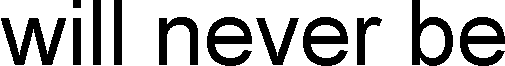
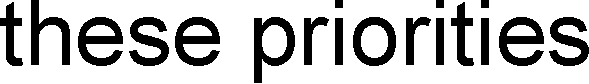
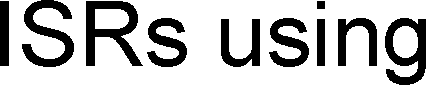
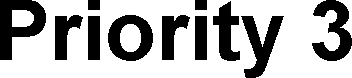
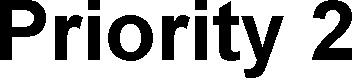
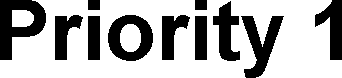
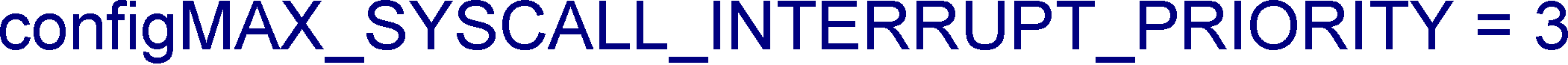


Figure 4 An example interrupt priority configuration

ISRs running above the configMAX\_SYSCALL\_INTERRUPT\_PRIORITY are never masked by the kernel itself, so their responsiveness is not affected by the kernel functionality. This is ideal for interrupts that require very high temporal accuracy – for example, interrupts that perform motor commutation. However, interrupts that have a priority above configMAX\_SYSCALL\_INTERRUPT\_PRIORITY cannot call any FreeRTOS API functions, even those that end in “FromISR” cannot be used.

configKERNEL\_INTERRUPT\_PRIORITY will nearly always, in not always, be set to the lowest available interrupt priority.

configMAX\_CO\_ROUTINE\_PRIORITIES

Sets the maximum priority that can be assigned to a co-routine. Co-routines can be assigned a priority from zero, which is the lowest priority, to (configMAX\_CO\_ROUTINE\_PRIORITIES – 1), which is the highest priority.

configMAX\_PRIORITIES

Sets the maximum priority that can be assigned to a task. Tasks can be assigned a priority from zero, which is the lowest priority, to (configMAX\_PRIORITIES – 1), which is the highest priority.

configMAX\_TASK\_NAME\_LEN

Sets the maximum number of characters that can be used for the name of a task. The NULL terminator is included in the count of characters.

configMAX\_SYSCALL\_INTERRUPT\_PRIORITY

See the description of the configKERNEL\_INTERRUPT\_PRIORITY configuration constant.

configMINIMAL\_STACK\_SIZE

Sets the size of the stack allocated to the Idle task. The value is specified in words, not bytes.

The kernel itself does not use configMINIMAL\_STACK\_SIZE for any other purpose, although the constant is used extensively by the standard demo tasks.

A demo application is provided for every official FreeRTOS port. The value of configMINIMAL\_STACK\_SIZE used in such a port specific demo application is the minimum recommended stack size for any task created using that port.

configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS

Thread local storage (or TLS) allows the application writer to store values inside a task's control block, making the value specific to (local to) the task itself, and allowing each task to have its own unique value.

Each task has its own array of pointers that can be used as thread local storage. The number of indexes in the array is set by configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS.

configQUEUE\_REGISTRY\_SIZE

Sets the maximum number of queues and semaphores that can be referenced from the queue registry at any one time. Only queues and semaphores that need to be viewed in a kernel aware debugging interface need to be registered.

The queue registry is only required when a kernel aware debugger is being used. At all other times it has no purpose and can be omitted by setting configQUEUE\_REGISTRY\_SIZE to 0, or by omitting the configQUEUE\_REGISTRY\_SIZE configuration constant definition altogether.

configTICK\_RATE\_HZ

Sets the tick interrupt frequency. The value is specified in Hz.

The pdMS\_TO\_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks. Block times specified this way will remain constant even when the configTICK\_RATE\_HZ definition is changed. pdMS\_TO\_TICKS() can only be used when configTICK\_RATE\_HZ is less than or equal to 1000. The standard demo tasks make extensive use of pdMS\_TO\_TICKS(), so they too can only be used when configTICK\_RATE\_HZ is less than or equal to 1000.

configTIMER\_QUEUE\_LENGTH

Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. configTIMER\_QUEUE\_LENGTH sets the maximum number of unprocessed commands that the timer command queue can hold at any one time.

Reasons the timer command queue might fill up include:

* Multiple timer API function calls being made before the scheduler has been started, and therefore before the timer service task has been created.
* Multiple (interrupt safe) timer API function calls being made from an interrupt service routine (ISR), and therefore not allowing the timer service task to process the commands.
* Multiple timer API function calls being made from a task that has a priority above that of the timer service task.

configTIMER\_TASK\_PRIORITY

Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. configTIMER\_TASK\_PRIORITY sets the priority of the timer service task. Like all tasks, the timer service task can run at any priority between 0 and ( configMAX\_PRIORITIES - 1 ).

This value needs to be chosen carefully to meet the requirements of the application. For example, if the timer service task is made the highest priority task in the system, then commands sent to the timer service task (when a timer API function is called), and expired timers, will both get processed immediately. Conversely, if the timer service task is given a low priority, then commands sent to the timer service task, and expired timers, will not be processed until the timer service task is the highest priority task that is able to run. It is worth noting however, that timer expiry times are calculated relative to when a command is sent, and not relative to when a command is processed.

configTIMER\_TASK\_STACK\_DEPTH

Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. configTIMER\_TASK\_STACK\_DEPTH sets the size of the stack (in words, not bytes) allocated to the timer service task.

Timer callback functions execute in the context of the timer service task. The stack requirement of the timer service task therefore depends on the stack requirements of the timer callback functions.

configTOTAL\_HEAP\_SIZE

The kernel allocates memory from the heap each time a task, queue or semaphore is created. The official FreeRTOS download includes three sample memory allocation schemes for this purpose. The schemes are implemented in the heap\_1.c, heap\_2.c, heap\_3.c and heap\_4.c source files respectively. The schemes defined by heap\_1.c, heap\_2.c and heap\_4.c allocate memory from a statically allocated array, known as the FreeRTOS heap. configTOTAL\_HEAP\_SIZE sets the size of this array. The size is specified in bytes.

The configTOTAL\_HEAP\_SIZE setting has no effect unless heap\_1.c, heap\_2.c or heap\_4.c are being used by the application.

configUSE\_16\_BIT\_TICKS

The tick count is held in a variable of type TickType\_t. When configUSE\_16\_BIT\_TICKS is set to 1, TickType\_t is defined to be an unsigned 16-bit type. When configUSE\_16\_BIT\_TICKS is set to 0, TickType\_t is defined to be an unsigned 32-bit type.

Using a 16-bit type can greatly improve efficiency on 8-bit and 16-bit microcontrollers, but at the cost of limiting the maximum block time that can be specified.

configUSE\_ALTERNATIVE\_API

Two sets of API functions are provided to send to, and receive from, queues – the standard API and the ‘alternative’ API. Only the standard API is documented in this manual. Use of the alternative API is no longer recommended.

Setting configUSE\_ALTERNATIVE\_API to 1 will include the alternative API functions in the build. Setting configUSE\_ALTERNATIVE\_API to 0 will exclude the alternative API functions from the build.

Note: Use of the alternative API is deprecated and therefore not recommended.

configUSE\_APPLICATION\_TASK\_TAG

Setting configUSE\_APPLICATION\_TASK\_TAG to 1 will include both the

vTaskSetApplicationTaskTag() and xTaskCallApplicationTaskHook() API functions in the build.

Setting configUSE\_APPLICATION\_TASK\_TAG to 0 will exclude both the vTaskSetApplicationTaskTag() and the xTaskCallApplicationTaskHook() API functions from the build.

configUSE\_CO\_ROUTINES

Co-routines are light weight tasks that save RAM by sharing a stack, but have limited functionality. Their use is omitted from this manual.

Setting configUSE\_CO\_ROUTINES to 1 will include all co-routine functionality and its associated API functions in the build. Setting configUSE\_CO\_ROUTINES to 0 will exclude all co-routine functionality and its associated API functions from the build.

configUSE\_COUNTING\_SEMAPHORES

Setting configUSE\_COUNTING\_SEMAPHORES to 1 will include the counting semaphore functionality and its associated API in the build. Setting configUSE\_COUNTING\_SEMAPHORES to 0 will exclude the counting semaphore functionality and its associated API from the build.

configUSE\_IDLE\_HOOK

The idle task hook function is a hook (or callback) function that, if defined and configured, will be called by the Idle task on each iteration of its implementation.

If configUSE\_IDLE\_HOOK is set to 1 then the application must define an idle task hook function. If configUSE\_IDLE\_HOOK is set to 0 then the idle task hook function will not be called, even if one is defined.

Idle task hook functions must have the name and prototype shown in Listing 203.

void vApplicationIdleHook( void );

Listing 203 The idle task hook function name and prototype.

configUSE\_MALLOC\_FAILED\_HOOK

The kernel uses a call to pvPortMalloc() to allocate memory from the heap each time a task, queue or semaphore is created. The official FreeRTOS download includes three sample memory allocation schemes for this purpose. The schemes are implemented in the heap\_1.c, heap\_2.c heap\_3.c and heap\_4.c source files respectively.

configUSE\_MALLOC\_FAILED\_HOOK is only relevant when one of these three sample schemes is being used.

The malloc() failed hook function is a hook (or callback) function that, if defined and configured, will be called if pvPortMalloc() ever returns NULL. NULL will be returned only if there is insufficient FreeRTOS heap memory remaining for the requested allocation to succeed.

If configUSE\_MALLOC\_FAILED\_HOOK is set to 1 then the application must define a malloc() failed hook function. If configUSE\_MALLOC\_FAILED\_HOOK is set to 0 then the malloc() failed hook function will not be called, even if one is defined.

Malloc() failed hook functions must have the name and prototype shown in Listing 204.

void vApplicationMallocFailedHook( void );

Listing 204 The malloc() failed hook function name and prototype.

configUSE\_MUTEXES

Setting configUSE\_MUTEXES to 1 will include the mutex functionality and its associated API in the build. Setting configUSE\_MUTEXES to 0 will exclude the mutex functionality and its associated API from the build.

configUSE\_NEWLIB\_REENTRANT

If configUSE\_NEWLIB\_REENTRANT is set to 1 then a newlib reent structure will be allocated for each created task.

Note Newlib support has been included by popular demand, but is not used by the FreeRTOS maintainers themselves. FreeRTOS is not responsible for resulting newlib operation. User must be familiar with newlib and must provide system-wide implementations of the necessary stubs. Be warned that (at the time of writing) the current newlib design implements a systemwide malloc() that must be provided with locks. configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION

Some FreeRTOS ports have two methods of selecting the next task to execute – a generic method, and a method that is specific to that port.

The Generic method:

* Is used when configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION is set to 0, or when a port specific method is not implemented.
* Can be used with all FreeRTOS ports.
* Is completely written in C, making it less efficient than a port specific method.
* Does not impose a limit on the maximum number of available priorities.

A port specific method:

* Is not available for all ports.
* Is used when configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION is set to 1.
* Relies on one or more architecture specific assembly instructions (typically a Count Leading Zeros [CLZ] of equivalent instruction) so can only be used with the architecture for which it was specifically written.
* Is more efficient than the generic method.
* Typically imposes a limit of 32 on the maximum number of available priorities.

configUSE\_PREEMPTION

Setting configUSE\_PREEMPTION to 1 will cause the pre-emptive scheduler to be used.

Setting configUSE\_PREEMPTION to 0 will cause the co-operative scheduler to be used.

When the pre-emptive scheduler is used the kernel will execute during each tick interrupt, which can result in a context switch occurring in the tick interrupt.

When the co-operative scheduler is used a context switch will only occur when either:

1. A task explicitly calls taskYIELD().
2. A task explicitly calls an API function that results in it entering the Blocked state.
3. An application defined interrupt explicitly performs a context switch.

configUSE\_QUEUE\_SETS

Setting configUSE\_QUEUE\_SETS to 1 will include queue set functionality (the ability to block on multiple queues at the same time) and its associated API in the build. Setting configUSE\_QUEUE\_SETS to 0 will exclude queue set functionality and its associated API from the build.

configUSE\_RECURSIVE\_MUTEXES

Setting configUSE\_RECURSIVE\_MUTEXES to 1 will cause the recursive mutex functionality and its associated API to be included in the build. Setting configUSE\_RECURSIVE\_MUTEXES to 0 will cause the recursive mutex functionality and its associated API to be excluded from the build.

configUSE\_STATS\_FORMATTING\_FUNCTIONS

Set configUSE\_TRACE\_FACILITY and configUSE\_STATS\_FORMATTING\_FUNCTIONS to 1 to include the vTaskList() and vTaskGetRunTimeStats().

functions in the build. Setting either to 0 will omit vTaskList() and vTaskGetRunTimeStates() from the build.

configUSE\_TASK\_NOTIFICATIONS

Setting configUSE\_TASK\_NOTIFICATIONS to 1 (or leaving configUSE\_TASK\_NOTIFICATIONS undefined) will include direct to task notification functionality and its associated API in the build. Setting configUSE\_TASK\_NOTIFICATIONS to 0 will exclude direct to task notification functionality and its associated API from the build.

Each task consumes 8 additional bytes of RAM when direct to task notifications are included in the build.

configUSE\_TICK\_HOOK

The tick hook function is a hook (or callback) function that, if defined and configured, will be called during each tick interrupt.

If configUSE\_TICK\_HOOK is set to 1 then the application must define a tick hook function. If configUSE\_TICK\_HOOK is set to 0 then the tick hook function will not be called, even if one is defined.

Tick hook functions must have the name and prototype shown in Listing 205.

void vApplicationTickHook( void );

Listing 205 The tick hook function name and prototype.

configUSE\_TICKLESS\_IDLE

Set configUSE\_TICKLESS\_IDLE to 1 to use the low power tickless mode, or 0 to keep the tick interrupt running at all times.

configUSE\_TIMERS

Setting configUSE\_TIMERS to 1 will include software timer functionality and its associated API in the build. Setting configUSE\_TIMERS to 0 will exclude software timer functionality and its associated API from the build.

If configUSE\_TIMERS is set to 1, then configTIMER\_TASK\_PRIORITY, configTIMER\_QUEUE\_LENGTH and configTIMER\_TASK\_STACK\_DEPTH must also be defined.

configUSE\_TIME\_SLICING

By default (if configUSE\_TIME\_SLICING is not defined, or if configUSE\_TIME\_SLICING is defined as 1) FreeRTOS uses prioritized preemptive scheduling with time slicing. That means the RTOS scheduler will always run the highest priority task that is in the Ready state, and will switch between tasks of equal priority on every RTOS tick interrupt. If configUSE\_TIME\_SLICING is set to 0 then the RTOS scheduler will still run the highest priority task that is in the Ready state, but will not switch between tasks of equal priority just because a tick interrupt executed.

configUSE\_TRACE\_FACILITY

Setting configUSE\_TRACE\_FACILITY to 1 will result in additional structure members and functions that assist with execution visualization and tracing being included in the build.

# APPENDIX 1: Data Types and Coding Style Guide

Data Types

Each port of FreeRTOS has a unique portmacro.h header file that contains (amongst other things) definitions for two special data types, TickType\_t and BaseType\_t. These data types are described in Table 3.

Table 3. Special data types used by FreeRTOS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Macro or typedef used | | | Actual type | |
| TickType\_t | This is used to store the tick count value, and by variables that specify block times.  TickType\_t can be either an unsigned 16-bit type or an unsigned 32-bit type, depending on the setting of configUSE\_16\_BIT\_TICKS within FreeRTOSConfig.h.  Using a 16-bit type can greatly improve efficiency on 8-bit and 16-bit architectures, but severely limits the maximum block period that can be specified. There is no reason to use a 16-bit type on a 32-bit architecture. | |
| BaseType\_t | This is always defined to be the most efficient data type for the | |

architecture. Typically, this is a 32-bit type on a 32-bit architecture, a 16-bit type on a 16-bit architecture, and an 8-bit type on an 8-bit architecture.

BaseType\_t is generally used for variables that can take only a very limited range of values, and for Booleans.

Standard data types other than ‘char’ are not used (see below), instead type names defined within the compiler’s stdint.h header file are used. ‘char’ types are only permitted to point to ASCII strings or reference single ASCII characters.

Variable Names

Variables are prefixed with their type: ‘c’ for char, ‘s’ for short, ‘l’ for long, and ‘x’ for BaseType\_t and any other types (structures, task handles, queue handles, etc.).

If a variable is unsigned, it is also prefixed with a ‘u’. If a variable is a pointer, it is also prefixed with a ‘p’. Therefore, a variable of type unsigned char will be prefixed with ‘uc’, and a variable of type pointer to char will be prefixed with ‘pc’.

Function Names

Functions are prefixed with both the type they return and the file they are defined in. For example:

* vTaskPrioritySet() returns a void and is defined within task.c.
* xQueueReceive() returns a variable of type BaseType\_t and is defined within queue.c.
* vSemaphoreCreateBinary() returns a void and is defined within semphr.h.

File scope (private) functions are prefixed with ‘prv’.

Formatting

One tab is always set to equal four spaces.

Macro Names

Most macros are written in upper case and prefixed with lower case letters that indicate where the macro is defined. Table 4 provides a list of prefixes.

Table 4. Macro prefixes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prefix | | Location of macro definition | | |
| port (for example, portMAX\_DELAY) | | portable.h |
| task (for example, taskENTER\_CRITICAL()) | | task.h |
| pd (for example, pdTRUE) | | projdefs.h |
| config (for example, configUSE\_PREEMPTION) | | FreeRTOSConfig.h |
| err (for example, errQUEUE\_FULL) | | projdefs.h |

Note that the semaphore API is written almost entirely as a set of macros, but follows the function naming convention, rather than the macro naming convention.

The macros defined in Table 5 are used throughout the FreeRTOS source code.

Table 5. Common macro definitions

|  |  |  |  |
| --- | --- | --- | --- |
| Macro | | Value | |
| pdTRUE | 1 |
| pdFALSE | 0 |
| pdPASS | 1 |
| pdFAIL | 0 |

Rationale for Excessive Type Casting

The FreeRTOS source code can be compiled with many different compilers, all of which differ in how and when they generate warnings. In particular, different compilers want casting to be used in different ways. As a result, the FreeRTOS source code contains more type casting than would normally be warranted.

# APPENDIX 2: 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 6.
* You wish to receive direct technical support.
* You wish to have assistance with your development.
* You require guarantees and indemnification.

Table 6. 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) modified by an exception.

The full text of the GPL is available at http://www.freertos.org/license.txt. The text of the exception that modified the GPL license 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

API Usage Restrictions, 17

C

configASSERT, 273 configCHECK\_FOR\_STACK\_OVERFLOW, 273 configCPU\_CLOCK\_HZ, 275 configGENERATE\_RUN\_TIME\_STATS, 276 configIDLE\_SHOULD\_YIELD, 277 configINCLUDE\_APPLICATION\_DEFINED\_PRIVILE

GED\_FUNCTIONS, 278 configKERNEL\_INTERRUPT\_PRIORITY, 279 configMAX\_CO\_ROUTINE\_PRIORITIES, 281 configMAX\_PRIORITIES, 30, 104, 281

configMAX\_SYSCALL\_INTERRUPT\_PRIORITY, 279,

281

configMAX\_TASK\_NAME\_LEN, 281 configMINIMAL\_STACK\_DEPTH, 29 configMINIMAL\_STACK\_SIZE, 281

configNUM\_THREAD\_LOCAL\_STORAGE\_POINTER

S, 281

configQUEUE\_REGISTRY\_SIZE, 282 configTICK\_RATE\_HZ, 282 configTIMER\_QUEUE\_LENGTH, 282 configTIMER\_TASK\_PRIORITY, 283 configTIMER\_TASK\_STACK\_DEPTH, 283 configTOTAL\_HEAP\_SIZE, 284 configUSE\_16\_BIT\_TICKS, 284 configUSE\_ALTERNATIVE\_API, 284 configUSE\_APPLICATION\_TASK\_TAG, 284 configUSE\_CO\_ROUTINES, 285 configUSE\_COUNTING\_SEMAPHORES, 285 configUSE\_IDLE\_HOOK, 285 configUSE\_MALLOC\_FAILED\_HOOK, 285 configUSE\_MUTEXES, 286 configUSE\_NEWLIB\_REENTRANT, 286

configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION,

287 configUSE\_PREEMPTION, 287 configUSE\_QUEUE\_SETS, 288 configUSE\_RECURSIVE\_MUTEXES, 288 configUSE\_STATS\_FORMATTING\_FUNCTIONS, 288 configUSE\_TICK\_HOOK, 288, 289 configUSE\_TICKLESS\_IDLE, 289 configUSE\_TIME\_SLICING, 289 configUSE\_TIMERS, 289 configUSE\_TRACE\_FACILITY, 290

D

Data Types, 292

E

eTaskGetState(), 65

F

Formatting, 293

FreeRTOSConfig.h, 268 Function Names, 293

H

high water mark, 63 highest priority, 30, 104 I

INCLUDE\_ xTimerPendFunctionCall, 272

INCLUDE\_eTaskGetState, 271

INCLUDE\_pcTaskGetTaskName, 270

INCLUDE\_uxTaskGetStackHighWaterMark, 270

INCLUDE\_uxTaskPriorityGet, 270

INCLUDE\_vTaskDelay, 269

INCLUDE\_vTaskDelayUntil, 270

INCLUDE\_vTaskDelete, 270

INCLUDE\_vTaskPrioritySet, 271

INCLUDE\_vTaskSuspend, 271

INCLUDE\_xEventGroupSetBitFromISR, 269

INCLUDE\_xSemaphoreGetMutexHolder, 269

INCLUDE\_xTaskGetCurrentTaskHandle, 270

INCLUDE\_xTaskGetIdleTaskHandle, 270

INCLUDE\_xTaskGetSchedulerState, 270

INCLUDE\_xTaskResumeFromISR, 271

INCLUDE\_xTimerGetTimerDaemonTaskHandle, 271

L

lowest priority, 30, 104

M

Macro Names, 293

P

pcTaskGetTaskName(), 73 pcTimerGetTimerName(), 222 portBASE\_TYPE, 292

portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS,

59, 276

portGET\_RUN\_TIME\_COUNTER\_VALUE, 60, 277 portMAX\_DELAY, 151, 155, 168 portSWITCH\_TO\_USER\_MODE(), 21 portTickType, 292

priority, 30 pvTaskGetTheadLocalStoragePointer(), 71 pvTimerGetTimerID(), 220

T

tabs, 293 task handle, 30, 33

|  |  |
| --- | --- |
| taskDISABLE\_INTERRUPTS(), 45 taskENABLE\_INTERRUPTS(), 47 taskENTER\_CRITICAL(), 48 taskEXIT\_CRITICAL(), 51  taskYIELD(), 127 Type Casting, 294  U  ulTaskNotifyTake(), 96 uxQueueMessagesWaiting(), 144 uxQueueMessagesWaitingFromISR(), 145  uxQueueSpacesAvailable(), 175 uxTaskGetNumberOfTasks(), 57 uxTaskGetStackHighWaterMark (), 63 uxTaskGetSystemState(), 67 uxTaskPriorityGet(), 102  V  Variable Names, 293 vEventGroupDelete(), 252 vQueueAddToRegistry(), 130 vQueueDelete(), 140 vSemaphoreCreateBinary(), 178 vSemaphoreDelete(), 189 vTaskDelay(), 38 vTaskDelayUntil(), 40 vTaskDelete(), 43 vTaskGetRunTimeStats(), 58 vTaskList(), 78 vTaskNotifyGiveFromISR(), 93  vTaskPrioritySet(), 104 vTaskResume(), 106  vTaskSetApplicationTaskTag(), 113 vTaskSetThreadLocalStoragePointer(), 115 vTaskSetTimeOutState(), 117 vTaskStartScheduler(), 119 vTaskStepTick(), 121 vTaskSuspend(), 123 vTaskSuspendAll(), 125 vTimerSetTimerID(), 235  X  xEventGroupClearBits(), 248 xEventGroupClearBitsFromISR(), 250 xEventGroupCreate(), 246 xEventGroupGetBits(), 253 xEventGroupGetBitsFromISR(), 254 xEventGroupSetBits(), 255 xEventGroupSetBitsFromISR(), 257 xEventGroupSync(), 260 xEventGroupWaitBits(), 264 xQueueAddToSet(), 132 xQueueCreate(), 134 xQueueCreateSet(), 136 xQueueIsQueueEmptyFromISR(), 142 xQueueIsQueueFullFromISR(), 143 | xQueueOverwrite(), 147 xQueueOverwriteFromISR(), 149 xQueuePeek(), 151 xQueuePeekFromISR(), 154 xQueueReceive(), 155 xQueueReceiveFromISR(), 158, 202 xQueueRemoveFromSet(), 161 xQueueReset(), 163 xQueueSelectFromSet(), 164 xQueueSelectFromSetFromISR(), 166 xQueueSend(), 168 xQueueSendFromISR(), 171 xQueueSendToBack(), 168 xQueueSendToBackFromISR(), 171 xQueueSendToFront(), 168 xQueueSendToFrontFromISR(), 171 xSemaphoreCreateBinary(), 181 xSemaphoreCreateCounting(), 183 xSemaphoreCreateMutex(), 185 xSemaphoreCreateRecursiveMutex(), 187 xSemaphoreGetMutexHolder(), 190 xSemaphoreGive(), 191 xSemaphoreGiveFromISR(), 193 xSemaphoreGiveRecursive(), 196 xSemaphoreTake(), 199 xSemaphoreTakeRecursive(), 204 xTaskAllocateMPURegions(), 22 xTaskCallApplicationHook(), 25 xTaskCheckForTimeOut(), 27 xTaskCreate(), 29 xTaskCreateRestricted(), 33 xTaskGetApplicationTaskTag(), 53 xTaskGetCurrentTaskHandle(), 55 xTaskGetIdleTaskHandle(), 56 xTaskGetSchedulerState(), 62 xTaskGetTickCount(), 74 xTaskGetTickCountFromISR(), 76  xTaskNotify(), 81 xTaskNotifyAndQuery(), 84 xTaskNotifyFromISR(), 87 xTaskNotifyGive(), 91 xTaskNotifyWait(), 99 xTaskResumeAll(), 108 xTaskResumeFromISR(), 110 xTimerChangePeriod(), 208 xTimerChangePeriodFromISR(), 211  xTimerCreate(), 213 xTimerDelete(), 217 xTimerGetTimerDaemonTaskHandle(), 219  xTimerIsTimerActive(), 223 xTimerPendFunctionCall (), 225 xTimerPendFunctionCallFromISR(), 227  xTimerReset(), 230 xTimerResetFromISR(), 233 xTimerStart(), 237 xTimerStartFromISR(), 239 xTimerStop(), 241 xTimerStopFromISR(), 243 |
|  |  |

1. The ‘alternative’ API is not included as its use is no longer recommended. The co-routine API is also omitted as co-routines are only useful to a small subset of applications. [↑](#footnote-ref-1)
2. Note care must be taken when assigning values to configKERNEL\_INTERRUPT\_PRIORITY and configMAX\_SYCALL\_INTERRUPT\_PRIORITY as some microcontrollers use zero or one to mean the lowest priority, while others use zero or one to mean the highest priority. [↑](#footnote-ref-2)