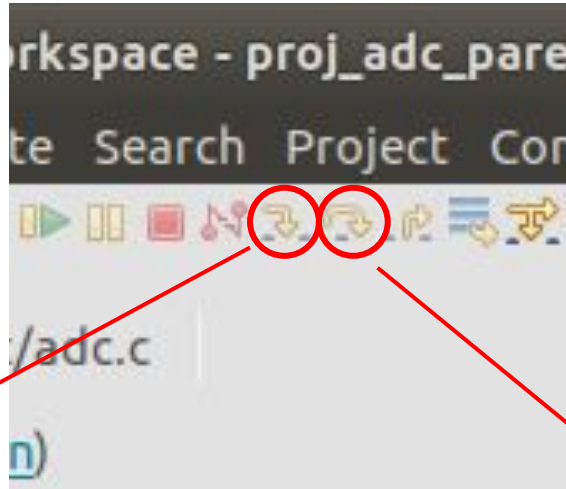


cross-debugging, semaphores & timers

ese3025

cross-debugging

“**sequential**”
debugging tools:



“**step into**”

(follows execution into called functions)

“**step over**”

(skips over function calls)

breakpoints (extremely useful in RTOS, use strategically!)

- can answer the question: “Does my task even run?” (put a breakpoint in the task body)
- can allow sequential debugging within a task is activated

FreeRTOS Software Timers

Software timers are used to schedule the execution of a function at a set time in the future, or periodically with a fixed frequency. The function executed by the software timer is called the software timer's callback function.

Software timers are implemented by, and are under the control of, the FreeRTOS kernel. They do not require hardware support, and are not related to hardware timers or hardware counters.

Note that, in line with the FreeRTOS philosophy of using innovative design to ensure maximum efficiency, software timers do not use any processing time unless a software timer callback function is actually executing.

Software timer functionality is optional. To include software timer functionality:

1. Build the FreeRTOS source file `FreeRTOS/Source/timers.c` as part of your project.
2. Set `configUSE_TIMERS` to 1 in `FreeRTOSConfig.h`.

FreeRTOS Software Timers (cont'd)

The RTOS Daemon (Timer Service) Task

All software timer callback functions execute in the context of the same RTOS daemon (or 'timer service') task¹.

The daemon task is a standard FreeRTOS task that is created automatically when the scheduler is started. Its priority and stack size are set by the `configTIMER_TASK_PRIORITY` and `configTIMER_TASK_STACK_DEPTH` compile time configuration constants respectively. Both constants are defined within `FreeRTOSConfig.h`.

Software timer callback functions must not call FreeRTOS API functions that will result in the calling task entering the Blocked state, as to do so will result in the daemon task entering the Blocked state.

The Timer Command Queue

Software timer API functions send commands from the calling task to the daemon task on a queue called the 'timer command queue'. This is shown in Figure 41. Examples of commands include 'start a timer', 'stop a timer' and 'reset a timer'.

The timer command queue is a standard FreeRTOS queue that is created automatically when the scheduler is started. The length of the timer command queue is set by the `configTIMER_QUEUE_LENGTH` compile time configuration constant in `FreeRTOSConfig.h`.

```

/* The periods assigned to the one-shot and auto-reload timers are 3.333 second and half a
second respectively. */
#define mainONE_SHOT_TIMER_PERIOD    pdMS_TO_TICKS( 3333 )
#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS( 500 )

int main( void )
{
TimerHandle_t xAutoReloadTimer, xOneShotTimer;
BaseType_t xTimer1Started, xTimer2Started;

/* Create the one shot timer, storing the handle to the created timer in xOneShotTimer. */
xOneShotTimer = xTimerCreate(
    /* Text name for the software timer - not used by FreeRTOS. */
    "OneShot",
    /* The software timer's period in ticks. */
    mainONE_SHOT_TIMER_PERIOD,
    /* Setting uxAutoReload to pdFALSE creates a one-shot software timer. */
    pdFALSE,
    /* This example does not use the timer id. */
    0,
    /* The callback function to be used by the software timer being created. */
    prvOneShotTimerCallback );

/* Create the auto-reload timer, storing the handle to the created timer in xAutoReloadTimer. */
xAutoReloadTimer = xTimerCreate(
    /* Text name for the software timer - not used by FreeRTOS. */
    "AutoReload",
    /* The software timer's period in ticks. */
    mainAUTO_RELOAD_TIMER_PERIOD,
    /* Setting uxAutoReload to pdTRUE creates an auto-reload timer. */
    pdTRUE,
    /* This example does not use the timer id. */
    0,
    /* The callback function to be used by the software timer being created. */
    prvAutoReloadTimerCallback );

/* Check the software timers were created. */
if( ( xOneShotTimer != NULL ) && ( xAutoReloadTimer != NULL ) )
{
    /* Start the software timers, using a block time of 0 (no block time). The scheduler has
    not been started yet so any block time specified here would be ignored anyway. */
    xTimer1Started = xTimerStart( xOneShotTimer, 0 );
    xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );

    /* The implementation of xTimerStart() uses the timer command queue, and xTimerStart()
    will fail if the timer command queue gets full. The timer service task does not get
    created until the scheduler is started, so all commands sent to the command queue will
    stay in the queue until after the scheduler has been started. Check both calls to
    xTimerStart() passed. */
    if( ( xTimer1Started == pdPASS ) && ( xTimer2Started == pdPASS ) )
    {
        /* Start the scheduler. */
        vTaskStartScheduler();
    }
}

/* As always, this line should not be reached. */
for( ;; );
}

```

```

static void prvOneShotTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow;

    /* Obtain the current tick count. */
    xTimeNow = xTaskGetTickCount();

    /* Output a string to show the time at which the callback was executed. */
    vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );

    /* File scope variable. */
    ulCallCount++;
}

```

Listing 76. The callback function used by the one-shot timer in Example 13

```

static void prvAutoReloadTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow;

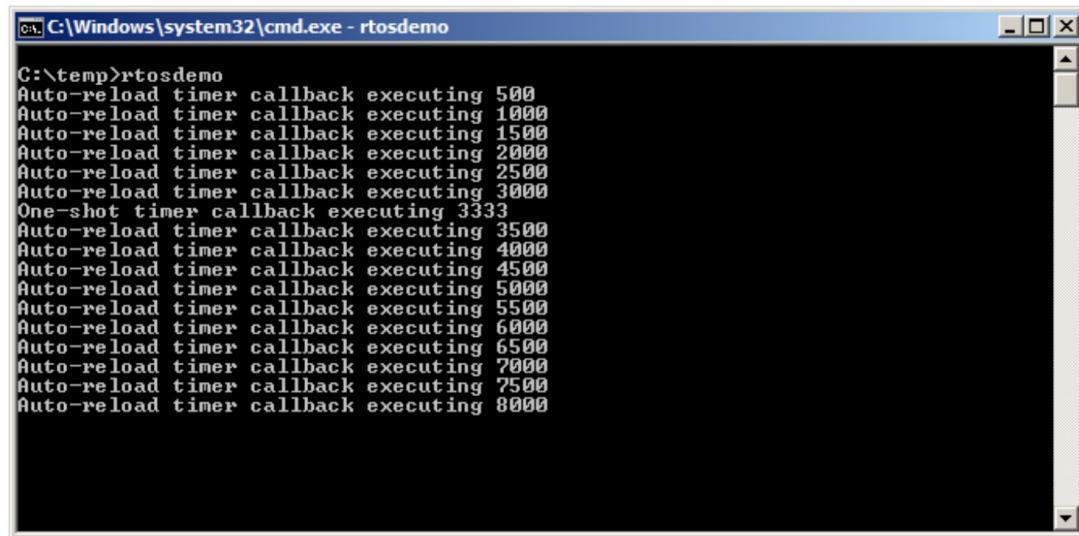
    /* Obtain the current tick count. */
    xTimeNow = uxTaskGetTickCount();

    /* Output a string to show the time at which the callback was executed. */
    vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow );

    ulCallCount++;
}

```

Listing 77. The callback function used by the auto-reload timer in Example 13



```
C:\Windows\system32\cmd.exe - rtosdemo

C:\temp>rtosdemo
Auto-reload timer callback executing 500
Auto-reload timer callback executing 1000
Auto-reload timer callback executing 1500
Auto-reload timer callback executing 2000
Auto-reload timer callback executing 2500
Auto-reload timer callback executing 3000
One-shot timer callback executing 3333
Auto-reload timer callback executing 3500
Auto-reload timer callback executing 4000
Auto-reload timer callback executing 4500
Auto-reload timer callback executing 5000
Auto-reload timer callback executing 5500
Auto-reload timer callback executing 6000
Auto-reload timer callback executing 6500
Auto-reload timer callback executing 7000
Auto-reload timer callback executing 7500
Auto-reload timer callback executing 8000
```

Figure 44 The output produced when Example 13 is executed

FreeRTOS (Counting) Semaphores

- while a mutex can be regarded as a queue of length one, a counting semaphore can be viewed as a queue of length greater than one
- according to Richard Barry: “Each time a counting semaphore is ‘given’, another space in its queue is used. The number of items in the queue is the semaphore’s ‘count’ value”

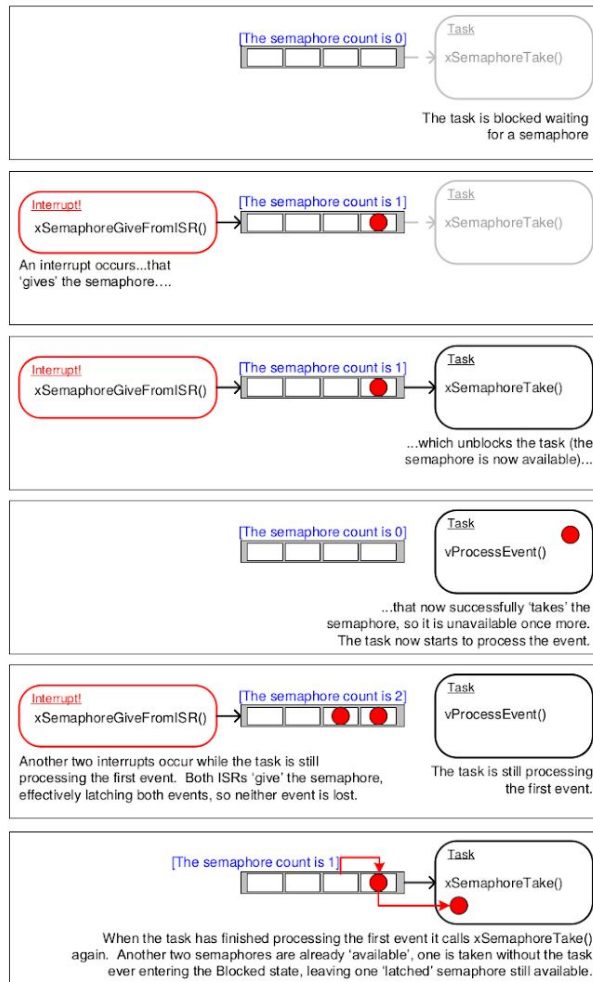


Figure 55. Using a counting semaphore to 'count' events

The xSemaphoreCreateCounting() API Function

FreeRTOS V9.0.0 also includes the xSemaphoreCreateCountingStatic() function, which allocates the memory required to create a counting semaphore statically at compile time: Handles to all the various types of FreeRTOS semaphore are stored in a variable of type SemaphoreHandle_t.

Before a semaphore can be used, it must be created. To create a counting semaphore, use the xSemaphoreCreateCounting() API function.

```
SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount,  
                                             UBaseType_t uxInitialCount );
```

Listing 97. The xSemaphoreCreateCounting() API function prototype

Table 36. xSemaphoreCreateCounting() parameters and return value

Parameter Name/ Returned Value	Description
uxMaxCount	<p>The maximum value to which the semaphore will count. To continue the queue analogy, the uxMaxCount value is effectively the length of the queue.</p> <p>When the semaphore is to be used to count or latch events, uxMaxCount is the maximum number of events that can be latched.</p> <p>When the semaphore is to be used to manage access to a collection of resources, uxMaxCount should be set to the total number of resources that are available.</p>
uxInitialCount	<p>The initial count value of the semaphore after it has been created.</p> <p>When the semaphore is to be used to count or latch events, uxInitialCount should be set to zero—as, presumably, when the semaphore is created, no events have yet occurred.</p> <p>When the semaphore is to be used to manage access to a collection of resources, uxInitialCount should be set to equal uxMaxCount—as, presumably, when the semaphore is created, all the resources are available.</p>

NOTE:

returns NULL if there is insufficient heap memory available to create the semaphore, so use this API function in conjunction with an if statement, in general

use in conjunction with xSemaphoreTake() and xSemaphoreGive() API functions, as you did with mutexes

be careful using FreeRTOS API functions within Interrupt Service Routines...

The xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() API Functions

```
BaseType_t xQueueSendToFrontFromISR( QueueHandle_t xQueue,  
                                     void *pvItemToQueue  
                                     BaseType_t *pxHigherPriorityTaskWoken  
                                     );
```

Listing 105. The xQueueSendToFrontFromISR() API function prototype

```
BaseType_t xQueueSendToBackFromISR( QueueHandle_t xQueue,  
                                     void *pvItemToQueue  
                                     BaseType_t *pxHigherPriorityTaskWoken  
                                     );
```

Listing 106. The xQueueSendToBackFromISR() API function prototype

xQueueSendFromISR() and xQueueSendToBackFromISR() are functionally equivalent.

Table 38. xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() parameters and return values

Parameter Name/ Returned Value	Description
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.