FreeRTOS: A Very Short Introduction

Real-Time Operative Systems

Paulo Pedreiras
DETI/UA/IT
2021/2022

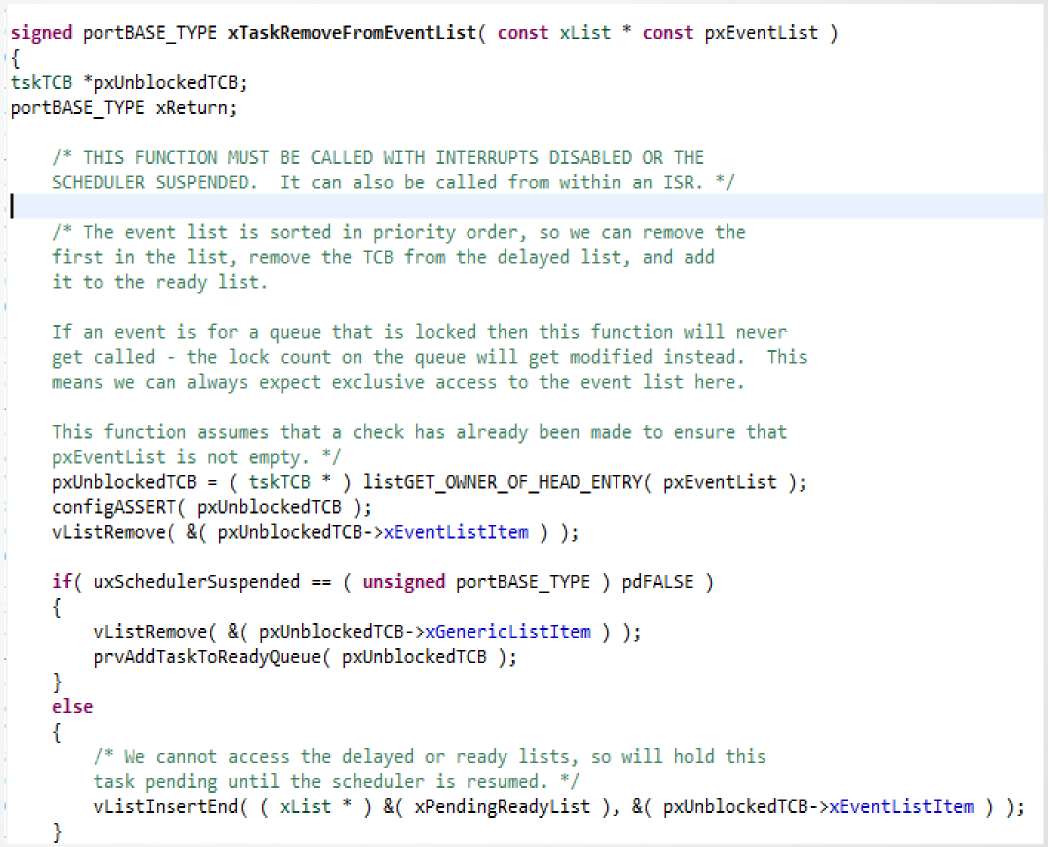
"… market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors."

FreeRTOS Features

- Currently owned by Amazon
- Open Source
- Portable
- Scalable
- Preemptive and co-operative scheduling
- Multitasking Services
- Interrupt management
- Advanced features
- Popular (used by Espressif ESP32, TI SimpleLink SDK, ...)

Source Code

- High quality
- Well documented
- Consistent and well organized
- Numerous examples
- Sample code for many platforms
- Big developer community



```
signed portBASE_TYPE xTaskRemoveFromEventList( const xList * const pxEventList )
{
tskTCB *pxUnblockedTCB;
portBASE_TYPE xReturn;

    /* THIS FUNCTION MUST BE CALLED WITH INTERRUPTS DISABLED OR THE
    SCHEDULER SUSPENDED.  It can also be called from within an ISR. */

    /* The event list is sorted in priority order, so we can remove the
    first in the list, remove the TCB from the delayed list, and add
    it to the ready list.

    If an event is for a queue that is locked then this function will never
    get called - the lock count on the queue will get modified instead.  This
    means we can always expect exclusive access to the event list here.

    This function assumes that a check has already been made to ensure that
    pxEventList is not empty. */
    pxUnblockedTCB = ( tskTCB * ) listGET_OWNER_OF_HEAD_ENTRY( pxEventList );
    configASSERT( pxUnblockedTCB );
    vListRemove( &( pxUnblockedTCB->xEventListItem ) );

    if( uxSchedulerSuspended == ( unsigned portBASE_TYPE ) pdFALSE )
    {
        vListRemove( &( pxUnblockedTCB->xGenericListItem ) );
        prvAddTaskToReadyQueue( pxUnblockedTCB );
    }
    else
    {
        /* We cannot access the delayed or ready lists, so will hold this
        task pending until the scheduler is resumed. */
        vListInsertEnd( ( xList * ) &( xPendingReadyList ), &( pxUnblockedTCB->xEventListItem ) );
    }
```

Portable

- Highly portable C
- More than 35 architectures supported
- Just a few core parts written in assembly
- Ports are freely available in source code.

Scalable

- Only use the services you need.
  - FreeRTOSConfig.h

- Small footprint
  - RL78 port can create 13 tasks, 2 queues and 4 software timers in under 4 kbytes of RAM!

- Modular. Services can be added on demand
  - Lists
  - Queues
  - Semaphores
  - Software timers
  - And many more ...

Preemptive and Cooperative Scheduling

- Preemptive scheduling:
  - Fully preemptive
  - Always runs the highest priority task that is ready to run
  - Comparable with other preemptive kernels
  - Used in conjunction with tasks

- Cooperative scheduling:
  - Context switch occurs if:
    - A task/co-routine blocks
    - Or a task/co-routine yields the CPU
  - Used in conjunction with tasks/co-routines

Task management

- Not restricted on:
  - # of tasks that can be created
  - # of priorities that can be used
  - Priority assignment
    - More than one task can be assigned the same priority.
    - RR with time slice = 1 RTOS tick

Services

- Queues
- Semaphores
  - Binary and counting
- Mutexes
  - With priority inheritance
  - Support recursion

Advanced Features

- Execution tracing
- Runtime statistics collection
- Memory management
- Memory protection support
- Stack overflow protection

Ecosystem

- TCP/IP stack
- USB stack
  - Host and device
- File systems
  - DOS compatible FAT
- CLI
- Safety and certification
- And many more ...

Licensing

- MIT Open Source License Text
- Optional Commercial Licensing
  - OpenRTOS: commercially licensed version of the FreeRTOS kernel, which includes indemnification and dedicated support. Shares source code with FreeRTOS.

  - SAFERTOS: derivative version of the FreeRTOS kernel that has been analyzed, documented and tested to meet the stringent requirements of industrial (IEC 61508 SIL 3), medical (IEC 62304 and FDA 510(K)) automotive (ISO 26262) and other international safety standards.

Preliminary details

- “Excessive” type casting
  - Approach to deal with the constraints of diverse compilers
  - e.g. “char” type without a qualifier is handled as signed by some compilers and as unsigned by others

- Rules:
  - Variable names preceded by
    - c: char / s: short / l: long / v:void p: pointer / ...
    - x: portBASE_TYPE (type that is architecture dependent – 8/16/32 bit)

Preliminary details

- Rules (cont):
  - Function names:
    - Return type+file name in which they are defined
      - e.g. vTaskPrioritySet()
      - Returns “void” and is implemented in “task.c”
  - Macros
    - Uppercase names with lowercase prefix identifying where they are defined:
      - e.g. portMAX_DELAY
        - Definied in “port.h”
        - Called “MAX_DELAY”
  - Some common macros:
    - pdTRUE (1) ; pdFALSE (0) ; pdPASS (1) ; pdFAIL (0)

Preliminary details

- Creating a project
  - Recommended to start from a working demo and adapt to the needs
  - Open the demo and see if it compiles
  - Remove all demo-specific files
  - Remove all functions from “main()” except “prvSetupHardware()”
  - Set to “0” the following constants at “FreeRTOSConfig.h”
    - config{USE_IDLE_HOOK;USE_TICK_HOOK;USE_MALLOC_FAILED_HOOK;CHECK_FOR_STACK_OVERFLOW}
    - Of course, set to 1 when needed

Preliminary details

- Template for a main function

```
short main( void )
{
        /* Init hardware */
        prvSetupHardware();

        /* Create application tasks */
        ...

        /* Set the scheduler running.  This function will not return unless a task calls
        vTaskEndScheduler() or there are no resources */
        vTaskStartScheduler();

        return 0;
}
```

Preliminary details

- System runs on “ticks
  - Every tick the kernel runs and figures out what to do next.
  - Interrupts have a different mechanism
- Hardware timer is set to generate regular interrupts and calls the scheduler.
  - <u>**FreeRTOS uses one of the CPU timers, which shouldn’t be shared**</u>

Tasks

- Each task is a function that must not return
  - So it's in an infinite loop. Standard in Real-Time and Embedded Operative Systems.
- The API allows to define:
  - The resources required by the task (stack space, priority, ...)
  - Any arguments the tasks needs
- All tasks must be of void return type and take a single void* as an argument.
  - The pointer is cast as needed to get the actual argument.

Task structure example

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Busy wait! Bad approach. It will be improved latter.

Task creation

From the task.h file in FreeRTOS

```
portBASE_TYPE xTaskCreate(
  pdTASK_CODE pvTaskCode,
  const char * const pcName,
  unsigned short usStackDepth,
  void *pvParameters,
  unsigned portBASE_TYPE uxPriority,
  xTaskHandle *pvCreatedTask
  );
```

Create a new task and add it to the list of tasks that are ready to run. **xTaskCreate()** can only be used to create a task that has unrestricted access to the entire microcontroller memory map. Systems that include MPU support can alternatively create an MPU constrained task using xTaskCreateRestricted().

- **pvTaskCode:** Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- **pcName:** A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by tskMAX_TASK_NAME_LEN – default is 16.
- **usStackDepth:** The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage.
- **pvParameters**: Pointer that will be used as the parameter for the task being created.
- **uxPriority:** The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE_BIT).
- **pvCreatedTask:** Used to pass back a handle by which the created task can be referenced.
- **pdPASS**: If the task was successfully created and added to a ready list, otherwise an error code defined in the file errors.h

Creating a task: example

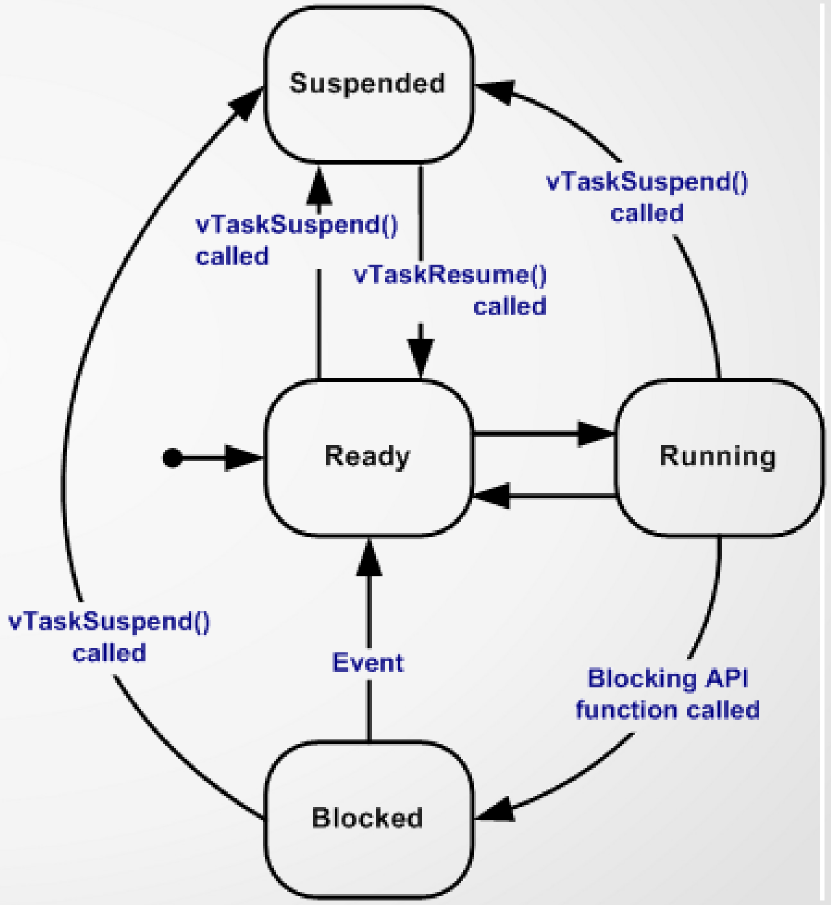
```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - most small microcontrollers will use
                             much less stack than this. */
                    NULL,    /* We are not using the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
```

Task scheduling and dispatching

- Tasks run if ready and there are no other tasks of higher priority
  - If other tasks of the same priority: Round-Robin.
- The kernel needs to know if a task is ready
  - In the previous example the task was always ready
    - Lower priority tasks will never execute
  - There are API calls that allow to inform the kernel that the task has no workload
    - E.g. **`vTaskDelay(x)`**

Task states in FreeRTOS

- **Running**
  - Task is actually executing
- **Ready**
  - Task is ready to execute but a task of equal or higher priority is Running.
- **Blocked**
  - Task is waiting for some event.
    - **Time**: if a task calls vTaskDelay() it will block until the delay period has expired.
    - **Resource**: Tasks can also block waiting for queue and semaphore events.
- **Suspended**
  - Much like blocked, but not waiting for anything.
  - Tasks will only enter or exit the suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.



Task management API

- The API allows several operations on tasks:
  - Change priority of a task
  - Delete a task
  - Suspend a task
  - Get priority of a task.
- Example
  - Set the priority of a task.

```
void vTaskPrioritySet(
xTaskHandle pxTask,
unsigned uxNewPriority );
```

- **pxTask:** Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set.
- **uxNewPriority:** The priority to which the task will be set.

Interrupts

- Interrupts are essentially handled at low-level.
  - **Very** port dependent.
  - Interrupts need to be configured completely
  - E.g. for serial device-driver (code should be familiar)

```
/* The UART interrupt handler.  As this uses the FreeRTOS assembly
interrupt entry point the IPL setting in the following prototype has no
effect.  The interrupt priority is set by the call to  ConfigIntUART2() in
xSerialPortInitMinimal(). */
void __attribute__( (interrupt(IPL0AUTO), vector(_UART2_VECTOR)))
vU2InterruptWrapper( void );
```

Deferred Interrupt Processing

- ISR should be kept short and simple
- Complex processing should be carried out by a task, not the ISR
- Synchronization can be carried out e.g. via a semaphore.
  - When the interrupt happens, the ISR carries the elemental processing and sets the semaphore before exiting.
    - Task can now be scheduled like any other. No need to worry about nesting interrupts (and thus interrupt priority).
    - FreeRTOS does support nested interrupts on some platforms though.
  - Concept similar to the Fast handler/Slow handler used e.g. in Linux

Semaphore example in FreeRTOS

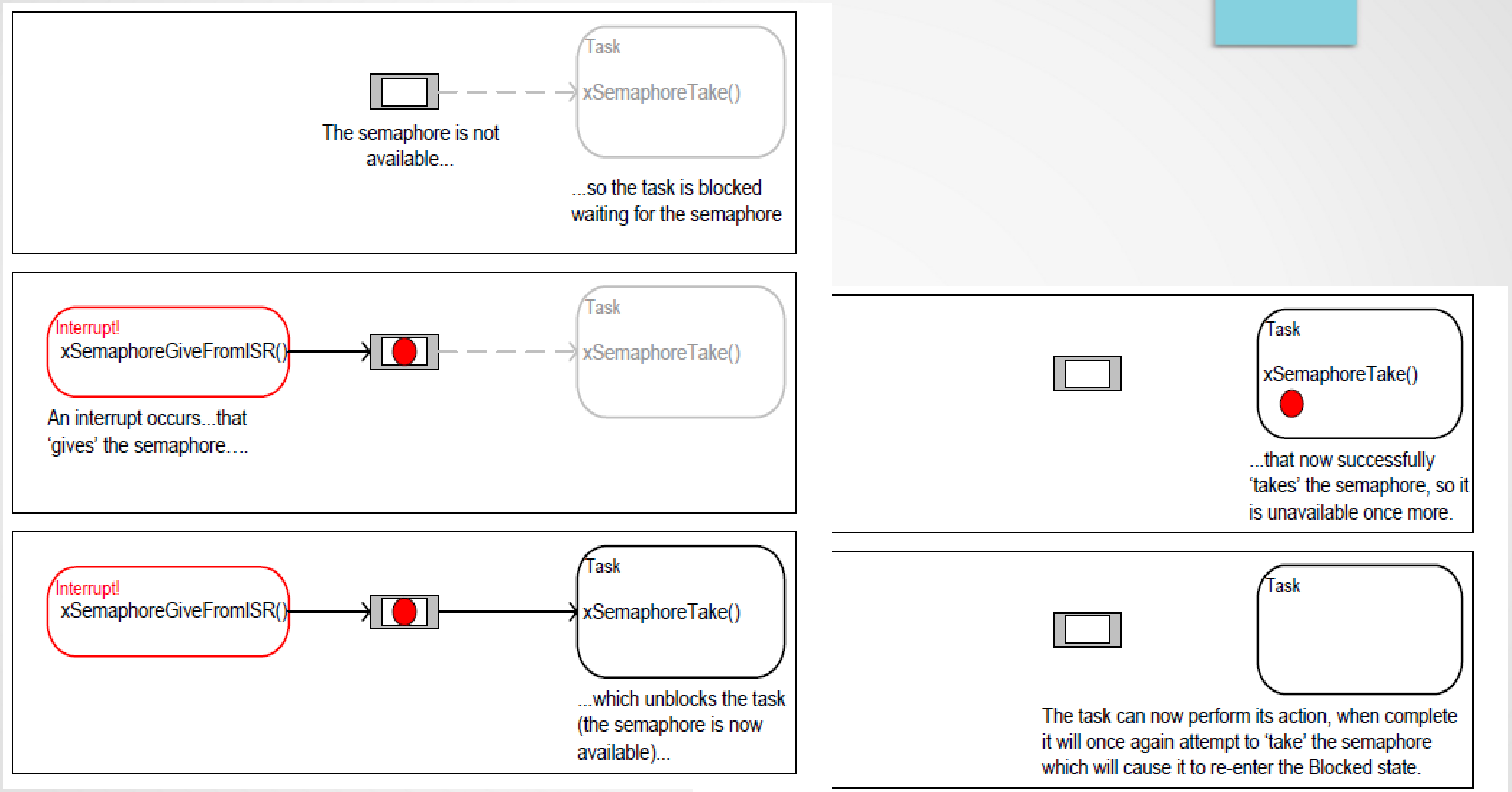

Task
xSemaphoreTake()
The semaphore is not available...
...so the task is blocked waiting for the semaphore
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...
Task
xSemaphoreTake()
...that now successfully 'takes' the semaphore, so it is unavailable once more.
Task
The task can now perform its action, when complete it will once again attempt to 'take' the semaphore which will cause it to re-enter the Blocked state.

Figure 26. Using a binary semaphore to synchronize a task with an interrupt

Semaphore take

```
xSemaphoreTake(  xSemaphoreHandle xSemaphore,
        portTickType xBlockTime )
```

- *Macro* to obtain a semaphore. The semaphore must have previously been created.
- **xSemaphore** A handle to the semaphore being taken - obtained when the semaphore was created.
- **xBlockTime** The time in ticks to wait for the semaphore to become available. The macro portTICK_RATE_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore.|
- TRUE if the semaphore was obtained.
- There are a handful of variations.
  - Faster but more locking version, non-binary version, etc.

Bibliography

•Mastering the FreeRTOS™Real Time Kernel - A Hands-On Tutorial Guide, Richard Barry, Real Time Engineers Ltd. 2016

•http://www.aosabook.org/en/freertos.html

•http://embedded-tips.blogspot.com/2010/07/freertos-course-semaphoremutex.html