Scheduling and Timing Services in FreeRTOS

Mojtaba Bagherzadeh, Adrien Lapointe

Royal Military College (RMC)

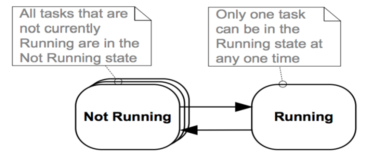
mojtaba@cs.queensu.ca,adrien.lapointe@rmc.ca

February 25, 2018

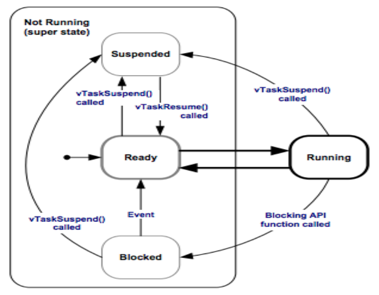
Overview

Task Status

Task High-level Status



Task Full Status



Task Status

- The task that is actually running (using processing time) is in the Running state. On a single core processor only one task can be in Running state.

Task Status

- The task that is actually running (using processing time) is in the Running state. On a single core processor only one task can be in Running state.
- Tasks that are not actually running, but are not in either the Blocked state or the Suspended state, are in the Ready state.

Task Status

- The task that is actually running (using processing time) is in the Running state. On a single core processor only one task can be in Running state.
- Tasks that are not actually running, but are not in either the Blocked state or the Suspended state, are in the Ready state.
- Tasks that are in the Ready state are available to be selected by the scheduler as the task to enter the Running state. The scheduler will always choose the highest priority Ready state task to enter the Running state.

Task Status

- The task that is actually running (using processing time) is in the Running state. On a single core processor only one task can be in Running state.
- Tasks that are not actually running, but are not in either the Blocked state or the Suspended state, are in the Ready state.
- Tasks that are in the Ready state are available to be selected by the scheduler as the task to enter the Running state. The scheduler will always choose the highest priority Ready state task to enter the Running state.
- Tasks in Blocked state wait for an event and are automatically moved back to the Ready state when a temporal or synchronization events occurs.

Task Scheduling

Scheduling Algorithm

The scheduling algorithm is the software routine that decides which Ready task to move into the Running state.

Scheduling Configuration

Scheduling algorithm can be changed using the `configUSE_PREEMPTION` and `configUSE_TIME_SLICING` configuration which are defined in `FreeRTOSConfig.h`.

Fixed-Priority Preemptive Scheduling with Time Slicing

Fixed-Priority

Fixed-Priority algorithms do not change the priority assigned to the tasks being scheduled.

Preemptive

Preemptive algorithms immediately preempt the Running state task if a higher priority task than the running task enters the Ready state.

Time Slicing

Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state. A time slice is equal to the tick period.

Relative Configuration

`configUSE_PREEMPTION=1` and `configUSE_TIME_SLICING=1`

Fixed-Priority Preemptive Scheduling with Time Slicing

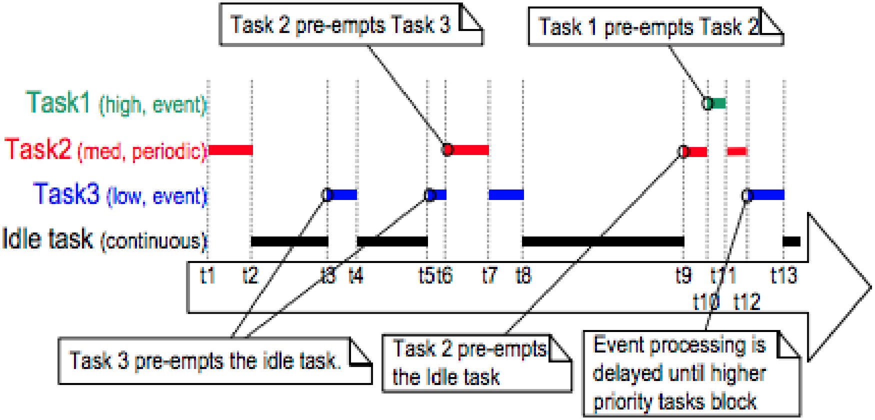


Figure: Example of Tasks with Different Priorities

Fixed-Priority Preemptive Scheduling with Time Slicing

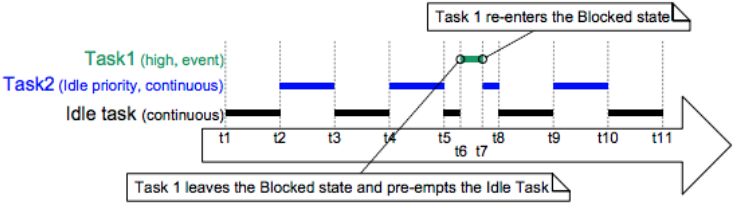


Figure: Example of Two Tasks with Same Priority

Fixed-Priority Preemptive Scheduling with Time Slicing

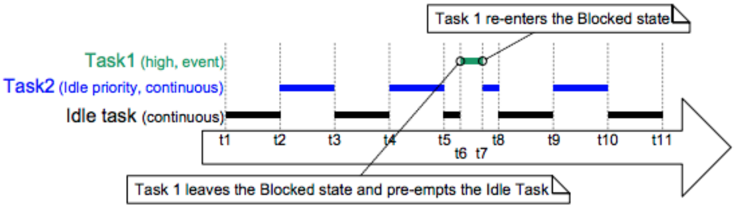


Figure: Example of Two Tasks with Same Priority

Note

When `configIDLE_SHOULD_YIELD` is set to 1, the task is selected to enter the Running state after the Idle task does not execute for an entire time slice, but instead executes for whatever remains of the time slice during which the Idle task yielded.

Prioritized Preemptive Scheduling (without Time Slicing)

Prioritized Preemptive Scheduling without time slicing maintains the same task selection and preemption algorithms as described in the previous section, but does not use time slicing to share processing time between tasks of equal priority.

Relative Configuration

`configUSE_PREEMPTION=1` and `configUSE_TIME_SLICING=0`

Prioritized Preemptive Scheduling (without Time Slicing)

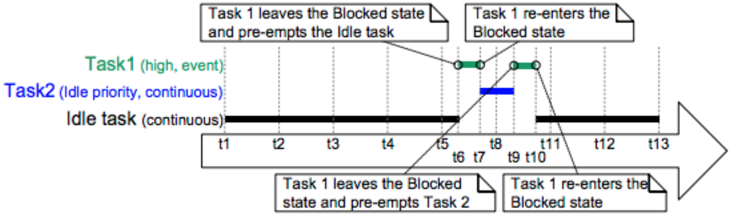


Figure: Example of Two Tasks with Same Priority

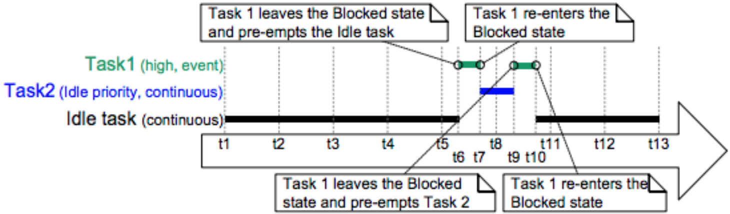


Figure: Example of Two Tasks with Same Priority

Note

- No time slicing is occurred between the two tasks with same priority. This algorithms minimize the context switching overhead.
- Turning time slicing off can also result in tasks of equal priority receiving greatly different amounts of processing time.

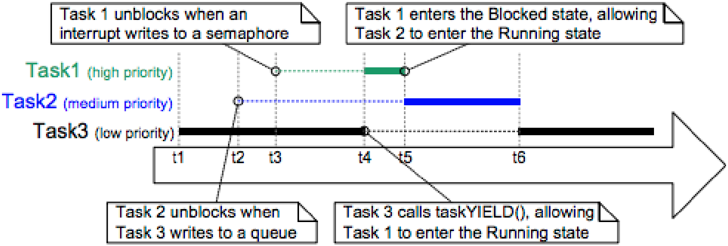
Co-operative Scheduling

When the co-operative scheduler is used, a context switch will only occur when the Running state task enters the Blocked state, or the Running state task explicitly yields (manually requests a re-schedule) by calling `taskYIELD()`. Tasks are never preempted, so time slicing cannot be used.

Relative Configuration

`configUSE_PREEMPTION=0` and `configUSE_TIME_SLICING=Any value`

Co-operative Scheduling



Software Timer

Defnition

Software timers allow to execute a function at a specific time in the future, or periodically with a fixed frequency.

Software Timer

Defnition

Software timers allow to execute a function at a specific time in the future, or periodically with a fixed frequency.

Software Timer's Callback Function

A function which is executed by the timer and implemented as a C function.

- It returns `void`, and takes a handle to a software timer as its only parameter.
- It executes from start to finish, and exits in the normal way.
- It should be kept short, and must not enter the Blocked state.

Software Timer

Defnition

Software timers allow to execute a function at a specific time in the future, or periodically with a fixed frequency.

Software Timer's Callback Function

A function which is executed by the timer and implemented as a C function.

- It returns `void`, and takes a handle to a software timer as its only parameter.
- It executes from start to finish, and exits in the normal way.
- It should be kept short, and must not enter the Blocked state.

Timer Activation

- Add source file `FreeRTOS/Source/timers.c` to your project.
- Set `configUSE_TIMERS` to 1 `in FreeRTOSConfig.h`.

Timer Service Task

- All software timer callback functions are executed in the context of the timer service task.

Timer Service Task

- All software timer callback functions are executed in the context of the timer service task.
- The timer service 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`.

Timer Service Task

- All software timer callback functions are executed in the context of the timer service task.
- The timer service 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`.
- All of timers' related commands are processed by timer service task.

Timer Service Task

- All software timer callback functions are executed in the context of the timer service task.
- The timer service 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`.
- All of timers' related commands are processed by timer service task.
- Calling blocking API by a timer callback function will block the timer service task. As the result, all timers will be affected.

Software Timer

There are two types of software timer:

- **One-shot timers:** Once started, a one-shot timer will execute its callback function once only. A one-shot timer can be restarted manually, but will not restart itself.

Software Timer

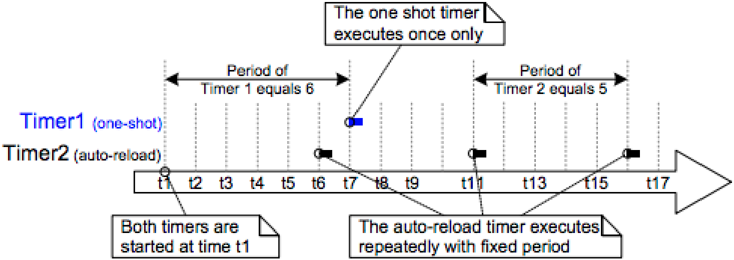
There are two types of software timer:

- **One-shot timers:** Once started, a one-shot timer will execute its callback function once only. A one-shot timer can be restarted manually, but will not restart itself.
- **Auto-reload timers:** Once started, an auto-reload timer will re-start itself each time it expires, resulting in the periodic execution of its callback function.

Software Timer

There are two types of software timer:

- **One-shot timers:** Once started, a one-shot timer will execute its callback function once only. A one-shot timer can be restarted manually, but will not restart itself.
- **Auto-reload timers:** Once started, an auto-reload timer will re-start itself each time it expires, resulting in the periodic execution of its callback function.



Software Timer

Software timer can be in one of the following two states:

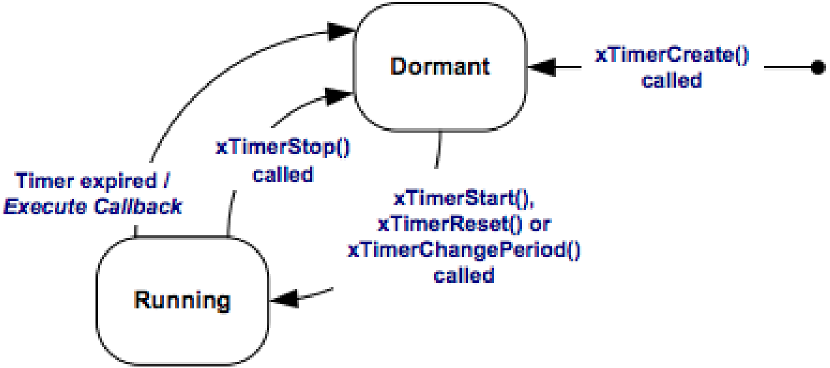
- **Dormant:** A dormant software timer exists, and can be referenced by its handle, but is not running, so its callback functions will not be executed.

Software Timer

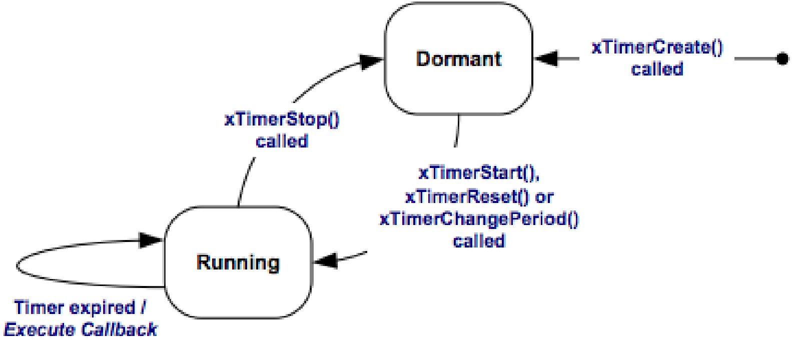
Software timer can be in one of the following two states:

- **Dormant:** A dormant software timer exists, and can be referenced by its handle, but is not running, so its callback functions will not be executed.
- **Running:** A Running software timer will execute its callback function after a time equal to its period has elapsed since the software timer entered the Running state, or since the time that software timer was reset.

One-shot Timer States



Auto-reload Timer States



Create Timer

```
TimerHandle_t xTimerCreate(const char * const pcTimerName,
    TickType_t xTimerPeriodInTicks, UBaseType_t uxAutoReload
    , void * pvTimerID, TimerCallbackFunction_t
    pxCallbackFunction);
```

Create Timer

```
TimerHandle_t xTimerCreate(const char * const pcTimerName,
    TickType_t xTimerPeriodInTicks, UBaseType_t uxAutoReload
    , void * pvTimerID, TimerCallbackFunction_t
    pxCallbackFunction);
```

Argument	Description
pcTimerName	name of the timer
xTimerPeriodInTicks	the timer's period specified in ticks.
uxAutoReload	set to pdTRUE to create an auto-reload timer.
pvTimerID	The timer ID.
pxCallbackFunction	the software timer callback function which is a simply C functions with the mentioned prototype.

Create Timer

```
TimerHandle_t xTimerCreate(const char * const pcTimerName,
    TickType_t xTimerPeriodInTicks, UBaseType_t uxAutoReload
    , void * pvTimerID, TimerCallbackFunction_t
    pxCallbackFunction);
```

Argument	Description
pcTimerName	name of the timer
xTimerPeriodInTicks	the timer's period specified in ticks.
uxAutoReload	set to pdTRUE to create an auto-reload timer.
pvTimerID	The timer ID.
pxCallbackFunction	the software timer callback function which is a simply C functions with the mentioned prototype.

Return values of the function:

- `NULL`: the time has not been created successfully.
- `non-NULL`: the time has been created and the time handle is returned .

Star a Timer

```
TimerHandle_t xTimerStart( TimerHandle_t xTimer, TickType_t
    xTicksToWait )
```

Star a Timer

```
TimerHandle_t xTimerStart( TimerHandle_t xTimer, TickType_t
    xTicksToWait )
```

Argument	Description
xTimer	the handle of the timer that will be started. The handle is returned from the call to xTimerCreate() used to create the software timer.
xTicksToWait	specifies the maximum amount of time the calling task should remain in the Blocked state to wait for space to become available on the timer command queue.

Star a Timer

```
TimerHandle_t xTimerStart( TimerHandle_t xTimer, TickType_t
    xTicksToWait )
```

Argument	Description
xTimer	the handle of the timer that will be started. The handle is returned from the call to xTimerCreate() used to create the software timer.
xTicksToWait	specifies the maximum amount of time the calling task should remain in the Blocked state to wait for space to become available on the timer command queue.

Return values of the function:

- `pdPASS`: successful execution.
- `pdFALSE`: un-successful execution .

Example

```
#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;
xOneShotTimer =xTimerCreate("OneShot",
    mainONE_SHOT_TIMER_PERIOD,pdFALS,0,
    prvOneShotTimerCallback );
xAutoReloadTimer = xTimerCreate("AutoReload",
    mainAUTO_RELOAD_TIMER_PERIOD,pdTRUE,0,
    prvAutoReloadTimerCallback );
if((xOneShotTimer != NULL )&&( xAutoReloadTimer != NULL )){
 xTimer1Started = xTimerStart( xOneShotTimer, 0 );
 xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );
 // rest of the code
}
```

Useful APIs

Get and Set TimerID

```
void vTimerSetTimerID( const TimerHandle_t xTimer, void *pvNewID)
void *pvTimerGetTimerID( TimerHandle_t xTimer )
```

Change the Period of a Timer

```
xTimerChangePeriod(TimerHandle_t xTimer,TickType_t
xNewTimerPeriodInTicks,TickType_t xTicksToWait )
```

Reseting a Timer

```
xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait )
```

References

Richard Barry. Mastering the FreeRTOS Real Time Kernel. FreeRTOS.org, 2016

Question?