HCMC University Of Technology, VNU-HCM
Faculty of Computer Science & Engineering

– Embedded System Lab 05 –

FreeRTOS Software Timer

Pham Hoang Anh & **Huynh Hoang Kha**

Goal In this lab, students are expected to be able to understand and use FreeRTOS Software Timer.

Content

- Introduction of FreeRTOS Software Timer
- Software Timer Callback Functions
- Attributes and States of a Software Timer
- The Context of a Software Timer
- Configure a Software Timer
- Exercises

Grading policy

- 40% in-class performance
- 60% report submission

1 About the FreeRTOS software timer

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. User can use it by including software timer functionality as follows:

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

2 Software Timer Callback Functions

Software timer callback functions are implemented as C functions. The only thing special about them is their prototype, which must return void, and take a handle to a software timer as its only parameter. The callback function prototype is demonstrated by Code 1.

```
1 void ATimerCallback(TimerHandle_t xTimer);
```

Code 1: The software timer callback function prototype

Software timer callback functions execute from start to finish, and exit in the normal way. They should be kept short, and must not enter the Blocked state.

Note: As will be seen, software timer callback functions execute in the context of a task that is created automatically when the FreeRTOS scheduler is started. Therefore, it is essential that software timer callback functions never call FreeRTOS API functions that will result in the calling task entering the Blocked state. It is ok to call functions such as `xQueueReceive()`, but only if the function's `xTicksToWait` parameter (which specifies the function's block time) is set to 0. It is not ok to call functions such as `vTaskDelay()`, as calling `vTaskDelay()` will always place the calling task into the Blocked state.

3 Attributes and States of a Software Timer

Period of a Software Timer

A software timer's "period" is the time between the software timer being started, and the software timer's callback function executing.

One-shot and Auto-reload Timers

There are two types of software timer:

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

2. **Auto-reload timers**: Once started, an auto-reload timer will re-start itself each time it expires, resulting in periodic execution of its callback function.

Figure 1 shows the difference in behavior between a one-shot timer and an auto-reload timer. The dashed vertical lines mark the times at which a tick interrupt occurs.

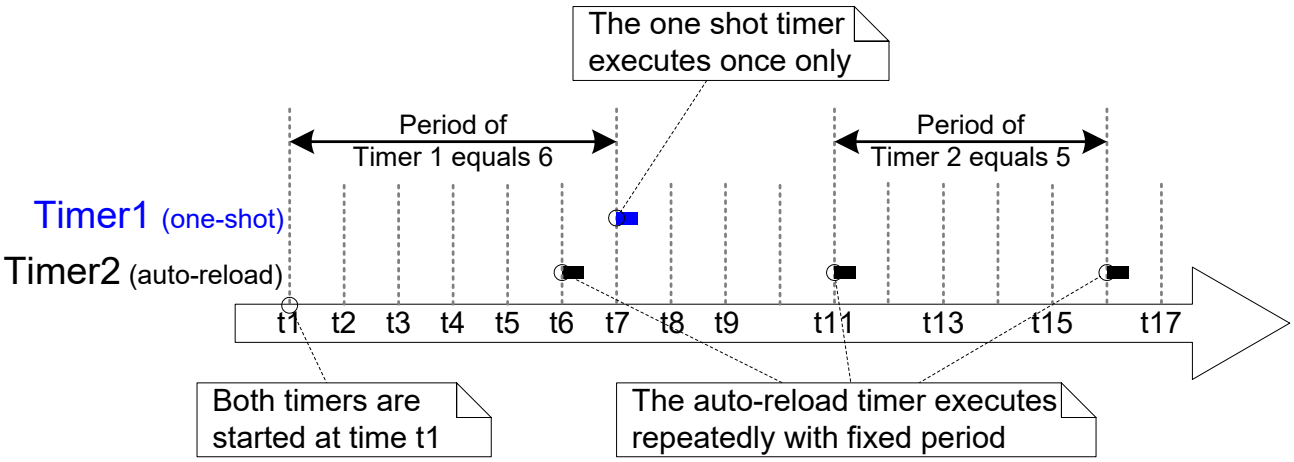

Figure 1: The difference in behavior between one-shot and auto-reload software timers

Timer 1 is a one-shot timer that has a period of 6 ticks. It is started at time t1, so its callback function executes 6 ticks later, at time t7. As timer 1 is a one-shot timer, its callback function does not execute again.

Timer 2 is an auto-reload timer that has a period of 5 ticks. It is started at time t1, so its callback function executes every 5 ticks after time t1. In Figure 1 this is at times t6, t11 and t16.

Software Timer States that 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 execute.

- 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 software timer was last reset.

Figure 2 and Figure 3 show the possible transitions between the Dormant and Running states for an auto-reload timer and a one-shot timer respectively. The key difference between the two diagrams is the state entered after the timer has expired; the auto-reload timer executes its callback function then re-enters the Running state, the one-shot timer executes its callback function then enters the Dormant state.

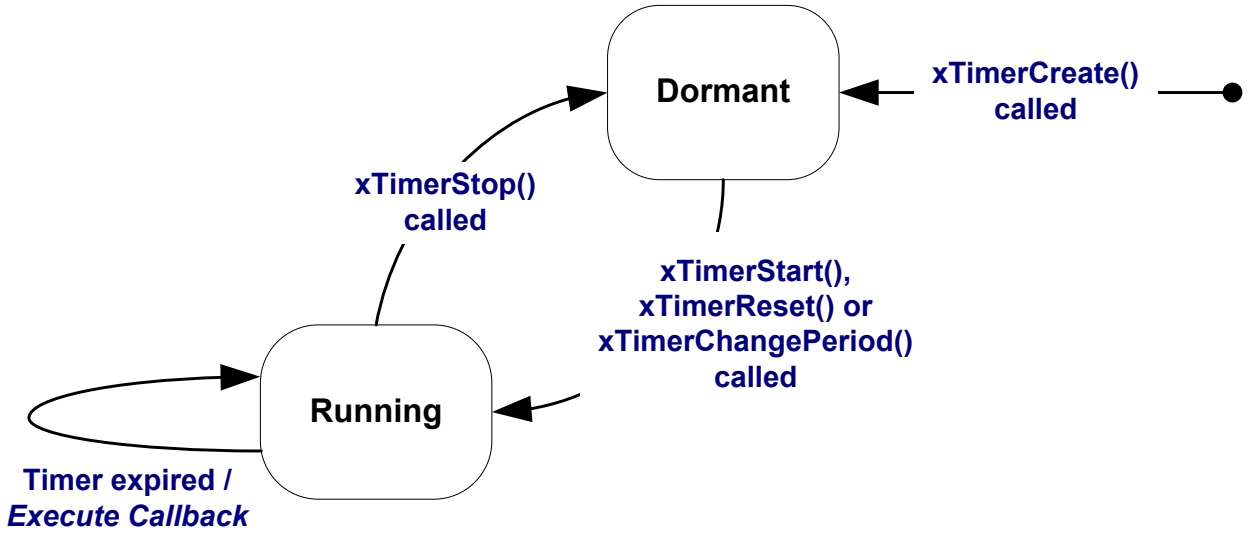

Figure 2: Auto-reload software timer states and transitions

4 The Context of a Software Timer

The RTOS Daemon (Timer Service) Task

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

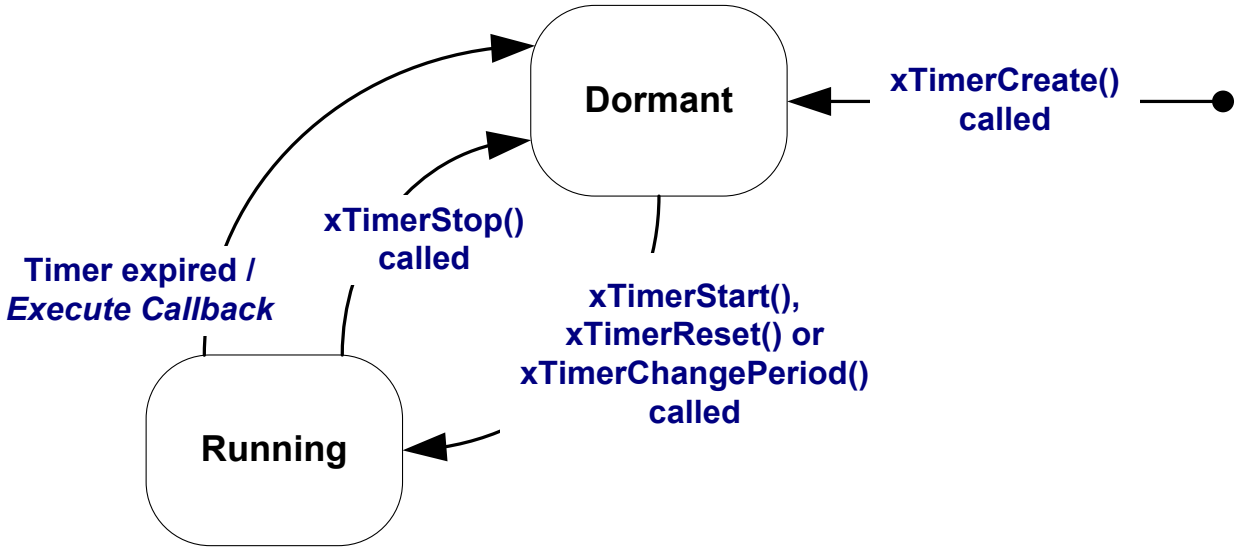

Figure 3: One-shot software timer states and transitions

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 4. Examples of commands include "start a timer", "stop a timer" and "reset a timer".

Daemon Task Scheduling

The daemon task is scheduled like any other FreeRTOS task; it will only process commands, or execute timer callback functions, when it is the highest priority task that is able to run.

Figure 5 and Figure 6 demonstrate how the `configTIMER_TASK_PRIORITY` setting affects the execution pattern.

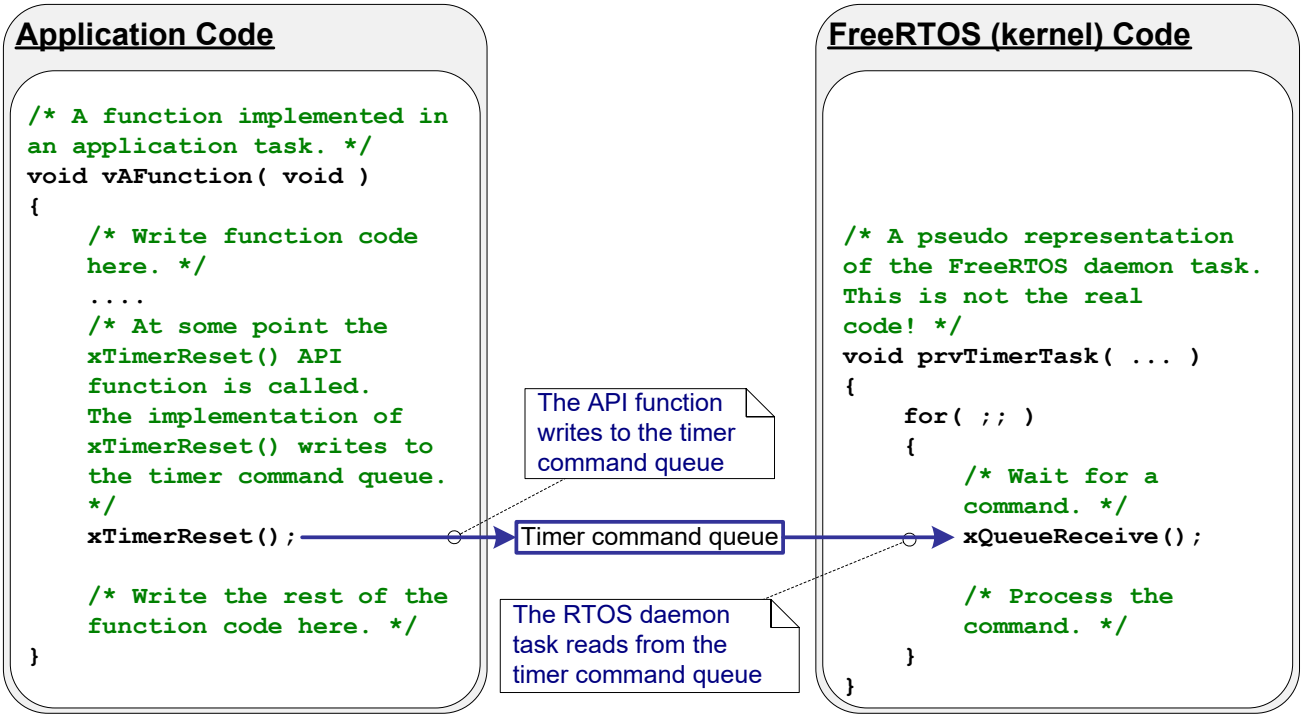

Figure 4: The timer command queue being used by a software timer API function to communicate with the RTOS daemon task

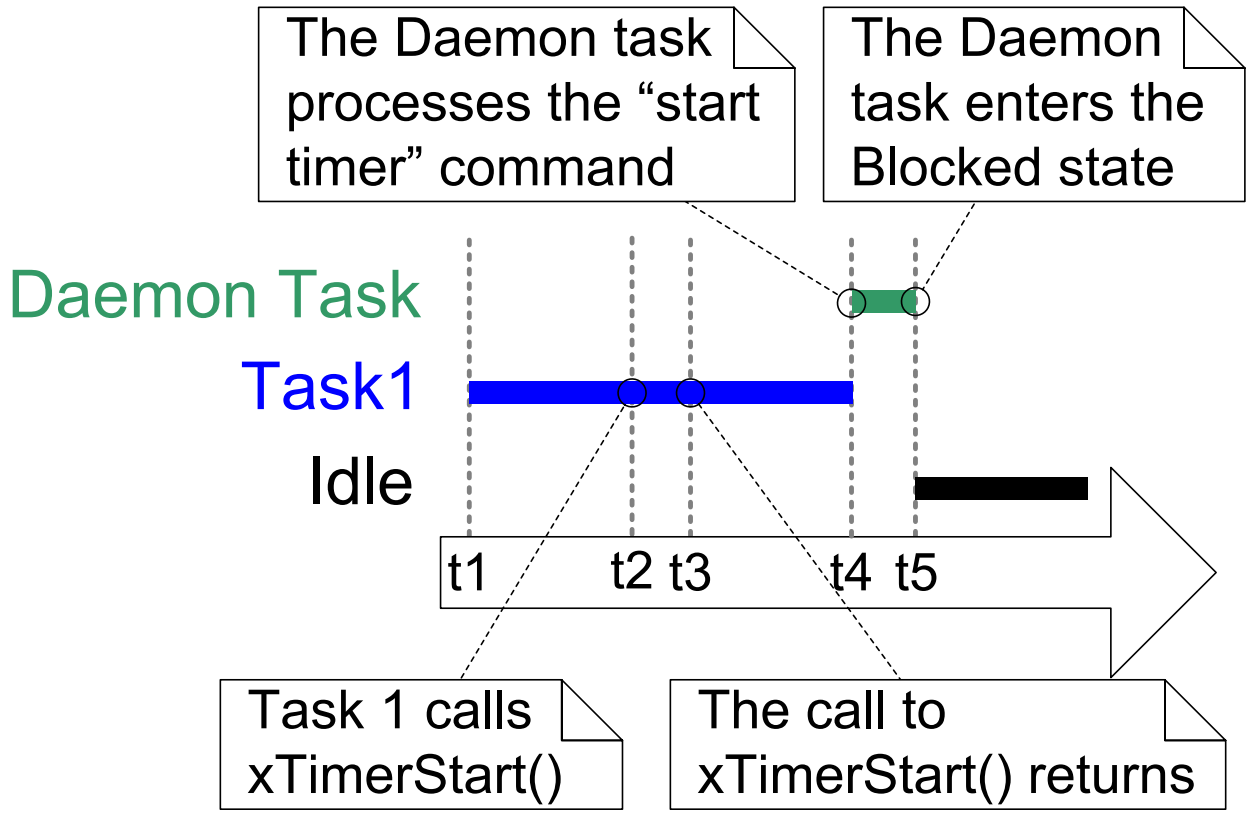

Figure 5: The execution pattern when the priority of a task calling `xTimerStart()` is above the priority of the daemon task

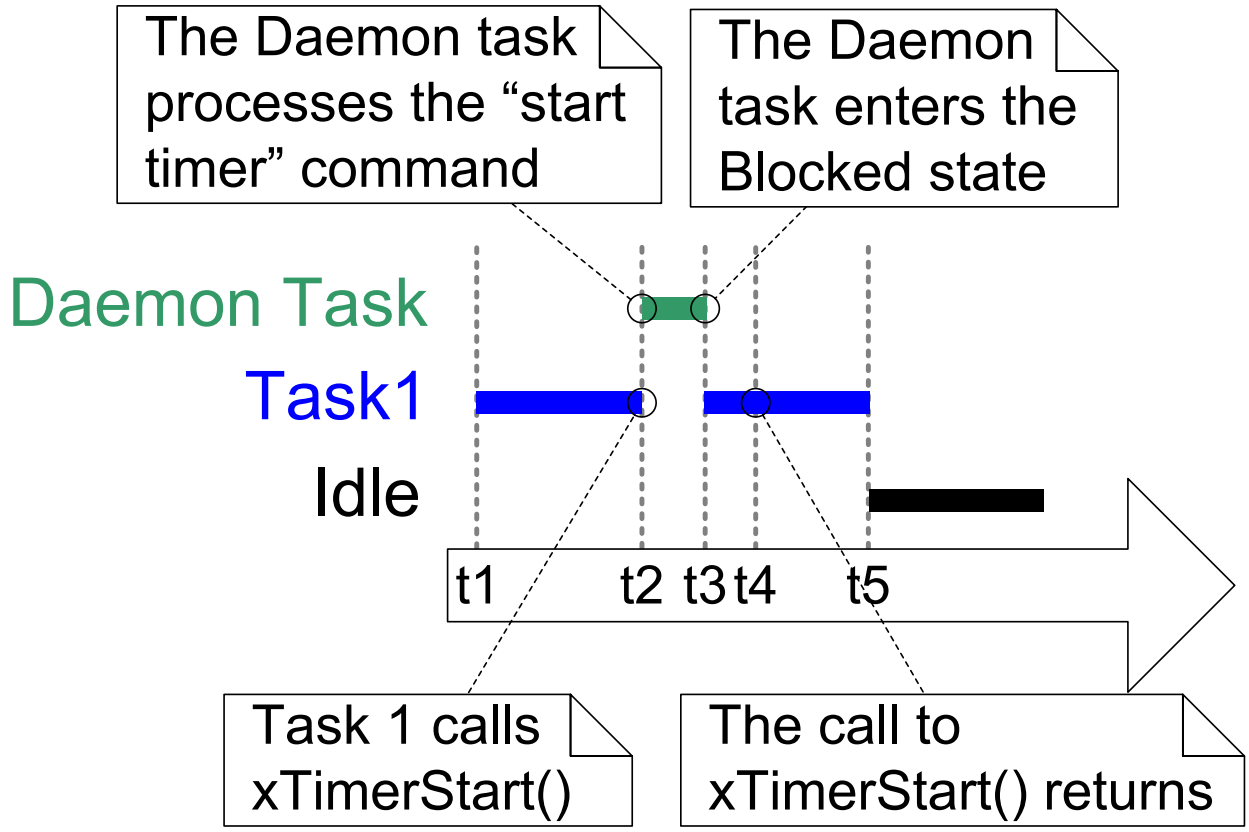

Figure 6: The execution pattern when the priority of a task calling `xTimerStart()` is below the priority of the daemon task

5 Creating and Starting a Software Timer

The `xTimerCreate()` API Function

A software timer must be explicitly created before it can be used. Software timers are referenced by variables of type `TimerHandle_t`. `xTimerCreate()` is used to create a software timer and returns a `TimerHandle_t` to reference the software timer it creates. Software timers are created in the Dormant state.

Software timers can be created before the scheduler is running, or from a task after the scheduler has been started.

```
1 TimerHandle_t xTimerCreate(const char * const pcTimerName,
2                            TickType_t xTimerPeriodInTicks,
3                            UBaseType_t uxAutoReload,
4                            void * pvTimerID,
5                            TimerCallbackFunction_t pxCallbackFunction);
```

Code 2: The `xTimerCreate()` API function prototype

In code 2, note that:

- Set `uxAutoReload` to `pdTRUE` to create an auto-reload timer. Set `uxAutoReload` to `pdFALSE` to create a one-shot timer.
- Each software timer has an ID value. The ID is a void pointer, and can be used by the application writer for any purpose. The ID is particularly useful when the same callback function is used by more than one software timer, as it can be used to provide timer specific storage. `pvTimerID` sets an initial value for the ID of the task being created.
- If `NULL` is returned, then the software timer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the necessary data structure. A non-`NULL` value being returned indicates that the software timer has been created successfully. The returned value is the handle of the created timer.

The `xTimerStart()` API Function

`xTimerStart()` is used to start a software timer that is in the Dormant state, or reset (re-start) a software timer that is in the Running state. `xTimerStop()` is used to stop a software timer that is in the Running state. Stopping a software timer is the same as transitioning the timer into the Dormant state.

`xTimerStart()` can be called before the scheduler is started, but when this is done, the software timer will not actually start until the time at which the scheduler starts.

Note: Never call `xTimerStart()` from an interrupt service routine. The interrupt-safe version `xTimerStartFromISR()` should be used in its place.

```
1 BaseType_t xTimerStart(TimerHandle_t xTimer, TickType_t xTicksToWait);
```

Code 3: The `xTimerStart()` API function prototype

In Code 3, `xTimerStart()` uses the timer command queue to send the 'start a timer' command to the daemon task. `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, should the queue already be full.

If `xTimerStart()` is called before the scheduler has been started then the value of `xTicksToWait` is ignored, and `xTimerStart()` behaves as if `xTicksToWait` had been set to zero.

6 The Timer ID

Each software timer has an ID, which is a tag value that can be used by the application writer for any purpose. The ID is stored in a void pointer (void *), so can store an integer value directly, point to any other object, or be used as a function pointer.

An initial value is assigned to the ID when the software timer is created - after which the ID can be updated using the `vTimerSetTimerID()` API function, and queried using the `pvTimerGetTimerID()` API function.

Unlike other software timer API functions, vTimerSetTimerID() and pvTimerGetTimerID() access the software timer directly—they do not send a command to the timer command queue.

The vTimerSetTimerID() API Function

```
1 void vTimerSetTimerID(const TimerHandle_t xTimer, void *pvNewID);
```

Code 4: The `vTimerSetTimerID()` API function prototype

The vTimerGetTimerID() API Function

```
1 void *pvTimerGetTimerID(TimerHandle_t xTimer);
```

Code 5: The `pvTimerGetTimerID()` API function prototype

7 Changing the Period of a Timer

Every official FreeRTOS port is provided with one or more example projects. Most example projects are self-checking, and an LED is used to give visual feedback of the project's status; if the self-checks have always passed then the LED is toggled slowly, if a self-check has ever failed then the LED is toggled quickly.

Some example projects perform the self-checks in a task, and use the `vTaskDelay()` function to control the rate at which the LED toggles. Other example projects perform the self-checks in a software timer callback function, and use the timer's period to control the rate at which the LED toggles.

The `xTimerChangePeriod()` API Function

The period of a software timer is changed using the `xTimerChangePeriod()` function. 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 is relative to when `xTimerChangePeriod()` was called, not relative to when the timer was originally started.

If `xTimerChangePeriod()` is used to change the period of a timer that is in the Dormant state (a timer that is not running), then the timer will calculate an expiry

time, and transition to the Running state (the timer will start running).

Note: Never call `xTimerChangePeriod()` from an interrupt service routine. The interrupt-safe version `xTimerChangePeriodFromISR()` should be used in its place.

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

Code 6: The `xTimerChangePeriod()` API function prototype

8 Resetting a Software Timer

Resetting a software timer means to re-start the timer; the timer's expiry time is recalculated to be relative to when the timer was reset, rather than when the timer was originally started. This is demonstrated by Figure 7, which shows a timer that has a period of 6 being started, then reset twice, before eventually expiring and executing its callback function.

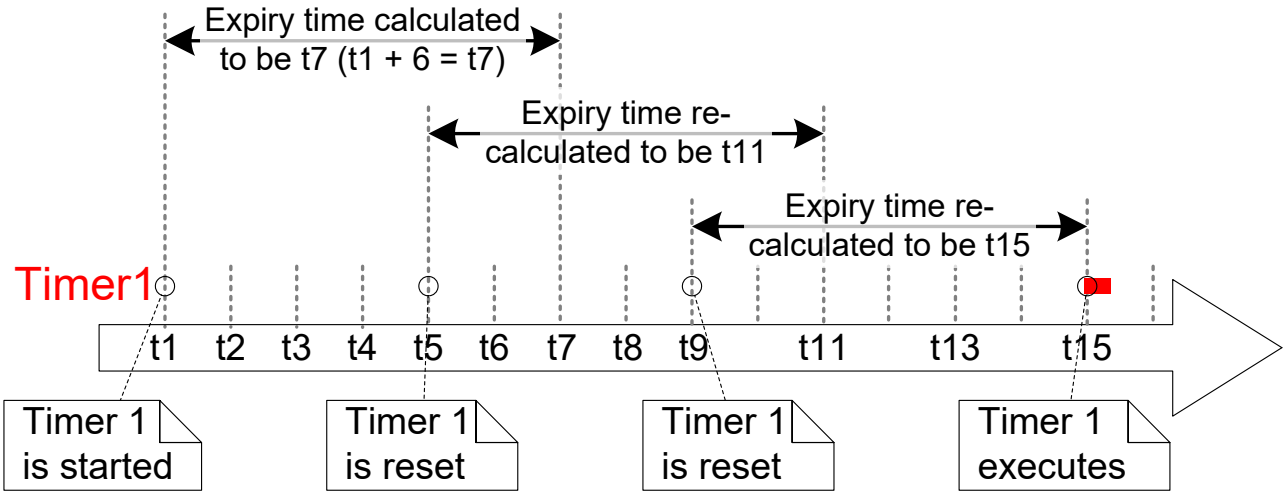

Figure 7: Starting and resetting a software timer that has a period of 6 ticks

The `xTimerReset()` API Function

A timer is reset using the `xTimerReset()` API function and can also be used to start a timer that is in the Dormant state.

Note: Never call `xTimerReset()` from an interrupt service routine. The interrupt-safe version `xTimerResetFromISR()` should be used in its place.

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

Code 7: The `xTimerReset()` API function prototype

9 Exercises

Students create 2 software timers sharing only one timer callback function, in which

- The first timer is used to print "ahihi" every 2 seconds and will stop after 10 times printing.
- The second timer is used to print "ihaha" every 3 seconds and will stop after 5 times printing.