SOFTWARE TIMER MANAGEMENT

INDEX:

1. Hardware timers

Hardware circuit generate the external interrupt regularly depending on the configuration at some frequency

Example: STM32 boards internal timers and crystal oscillators, watch dog timer

1.1 Watch dog timer

In an operating system context, including real-time operating systems (RTOS) like FreeRTOS, the watchdog timer plays a crucial role in ensuring system stability and reliability. Here's how it typically works:

1. **Monitoring System Health**: The watchdog timer is essentially a countdown timer that needs to be periodically reset by the system software. If the software fails to reset the timer within a predefined interval, it triggers a system reset or takes another predefined action.
2. **Preventing System Hangs**: The primary purpose of the watchdog timer is to prevent the system from becoming unresponsive or "hanging" due to software bugs, race conditions, or other unforeseen issues. By regularly resetting the watchdog timer, the system demonstrates that it is still functioning correctly.
3. **Automatic Recovery**: If the system stops resetting the watchdog timer, indicating a fault or malfunction, the watchdog timer initiates corrective action. This action may include resetting the system, performing a graceful shutdown, or executing a specific recovery routine, depending on the system's design.
4. **Fault Tolerance**: Watchdog timers contribute to fault tolerance by providing a mechanism for automatic recovery from software failures. This is particularly important in safety-critical systems, where system reliability is paramount.
5. **System Health Monitoring**: Beyond just preventing system hangs, watchdog timers can also be used for monitoring system health. For example, they may trigger an alert if a critical system process stops running or if system resources are depleted.

In the context of an operating system like FreeRTOS, the watchdog timer is often integrated into the kernel and exposed through APIs. Application developers can configure the watchdog timer's timeout period and reset it periodically within their tasks to demonstrate that the system is still operational.

Overall, the watchdog timer serves as a safety net, ensuring that even in the face of unexpected software failures, the system can maintain a basic level of functionality and recover automatically whenever possible.

2. Software timers

- implemented using C programming
- it cannot identify the system hangs since the code execution get stuck
- It is used to schedule execution of function at a set time in future/periodically with fixed frequency
- Function execution by software timer is called software timer 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
- 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.

2.1 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

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

- 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. (example do not use vTaskDelay())
- 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 of Software Timer

3.1 Period of a Software Timer

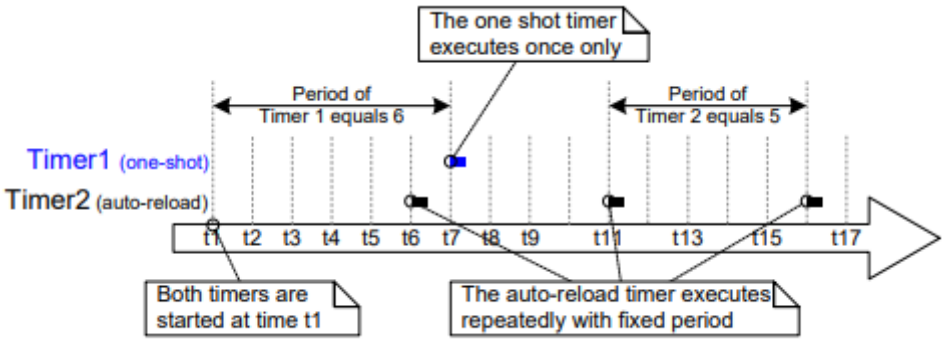
A software timer's 'period' is the time between the software timer being started, and the software timer's callback function executing.

3.2 One-shot and Auto-reload Timers

There are two types of software timer:

1. 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.

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.



- Referring to above figure
    - 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 above figure this is at times t6, t11 and t1

3.3 Software Timer States

A 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 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

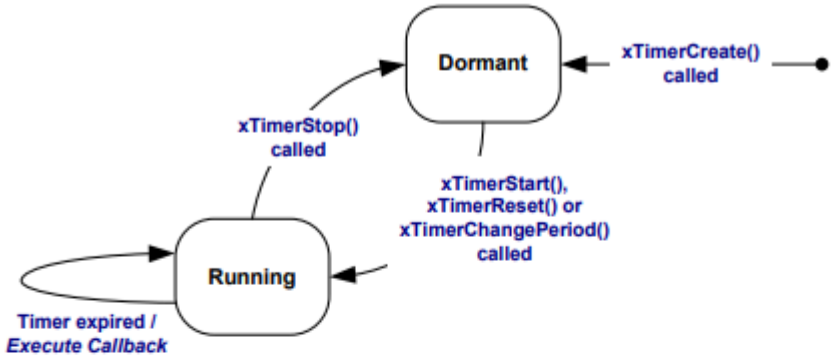


Figure 39 Auto-reload software timer states and transitions

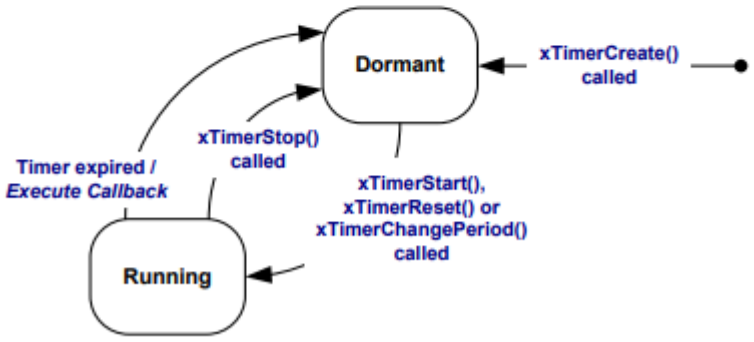


Figure 40 One-shot software timer states and transitions

Figure 39 and Figure 40 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.

The **xTimerDelete()** API function deletes a timer. A timer can be deleted at any time.

4. The Context of a Software Timer

4.1 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 task used to be called the 'timer service task', because originally it was only used to execute software timer callback functions. Now the same task is used for other purposes too, so it is known by the more generic name of the 'RTOS daemon task'
- 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.

4.2 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'.
- 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.

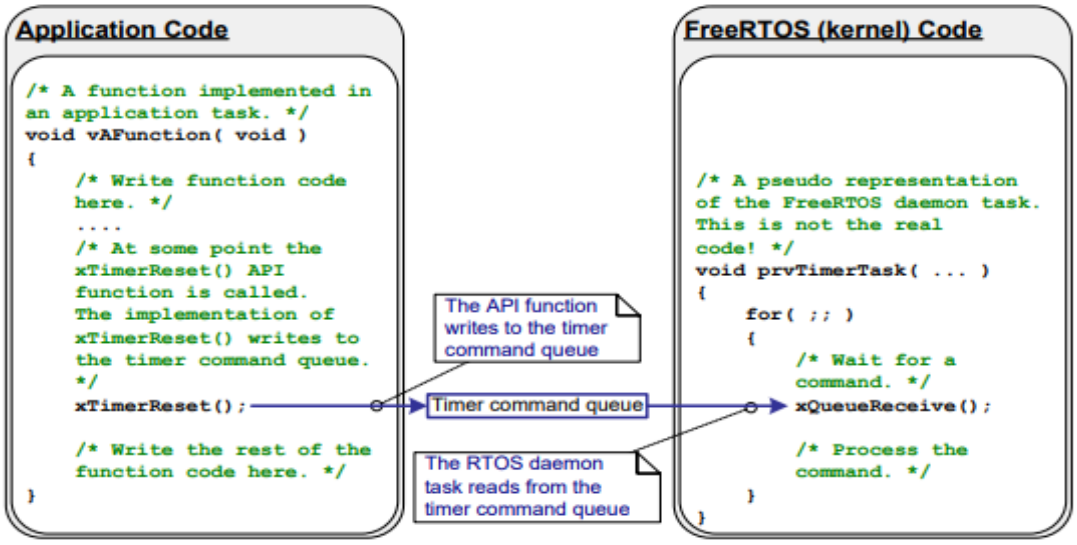


Figure 41 The timer command queue being used by a software timer API function to communicate with the RTOS daemon task

4.3 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.

Condition-1

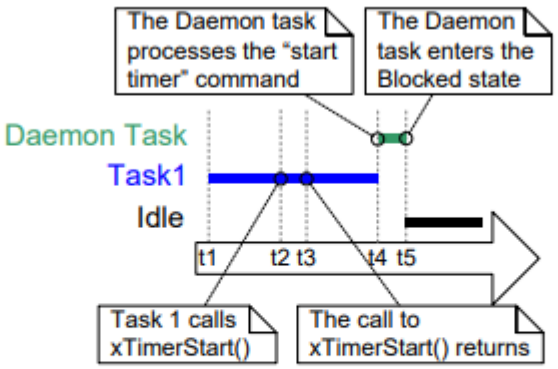


Figure 42 The execution pattern when the priority of a task calling xTimerStart() is above the priority of the daemon task

- Referring to Figure 42, in which the priority of Task 1 is higher than the priority of the daemon task, and the priority of the daemon task is higher than the priority of the Idle task:

  1. At time t1

     Task 1 is in the Running state, and the daemon task is in the Blocked state. The daemon task will leave the Blocked state if a command is sent to the timer command queue, in which case it will process the command, or if a software timer expires, in which case it will execute the software timer's callback function.

  2. At time t2

     Task 1 calls xTimerStart(). xTimerStart() sends a command to the timer command queue, causing the daemon task to leave the Blocked state. The priority of Task 1 is higher than the priority of the daemon task, so the daemon task does not pre-empt Task 1. Task 1 is still in the Running state, and the daemon task has left the Blocked state and entered the Ready state.

  3. At time t3

     Task 1 completes executing the xTimerStart() API function. Task 1 executed xTimerStart() from the start of the function to the end of the function, without leaving the Running state.

  4. At time t4

     Task 1 calls an API function that results in it entering the Blocked state. The daemon task is now the highest priority task in the Ready state, so the scheduler selects the daemon task as the task to enter the Running state. The daemon task then starts to process the command sent to the timer command queue by Task 1.

     Note: The time at which the software timer being started will expire is calculated from the time the 'start a timer' command was sent to the timer command queue—it is not calculated from the time the daemon task received the 'start a timer' command from the timer command queue.

  5. At time t5

     The daemon task has completed processing the command sent to it by Task 1, and attempts to receive more data from the timer command queue. The timer command queue is empty, so the daemon task re-enters the Blocked state. The daemon task will leave the Blocked state again if a command is sent to the timer command queue, or if a software timer expires.

The Idle task is now the highest priority task in the Ready state, so the scheduler selects the Idle task as the task to enter the Running state

Condition-2

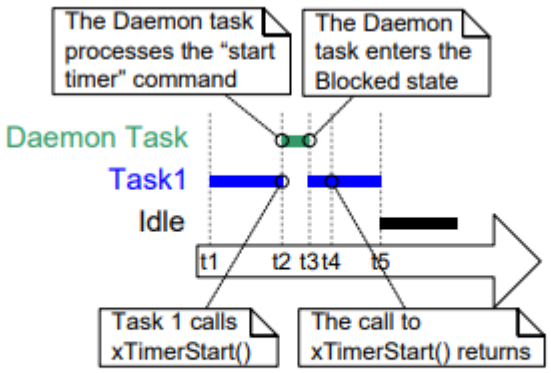


Figure 43 The execution pattern when the priority of a task calling xTimerStart() is below the priority of the daemon task

Referring to Figure 43, in which the priority of the daemon task is higher than the priority of Task 1, and the priority of the Task 1 is higher than the priority of the Idle task:

1. At time t1

    As before, Task 1 is in the Running state, and the daemon task is in the Blocked state.

2. At time t2

    Task 1 calls xTimerStart().

    xTimerStart() sends a command to the timer command queue, causing the daemon task to leave the Blocked state. The priority of the daemon task is higher than the priority of Task 1, so the scheduler selects the daemon task as the task to enter the Running state.

    Task 1 was pre-empted by the daemon task before it had completed executing the xTimerStart() function, and is now in the Ready state.

    The daemon task starts to process the command sent to the timer command queue by Task 1.

3. At time t3

The daemon task has completed processing the command sent to it by Task 1, and attempts to receive more data from the timer command queue. The timer command queue is empty, so the daemon task re-enters the Blocked state.

Task 1 is now the highest priority task in the Ready state, so the scheduler selects Task 1 as the task to enter the Running state.

4. At time t4

Task 1 was pre-empted by the daemon task before it had completed executing the xTimerStart() function, and only exits (returns from) xTimerStart() after it has re-entered the Running state.

5. At time t5

Task 1 calls an API function that results in it entering the Blocked state. The Idle task is now the highest priority task in the Ready state, so the scheduler selects the Idle task as the task to enter the Running state

5. Software Timer APIs

5.1 Creating and starting a software timer

5.1.1 xTimerCreate()

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.

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

PARAMETER NAME/RETURNED VALUE	DESCRIPTION
pcTimerName	A descriptive name for the timer. This is not used by FreeRTOS in any way. It is included purely as a debugging aid. Identifying a timer by a human readable name is much simpler than attempting to identify it by its handle.
xTimerPeriodInTicks	The timer's period specified in ticks. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in ticks.
uxAutoReload	Set uxAutoReload to pdTRUE to create an auto-reload timer. Set uxAutoReload to pdFALSE to create a one-shot timer
pvTimerID	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.
pxCallbackFunction	The pxCallbackFunction parameter is a pointer to the function (in effect, just the function name) to use as the callback function for the software timer being created
Returned value	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.

5.1.2 xTimerStart()

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

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

- **xTimer** □ The handle of the software timer being started or reset. The handle will have been returned from the call to xTimerCreate() used to create the software timer
- **xTicksToWait** □ 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.
    - xTimerStart() will return immediately if xTicksToWait is zero and the timer command 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 macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks.
    - If INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h then setting xTicksToWait to portMAX_DELAY will result in the calling task remaining in the Blocked state indefinitely (without a timeout) to wait for space to become available in the timer command queue.
- 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.
- **Returned value** □ **pdPASS**(it will be returned only if the 'start a timer' command was successfully sent to the timer command queue.) or **pdFALSE** (it will be returned if the 'start a timer' command could not be written to the timer command queue because the queue was already full.)

5.2 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.

5.2.1 vTimerSetTimerID()

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

Parameter Name/ Returned Value	Description
xTimer	The handle of the software timer being updated with a new ID value. The handle will have been returned from the call to xTimerCreate() used to create the software timer.
pvNewID	The value to which the software timer's ID will be set

5.2.2 pvTimerGetTimerID()

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

Returned value : The ID of the software timer being queried.

prvTimerCallback()

- it will execute when either timer expires. The implementation of prvTimerCallback() uses the function's parameter to determine if it was called because the one-shot timer expired, or because the auto-reload timer expired.
- prvTimerCallback() also demonstrates how to use the software timer ID as timer specific storage; each software timer keeps a count of the number of times it has expired in its own ID

5.3 Changing the Period of a Timer

5.3.1 xTimerChangePeriod()

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

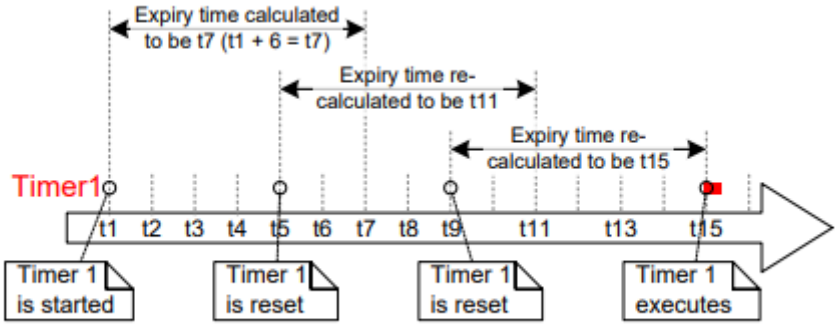
```
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer, TickType_t
xNewTimerPeriodInTicks, TickType_t xTicksToWait );
```

Parameter Name/ Returned Value	Description
xTimer	The handle of the software timer being updated with a new ID value. The handle will have been returned from the call to xTimerCreate() used to create the software timer.
xTimerPeriodInTicks	The new period for the software timer, specified in ticks. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in tick
xTicksToWait	Same as mentioned above
Returned value	There are two possible return values: **1. pdPASS** pdPASS will be returned only if data was successfully sent to the timer command queue. **2. pdFALSE** pdFALSE will be returned if the ‘change period’ command could not be written to the timer

	command queue because the queue was already full.

5.4 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



Referring to Figure

· Timer 1 is started at time t1. It has a period of 6, so the time at which it will execute its callback function is originally calculated to be t7, which is 6 ticks after it was started.

· Timer 1 is reset before time t7 is reached, so before it had expired and executed its callback function. Timer 1 is reset at time t5, so the time at which it will execute its callback function is re-calculated to be t11, which is 6 ticks after it was reset.

· Timer 1 is reset again before time t11, so again before it had expired and executed its callback function. Timer 1 is reset at time t9, so the time at which it will execute its callback function is re-calculated to be t15, which is 6 ticks after it was last reset.

· Timer 1 is not reset again, so it expires at time t15, and its callback function is executed accordingly.

5.4.1 xTimerReset()

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

- The parameters description is same as above

6. Example codes

6.1 Creating one shot and auto reload timers

```
#include "FreeRTOS.h"
#include "task.h"
#include "timers.h"
#include <stdarg.h>
#include <stdio.h>

void vApplicationIdleHook(void)
{
}

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

/* Define callback functions */
static void prvOneShotTimerCallback(TimerHandle_t xTimer);
static void prvAutoReloadTimerCallback(TimerHandle_t xTimer);

/* Define timer periods */
#define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS(3333) /* 3.333
seconds */
```

```
#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS(500) /* Half a
second */


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


    /* Create one-shot timer */
    xOneShotTimer = xTimerCreate("OneShot",
mainONE_SHOT_TIMER_PERIOD, pdFALSE, 0, prvOneShotTimerCallback);


    /* Create auto-reload timer */
    xAutoReloadTimer = xTimerCreate("AutoReload",
mainAUTO_RELOAD_TIMER_PERIOD, pdTRUE, 0,
prvAutoReloadTimerCallback);


    /* Check if timers were created successfully */
    if ((xOneShotTimer != NULL) && (xAutoReloadTimer != NULL)) {
        /* Start the timers */
        xTimer1Started = xTimerStart(xOneShotTimer, 0);
        xTimer2Started = xTimerStart(xAutoReloadTimer, 0);


        /* Start the scheduler if timers started successfully */
        if ((xTimer1Started == pdPASS) && (xTimer2Started ==
pdPASS)) {
            vTaskStartScheduler(); /* This call will never return
*/
        }
    }


    /* This line should not be reached */
    for (;;) {
        /* Do nothing */
    }
}
```

```
/* One-shot timer callback function */
static void prvOneShotTimerCallback(TimerHandle_t xTimer) {
    TickType_t xTimeNow;
    xTimeNow = xTaskGetTickCount();
    vPrintString("One-shot timer callback executing %d\n",
xTimeNow);
}

/* Auto-reload timer callback function */
static void prvAutoReloadTimerCallback(TimerHandle_t xTimer) {
    TickType_t xTimeNow;
    xTimeNow = xTaskGetTickCount();
    vPrintString("Auto-reload timer callback executing%d\n",
xTimeNow);
}
```

Output:

Auto-reload timer callback executing500

Auto-reload timer callback executing1000

Auto-reload timer callback executing1500

Auto-reload timer callback executing2000

Auto-reload timer callback executing2500

Auto-reload timer callback executing3000

One-shot timer callback executing 3333

Auto-reload timer callback executing3500

Auto-reload timer callback executing4000

Auto-reload timer callback executing4500

Auto-reload timer callback executing5000

.

.

6.2 use of prvtTimerCallback

```
#include "FreeRTOS.h"
#include "task.h"
#include "timers.h"
#include <stdarg.h>
#include <stdio.h>

void vApplicationIdleHook(void)
{
}

TickType_t xTimeNow;
uint32_t ulExecutionCount;

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

/* Define callback functions */
static void prvOneShotTimerCallback(TimerHandle_t xTimer);
static void prvAutoReloadTimerCallback(TimerHandle_t xTimer);

/* Define timer periods */
#define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS(3333) /* 3.333
seconds */
#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS(500) /* Half a
second */
```

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

    /* Create one-shot timer */
    xOneShotTimer = xTimerCreate("OneShot",
mainONE_SHOT_TIMER_PERIOD, pdFALSE, 0, prvOneShotTimerCallback);

    /* Create auto-reload timer */
    xAutoReloadTimer = xTimerCreate("AutoReload",
mainAUTO_RELOAD_TIMER_PERIOD, pdTRUE, 0,
prvAutoReloadTimerCallback);

    // Check if timers were created successfully

    if (xOneShotTimer != NULL && xAutoReloadTimer != NULL) {
        /* Start the software timers */
        xTimer1Started = xTimerStart(xOneShotTimer, 0);
        xTimer2Started = xTimerStart(xAutoReloadTimer, 0);

        // Check if timers were started successfully

        if (xTimer1Started == pdPASS && xTimer2Started == pdPASS)
{
            /* Start the scheduler */
            vTaskStartScheduler();
        }
    }

    /* As always, this line should not be reached */
    for (;;) {
        // Do nothing
    }
```

```
}


static void prvOneShotTimerCallback(TimerHandle_t xTimer) {

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

    // Output a string to show the time at which the callback was
executed.
    vPrintString("One-shot timer callback executing %lu\n",
xTimeNow);
}

static void prvAutoReloadTimerCallback(TimerHandle_t xTimer) {

    // A count of the number of times this software timer has
expired is stored in the timer's ID.
    // Obtain the ID, increment it, then save it as the new ID
value.
    ulExecutionCount = (uint32_t)pvTimerGetTimerID(xTimer);
    ulExecutionCount++;
    vTimerSetTimerID(xTimer, (void *)ulExecutionCount);

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

    // Output a string to show the time at which the callback was
executed.
    vPrintString("Auto-reload timer callback executing %lu\n",
xTimeNow);

    // Stop the auto-reload timer after it has executed 5 times.
    // This callback function executes in the context of the RTOS
daemon task so must not call
```

```
    // any functions that might place the daemon task into the
Blocked state.
    if (ulExecutionCount == 5) {
        xTimerStop(xTimer, 0);
    }
}
```

Output:

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

One-shot timer callback executing 3333

6.3 Change time period of timer

```
#include "FreeRTOS.h"
#include "task.h"
#include "timers.h"
#include <stdarg.h>
#include <stdio.h>

TickType_t xTimeNow;
int tickCounter = 0; // Counter to track the number of ticks

void vApplicationIdleHook(void) {}

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
```

```
    fflush(stdout);
}

/* Define constants for timer periods */
#define NORMAL_PERIOD_MS 2000
#define FAST_PERIOD_MS   500

/* Timer callback function */
static void TimerCallback(TimerHandle_t xTimer) {
    TickType_t xTimeNow;
    xTimeNow = xTaskGetTickCount();
    vPrintString("Timer callback executing at tick %d\n",
xTimeNow);

    // Increment the tick counter
    tickCounter++;

    // Change the period after 5 ticks
    if (tickCounter == 5) {
        xTimerChangePeriod(xTimer, pdMS_TO_TICKS(FAST_PERIOD_MS),
0);

    }
}

int main(void) {
    // Create the timer with the initial normal period
    TimerHandle_t xTimer = xTimerCreate("MyTimer",
pdMS_TO_TICKS(NORMAL_PERIOD_MS), pdTRUE, NULL, TimerCallback);

    if (xTimer != NULL) {
        // Start the timer
        if (xTimerStart(xTimer, 0) != pdPASS) {
            // Timer start failed
```

```
            printf("Timer start failed\n");
        }
    } else {
        // Timer creation failed
        printf("Timer creation failed\n");
    }

    // The FreeRTOS scheduler should never return
    vTaskStartScheduler();

    // The following code should not be reached
    for (;;) {}
}
```

Output:

Timer callback executing at tick 2000

Timer callback executing at tick 4000

Timer callback executing at tick 6000

Timer callback executing at tick 8000

Timer callback executing at tick 10000

Timer callback executing at tick 10505

Timer callback executing at tick 11005

Timer callback executing at tick 11505

Timer callback executing at tick 12005

.

.

6.4 timer reset

```
#include "FreeRTOS.h"
#include "task.h"
#include "timers.h"
```

```
#include <stdio.h>
#include <stdarg.h>

/* Define constants for timer period and reset intervals */
#define TIMER_PERIOD_MS 1000

/* Timer handle */
TimerHandle_t xTimer;

/* Task handle */
TaskHandle_t xResetTaskHandle = NULL;
void vApplicationIdleHook(void) {}

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

/* Timer callback function */
static void TimerCallback(TimerHandle_t xTimer) {
    TickType_t xTimeNow = xTaskGetTickCount();
    vPrintString("Timer callback executing at %d ticks\n",
xTimeNow);
}

/* Task to reset the timer periodically */
static void vResetTimerTask(void *pvParameters) {
    int resetCount = 0;

    while (resetCount < 3) {
```

```
        // Simulated delay before each reset (less than timer
period to ensure reset before expiry)
        vTaskDelay(pdMS_TO_TICKS(TIMER_PERIOD_MS / 2));

        // Reset the timer
        if (xTimerReset(xTimer, 0) != pdPASS) {
            // Timer reset failed
            vPrintString("Timer reset failed\n");
        } else {
            TickType_t xTimeNow = xTaskGetTickCount();
            vPrintString("Timer reset at %d ticks\n", xTimeNow);
            resetCount++;
        }
    }

    vPrintString("Timer reset task completed\n");

    // Suspend the reset task as its job is done
    vTaskSuspend(NULL);
}

int main(void) {
    // Create the timer with the initial period
    xTimer = xTimerCreate("MyTimer",
pdMS_TO_TICKS(TIMER_PERIOD_MS), pdTRUE, NULL, TimerCallback);

    if (xTimer != NULL) {
        // Start the timer
        if (xTimerStart(xTimer, 0) != pdPASS) {
            // Timer start failed
            vPrintString("Timer start failed\n");
        }
    } else {
```

```
        // Timer creation failed
        vPrintString("Timer creation failed\n");
    }

    // Create the task to reset the timer
    if (xTaskCreate(vResetTimerTask, "ResetTimerTask",
configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 1,
&xResetTaskHandle) != pdPASS) {
        // Task creation failed
        vPrintString("Reset task creation failed\n");
        return 1;
    }

    // Start the scheduler
    vTaskStartScheduler();

    // The following code should not be reached
    for (;;) {}

}
```

Output:

Timer reset at 500 ticks

Timer reset at 1003 ticks

Timer reset at 1506 ticks

Timer reset task completed

Timer callback executing at 2506 ticks

Timer callback executing at 3506 ticks

Timer callback executing at 4506 ticks

Timer callback executing at 5506 ticks