Module 4 : Software Timers & Interrupts

Module 4 : Software Timers & Interrupts

Software Timer

Software timer is a feature of FreeRTOS that can call a function when the timer expires. This function is known as a callback function and is passed to the timer as an argument. The callback function must be quick and non-blocking, similar to an ISR.

Software timers rely on a tick timer, which is a hardware timer that generates interrupts at a fixed frequency. The tick timer determines the resolution of the software timer, meaning that we cannot create a software timer with a period or delay of less than one tick.

There are two types of software timers: one-shot and auto-reload. A one-shot timer will run the callback function only once after the specified delay. An auto-reload timer will run the callback function repeatedly at the specified period.

How to Use Software Timer

To use a software timer in FreeRTOS (the vanilla version, not the one currently in use), we need to include the header file `timers.h`, which contains API functions for creating, deleting, starting, stopping, and resetting the timer. We also need to enable the `configUSE_TIMERS` setting in the `FreeRTOSConfig.h` file, which will create a background task that manages the software timers.

This background task is called the timer service task or timer daemon. It maintains a list of timers and calls their callback functions when the timers expire. This task does not run continuously but wakes up only when the tick timer reaches one of the expiration times.

We do not control the timer service task directly, but instead, we send commands to it via a queue. The queue is accessed by the API functions, which place commands in the queue to create, start, stop, and reset timers.

The API functions return a boolean value indicating whether the command was successfully sent to the queue or not. If the queue is full, we can specify a timeout value to wait for space to become available in the queue.

How to Create a Software Timer

To create a software timer, we use the `xTimerCreate` function, which returns a handle to the timer. This handle is used to identify and control the timer in other API functions.

The `xTimerCreate` function takes five parameters:

1. **Timer name**, which is a string for debugging purposes.
2. **Timer period or delay**, in ticks. We can use the macro `pdMS_TO_TICKS` to convert milliseconds to ticks.
3. **Auto-reload setting**, which can be `pdTRUE` for an auto-reload timer or `pdFALSE` for a one-shot timer.
4. **Timer ID**, which is a pointer to any data type. We can use it to store some information about the timer or to identify it in the callback function.
5. **Callback function**, which is the name of the function to be called when the timer expires.

For example, we can create a one-shot timer that will call a function named `vOneShotCallback` after 2000 milliseconds with the following code:

```
TimerHandle_t xOneShotTimer;
xOneShotTimer = xTimerCreate("OneShot", pdMS_TO_TICKS(2000), pdFALSE, (void *) 0, vOneShotCallback);
```

We can create an auto-reload timer that will call the `vAutoReloadCallback` function every 1000 milliseconds with the following code:

```
TimerHandle_t xAutoReloadTimer;
xAutoReloadTimer = xTimerCreate("AutoReload", pdMS_TO_TICKS(1000), pdTRUE, (void *) 1,
vAutoReloadCallback);
```

Note that we have used different values for the timer ID parameter to distinguish between the two timers.

We should always check whether the `xTimerCreate` function returns a valid handle or not. If it returns `NULL`, it means that there is not enough heap memory allocated for the timer.

How to Start a Software Timer

To start a software timer, we use the `xTimerStart` function, which takes two parameters:

1. **Timer handle** that will be started.
2. **Timeout value** to send the command to the queue.

For example, we can start a one-shot timer and an auto-reload timer with the following code:

```
if (xOneShotTimer != NULL) {
    xTimerStart(xOneShotTimer, portMAX_DELAY);
}

if (xAutoReloadTimer != NULL) {
    xTimerStart(xAutoReloadTimer, portMAX_DELAY);
}
```

We have used `portMAX_DELAY` as the timeout value, which means we will wait indefinitely if the queue is full.

Note that the `xTimerStart` function will also restart the timer if it is already running. This means the timer will be reset to its initial value.

How to Stop a Software Timer

To stop a software timer, use the following code:

```
if (xAutoReloadTimer != NULL) { xTimerStop(xAutoReloadTimer, portMAX_DELAY); }
```

How to Reset a Software Timer

To reset a software timer, use the following code:

```
if (xOneShotTimer != NULL) { xTimerReset(xOneShotTimer, portMAX_DELAY); }
```

How to Delete a Software Timer

To delete a software timer, use the following code:

```
if (xOneShotTimer != NULL) { xTimerDelete(xOneShotTimer, portMAX_DELAY); }

if (xAutoReloadTimer != NULL) { xTimerDelete(xAutoReloadTimer, portMAX_DELAY); }
```

How to Create a Callback Function

A callback function is a function given to the timer as an argument and called when the timer expires. This function should not return anything and should take the timer handle as a parameter.

We can use the timer handle to identify which timer called the function or to access the ID or other information. We can also use the `xTimerChangePeriod` function to change the period of a running timer from within the callback function.

We should write our callback function in a manner similar to an ISR: it should be quick and non-blocking, avoiding the use of delay functions or blocking operations with queues, mutexes, and semaphores. The callback function should avoid calling API functions that are not interrupt-safe.

For example, we can write one-shot and auto-reload callback functions with the following code:

```
void vOneShotCallback(TimerHandle_t xTimer) { // Get the timer ID
    uint32_t ulTimerID = (uint32_t) pvTimerGetTimerID(xTimer);
    // Check if it is our one-shot timer
    if (ulTimerID == 0) {
        // Do something once
        Serial.println("One-shot timer expired");
    }
}

void vAutoReloadCallback(TimerHandle_t xTimer) { // Get the timer ID
    uint32_t ulTimerID = (uint32_t) pvTimerGetTimerID(xTimer);
    // Check if it is our auto-reload timer
    if (ulTimerID == 1) {
        // Do something repeatedly
        Serial.println("Auto-reload timer expired");
    }
}
```

Real-Time Operating System (RTOS) and Hardware Interrupts

A Real-Time Operating System (RTOS) is a software platform that allows embedded systems to run multiple tasks concurrently and efficiently. One of the key features of an RTOS is the ability to handle hardware interrupts, which are signals that notify the processor of asynchronous events requiring immediate attention.

Hardware interrupts can be generated by various sources, such as timers, buttons, communication buses, or sensors. For example, a timer can trigger an interrupt when it reaches a certain count, or a button can trigger an interrupt when pressed by the user. These interrupts can be used to

perform specific actions or gather data from devices.

However, hardware interrupts also pose several challenges for RTOS developers. For instance, how to synchronize shared data and variables between interrupt service routines (ISRs) and tasks? How to avoid blocking or delaying other interrupts or tasks when processing an interrupt? How to use RTOS API functions correctly and safely within an ISR?

Setting Up Hardware Interrupts

Let's start with a simple example of setting up a hardware timer interrupt on the ESP32. The ESP32 has four timers, each with a 16-bit prescaler and a 64-bit counter. The default timer base clock is 80 MHz, which means the timer ticks 80 million times per second.

We can use a prescaler to reduce the timer frequency. For example, if we set the prescaler to 80, the timer will tick at 1 MHz, or one million times per second. We can also set a maximum count value for the timer, which determines when the timer will trigger an interrupt. For instance, if we set the maximum count to one million, the timer will trigger an interrupt every second.

We can use the ESP32 HAL timer library included with the Arduino package to configure and start the timer. We also need to define an ISR function that will execute when the timer interrupt occurs. In this example, we will simply toggle an LED within the ISR.

Code to Set Up Timer Interrupt

```
// Define LED pin
#define LED_PIN 2

// Define timer handle
hw_timer_t *timer = NULL;

// Define ISR function
void IRAM_ATTR onTimer() {
  // Toggle LED
  digitalWrite(LED_PIN, !digitalRead(LED_PIN));
}

void setup() {
  // Configure LED pin as output
  pinMode(LED_PIN, OUTPUT);
```

```
  // Create and start hardware timer number 0
  timer = timerBegin(0, 80, true);

  // Configure ISR as a callback function
  timerAttachInterrupt(timer, &onTimer, true);

  // Set maximum count value
  timerAlarmWrite(timer, 1000000, true);

  // Enable timer interrupt
  timerAlarmEnable(timer);
}

void loop() {
  // Do nothing
}
```

Timer Interrupt and Synchronizing Variables between ISRs and Tasks

If we upload this code to an ESP32 board, we should see the LED blinking with a one-second interval.

Synchronizing Variables between ISRs and Tasks

A common scenario in embedded systems is to collect data from a device using an ISR and then process that data in a task. For example, we might want to sample an analog value from a sensor at a regular interval using a timer interrupt, and then compute some statistics or perform calculations on that value in a task.

However, this also means we need to share some variables between the ISR and the task. For instance, we may need to store the sampled values in a global variable or buffer that can be accessed by both the ISR and the task. This introduces some challenges for synchronization and concurrency control.

One of the main challenges is that an ISR can interrupt a task at any time and modify shared variables while the task is using them. This can result in inconsistent or corrupted data. For example, consider a global variable storing an integer value. The ISR increments this variable by one every time it runs. The task decrements this variable by one in a loop and prints its value.

If we don't synchronize this variable properly, we might encounter issues. For instance, if the initial value of the variable is zero. The task reads this value and prepares to decrement it by one. However, before it can write the new value of negative one, an ISR occurs and increments the variable by one. The ISR finishes and returns to the task. The task then writes back its computed value, negative one, overwriting the one that was just set by the ISR. The result is incorrect, negative one instead of zero.

To avoid such issues, we need to protect the shared variable from being accessed simultaneously by both the ISR and the task. One way to do this is by using critical sections. A critical section is a code segment that disables interrupts and prevents context switching while it is executing. This ensures that the shared variable is accessed by only one entity at a time.

In FreeRTOS, we can use special functions to enter and exit critical sections. For example, we can use `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()` in tasks, and `portENTER_CRITICAL_ISR()` and `portEXIT_CRITICAL_ISR()` in ISRs. These functions also use spinlocks to prevent tasks on other cores from entering the critical section.

Here is an example of using critical sections to protect shared variables between an ISR and a task:

Code for Synchronizing Variables between ISRs and Tasks

```
// Define LED pin
#define LED_PIN 2

// Define timer handle
hw_timer_t *timer = NULL;

// Define global variable
volatile int counter = 0;

// Define ISR function
void IRAM_ATTR onTimer() {
  // Enter critical section
  portENTER_CRITICAL_ISR(&timerMux);

  // Increment global variable
  counter++;

  // Exit critical section
```

```
  portEXIT_CRITICAL_ISR(&timerMux);
}

// Define task function
void printValues(void * parameter) {
  // Loop forever
  while (true) {
    // Enter critical section
    taskENTER_CRITICAL();

    // Decrement global variable
    counter--;

    // Exit critical section
    taskEXIT_CRITICAL();

    // Print global variable
    Serial.println(counter);

    // Wait for two seconds
    delay(2000);
  }
}

void setup() {
  // Start serial terminal
  Serial.begin(115200);

  // Configure LED pin as output
  pinMode(LED_PIN, OUTPUT);

  // Create and start hardware timer number 0
  timer = timerBegin(0, 8, true);

  // Configure ISR as a callback function
  timerAttachInterrupt(timer, &onTimer, true);

  // Set maximum count value
  timerAlarmWrite(timer, 100000, true);
```

```
  // Enable timer interrupt
  timerAlarmEnable(timer);

  // Create and start task
  xTaskCreatePinnedToCore(printValues, "Print Values", 10000, NULL, 1, NULL, app_cpu);
}

void loop() {
  // Do nothing
}
```

If we upload this code to our ESP32 board, we should see the global variable counting down from some value every two seconds. We may also observe some repeated values, indicating that the ISR is running between the serial print statements. This behavior is expected, as we want the ISR to run asynchronously and separately from the task.

Using Semaphores for Synchronizing ISR and Tasks

Another way to synchronize an ISR and a task is by using semaphores. A semaphore is a signaling mechanism that can be used to control access to shared resources or to notify a task about an event. Semaphores can have binary or counting values. A binary semaphore can only have two states: available or taken. A counting semaphore can have multiple states: from zero to a maximum value.

Semaphores can be taken or given by tasks or ISRs. Taking a semaphore means acquiring or locking the semaphore. Giving a semaphore means releasing or unlocking it. A task can be blocked on a semaphore until it becomes available. An ISR cannot be blocked on a semaphore but can give a semaphore to unlock a waiting task.

One common use case for a semaphore is to signal a task when some data is ready to be processed by an ISR. For example, we might want to sample analog values from different sensors at a regular interval using a timer interrupt and then compute some statistics or perform calculations on those values in a task.

In this case, we can use a binary semaphore to notify the task when the ISR has sampled new values. The ISR will store the sampled values in a global variable and give the semaphore. The task will wait on the semaphore and take it when available. The task will then read the global variable and process the sampled values.

However, we need to use special functions ending with `FromISR` when using semaphores within an ISR. These functions will never block and will also check if giving the semaphore has unlocked a

higher-priority task. If so, they will request a context switch so that the higher-priority task can run immediately after the ISR finishes.

Here is an example of using a binary semaphore to synchronize an ISR and a task:

```
// Define global variables
SemaphoreHandle_t binarySemaphore;
volatile int sampledValue = 0;

// Define ISR function
void IRAM_ATTR onTimer() {
  // Store sampled value
  sampledValue = analogRead(A0);

  // Give semaphore
  xSemaphoreGiveFromISR(binarySemaphore, NULL);
}

// Define task function
void processValues(void * parameter) {
  while (true) {
    // Wait for semaphore
    if (xSemaphoreTake(binarySemaphore, portMAX_DELAY) == pdTRUE) {
      // Process sampled value
      Serial.println(sampledValue);
    }
  }
}

void setup() {
  // Start serial terminal
  Serial.begin(115200);

  // Create binary semaphore
  binarySemaphore = xSemaphoreCreateBinary();

  // Create and start hardware timer
  timer = timerBegin(0, 80, true);
  timerAttachInterrupt(timer, &onTimer, true);
  timerAlarmWrite(timer, 1000000, true);
```

```
  timerAlarmEnable(timer);

  // Create and start task
  xTaskCreatePinnedToCore(processValues, "Process Values", 10000, NULL, 1, NULL, app_cpu);
}

void loop() {
  // Do nothing
}
```

In this example, the ISR samples a value and gives the semaphore. The task waits for the semaphore, takes it when available, and processes the sampled value.

Using Semaphores for Synchronizing ISR and Tasks

Another way to synchronize an ISR and a task is by using semaphores. A semaphore is a signaling mechanism that can be used to control access to shared resources or notify a task about an event. Semaphores can have binary or counting values. A binary semaphore can only have two states: available or taken. A counting semaphore can have multiple states: from zero to a maximum value.

Semaphores can be taken or given by tasks or ISRs. Taking a semaphore means acquiring or locking the semaphore. Giving a semaphore means releasing or unlocking it. A task can be blocked on a semaphore until it becomes available. An ISR cannot be blocked on a semaphore but can give a semaphore to unlock a waiting task.

One common use case for a semaphore is to signal a task when some data is ready to be processed by an ISR. For example, we might want to sample analog values from different sensors at a regular interval using a timer interrupt and then compute some statistics or perform calculations on those values in a task.

In this case, we can use a binary semaphore to notify the task when the ISR has sampled new values. The ISR will store the sampled values in a global variable and give the semaphore. The task will wait on the semaphore and take it when available. The task will then read the global variable and process the sampled values.

However, we need to use special functions ending with `FromISR` when using semaphores within an ISR. These functions will never block and will also check if giving the semaphore has unlocked a higher-priority task. If so, they will request a context switch so that the higher-priority task can run immediately after the ISR finishes.

Here is an example of using a binary semaphore to synchronize an ISR and a task:

```
// Define LED pin
#define LED_PIN 2

// Define timer handler
hw_timer_t *timer = NULL;

// Define global variable
volatile int adcValue = 0;

// Define binary semaphore
SemaphoreHandle_t binSemaphore = NULL;

// Define ISR function
void IRAM_ATTR onTimer() {
  // Sample analog value from pin 34
  adcValue = analogRead(34);

  // Give binary semaphore
  xSemaphoreGiveFromISR(binSemaphore, NULL);
}

// Define task function
void processValues(void * parameter) {
  // Loop forever
  while (true) {
    // Wait for binary semaphore
    xSemaphoreTake(binSemaphore, portMAX_DELAY);

    // Process the sampled value
    Serial.println(adcValue);

    // Wait for one second
    delay(1000);
  }
}

void setup() {
  // Start serial terminal
  Serial.begin(115200);
```

```
  // Configure LED pin as output
  pinMode(LED_PIN, OUTPUT);

  // Create and start hardware timer number 0
  timer = timerBegin(0, 8, true);

  // Configure ISR as callback function
  timerAttachInterrupt(timer, &onTimer, true);

  // Set maximum alarm value
  timerAlarmWrite(timer, 100000, true);

  // Enable timer interrupt
  timerAlarmEnable(timer);

  // Create binary semaphore
  binSemaphore = xSemaphoreCreateBinary();

  // Create and start task
  xTaskCreatePinnedToCore(processValues, "Process Values", 10000, NULL, 1, NULL, app_cpu);
}

void loop() {
  // Do nothing
}
```

Using Queues to Pass Data Between ISR and Tasks

Another way to pass data between an ISR and a task is by using queues. A queue is a data structure that can hold multiple items of the same type in a FIFO (first-in, first-out) order. Queues can be created with fixed sizes and fixed item sizes. Queues can be filled or emptied by tasks or ISRs. Filling a queue means adding items to the end of the queue, while emptying a queue means removing items from the front.

A task can be blocked on a queue until it is full or empty. An ISR cannot be blocked on a queue but can fill or empty a queue and check if filling the queue has unlocked a higher-priority task. If so, the ISR will request a context switch so that the higher-priority task can run immediately after the ISR

completes.

A common use case for a queue is to pass several data items from an ISR to a task. For example, we might want to sample analog values from different sensors at a regular interval using a timer interrupt and then compute some statistics or perform calculations on those values in a task.

In this case, we can use a queue to store the sampled values in an array or structure. The ISR will place the array or structure into the queue. The task will take the array or structure from the queue and process the sampled values.

Here is an example of using a queue to pass data between an ISR and a task:

```
// Define LED pin
#define LED_PIN 2

// Define timer handler
hw_timer_t *timer = NULL;

// Define data structure
typedef struct {
  int adcValue1;
  int adcValue2;
} sensorData_t;

// Define queue handler
QueueHandle_t sensorQueue = NULL;

// Define ISR function
void IRAM_ATTR onTimer() {
  // Create an instance of the data structure
  sensorData_t data;

  // Sample analog values from pin 34 and 35
  data.adcValue1 = analogRead(34);
  data.adcValue2 = analogRead(35);

  // Send the data structure to the queue
  xQueueSendFromISR(sensorQueue, &data, NULL);
}

// Define task function
```

```
void processValues(void * parameter) {
  // Create an instance of the data structure
  sensorData_t data;

  // Loop forever
  while (true) {
    // Receive the data structure from the queue
    xQueueReceive(sensorQueue, &data, portMAX_DELAY);

    // Process the sampled values
    Serial.print(data.adcValue1);
    Serial.print(" ");
    Serial.println(data.adcValue2);

    // Wait for one second
    delay(1000);
  }
}

void setup() {
  // Start serial terminal
  Serial.begin(115200);

  // Configure LED pin as output
  pinMode(LED_PIN, OUTPUT);

  // Create and start hardware timer number 0
  timer = timerBegin(0, 8, true);

  // Configure ISR as callback function
  timerAttachInterrupt(timer, &onTimer, true);

  // Set maximum alarm value
  timerAlarmWrite(timer, 100000, true);

  // Enable timer interrupt
  timerAlarmEnable(timer);

  // Create a queue with size 10 and item size sensorData_t
  sensorQueue = xQueueCreate(10, sizeof(sensorData_t));
```

```
  // Create and start task
  xTaskCreatePinnedToCore(processValues, "Process Values", 10000, NULL, 1, NULL, app_cpu);
}

void loop() {
  // Do nothing
}
```

External Reference

Check out the external reference by digikey: Software Timers & Hardware Interrupts