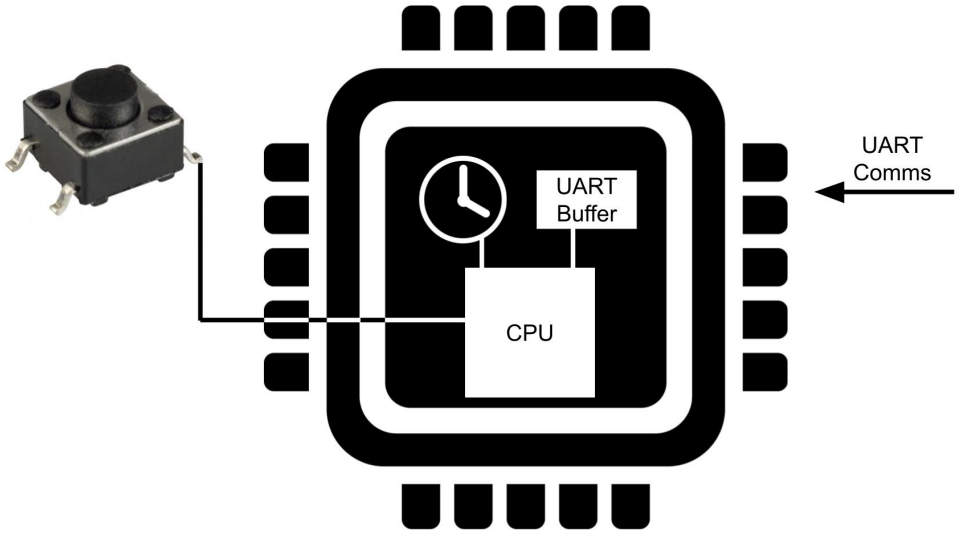
Introduction to RTOS - Solution to Part 9 (Hardware Interrupts)

By ShawnHymel

Concepts

Hardware interrupts are an important part of many embedded systems. They allow events to occur asynchronously (not as part of any executing program) and notify the CPU that it should take some action. These types of interrupts can cause the CPU to stop whatever it was doing and execute some other function, known as an “interrupt service routine” (ISR).

Such hardware interrupts can include things like button presses (input pin voltage change), a hardware timer expiring, or a communication buffer being filled.



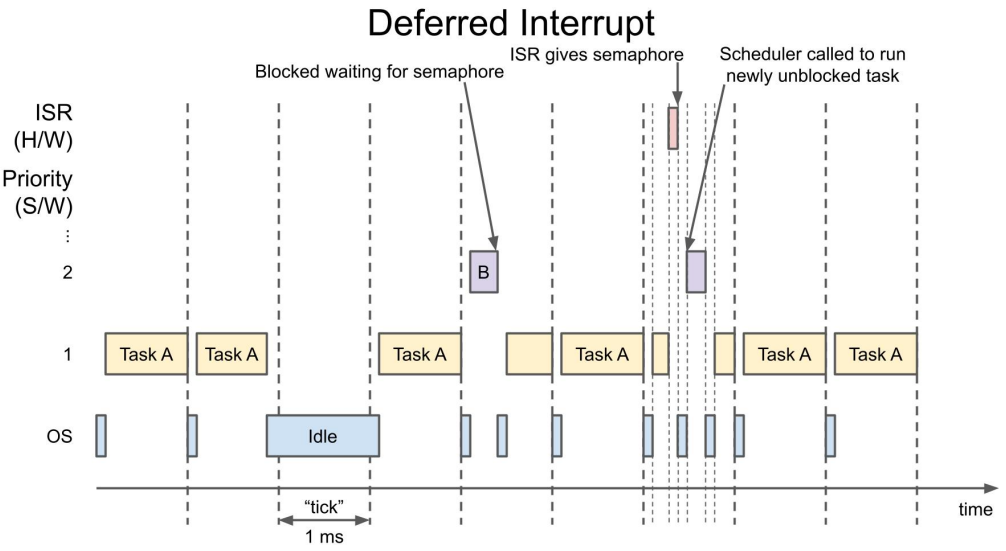
In most RTOSes (FreeRTOS included), hardware interrupts have a higher priority than any task (unless we purposely disable hardware interrupts). When working with hardware interrupts, there are few things to keep in mind.

First, an ISR should never block itself. ISRs are not executed as part of a task, and as a result, cannot be blocked. Because of this, you should only use FreeRTOS function calls that end in *FromISR inside an ISR. You cannot wait for a queue, mutex, or semaphore, so you will need to come up with alternatives if writing to one fails!

Second, you should keep ISRs as short as possible. You probably do not want to delay the execution of all your waiting tasks!

Third, if a variable (such as a global) is updated inside an ISR, you likely need to declare it with the “volatile” qualifier. This lets the compiler know that the “volatile” variable can change outside the current thread of execution. Without it, compilers may (depending on their optimization settings) simply remove the variable, as they don’t think it’s being used (anywhere inside main() or various tasks).

Finally, one of the easiest ways to synchronize a task to an ISR is to use what's known as a "deferred interrupt." Here, we defer processing the data captured inside the ISR to another task. Whenever such data has been captured, we can give a semaphore (or send a task notification) to let some other task know that data is ready for processing.



In the diagram above, Task B is blocked waiting for a semaphore. Only the ISR gives the semaphore. So, as soon as the ISR runs (and, say, collects some data from a sensor), it gives the semaphore. When the ISR has finished executing, Task B is immediately unblocked and runs to process the newly collected data.

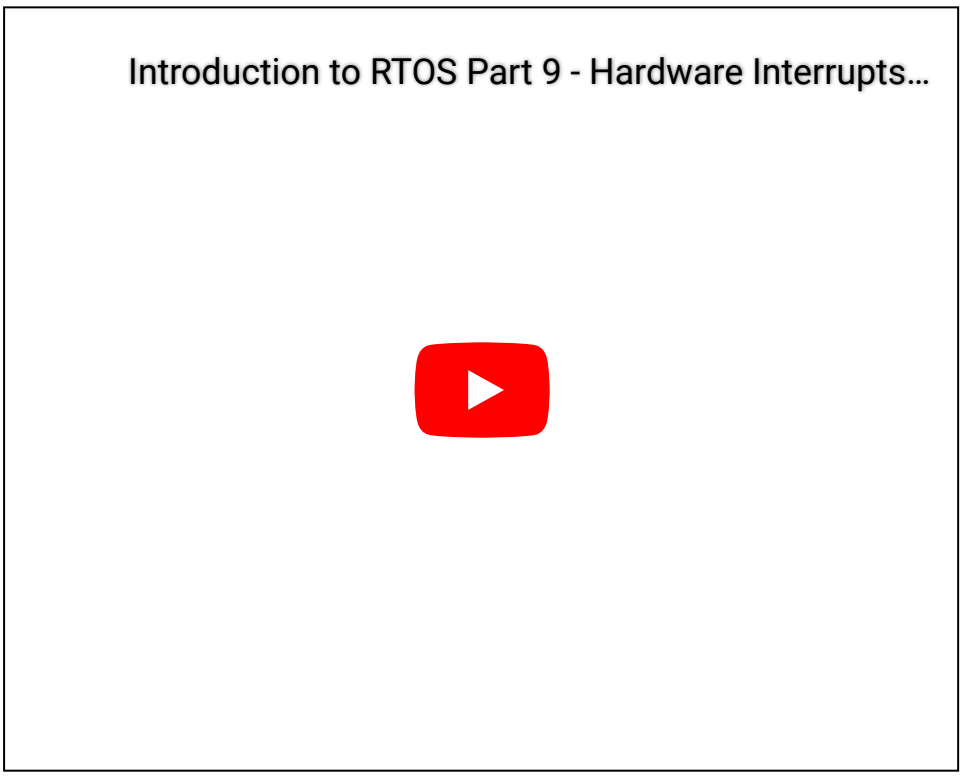
Required Hardware

Any ESP32 development board should work, so long as it's supported in the Arduino IDE. See here for a list of supported ESP32 boards. You may also use any development board capable of running FreeRTOS, although my solution will likely vary some (as the ESP32 runs a modified version of FreeRTOS called ESP-IDF).

This solution uses the Adafruit Feather HUZZAH32.

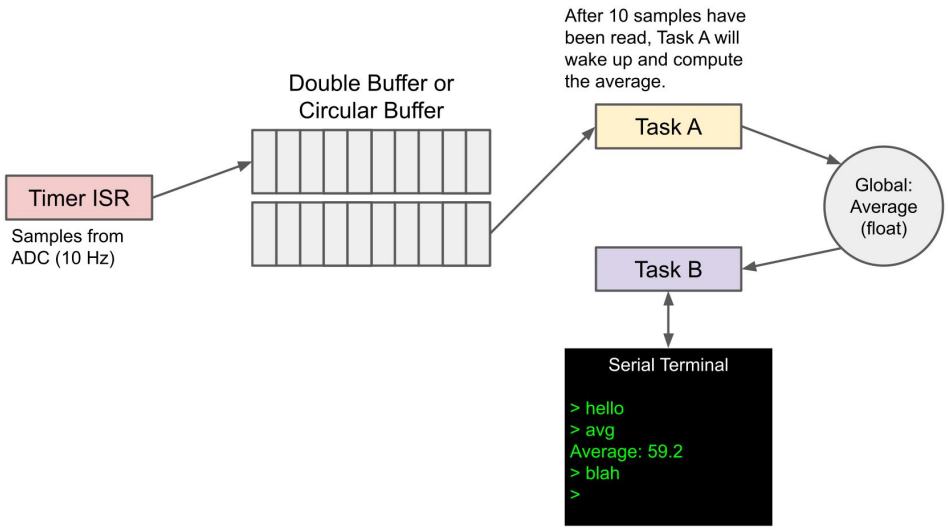
Video

If you have not done so, please watch the following video, which provides the steps necessary to creating tasks and assigning priorities. It also demonstrates a working version of the challenge:



Challenge

Your challenge is to create a sampling, processing, and interface system using hardware interrupts and whatever kernel objects (e.g. queues, mutexes, and semaphores) you might need.



You should implement a hardware timer in the ESP32 (here is a good article showing how to do that) that samples from an ADC pin once every 100 ms. This sampled data should be copied to a double buffer (you could also use a circular buffer). Whenever one of the buffers is full, the ISR should notify Task A.

Task A, when it receives notification from the ISR, should wake up and compute the average of the previously collected 10 samples. Note that during this calculation time, the ISR may trigger again. This is where a double (or circular) buffer will help: you can process one buffer while the other is filling up.

When Task A is finished, it should update a global floating point variable that contains the newly computed average. Do not assume that writing to this floating point variable will take a single instruction cycle! You will need to protect that action as we saw in the queue episode.

Task B should echo any characters received over the serial port back to the same serial port. If the command “avg” is entered, it should display whatever is in the global average variable.

This is like a “final project” in that you will need to use many of the concepts we covered in previous lectures and challenges.

Solution

Spoilers below! I highly encourage you to try the challenge on your own before comparing your answer to mine. Note that my solution may not be the only way to solve the challenge.

Copy Code

```
/**
 * ESP32 Sample and Process Solution
 *
 * Sample ADC in an ISR, process in a task.
 *
 * Date: February 3, 2021
 * Author: Shawn Hymel
 * License: 0BSD
 */

// You'll likely need this on vanilla FreeRTOS
//#include semphr.h

// Use only core 1 for demo purposes
#if CONFIG_FREERTOS_UNICORE
  static const BaseType_t app_cpu = 0;
#else
  static const BaseType_t app_cpu = 1;
#endif

// Settings
static const char command[] = "avg";              // Command
static const uint16_t timer_divider = 8;          // Divide 80 MHz by this
static const uint64_t timer_max_count = 1000000;  // Timer counts to this value
static const uint32_t cli_delay = 20;             // ms delay
enum { BUF_LEN = 10 };      // Number of elements in sample buffer
enum { MSG_LEN = 100 };     // Max characters in message body
enum { MSG_QUEUE_LEN = 5 }; // Number of slots in message queue
enum { CMD_BUF_LEN = 255};  // Number of characters in command buffer

// Pins
static const int adc_pin = A0;

// Message struct to wrap strings for queue
typedef struct Message {
  char body[MSG_LEN];
} Message;

// Globals
static hw_timer_t *timer = NULL;
static TaskHandle_t processing_task = NULL;
static SemaphoreHandle_t sem_done_reading = NULL;
static portMUX_TYPE spinlock = portMUX_INITIALIZER_UNLOCKED;
static QueueHandle_t msg_queue;
static volatile uint16_t buf_0[BUF_LEN];      // One buffer in the pair
static volatile uint16_t buf_1[BUF_LEN];      // The other buffer in the pair
static volatile uint16_t* write_to = buf_0;   // Double buffer write pointer
static volatile uint16_t* read_from = buf_1;  // Double buffer read pointer
static volatile uint8_t buf_overrun = 0;      // Double buffer overrun flag
static float adc_avg;

//*****************************************************************************
// Functions that can be called from anywhere (in this file)

// Swap the write_to and read_from pointers in the double buffer
// Only ISR calls this at the moment, so no need to make it thread-safe
void IRAM_ATTR swap() {
  volatile uint16_t* temp_ptr = write_to;
  write_to = read_from;
  read_from = temp_ptr;
```

```
}

//*****************************************************************************
// Interrupt Service Routines (ISRs)

// This function executes when timer reaches max (and resets)
void IRAM_ATTR onTimer() {

  static uint16_t idx = 0;
  BaseType_t task_woken = pdFALSE;

  // If buffer is not overrun, read ADC to next buffer element. If buffer is
  // overrun, drop the sample.
  if ((idx < BUF_LEN) && (buf_overrun == 0)) {
    write_to[idx] = analogRead(adc_pin);
    idx++;
  }

  // Check if the buffer is full
  if (idx >= BUF_LEN) {

    // If reading is not done, set overrun flag. We don't need to set this
    // as a critical section, as nothing can interrupt and change either value.
    if (xSemaphoreTakeFromISR(sem_done_reading, &task_woken) == pdFALSE) {
      buf_overrun = 1;
    }

    // Only swap buffers and notify task if overrun flag is cleared
    if (buf_overrun == 0) {

      // Reset index and swap buffer pointers
      idx = 0;
      swap();

      // A task notification works like a binary semaphore but is faster
      vTaskNotifyGiveFromISR(processing_task, &task_woken);
    }
  }

  // Exit from ISR (Vanilla FreeRTOS)
  //portYIELD_FROM_ISR(task_woken);

  // Exit from ISR (ESP-IDF)
  if (task_woken) {
    portYIELD_FROM_ISR();
  }
}

//*****************************************************************************
// Tasks

// Serial terminal task
void doCLI(void *parameters) {

  Message rcv_msg;
  char c;
  char cmd_buf[CMD_BUF_LEN];
  uint8_t idx = 0;
  uint8_t cmd_len = strlen(command);

  // Clear whole buffer
  memset(cmd_buf, 0, CMD_BUF_LEN);

  // Loop forever
  while (1) {

    // Look for any error messages that need to be printed
    if (xQueueReceive(msg_queue, (void *)&rcv_msg, 0) == pdTRUE) {
      Serial.println(rcv_msg.body);
    }

    // Read characters from serial
    if (Serial.available() > 0) {
      c = Serial.read();
```

```
      // Store received character to buffer if not over buffer limit
      if (idx < CMD_BUF_LEN - 1) {
        cmd_buf[idx] = c;
        idx++;
      }

      // Print newline and check input on 'enter'
      if ((c == '\n') || (c == '\r')) {

        // Print newline to terminal
        Serial.print("\r\n");

        // Print average value if command given is "avg"
        cmd_buf[idx - 1] = '\0';
        if (strcmp(cmd_buf, command) == 0) {
          Serial.print("Average: ");
          Serial.println(adc_avg);
        }

        // Reset receive buffer and index counter
        memset(cmd_buf, 0, CMD_BUF_LEN);
        idx = 0;

      // Otherwise, echo character back to serial terminal
      } else {
        Serial.print(c);
      }
    }

    // Don't hog the CPU. Yield to other tasks for a while
    vTaskDelay(cli_delay / portTICK_PERIOD_MS);
  }
}

// Wait for semaphore and calculate average of ADC values
void calcAverage(void *parameters) {

  Message msg;
  float avg;

  // Loop forever, wait for semaphore, and print value
  while (1) {

    // Wait for notification from ISR (similar to binary semaphore)
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

    // Calculate average (as floating point value)
    avg = 0.0;
    for (int i = 0; i < BUF_LEN; i++) {
      avg += (float)read_from[i];
      //vTaskDelay(105 / portTICK_PERIOD_MS); // Uncomment to test overrun flag
    }
    avg /= BUF_LEN;

    // Updating the shared float may or may not take multiple isntructions, so
    // we protect it with a mutex or critical section. The ESP-IDF critical
    // section is the easiest for this application.
    portENTER_CRITICAL(&spinlock);
    adc_avg = avg;
    portEXIT_CRITICAL(&spinlock);

    // If we took too long to process, buffer writing will have overrun. So,
    // we send a message to be printed out to the serial terminal.
    if (buf_overrun == 1) {
      strcpy(msg.body, "Error: Buffer overrun. Samples have been dropped.");
      xQueueSend(msg_queue, (void *)&msg, 10);
    }

    // Clearing the overrun flag and giving the "done reading" semaphore must
    // be done together without being interrupted.
    portENTER_CRITICAL(&spinlock);
    buf_overrun = 0;
    xSemaphoreGive(sem_done_reading);
    portEXIT_CRITICAL(&spinlock);
  }
```

```
}

//*****************************************************************************
// Main (runs as its own task with priority 1 on core 1)

void setup() {

  // Configure Serial
  Serial.begin(115200);

  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);
  Serial.println();
  Serial.println("---FreeRTOS Sample and Process Demo---");

  // Create semaphore before it is used (in task or ISR)
  sem_done_reading = xSemaphoreCreateBinary();

  // Force reboot if we can't create the semaphore
  if (sem_done_reading == NULL) {
    Serial.println("Could not create one or more semaphores");
    ESP.restart();
  }

  // We want the done reading semaphore to initialize to 1
  xSemaphoreGive(sem_done_reading);

  // Create message queue before it is used
  msg_queue = xQueueCreate(MSG_QUEUE_LEN, sizeof(Message));

  // Start task to handle command line interface events. Let's set it at a
  // higher priority but only run it once every 20 ms.
  xTaskCreatePinnedToCore(doCLI,
                          "Do CLI",
                          1024,
                          NULL,
                          2,
                          NULL,
                          app_cpu);

  // Start task to calculate average. Save handle for use with notifications.
  xTaskCreatePinnedToCore(calcAverage,
                          "Calculate average",
                          1024,
                          NULL,
                          1,
                          &processing_task,
                          app_cpu);

  // Start a timer to run ISR every 100 ms
  timer = timerBegin(0, timer_divider, true);
  timerAttachInterrupt(timer, &onTimer, true);
  timerAlarmWrite(timer, timer_max_count, true);
  timerAlarmEnable(timer);

  // Delete "setup and loop" task
  vTaskDelete(NULL);
}

void loop() {
  // Execution should never get here
}
```

Explanation

The swap() function allows us to swap pointers in the double buffer.

Copy Code

```
// Swap the write_to and read_from pointers in the double buffer
// Only ISR calls this at the moment, so no need to make it thread-safe
void IRAM_ATTR swap() {
  volatile uint16_t* temp_ptr = write_to;
```

```
  write_to = read_from;
  read_from = temp_ptr;
}
```

We have two buffers: buf_0 and buf_1. We also have two pointers: write_to and read_from. Each points to one of the buffers. Whenever swap() is called, these pointers switch which buffer they're pointing to. This function is to be called whenever one of the buffers fills up so that the "producer" (the ISR) can start writing to the next buffer while the "consumer" (Task A) reads from the first buffer.

The ISR (onTimer()) simply copies the analog-to-digital converter (ADC) value to an available element in the write_to buffer (which could be either buf_0 or buf_1). Note that we must check first that the double buffer has not been overrun!

Copy Code

```
// If buffer is not overrun, read ADC to next buffer element. If buffer is
// overrun, drop the sample.
if ((idx < BUF_LEN) && (buf_overrun == 0)) {
  write_to[idx] = analogRead(adc_pin);
  idx++;
}
```

We check for an overrun condition in the following if block, where we wait for the write_to buffer to be full.

Copy Code

```
// Check if the buffer is full
if (idx >= BUF_LEN) {

  // If reading is not done, set overrun flag. We don't need to set this
  // as a critical section, as nothing can interrupt and change either value.
  if (xSemaphoreTakeFromISR(sem_done_reading, &task_woken) == pdFALSE) {
    buf_overrun = 1;
  }

  // Only swap buffers and notify task if overrun flag is cleared
  if (buf_overrun == 0) {

    // Reset index and swap buffer pointers
    idx = 0;
    swap();

    // A task notification works like a binary semaphore but is faster
    vTaskNotifyGiveFromISR(processing_task, &task_woken);
  }
}
```

In that block, we check for a semaphore to be set by Task A. If that semaphore has not been set by the time we get here, it means that Task A has not finished reading from the read_from buffer even though the write_to buffer is full (i.e. the buffer has overrun). We set a global flag and drop any subsequent ADC samples until that flag is cleared.

If we've made it this far and the overrun flag is not set, we reset the counter index and call the swap() function (to swap the buffer pointers). We also give a task notification to our processing task (Task A). The second parameter in the vTaskNotifyGiveFromISR() function is a boolean that lets the scheduler know if we need to perform a context switch as soon as the ISR ends. This function will write a value to that variable (hence why we give it the address), and we pass that boolean to portYIELD_FROM_ISR().

Note that in ESP-IDF, this yield function does not accept a parameter, so we can replicate the action of the vanilla FreeRTOS function with the following if statement:

Copy Code

```
// Exit from ISR (Vanilla FreeRTOS)
//portYIELD_FROM_ISR(task_woken);
```

```
// Exit from ISR (ESP-IDF)
if (task_woken) {
  portYIELD_FROM_ISR();
}
```

The calcAverage() function is our Task A. It waits for a notification from the ISR with the following function:

Copy Code

```
// Wait for notification from ISR (similar to binary semaphore)
ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
```

It then computes the average of all the values stored in the read_from array. This average is stored in the global adc_avg variable. Note that writing to this variable might not happen in a single instruction cycle, so we need to protect it as a critical section. We could use a mutex, but I decided to show it using the critical section guards.

Copy Code

```
// Updating the shared float may or may not take multiple isntructions, so
// we protect it with a mutex or critical section. The ESP-IDF critical
// section is the easiest for this application.
portENTER_CRITICAL(&spinlock);
adc_avg = avg;
portEXIT_CRITICAL(&spinlock);
```

Note that these guards also disable interrupts (up to the priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY in the FreeRTOS config file). In ESP-IDF we must also provide these critical section guards with a spinlock. This spinlock works much like a mutex, but it forces tasks in the other core to wait (spin doing nothing rather than block) if they attempt to take the spinlock. You can read more about these critical section guards here.

We also check the buffer overrun flag. If it's set, we send a message to the interface task, which will print it to the serial terminal. Notice that we're using a queue here to send the message.

Copy Code

```
// If we took too long to process, buffer writing will have overrun. So,
// we send a message to be printed out to the serial terminal.
if (buf_overrun == 1) {
  strcpy(msg.body, "Error: Buffer overrun. Samples have been dropped.");
  xQueueSend(msg_queue, (void *)&msg, 10);
}
```

Regardless of the state of the overrun flag, we clear it and let the ISR know that we're done reading by setting the sem_done_reading semaphore. Because these two actions need to occur without being interrupted by the ISR (weird things happen if the ISR interrupts while the overrun flag is set and sem_done_reading is not set), we disable interrupts from occurring using the same set of critical section guards.

Copy Code

```
// Clearing the overrun flag and giving the "done reading" semaphore must
// be done together without being interrupted.
portENTER_CRITICAL(&spinlock);
buf_overrun = 0;
xSemaphoreGive(sem_done_reading);
portEXIT_CRITICAL(&spinlock);
```

The interface task (Task B) simply echoes all received characters back out over the same serial port (as we've seen before). If the string "avg" is received, it will print out "Average: " followed by the global adc_avg value.

Because writing to adc_avg was protected by the critical section guards, the scheduler will not allow Task B to run while the writing happens. As a result, we should not run into the race condition where we try to read a

partially written value. Alternatively, you could use a mutex in both tasks to protect the adc_avg variable.

I set the interface task to run at a higher priority than the adc_avg task so that the user won't experience lag if Task A is taking a long time (it might if the computation happened to be complex, like running a Fast Fourier Transform). However, we must make sure to delay (yield) periodically to allow other tasks to run.

Copy Code

```
// Don't hog the CPU. Yield to other tasks for a while
vTaskDelay(cli_delay / portTICK_PERIOD_MS);
```

In setup(), I create the kernel objects, start the tasks, start the hardware timer (with the attached callback ISR), and delete the "setup and loop" task.

Recommended Reading

All demonstrations and solutions for this course can be found in this GitHub repository.

If you would like to dig into these topics further, I recommend checking out the following excellent articles:

- FreeRTOS API reference (you will want to look for function calls that end in
- FromISR): https://www.freertos.org/a00106.html
- Chapter 6 (Interrupt Management) in the "Mastering the FreeRTOS Real Time Kernel" book is particularly helpful: https://www.freertos.org/Documentation/RTOS_book.html
- Critical sections in ESP-IDF: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/freertos-smp.html#critical-sections-disabling-interrupts
- What is a Real-Time Operating System Part 1(RTOS)?
- Introduction to RTOS - Solution to Part 2 (FreeRTOS)
- Introduction to RTOS - Solution to Part 3 (Task Scheduling)
- Introduction to RTOS - Solution to Part 4 (Memory Management)
- Introduction to RTOS - Solution to Part 5 (FreeRTOS Queue Example)
- Introduction to RTOS - Solution to Part 6 (FreeRTOS Mutex Example)
- Introduction to RTOS - Solution to Part 7 (FreeRTOS Semaphore Example)
- Introduction to RTOS - Solution to Part 8 (Software Timers)
- Introduction to RTOS - Solution to Part 10 (Deadlock and Starvation)
- Introduction to RTOS - Solution to Part 11 (Priority Inversion)
- Introduction to RTOS - Solution to Part 12 (Multicore Systems)

Key Parts and Components

1 Items

Mfr Part # 3405

HUZZAH32 ESP32 FEATHER LOOSE HDR

Adafruit Industries LLC

$19.95 Details

Add all Digi-Key Parts to Cart

TechForum

Have questions or comments? Continue the conversation on TechForum, DigiKey's online community and technical resource.

Visit TechForum

Arduino | 3D Printing | Raspberry Pi

Project Details

Platforms

Arduino

Development

C

C++

Tags

Arduino

RTOS

License

Attribution

Get Involved

Like

Save

1-800-344-4539 218-681-6674 sales@digikey.com 218-681-3380