

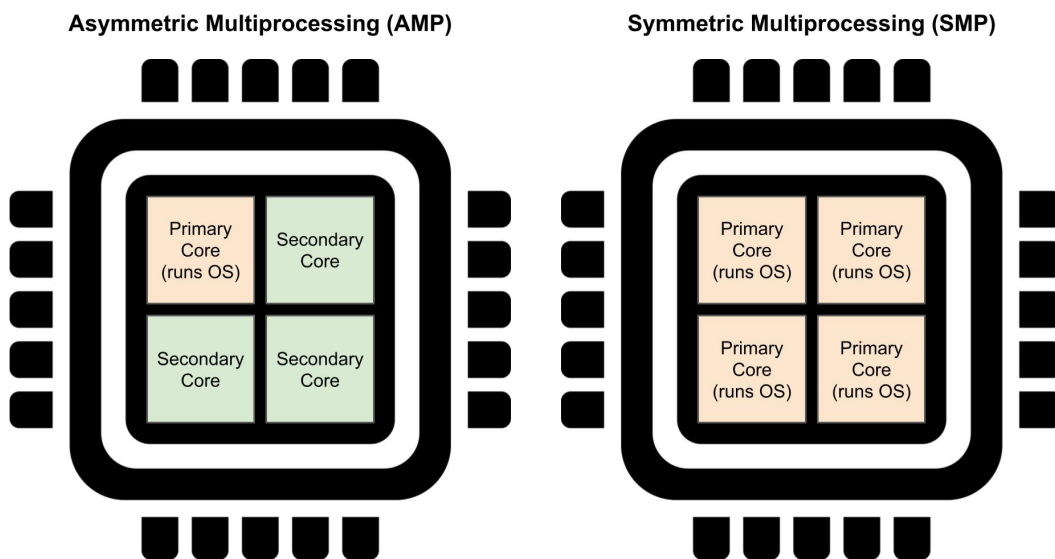
## Introduction to RTOS - Solution to Part 12 (Multicore Systems)

[By ShawnHymel](#)

### Concepts

Running an RTOS on a multicore system requires you to keep a few things in mind. FreeRTOS does not support multicore processors by default, but some variants, such as ESP-IDF, do support them. In this tutorial, we'll examine ESP-IDF and provide a solution to the challenge posed in the video to running a multithreaded application on a multicore system.

To begin, let's look at the difference between Asymmetric multiprocessing (AMP) and symmetric multiprocessing (SMP).

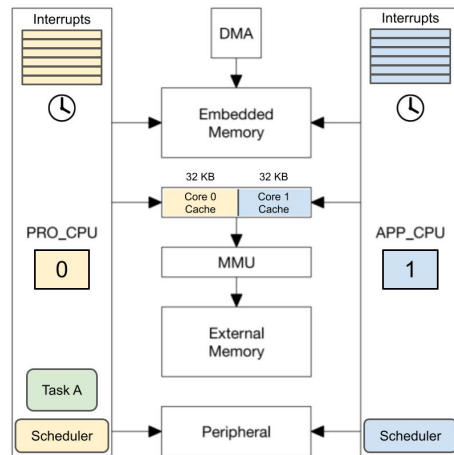


AMP is a programming paradigm that uses multiple cores or processors to run multiple tasks at the same time. It requires one core/processor to be the primary core running the operating system (OS). It sends jobs to the other cores, known as secondary cores. Note that these cores may or may not be the same architecture. In fact, you can set up an AMP configuration across separate computers.

SMP is also a paradigm that allows a multithreaded program to run on multiple cores. However, in SMP, each core runs a copy of the OS. The scheduler in each core operates independently, choosing to run tasks from a shared list. SMP requires the cores to be tightly coupled, often sharing RAM and other resources. As a result, you'll usually find that SMP is built on top of multiple cores of the same architecture.

ESP-IDF is a modification of FreeRTOS and is configured for SMP operation. The ESP32 uses an Xtensa LX6 processor, and most variants you come across will contain two cores.

ESP32 Block Diagram  
(example of SMP architecture)



**Figure 1: System Structure**  
(diagram is from the ESP32 Technical Reference Manual)

The ESP-IDF OS supports pinning tasks to cores, which means that you assign one of the cores to run a particular task. It also supports tasks having “no affinity,” which means the task can run on either core.

The cores in the ESP32 are labeled “Core 0” and “Core 1.” Core 0 is known as the “Protocol Core” or “PRO CPU.” In default ESP32 applications, protocol-related tasks, like WiFi and Bluetooth, are assigned to this core. Core 1 is known as the “Application Core” or “APP CPU,” which is responsible for running user applications.

Note that this is the default behavior. You can freely assign tasks to either core or choose “no affinity” for tasks. However, we recommend pinning tasks to a core for most microcontroller applications (as you will see shortly).

Each core maintains its own list of interrupts, timers, and if running ESP-IDF, scheduler. RAM is shared between cores, but a section of RAM is set aside for Core 0 cache, and another section is set aside for Core 1 cache.

Whenever the scheduler in either core is run, it looks at a shared list of tasks to see which task is in the ready state. It will choose whichever task is ready and has the highest priority. However, it will only run a task if the xCoreID field for that task is “no affinity” or matches the calling processor core number.

Because your particular dual-core microcontroller may be different from the ESP32, I encourage you to read the documentation for your processor. Here is a great article showing you how ESP-IDF works in the ESP32: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/freertos-smp.html>

For many general-purpose operating systems, it’s often easier to list a task as “no affinity” and let the scheduler choose which core it runs on. This has the benefit of being easier to use and will automatically optimize CPU utilization time among tasks while balancing the processing load.

However, many processors (including the ESP32) require that the interrupt service routine (ISR) runs in the core that sets up the interrupt. As a result, you may not know when a task with “no affinity” will be interrupted. Additionally, whenever a task moves to a different core, the new core will experience [cache misses](#) and will need to update its cache with data for the task.

Advantages of pinning a task to a core	Advantages of letting the scheduler(s) decide
<ul style="list-style-type: none"> <li>+ Knowing location of interrupts</li> <li>+ More control of core usage</li> <li>+ Fewer cache misses</li> <li>+ More deterministic</li> </ul>	<ul style="list-style-type: none"> <li>+ Easier to use</li> <li>+ Optimized core usage</li> <li>+ Automatic load balancing</li> </ul>

These issues make it more difficult to precisely predict the timing of tasks in a processor. If your application requires a high level of determinism, you likely will want to pin tasks to cores.

## 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

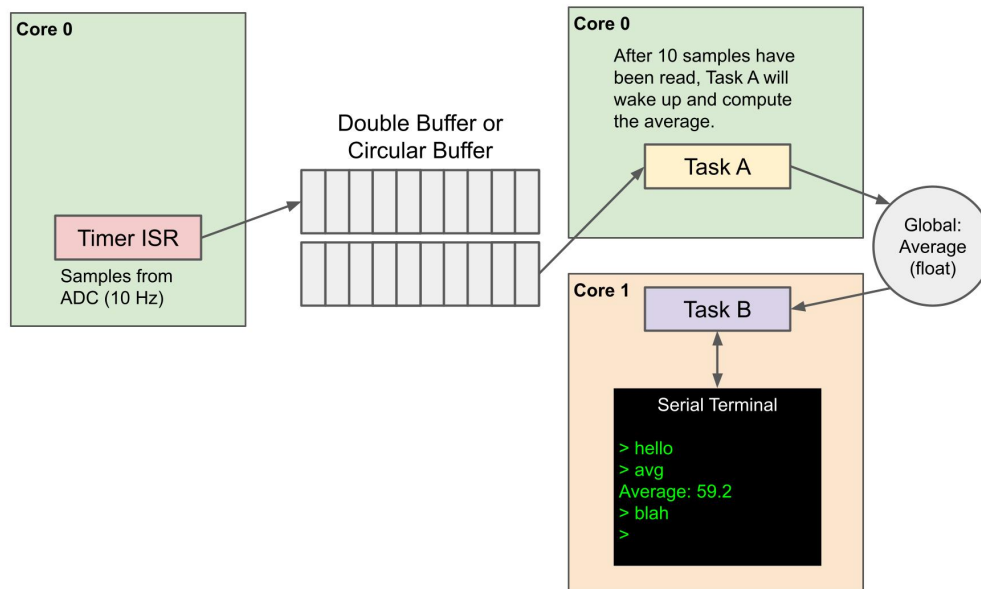
If you have not done so, please watch the following video, which provides the steps necessary to creating tasks and assigning priorities. It also demonstrates a working version of the challenge:

### Introduction to RTOS Part 12 - Multicore Systems...



## Challenge

Your challenge is to start with your [hardware interrupt solution](#) and make it run with dual-core support on the ESP32. Specifically, the timer ISR (for sampling the ADC) and the task that computes the average of 10 samples should run on Core 0. The user interface task (the one that handles echoing characters to the serial terminal and responding to the "avg" command) should run on Core 1.



## 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 on one core, handle CLI on the other.
 *
 * Note: Changes from original, single-core version are marked with a
 * "%%" in the comment.
 *
 * Date: March 3, 2021
 * Author: Shawn Hymel
 * License: 0BSD
 */

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

// Core definitions (assuming you have dual-core ESP32)
// %% We can use both cores now
static const BaseType_t pro_cpu = 0;
static const BaseType_t app_cpu = 1;

// 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 = 10; // 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;

    // Start a timer to run ISR every 100 ms
    // %% We move this here so it runs in core 0
    timer = timerBegin(0, timer_divider, true);
    timerAttachInterrupt(timer, &onTimer, true);
    timerAlarmWrite(timer, timer_max_count, true);
    timerAlarmEnable(timer);

    // 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 instructions, 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.
    // %% We set this task to run in Core 0
    xTaskCreatePinnedToCore(calcAverage,
                           "Calculate average",
                           1024,
                           NULL,
                           1,
                           &processing_task,
                           pro_cpu);
}

```

```

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

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

```

## Explanation

After all the time spent explaining the differences between AMP and SMP along with details about ESP-IDF, it turns out that running tasks in different cores is fairly trivial. Because memory is shared, you can use kernel objects (e.g. queues, semaphores, and mutexes) just like you would on a single-core system.

All changes from my previous hardware interrupt solution are noted with a “%%” string in the comments (so you can easily search for them).

First, we allow both cores to be used. PRO CPU is 0 and APP CPU is 1.

Copy Code

```

// Core definitions (assuming you have dual-core ESP32)
// %% We can use both cores now
static const BaseType_t pro_cpu = 0;
static const BaseType_t app_cpu = 1;

```

The tasks are then pinned to their respective cores.

Copy Code

```

// Start task to handle command line interface events. Let's set it at a
// higher priority but only run it once every 10 ms.
xTaskCreatePinnedToCore(doCLI,
    "Do CLI",
    1024,
    NULL,
    2,
    NULL,
    app_cpu);

// Start task to calculate average. Save handle for use with notifications.
// %% We set this task to run in Core 0
xTaskCreatePinnedToCore(calcAverage,
    "Calculate average",
    1024,
    NULL,
    1,
    &processing_task,
    pro_cpu);

```

Important! The setup() and loop() functions run in a task that is priority 1 pinned to core 1. Previously, we started the hardware timer and assigned the ISR inside setup(), which means that the ISR would run in Core 1. Since we want to run the ISR in Core 0, we must move the initialization functions to a task that runs in Core 0. calcAverage() runs in Core 0 now, so we moved the timer initialization functions there.

Copy Code

```

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

    Message msg;
    float avg;

    // Start a timer to run ISR every 100 ms
    // %% We move this here so it runs in core 0
    timer = timerBegin(0, timer_divider, true);
    timerAttachInterrupt(timer, &onTimer, true);
}

```



```
timerAlarmWrite(timer, timer_max_count, true);
timerAlarmEnable(timer);

// ...rest of calcAverage...

}
```

## 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:

- ESP-IDF FreeRTOS SMP Changes: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/freertos-smp.html>
- ESP32 Interrupt Allocation (specifically, see the “Multicore Issues” section): [https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/intr\\_alloc.html](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/intr_alloc.html)
- Adding simple multicore support to FreeRTOS: <https://www.freertos.org/2020/02/simple-multicore-core-to-core-communication-using-freertos-message-buffers.html>
- [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 9 \(Hardware Interrupts\)](#)
- [Introduction to RTOS - Solution to Part 10 \(Deadlock and Starvation\)](#)
- [Introduction to RTOS - Solution to Part 11 \(Priority Inversion\)](#)

## 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](#)



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

### Tags

Arduino

RTOS

### License

Attribution

## Get Involved

Like

Save



1-800-344-4539

218-681-6674



[sales@digikey.com](mailto:sales@digikey.com)



218-681-3380



United States | Copyright © 1995-2023, DigiKey. | All Rights Reserved.

Local Support: 701 Brooks Avenue South, Thief River Falls, MN 56701 USA

Do Not Sell / Do Not Share My Personal Information