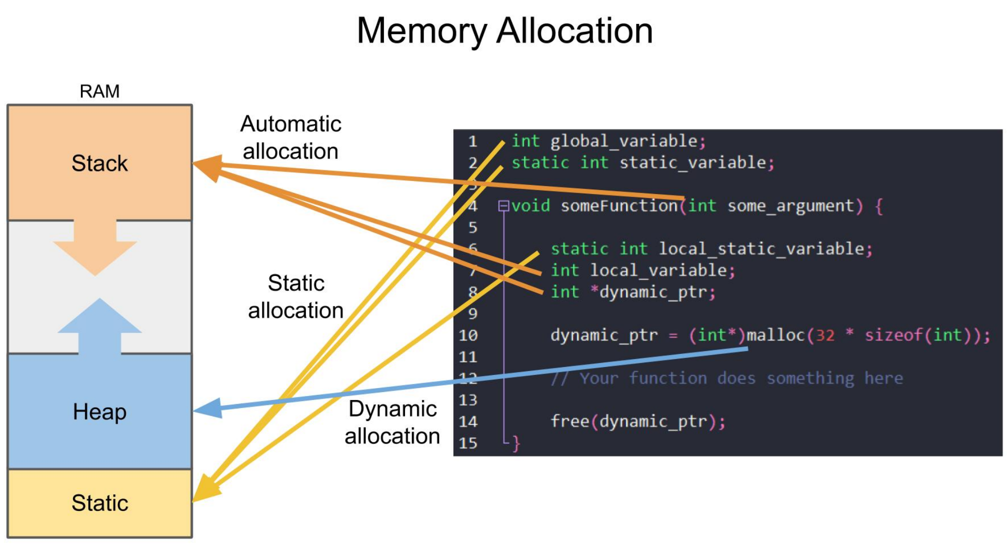
Introduction to RTOS - Solution to Part 4 (Memory Management)

By ShawnHymel

In lower-level languages, like C and C , we are often tasked with keeping track of how we use the memory in our system, both working memory (e.g. RAM) and non-volatile memory (e.g. flash). A real-time operating system (RTOS) adds another level of complexity on top of this.

Concepts

Volatile memory (e.g. RAM) in most microcontroller systems is divided up into 3 sections: static, stack, and heap.



```
int global_variable;
static int static_variable;

void someFunction(int some_argument) {

    static int local_static_variable;
    int local_variable;
    int *dynamic_ptr;

    dynamic_ptr = (int*)malloc(32 * sizeof(int));

    // Your function does something here

    free(dynamic_ptr);
}
```

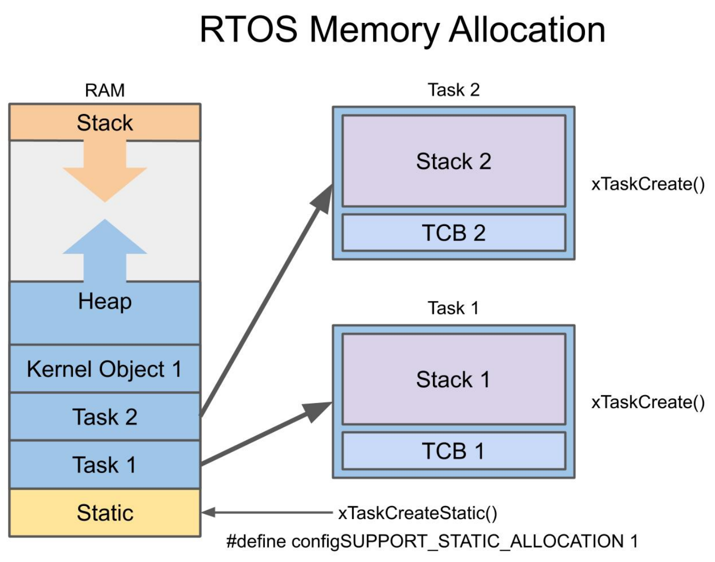
Static memory is used for storing global variables and variables designated as “static” in code (they persist between function calls).

The stack is used for automatic allocation of local variables. Stack memory is organized as a last-in-first-out (LIFO) system so that the variables of one function can be “pushed” to the stack when a new function is called. Upon returning to the first function, that functions’ variables can be “popped” off, which the function can use to continue running where it left off.

Heap must be allocated explicitly by the programmer. Most often in C, you will use the malloc() function to allocate heap for your variables, buffers, etc. This is known as “dynamic allocation.” Note that in languages without a garbage collection system (e.g. C and C), you must deallocate heap memory when it’s no longer used. Failing to do so will result in a memory leak, and could cause undefined effects, such as corrupting other parts of memory.

In most systems, the stack and heap will grow toward each other, taking up unallocated memory where needed. If left unchecked, they could collide and begin overwriting each other’s data.

When you create a task in FreeRTOS (e.g. with xTaskCreate()), the operating system will allocate a section of heap memory for the task.



One part of that allocated memory is the Task Control Block (TCB), which is used to store information about the task, such as its priority and local stack pointer. The other section is reserved as a local stack that operates just like the global stack (but on a smaller scale for just that task).

Local variables created during function calls within a task are pushed to the task's local stack. Because of this, it's important to calculate the predicted stack usage of a task ahead of time and include that as the stack size parameter in xTaskCreate().

When using FreeRTOS, malloc() and free() are not considered thread safe. As a result, it's recommended that you use pvPortMalloc() and vPortFree() instead. When using these, memory will be allocated from the system's global heap (instead of the heap allocated for the task).

Recent versions of FreeRTOS allow for the creation of static tasks. These use only static memory (allocating their own local stack and TCB in static memory instad of the heap). This is useful for situations where you cannot or do not want to use heap memory to prevent heap overflows.

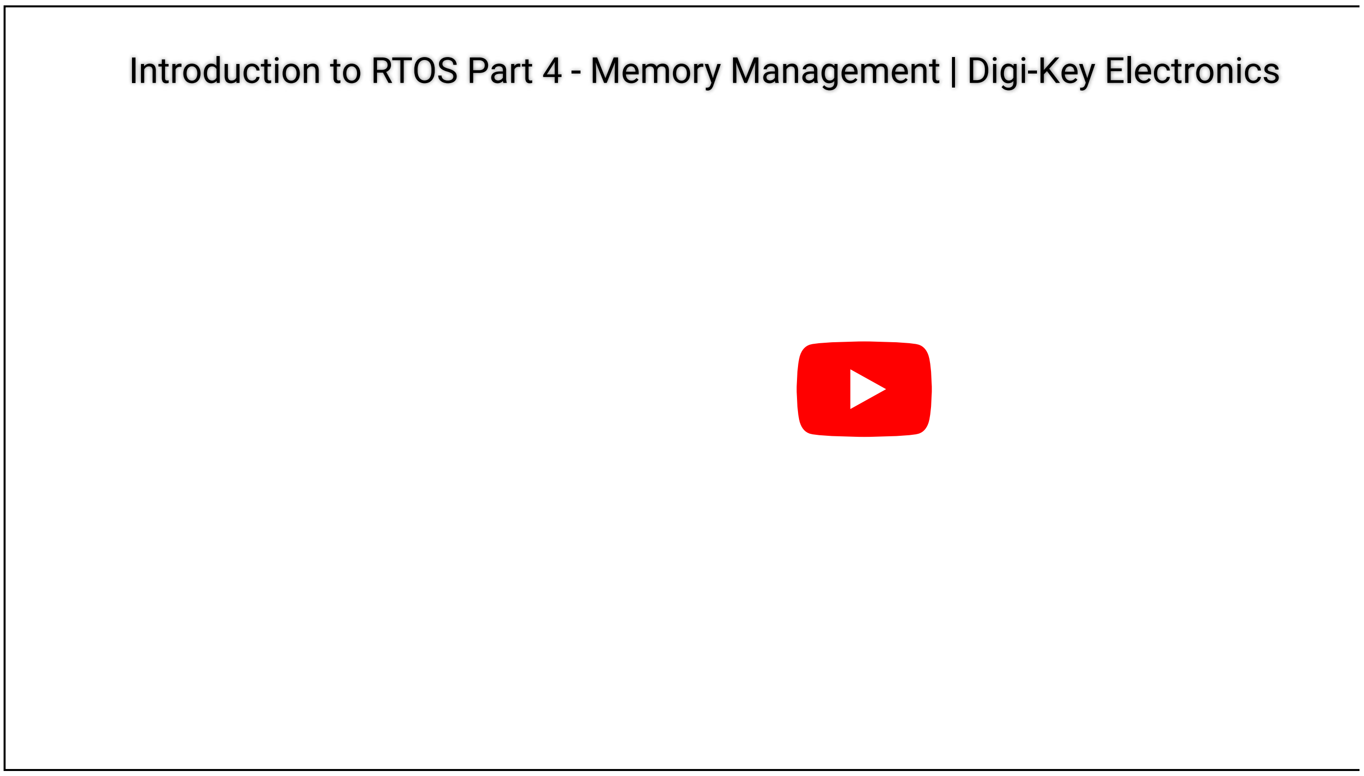
Required Hardware

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

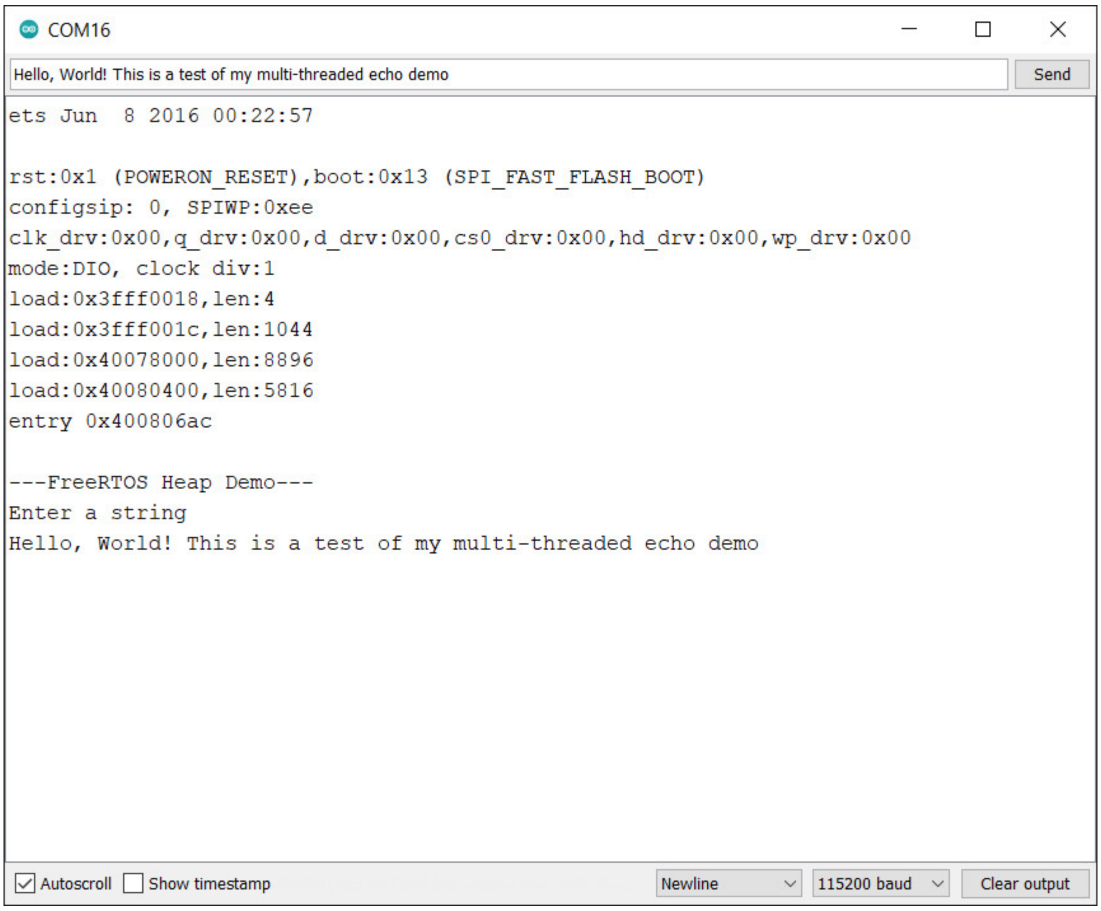
If you have not done so, please watch the following video, which provides the steps necessary to creating tasks and assigning priorities. It also demonstrates a working version of the challenge:


Introduction to RTOS Part 4 - Memory Management | Digi-Key Electronics

Challenge

Using FreeRTOS, create two separate tasks. One listens for input over UART (from the Serial Monitor). Upon receiving a newline character (‘\n’), the task allocates a new section of heap memory (using pvPortMalloc()) and stores the string up to the newline character in that section of heap. It then notifies the second task that a message is ready.

The second task waits for notification from the first task. When it receives that notification, it prints the message in heap memory to the Serial Monitor. Finally, it deletes the allocated heap memory (using vPortFree()).



```
Hello, World! This is a test of my multi-threaded echo demo
```

```
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1044
load:0x40078000,len:8896
load:0x40080400,len:5816
entry 0x400806ac

---FreeRTOS Heap Demo---
Enter a string
Hello, World! This is a test of my multi-threaded echo demo
```

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

```
/**
 * FreeRTOS Heap Demo
 *
 * One task reads from Serial, constructs a message buffer, and the second
 * prints the message to the console.
 *
 * Date: December 5, 2020
 * Author: Shawn Hymel
 * License: 0BSD
 */

// 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 uint8_t buf_len = 255;

// Globals
static char *msg_ptr = NULL;
static volatile uint8_t msg_flag = 0;

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

// Task: read message from Serial buffer
void readSerial(void *parameters) {

  char c;
  char buf[buf_len];
  uint8_t idx = 0;

  // Clear whole buffer
  memset(buf, 0, buf_len);

  // Loop forever
  while (1) {

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

      // Store received character to buffer if not over buffer limit
      if (idx < buf_len - 1) {
        buf[idx] = c;
        idx  ;
      }

      // Create a message buffer for print task
      if (c == '\n') {

        // The last character in the string is '\n', so we need to replace
        // it with '\0' to make it null-terminated
        buf[idx - 1] = '\0';

        // Try to allocate memory and copy over message. If message buffer is
        // still in use, ignore the entire message.
        if (msg_flag == 0) {
          msg_ptr = (char *)pvPortMalloc(idx * sizeof(char));

          // If malloc returns 0 (out of memory), throw an error and reset
          configASSERT(msg_ptr);

          // Copy message
          memcpy(msg_ptr, buf, idx);

          // Notify other task that message is ready
          msg_flag = 1;
        }

        // Reset receive buffer and index counter
        memset(buf, 0, buf_len);
        idx = 0;
      }
    }
  }
}

// Task: print message whenever flag is set and free buffer
void printMessage(void *parameters) {
  while (1) {

    // Wait for flag to be set and print message
    if (msg_flag == 1) {
      Serial.println(msg_ptr);

      // Give amount of free heap memory (uncomment if you'd like to see it)
//      Serial.print("Free heap (bytes): ");
//      Serial.println(xPortGetFreeHeapSize());

      // Free buffer, set pointer to null, and clear flag
      vPortFree(msg_ptr);
      msg_ptr = NULL;
      msg_flag = 0;
    }
  }
}

//*****************************************************************************
// 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 Heap Demo---");
    Serial.println("Enter a string");

    // Start Serial receive task
    xTaskCreatePinnedToCore(readSerial,
                            "Read Serial",
                            1024,
                            NULL,
                            1,
                            NULL,
                            app_cpu);

    // Start Serial print task
    xTaskCreatePinnedToCore(printMessage,
                            "Print Message",
                            1024,
                            NULL,
                            1,
                            NULL,
                            app_cpu);

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

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

Explanation

In the first task, we wait for a newline character to appear from the Serial Monitor. When we see it, we replace it with the null character ('\0') to make sure that the string is null-terminated. We then use pvPortMalloc() to allocate a section of heap the exact size of the string, and we copy the string (using memcpy) to that section of memory.

The configASSERT() call is a macro in FreeRTOS that allows us to quickly check if something is 0 (NULL) or not. If something is NULL, it will throw an error (in this case, halting the processor). This has the same effect as using an if statement to see if pvPortMalloc() returned NULL (out of heap memory) and running some code to print a message or restart. See here for more information about configASSERT().

Note that while it is possible to have the second task wait for our global pointer to that heap (msg_ptr) to be non-NULL, this method is not thread-safe! As soon as the pvPortMalloc() returns, msg_ptr will be set to a non-NULL value (assuming pvPortMalloc() did not return NULL, indicating that we are out of heap). If the scheduler were to stop the first thread and start running the second thread at this point, there would be no message inside the heap memory! It would be random bytes, all 0s, or perhaps even an old message!

To make this thread-safe, we need to use another global variable as a flag (msg_flag). This flag only gets set to 1 after we are done copying the message over to the heap buffer. We also check to make sure this flag is 0 before allocating new heap in the first task.

In the second task, we wait for our message flag (msg_flag) to be 1 and then print out the message in heap when it is. We then immediately free the heap memory, set it to NULL, and reset the message flag.

In a future lecture, we'll go over kernel objects, like queues, mutexes, and semaphores, to make sharing resources between tasks easier.

Recommended Reading

Example code from the video and the solution can also be found in the following repository: https://github.com/ShawnHymel/introduction-to-rtos.

If you'd like to dig deeper into RTOS memory management, I recommend checking out these excellent articles:

- Memory in C - Stack, Heap, and Static: https://craftofcoding.wordpress.com/2015/12/07/memory-in-c-the-stack-the-heap-and-static/
- The C Build Process: https://blog.feabhas.com/2012/06/the-c-build-process/
- What and where are the stack and heap? https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap/13326916#13326916
- FreeRTOS Memory Management: https://www.freertos.org/a00111.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 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)
- 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

Tags

Arduino

RTOS

License

Attribution

Get Involved

Like

Save