Using FreeRTOS with the Raspberry Pi Pico

By **Daniel Gross**
SENIOR DEVELOPER ADVOCATE
AMAZON WEB SERVICES

October 19, 2022

BLOG

In this blog series, we will explore writing embedded applications for the Raspberry Pi Pico using FreeRTOS. We will highlight some of the key features of FreeRTOS with sample code. We will also dive into the multicore capabilities of the Raspberry Pi Pico, which we can take advantage of with FreeRTOS. This is the first blog in a series in which we will discuss various aspects of FreeRTOS.

First off, what is FreeRTOS? As the name suggests, FreeRTOS is a real-time operating system (RTOS) that is free to use. It is entirely open source with a permissive MIT license, and is intended for embedded applications that run on microcontrollers such as the one on the Raspberry Pi Pico, and small microprocessors. FreeRTOS has been available for over 18 years with a strong community of partners and embedded developers utilizing it for their products and services.

The Raspberry Pi Pico is a small development board built around the RP2040 microcontroller, which was designed by Raspberry Pi. The RP2040 is a dual-core Arm Cortex-M0+ processor running up to 133 MHz with 264kB on-chip SRAM. The Pico board has 2MB flash with 26 multifunctional GPIO pins. These hardware specs give us a range of capabilities for creating embedded applications with FreeRTOS, including the dual-core processor. Raspberry Pi also provides a C/C++ SDK for the Pico, which we will be using later on.

So, why do we need an RTOS such as FreeRTOS for a microcontroller-based board like the Raspberry Pi Pico? The short answer is, technically, you do not need an RTOS in all cases. Some embedded applications can be really simple such as reading values from a sensor and sending those values to an output. An RTOS is not required in this case and you can find examples illustrating that. However, as microcontrollers have become more capable devices and embedded applications have become increasingly more complex to take advantage of the hardware, the benefits of using an RTOS have grown.

Multitasking is one of the important reasons for using an RTOS, and is a key feature of FreeRTOS. What is multitasking? It is a way to run multiple tasks in your embedded application so that those tasks can execute independently of one another while still sharing the same processor. Task execution is managed by what is known as a scheduler. In the case of a single-core microcontroller, it is the job of the scheduler to give tasks execution time based on specified priorities and other factors. This can give the impression that multiple tasks are executing simultaneously even though the scheduler is actually managing execution by switching between tasks in quick succession at the millisecond level.

With the sample code shown later in this blog, we demonstrate how to create and run tasks with FreeRTOS on the Raspberry Pi Pico. First, however, we will cover how to setup a development environment.

There are different ways to setup a development environment for FreeRTOS and the Raspberry Pi Pico. This particular setup is agnostic to a specific editor or IDE, and it aims to be simple and flexible. The essential tools needed are: Git, CMake, Make, and the GNU Arm Embedded Toolchain. Below are installation instructions for Windows, Linux, and MacOS.

Windows

- Install Git (includes Bash terminal): https://git-scm.com/downloads
- Install CMake: https://cmake.org/download/
- Install the Arm GNU Toolchain: https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain
- Install Make using Chocolatey
    - Chocolatey should already be installed from the Git install, otherwise install it: https://chocolatey.org/
    - Run this from Git Bash: $ choco install make

Linux

All the tools we need, including Git and Make, should be available after installing these packages:

```
$ sudo apt install cmake

$ sudo apt install gcc-arm-none-eabi

$ sudo apt install build-essential
```

MacOS

For MacOS, ensure that Homebrew is installed and then follow these steps in the Terminal:

```
$ brew install cmake

$ brew tap ArmMbed/homebrew-formulae

$ brew install arm-none-eabi-gcc
```

Once you have completed the OS-specific setup, launch your terminal (Git Bash on Windows) and complete the steps below to setup the Blink project from the command line:

```
$ mkdir freertos-pico

$ cd freertos-pico

$ git clone https://github.com/RaspberryPi/pico-sdk --recurse-submodules

$ git clone -b smp https://github.com/FreeRTOS/FreeRTOS-Kernel --recurse-submodules

$ export PICO_SDK_PATH=$PWD/pico-sdk

$ export FREERTOS_KERNEL_PATH=$PWD/FreeRTOS-Kernel

$ mkdir blink
$ cd blink
```

Now we are ready for the code. Below is a simple FreeRTOS application written in C that you can run on the Raspberry Pi Pico. It creates a task to blink the LED on the board. Create a file in the ‘blink’ directory and name it ‘main.c’ using any code editor you wish. Notepad++, vi, Sublime Text, Atom, and VSCode are some examples.

```
#include "pico/stdlib.h"

#include "FreeRTOS.h"

#include "task.h"

void vBlinkTask() {

  for (;;) {

    gpio_put(PICO_DEFAULT_LED_PIN, 1);

    vTaskDelay(250);

    gpio_put(PICO_DEFAULT_LED_PIN, 0);

    vTaskDelay(250);

  }

}

void main() {

  gpio_init(PICO_DEFAULT_LED_PIN);

  gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);

  xTaskCreate(vBlinkTask, "Blink Task", 128, NULL, 1, NULL);

  vTaskStartScheduler();

}
```

Once we have the C file in place, we need to build the project. We will use CMake to build our project, which requires a ‘CMakeLists.txt’ file. Copy this pre-made ‘CMakeLists.txt’ file into the blink directory.

In addition to the ‘main.c’ and ‘CMakeLists.txt’ files, we also need a ‘FreeRTOSConfig.h’ file to configure FreeRTOS for our project. Copy this pre-made ‘FreeRTOSConfig.h’ file into the ‘blink’ directory. Next, build the project by following the steps below from the ‘blink’ directory:

```
$ mkdir build

$ cd build
```

For Windows only:

```
$ cmake -G "MinGW Makefiles" ..
```

For Linux or MacOS:

```
$ cmake ..
```

Finally, from any environment:

```
$ make
```

Once the project successfully builds, there should now be a 'blink.uf2' in the 'build' directory. This file is the binary we will flash to the Pico. In order to flash this file, first hold down the BOOTSEL button on the Pico board while plugging it in to the USB interface. This will mount the Pico as a drive. Then copy the 'blink.uf2' file to the drive location and the Pico will automatically reboot and run the application. For example, if your drive location is D:, here is how to copy from the command line:

```
$ cp blink.uf2 /d/
```

Now that you have flashed the application, the LED on the Raspberry Pi Pico should be blinking. Congratulations, you are now executing a task with FreeRTOS!

For the next blog in this series, we will describe how and when the scheduler decides which task to run, then extend the Blink demo to demonstrate FreeRTOS features like Queues and Stream Buffers while exploring event-driven designs.

Click to access the complete CMakeLists.txt and FreeRTOSConfig.h files.

SUBSCRIBE

Daniel Gross is a Senior Developer Advocate at Amazon Web Services (AWS). Focused on FreeRTOS and embedded devices interacting with cloud services, Daniel is passionate about delivering a great developer experience for IoT builders.

More from Daniel

Categories

OPEN SOURCE - LINUX, FREERTOS & RELATED

Comment (1)

Login

Sort by: **Date** Rating Last Activity

oscar · *8 weeks ago* 0

Hi Daniel, thank you very much for the tutorial it is very interesting. I am facing a problem, I am using a raspberry Pi B in order to compile the code but I am getting the next error when executing the "make" command:
cc: error: nosys.specs no such file or directory

Could you please advice?

Reply

Post a new comment

Enter text right here!

Comment as a Guest, or login: intensedebate WordPress.com

Name	Email	Website (optional)
Displayed next to your comments.	*Not displayed publicly.*	*If you have a website, link to it here.*

Subscribe to None

Submit Comment

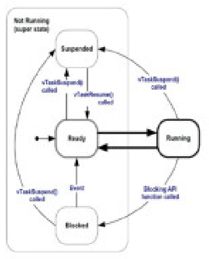
Using FreeRTOS with the Raspberry Pi Pico: Part 2

By Daniel Gross
SENIOR DEVELOPER ADVOCATE
AMAZON WEB SERVICES

October 31, 2022

BLOG

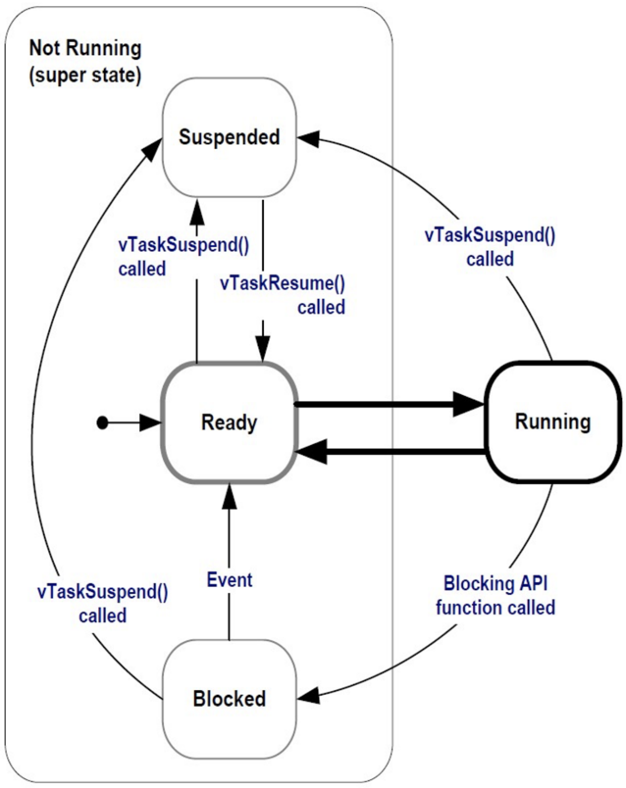
This is the second blog in the series where we explore writing embedded applications for the Raspberry Pi Pico using FreeRTOS.



In this blog, we will cover how and when the FreeRTOS Scheduler decides which task to run. We will also build on our code example from the first blog to demonstrate Queues and Message Buffers, which are features of FreeRTOS. Finally, we will briefly touch on event-driven design. So, get your compilers ready, here we go.

In the first blog, we created a task with xTaskCreate() and started the Scheduler with vTaskStartScheduler(). Then, the task function we defined called vBlinkTask() was executed. This is a simple example of the Scheduler managing a single task. But, what happens when we create multiple tasks to run in our application? After all, that is the main point of multitasking. In the next section, we will dive deeper into the fundamentals of scheduling and tasks.

The Scheduler in FreeRTOS implements a scheduling algorithm to determine what task should run at what time. For scheduling, time is determined by what is known as a “tick.” The time between ticks depends on the hardware clock speed and configuration, but is generally between 1-10 milliseconds based on the timing needs of the application. The time between ticks is known as a “time slice.” In its default configuration, FreeRTOS uses a scheduling policy where it will switch between tasks that are the same priority in round-robin order on each tick, and task execution happens within a time slice. The priority of a task is set at creation time, which we saw in the code example from the first blog as an argument passed to xTaskCreate() with a value of 1. However, this can be changed at runtime with vTaskPrioritySet(). Also, the scheduling policy is “preemptive,” meaning that the Scheduler will always run the highest priority task that is available to run. So, if a higher priority task becomes available to run while a lower priority task is already running, the Scheduler will start the higher priority task, causing the lower priority task to be immediately preempted. The following diagram illustrates the full task state machine, including the task states and transitions. When a task is executing, it is in the “running” state. When a task is not executing, it is in the “suspended,” “blocked,” or “ready” state.



Given the mechanics above around scheduling, there are a couple of points worth noting. First, because the Scheduler uses a preemptive scheduling policy, a high priority task that is never “blocked” or “suspended” can “starve” any lower priority tasks from ever executing. Task starvation can be avoided with techniques like event-driven design, which we will discuss later. Second, the scheduling policy described above holds true for a single core system. However, multiple cores add another dimension to how the Scheduler may behave, because with multiple cores tasks are able to run not only serially but also concurrently. This is especially relevant for the Raspberry Pi Pico as it has a dual core M0+ processor, and FreeRTOS can take advantage of this multicore capability with symmetric multiprocessing (SMP). SMP will be covered in more detail later in this blog series, but keep this in mind when considering Scheduler behavior.

Now that we understand how tasks are managed by the Scheduler, let’s look at how tasks can communicate with each other. This concept is known as inter-task communication. To accomplish this, we can use other built-in features of FreeRTOS such as Queues, Stream Buffers, and Message Buffers. Below is a code example using a Queue. In this example, one task determines how many LED blinks to perform while the other task executes the number of blinks based on the value from the queue. Rename the existing “main.c” file from the example in the first blog and create a new “main.c” file, adding the block of code below. Then, follow the instructions from the first blog to build and flash the application to the Raspberry Pi Pico.

```
#include "pico/stdlib.h"

#include "FreeRTOS.h"

#include "task.h"

#include "queue.h"



QueueHandle_t blinkQueue;
```

```
void vBlinkReceiverTask() {

    for (;;) {

    int blinks = 0;

        if (xQueueReceive(blinkQueue, &blinks, 0) == pdPASS) {

            for (int i = 0; i < blinks; i++) {

                gpio_put(PICO_DEFAULT_LED_PIN, 1);

                vTaskDelay(200);

                gpio_put(PICO_DEFAULT_LED_PIN, 0);

                vTaskDelay(200);

            }

        }

    }

}



void vBlinkSenderTask() {

    int loops = 4;

    for (;;) {

        for (int i = 1; i <= loops; i++) {

            xQueueSend(blinkQueue, &i, 0);

            vTaskDelay(500 + (i * 500));

        }

    }

}



void main() {

    gpio_init(PICO_DEFAULT_LED_PIN);

    gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);

    blinkQueue = xQueueCreate(1, sizeof(int));

    xTaskCreate(vBlinkSenderTask, "Blink Sender", 128, NULL, 1, NULL);

    xTaskCreate(vBlinkReceiverTask, "Blink Receiver", 128, NULL, 1, NULL);

    vTaskStartScheduler();

}
```

Running the example above, you should see a blink sequence on the LED of 1, 2, 3, and 4 short blinks which then repeats. The Queue, "blinkQueue," is dynamically allocated to hold integer values. Queues can be dynamically allocated from the heap, or statically allocated at compile time. Queues in FreeRTOS use the copy method, so values sent to the queue are copied byte for byte. That enables queues to pass data across memory boundaries, and pointers to data to be queued when the data is large. Beyond our simple example, there is a rich API for Queue Management that you can explore and experiment with further.

As mentioned earlier, other options available with FreeRTOS for communicating between tasks include using a Stream Buffer or a Message Buffer. A Message Buffer is a type of Stream Buffer and below is a simple example of Message Buffer usage. For this example, the "CMakeLists.txt" file in our project folder must be modified to enable output which can be read from the USB serial port. Add the following lines to the end of your "CMakeLists.txt" file:

```
# enable usb output, disable uart output

pico_enable_stdio_usb(blink 1)

pico_enable_stdio_uart(blink 0)
```

Once the "CMakeLists.txt" file has been modified, rename the existing "main.c" file again and create a new "main.c" file with the following contents:

```
#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include "pico/stdlib.h"

#include "FreeRTOS.h"

#include "task.h"

#include "message_buffer.h"



const size_t BUFFER_SIZE = 32;



void vReceiverTask(void *pvParameters) {

  MessageBufferHandle_t buffer = (MessageBufferHandle_t) pvParameters;

  size_t messageSize = BUFFER_SIZE - 4;

  char *message = malloc(messageSize);

  memset(message, '\0', messageSize);

  size_t lengthReceived;

  for (;;) {

    lengthReceived = xMessageBufferReceive(buffer, (void *)message, BUFFER_SIZE, 0);

    if (lengthReceived > 0) {

      printf("length: %d, message: %s\n", lengthReceived, message);
```

```
            memset(message, '\0', messageSize);

        }

    }

}



void vSenderTask(void *pvParameters) {

    MessageBufferHandle_t buffer = (MessageBufferHandle_t) pvParameters;

    char message[] = "FreeRTOS + Pi Pico";

    for (;;) {

        xMessageBufferSend(buffer, (void *)message, strlen(message), 0);

        vTaskDelay(1000);

    }

}



void main() {

    stdio_init_all();

    busy_wait_ms(1000);

    MessageBufferHandle_t buffer = xMessageBufferCreate(BUFFER_SIZE);

    xTaskCreate(vSenderTask, "Sender", 128, (void *)buffer, 1, NULL);

    xTaskCreate(vReceiverTask, "Receiver", 128, (void *)buffer, 1, NULL);

    vTaskStartScheduler();

}
```

Follow the build and flash procedure used from the first blog and the example earlier to run the code. Then, launch an application that can read output from the Raspberry Pi Pico over the USB serial port. There are a number of ways to do this depending on your environment. For Windows, tools like PuTTY or Tera Term are good examples. On MacOS, "screen" will do nicely. On Linux, you can use "minicom" or even the pySerial "miniterm" tool. In all cases you will need to find the port that USB serial is connecting to on your computer and configure your tool with that port to display the output.

You should now see the following output if everything is working correctly: "length: 18, message: FreeRTOS + Pi Pico". You might notice the allocated message size was created 4 bytes less than the allocated buffer size in the code. This is because the Message Buffer uses 4 bytes to store the message length. Remember to account for this when using Message Buffer. Like Queues, Stream Buffers and Message Buffers have extensive APIs that you can explore and experiment with further.

Congratulations, you have now used two different methods for inter-task communication with FreeRTOS on the Raspberry Pi Pico. You may have noticed something similar about both methods demonstrated. In each code example, the receiver task sits in an infinite loop while checking to see if data was received. If you recall the point made earlier in this blog about the preemptive scheduling policy and task starvation, you might recognize that this existing design could lead to a problematic situation if we introduce other tasks with different priorities or change the priority of our sender or receiver task. Furthermore, resources like processor cycles could be better optimized. Taking these points into consideration, we should consider moving toward an event-driven design for our application so that tasks only run when key events occur. The next blog in this series will cover precisely that and more.

SUBSCRIBE

Daniel Gross is a Senior Developer Advocate at Amazon Web Services (AWS). Focused on FreeRTOS and embedded devices interacting with cloud services, Daniel is passionate about delivering a great developer experience for IoT builders.

More from Daniel

Categories

OPEN SOURCE - LINUX, FREERTOS & RELATED

Comment (1)

Login

Sort by: **Date** Rating Last Activity

keshav · *4 weeks ago* 0

This was very informative, thank you for contributing to the discussion on embedded applications for Raspberry Pi. Using this we can build many projects.

Reply

Post a new comment

Enter text right here!

Comment as a Guest, or login: intensedebate WordPress.com

Name	Email	Website (optional)
Displayed next to your comments.	*Not displayed publicly.*	*If you have a website, link to it here.*

Subscribe to None

Submit Comment

Embedded
COMPUTING DESIGN

Using FreeRTOS with the Raspberry Pi Pico: Part 3

By Daniel Gross
SENIOR DEVELOPER ADVOCATE
AMAZON WEB SERVICES

November 14, 2022

BLOG

This is the third blog in the series where we explore writing embedded applications for the Raspberry Pi Pico using FreeRTOS.

As mentioned in the previous blog, this blog will cover event-driven design and its benefits. We will also look at Semaphores, another built-in feature of FreeRTOS. To demonstrate both of these concepts, we have a concise code example to share that you can run on your own Raspberry Pi Pico hardware.

First, what do we mean by "event" when we say "event-driven design" for embedded applications? Generally, an event represents an action that takes place at a particular point in time that can then be captured and acted upon programmatically. An event could be something like input from a button press or data arriving from a connected peripheral. Events happen asynchronously to the running application, and are usually triggered by what are known as interrupts on embedded systems. An interrupt is a signal produced by an embedded system that indicates a change in state. The specific source of an interrupt can be identified by an Interrupt Request or IRQ. The RP2040 microcontroller on the Raspberry Pi Pico has 26 IRQs built-in with a range of sources. See the RP2040 Datasheet (Section 2.3.2) for more details on the specific IRQs supported.

Knowing that events are based on interrupts in event-driven design, how do we actually utilize this in an embedded application? Interrupts can happen at any time, so they will not always align perfectly with the timing of a running application. To properly capture an interrupt, we usually have to implement an interrupt handler. This is also known as an Interrupt Service Routine or ISR. From a coding perspective, this can be represented with a callback function. A callback function is called from outside the application by the system. The application will define a callback function and pass the name of that function to an API so that the system can call that function when an interrupt occurs. With the Raspberry Pi Pico C/C++ SDK, two such examples are: gpio_set_irq_callback (gpio_irq_callback_t callback) and irq_set_exclusive_handler (uint num, irq_handler_t handler). We will use yet another API call from this SDK in our example code later.

Now that we understand how event-driven design works, why is this approach preferable? For one, an event-driven design does not waste execution cycles by continuously looping. With FreeRTOS, tasks can be stopped from executing to wait for an event, then re-started when an event occurs. This is more efficient for an embedded system in terms of processor usage because tasks that are not executing do not consume processor cycles. Another reason is that we can reduce the amount of code we write and maintain to constantly poll for state changes that may have occurred. Next, we can eliminate certain types of sleeps, delays, and other less elegant timing techniques used as an attempt to align to when we think events might occur in the application. This allows the system to respond to events more quickly since tasks can automatically restart when an event occurs. Finally, with event-driven design, we can simplify our task priorities and timings with FreeRTOS to avoid potential task starvation or race conditions.

Before we demonstrate a simple event-driven design with FreeRTOS and the Raspberry Pi Pico, we must first take a short detour to cover an important topic related to multitasking. When we have multiple tasks executing in quick succession, what happens when we want to share a particular resource that only one task can access at a time? In order to avoid possible collisions between tasks, another built-in feature of FreeRTOS available to us is the Semaphore API. Semaphores allow a developer to manage serial access to a shared resource within an application. There are 3 main types of Semaphores in FreeRTOS – binary, counting, and mutex. We will use a mutex in the example below, which is shorthand for "mutual exclusion." Once a mutex is defined, a task can "take" it to indicate that a shared resource is being used, then "give" it back to indicate the resource is no longer being used. If a mutex is already taken by one task, another task must wait until the mutex is given back before it can be taken again. We use a mutex in the example below to enable multiple tasks to print to STDOUT, which is a shared resource that only one task can access at a time.

As mentioned previously, there are different types of events that can be captured in an embedded application. An embedded system may have sensors or actuators attached to GPIO pins, there may be communication or networking attached to a UART, or there could be IRQs triggered from a programmable I/O (PIO) state machine. Each of these will have different implementations, but can all benefit from event-driven design. One other type of interrupt that can be captured is a hardware timer. There are 4 hardware timer slots on the Raspberry Pi Pico according to the datasheet referenced above, and we will use hardware timers to demonstrate event-driven design in the upcoming example.

In the example below, direct-to-task notification as a lightweight mailbox is used to communicate between an ISR and a task. The application creates and runs 3 tasks, all defined by the same vNotifyTask() function. Each task begins by getting its task number and handle, then creating a hardware timer with add_repeating_timer_ms(). A callback function defined in the application, vTimerCallback(), is passed to the timer, which will be called when the interrupt occurs. Also, the handle of the task is passed to the timer as data so that the callback can identify the task to notify. Each task then enters its 'for' loop and calls xTaskNotifyWait(), which causes each task to wait until a notification is received. The task is no longer executing at this point, so no processor cycles are consumed. The timer interrupt will trigger execution of the callback function, which then gets the current time in milliseconds since boot and notifies the task by calling xTaskNotifyFromISR(). At that point, the vNotifyTask() calls vPrintTime() to format the output and print the value using the vSafePrint() function, which utilizes a mutex. Then, the task waits for the next notification from the ISR.

```
#include <stdio.h>

#include "pico/stdlib.h"

#include "FreeRTOS.h"

#include "task.h"

#include "semphr.h"



SemaphoreHandle_t mutex;
```

```
void vSafePrint(char *out) {
    xSemaphoreTake(mutex, portMAX_DELAY);
    puts(out);
    xSemaphoreGive(mutex);
}

void vPrintTime(int task, uint32_t time) {
    char out[32];
    sprintf(out, "Task: %d, Timestamp: %d", task, time);
    vSafePrint(out);
}

bool vTimerCallback(struct repeating_timer *timer) {
    uint32_t time = time_us_32();
    TaskHandle_t handle = timer->user_data;
    xTaskNotifyFromISR(handle, time, eSetValueWithOverwrite, NULL);
    return true;
}

void vNotifyTask(void *pvParameters) {
    int task = (int)pvParameters;
    TaskHandle_t handle = xTaskGetCurrentTaskHandle();
    struct repeating_timer timer;
    add_repeating_timer_ms(1000, vTimerCallback, (void *)handle, &timer);
    uint32_t time;
    for (;;) {
        xTaskNotifyWait(0, 0, &time, portMAX_DELAY);
        vPrintTime(task, time);
    }
}
```

```
void main() {

    stdio_init_all();

    mutex = xSemaphoreCreateMutex();

    xTaskCreate(vNotifyTask, "Notify Task 1", 256, (void *)1, 1, NULL);

    xTaskCreate(vNotifyTask, "Notify Task 2", 256, (void *)2, 1, NULL);

    xTaskCreate(vNotifyTask, "Notify Task 3", 256, (void *)3, 1, NULL);

    vTaskStartScheduler();

}
```

To run this application, you can reuse the same project from the second blog in this series. Simply rename the existing 'main.c' file and create a new 'main.c' file inserting the code above. Then, follow the build and flash instructions from the first blog in this series. The output will be written to the USB serial as shown in the previous blog. You should see the task number and timestamp from each task printed out, with the hardware timer repeating every second. This is a sample of the expected output:

```
Task: 1, Timestamp: 23002971

Task: 2, Timestamp: 23002983

Task: 3, Timestamp: 23002991

Task: 1, Timestamp: 24002984

Task: 2, Timestamp: 24002997

Task: 3, Timestamp: 24003005
```

Notice how there are no continuously executing loops or artificial delays set in our application code. Instead, we use event-driven design to capture interrupts from the hardware to continue task execution, saving processor cycles. This example is less than 50 lines of code, but these basic concepts can be applied to much larger applications.

The next blog will continue our exploration of FreeRTOS with the Raspberry Pi Pico by looking into the multicore capabilities of the hardware. We will compare Asymmetric Multiprocessing (AMP) and Symmetric Multiprocessing (SMP), which are both options available with FreeRTOS.

SUBSCRIBE

Daniel Gross is a Senior Developer Advocate at Amazon Web Services (AWS). Focused on FreeRTOS and embedded devices interacting with cloud services, Daniel is passionate about delivering a great developer experience for IoT builders.

More from Daniel

Categories

OPEN SOURCE - LINUX, FREERTOS & RELATED

Comments

Login

There are no comments posted yet. Be the first one!

Post a new comment

Enter text right here!

Comment as a Guest, or login: intensedebate WORDPRESS.COM

Name	Email	Website (optional)
Displayed next to your comments.	*Not displayed publicly.*	*If you have a website, link to it here.*

Subscribe to None

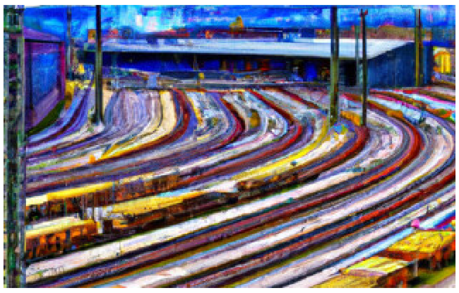
Submit Comment

Using FreeRTOS with the Raspberry Pi Pico: Part 4

By Daniel Gross
SENIOR DEVELOPER ADVOCATE
AMAZON WEB SERVICES

December 19, 2022

BLOG



This is the fourth blog in the series where we explore writing embedded applications for the Raspberry Pi Pico using FreeRTOS.

In this blog, we will cover how to develop code with FreeRTOS that utilizes the dual-core processor onboard the Raspberry Pi Pico. We will also explain the differences between Asymmetric Multiprocessing (AMP) and Symmetric Multiprocessing (SMP). Furthermore, we will walk through the various configuration options available with SMP, and address how the use of SMP can lead to non-deterministic behavior on microcontrollers.

As mentioned from the first blog in this series, the Raspberry Pi Pico is a development board built around the RP2040 microcontroller. The RP2040 has a dual-core Arm Cortex-M0+ processor, with both cores available for application development. Multiple cores can provide added processing capability for an embedded application. It is worth mentioning that FreeRTOS has been supporting multicore processing with AMP for many years. In 2021, FreeRTOS introduced support for multicore processing using SMP. The RP2040 is one of the hardware platforms now supported with the FreeRTOS SMP kernel. You may have noticed from the first blog how we used the '-b smp' flag when cloning the FreeRTOS kernel repository for building the example applications. This means we have been using the SMP branch of the FreeRTOS kernel all along.

Before we proceed, let us first cover what AMP and SMP are, and how they are different from each other in the context of FreeRTOS. As the name implies, Asymmetric Multiprocessing (AMP) treats each processor core independently in an embedded system. For AMP, each core in the system runs its own instance of FreeRTOS. These instances are entirely separate with just a single processor core available to each application. You can establish inter-core communication with AMP by enabling shared memory in the system, then use a stream buffer or message buffer to send and receive data between instances. This article from Richard Barry, Senior Principal Engineer at AWS and founder of the FreeRTOS project, describes how to implement inter-core communication with AMP.

Symmetric Multiprocessing (SMP), on the other hand, allows a multi-core embedded system to run a single instance of FreeRTOS with access to multiple processor cores within the same application. The term 'symmetric' refers to the fact that all cores in the system must have an identical hardware architecture. The

RP2040 has two M0+ cores, which satisfies the requirement for SMP with this microcontroller. To understand the scheduler behavior for multitasking in FreeRTOS, refer to our second blog in this series. With SMP, the scheduler in FreeRTOS can run tasks across multiple cores, which introduces important considerations in terms of scheduler behavior and how you design your applications.

Using SMP on the RP2040 with FreeRTOS, there are notable configuration options available that affect how the scheduler behaves and the functions available to the application. In the next section, we will dive into these specific options and what they mean. The 'FreeRTOSConfig.h' file we used in the first blog to build our application contains the configuration definitions for FreeRTOS, and in that file are the specific options for SMP you can alter.

First, the 'configNUM_CORES' definition sets the number of cores available for FreeRTOS. To use both cores on the RP2040, set the value to '2'.

The next important definition is 'configRUN_MULTIPLE_PRIORITIES'. Normally, FreeRTOS allows tasks to be given different priorities, which affects the order of execution and preemption by the scheduler. However, running tasks with multiple cores available can lead to simultaneous task execution, even if one of the tasks is a lower priority task. This could lead to unexpected and unwanted behavior within applications that assume only single core execution, so the developer should consider this setting carefully. By setting 'configRUN_MULTIPLE_PRIORITIES' to '0', tasks will run simultaneously only if they have the same priority. Setting this value to '1' means that tasks with different priorities could run simultaneously.

Another key definition for SMP is 'configUSE_CORE_AFFINITY'. With multiple cores available, the application developer may want to bind or pin certain tasks to run on specific cores. Pinning a task to one or more specific cores is known as core affinity. By setting this option to '1', a developer can use the vTaskCoreAffinitySet() and vTaskCoreAffinityGet() functions in the application. These functions allow the developer to set and get the core affinity mask for each task, which represents the cores a task can run on. These functions may also be helpful for transitioning an application from AMP or single-core to SMP. We will demonstrate the use of these functions in the code sample provided later in this blog.

Finally, the 'configUSE_PREEMPTION' definition allows the use of the vTaskPreemptionDisable() and the vTaskPreemptionEnable() functions. These functions allow the application developer to disable and enable preemption of a task over a specific section of code. This prevents the scheduler from preempting the task while the section of code is executing, allowing for even greater control of the behavior within a multicore application using SMP.

Below, we have a code example that demonstrates the use of SMP. You can follow the same instructions from the previous blogs in this series to build and run the example on your own Raspberry Pi Pico. In the code, we create four tasks, named A, B, C, and D. Task A is pinned to core 0 on the RP2040 using the vTaskCoreAffinitySet() function, while task B is pinned to core 1. Tasks C and D are not pinned to any particular core, meaning they can run on either core. On execution, each task gets the core affinity mask for itself by using vTaskCoreAffinityGet(). Then, each task prints its name, the core that it is currently running on, the current tick count, and the core affinity mask to the serial console. The get_core_num() function is provided by the Raspberry Pi Pico C/C++ SDK. You might notice we borrowed the vSafePrint() function that uses the Semaphore API from the previous blog.

```
#include <stdio.h>

#include "pico/stdlib.h"

#include "FreeRTOS.h"

#include "task.h"
```

```
#include "semphr.h"


const int taskDelay = 100;

const int taskSize = 128;


SemaphoreHandle_t mutex;


void vSafePrint(char *out) {

    xSemaphoreTake(mutex, portMAX_DELAY);

    puts(out);

    xSemaphoreGive(mutex);

}


void vTaskSMP(void *pvParameters) {

    TaskHandle_t handle = xTaskGetCurrentTaskHandle();

    UBaseType_t mask = vTaskCoreAffinityGet(handle);

    char *name = pcTaskGetName(handle);

    char out[32];

    for (;;) {

        sprintf(out,"%s  %d  %d  %d", name,

            get_core_num(), xTaskGetTickCount(), mask);

        vSafePrint(out);

        vTaskDelay(taskDelay);

    }

}


void main() {

    stdio_init_all();

    mutex = xSemaphoreCreateMutex();

    TaskHandle_t handleA;

    TaskHandle_t handleB;

    xTaskCreate(vTaskSMP, "A", taskSize, NULL, 1, &handleA);
    xTaskCreate(vTaskSMP, "B", taskSize, NULL, 1, &handleB);

    xTaskCreate(vTaskSMP, "C", taskSize, NULL, 1, NULL);

    xTaskCreate(vTaskSMP, "D", taskSize, NULL, 1, NULL);

    vTaskCoreAffinitySet(handleA, (1 << 0));

    vTaskCoreAffinitySet(handleB, (1 << 1));

    vTaskStartScheduler();

}
```

When you run this application on the Raspberry Pi Pico, you should see output in the serial console similar to this:

```
D  1  9101  -1

C  0  9101  -1

B  1  9102  2

A  0  9102  1



A  0  16000  1

C  1  16001  -1

D  0  16001  -1

B  1  16002  2



D  1  60701  -1

A  0  60701  1

C  1  60702  -1

B  1  60702  2
```

Above, we have extracted three sections of the output to illustrate potential behavior of tasks when using SMP. Each section shows that all four tasks are executing in close proximity to one another, just within a tick or two. However, you will notice that the order of execution for each task changes between the sections. You should also notice that tasks A and B run only on the cores they were configured to run on (0 and 1 respectively), but tasks C and D run on either core because of the affinity mask. Also, recognize that C may execute before D, D may execute before C, or they may execute simultaneously. This kind of behavior can be considered non-deterministic behavior, because we cannot predict the exact order the tasks will execute or where they will execute due to the affinity mask and preemption across cores. Therefore, it is important for a developer employing SMP to properly configure and use affinity masks, task priorities, and preemption to suit the needs of the application. This may mean consciously limiting some of the available capabilities to ensure the desired behavior.

Historically, embedded developers working on single-core microcontrollers did not have to consider certain kinds of non-deterministic behavior introduced by using multiple cores. As we covered in this blog, there are several different ways to use SMP, depending on the configuration and usage. SMP with FreeRTOS on the

Raspberry Pi Pico provides new capabilities, and developers should design their applications thoughtfully when leveraging these capabilities. Additional resources for learning more about SMP can be found here and here.

We hope you have found this blog series instructive and informative. Using FreeRTOS with the Raspberry Pi Pico for embedded development is straightforward and offers many capabilities for embedded applications as we have shown. Use the example code provided as a starting point to experiment. Remember to visit freertos.org for more information as you explore and build.

Daniel Gross is a Senior Developer Advocate at Amazon Web Services (AWS). Focused on FreeRTOS and embedded devices interacting with cloud services, Daniel is passionate about delivering a great developer experience for IoT builders.

More from Daniel

Categories

OPEN SOURCE - LINUX, FREERTOS & RELATED