LAB 3: Using a Real-time Operating System

Say you have a robot that is exploring an area. The computer controlling the robot has a number of tasks to do: getting sensor input, driving the wheels, and running a navigational program that may have a fairly high computational load. One key issue in such an environment is to ensure that all the tasks are done in a timely way. On a microcontroller you might use a timer to generate interrupts that regularly address the motors and sensors while having the navigational task running as the main program. But that fairly simple model breaks down in a number of situations. For example, what if the navigational program is actually a collection of programs each of which need to run at different intervals for different durations (perhaps image processing, routing, and mapping programs)? It would be quite possible to write code to handle this, but having libraries (or an OS) which provides APIs for scheduling would be very helpful. A real-time operating system (RTOS) provides tools that allow us to schedule these tasks with a fairly strong degree of reliability—more so than standard OSes like Windows and Linux.

An RTOS is an OS which is intended to serve real-time application requests. It must be able to process data as it comes in, typically without significant delays[1]. RTOSes come in a wide variety of forms. Some are full-fledged OSes with an emphasis on real-time responses. Those may have memory management, disk protection, support for multiple simultaneous users, etc. Others are more "bare-bones" and can be viewed as just of a collection of libraries to do specific tasks. In this lab we'll be using an RTOS that leans toward the "bare-bones" side of things—FreeRTOS.

As its name implies, FreeRTOS is a free real-time operating system. It is a fairly limited RTOS—and that can be a good thing. It will allow us to easily schedule tasks, deal with interrupts and work with at least some of the peripherals with relatively little overhead (both in terms of coming up to speed with the software and in terms of resources used). In this lab you will get a chance to use the RTOS and get familiar with the various issues associated with scheduling tasks.

People doing this lab should already have a familiarity with memory-mapped I/O, basic electronics (including the use of function generators and oscilloscopes), and an understanding of scheduling algorithms in general and rate-monotonic scheduling in particular. Those doing this lab also are expected to have a basic degree of familiarity with the Linux command-line interface.

[1] Those last two sentences are taken nearly verbatim from Wikipedia.

1. Prelab

The prelab consists of two parts: a quick review of the terminology and ideas of scheduling algorithms as well as an introduction to FreeRTOS.

A. Scheduling algorithms

In order to effectively use the RTOS we need to understand how to effectively schedule tasks and which scheduling algorithms are available to us. In order to do that, we need a vocabulary that lets us easily discuss the tasks and their requirements. *This lab assumes that you already have a basic understanding of scheduling in general and rate-monotonic scheduling in particular.* But even so, to be certain that the vocabulary is consistent (and to remind you of the basics), a brief overview is provided below.

Definitions

- **Execution time of a task** – time it takes for a task to run to completion
- **Period of a task** – how often a task is being called to execute; can generally assume tasks are periodic although this may not be the case in real-world systems.
- **CPU utilization** – the percentage of time that the processor is being used to execute a specific scheduled task

$$U = \frac{e_i}{P_i}$$

  - where e_i is the execution time of task i, and P_i is its period
- **Total CPU utilization** - the summation of the utilization of all tasks to be scheduled

$$\sum_{i=1}^{n} \frac{e_i}{P_i}$$

- **Preemption** – this occurs when a higher priority task is run displacing a lower priority task is already running. The higher priority task preempts the lower priority one and the higher priority task runs to completion (unless it is preempted by an even higher priority task).
- **Release time** – the time when an instance of the job is ready to run. The task's deadline is its period.
- **Deadline** - a ending time for which a system must finish executing a given task. Missing a deadline has different consequences, depending on the type of deadline that was missed.
  - **soft deadline** – when the result has utility even after the deadline has passed.
  - **firm deadline –** when the result has *no* utility after the deadline has passed.
  - **hard deadline** – If "catastrophe" could result if a (usually firm) deadline is missed. Catastrophe can vary by context, but "failure of the system to do its job" generally counts.
- **Priority** - importance assigned to each task; determines the order in which tasks execute/preempt each other; priorities are generally assigned in 2 major ways
  - **static scheduling policy** – priorities are assigned by the user at design time and remain fixed.
  - **dynamic scheduling policy** -- priorities may change during program execution and priority assignment is handled by the scheduler, not the user.
- **Critical instant** – an instant during which all tasks in the system are ready to *start* executing simultaneously.

Common Scheduling Policies

There are a number of fairly standard scheduling algorithms that see use in RTOSes. An ideal scheduling algorithm would be able to: schedule any set of tasks for which there is a schedule, fail gracefully when the tasks aren't schedulable, and would be simple to work with (having fixed or "static" priorities for each task would be a good start on being simple…). Of course, no such algorithm exists, so we have a number of different algorithms we can consider, each with its own set of advantages and disadvantages.

In this lab, the details of the scheduling schemes are not directly relevant. None-the-less, it is helpful to start thinking about the scheduling of jobs. As such, this section will provide a *very* terse introduction to one scheme: rate-monotonic scheduling. You should have already learned about this in class, but a summary is included here for quick reference (some of the lab questions ask about it).

Rate-Monotonic Scheduling (RMS)

Rate-monotonic scheduling (RMS) is a popular and easy to understand static policy which has a number of useful properties. Priorities are assigned in rank order of task period, so the highest priority is given to the task with the shortest period and the lowest priority is given to the task with the longest period.

However, when working with RMS, one must take into account the following assumptions:

1. The requests for all tasks for which hard deadlines exist are periodic.
2. Deadlines consist of runability constraints only, that is, each task must be completed before the end of its period.
3. The tasks are independent—each task does not depend on any other tasks.
4. Task run time is constant (or at least has a known worst-case bound).
5. Any non-periodic tasks are special (initialization or failure recovery, for example), and do not have hard critical deadlines.

Of course, these assumptions are often violated in real-world systems, i.e. some tasks can become aperiodic due to event-driven interrupts, and determining actual worst-case runtime can be quite difficult. Below is an example of an RTOS running an RMS policy with three tasks.

Task	Execution Time	Period	Priority
T1	1	4	High
T2	2	6	Medium
T3	3	12	Low

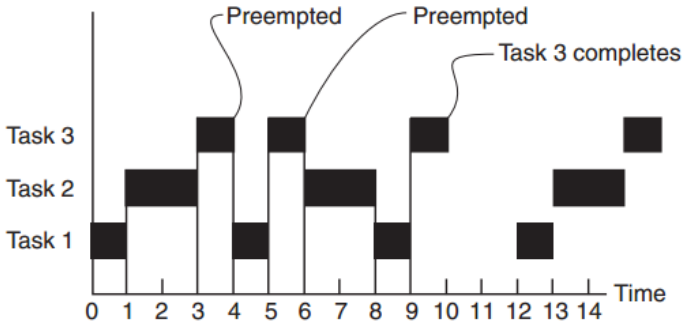


Table 1: A scheduling problem from [1]
Figure 1: "Critical instant" analysis, also from [1]

Q1. Explain why, at time 3, task #3 is running. Also explain why it stops running at time 4.

Q2. What is the largest value T2's execution time could be and still have these tasks be schedulable? You can use fractions as needed. Justify your answer. Do the same for T3. Assume that the scheduling algorithm has no overhead.

B. FreeRTOS

FreeRTOS is a fairly stripped-down RTOS when compared to large commercial RTOSes (such as VxWorks, Windows CE, QNX, etc.) We've chosen to work with FreeRTOS because it provides source code, is relatively easy to understand and work with, is free for commercial use with minimal "viral" issues[2], and is quite heavily used[3]. The major downside to working with it is the lack of support for more complex functions.

To begin with, we will work with Arduino FreeRTOS, a very simple port provided as a static library. We will use it to gain familiarity in programming with a real-time operating system. Despite Arduino's many good qualities, we will begin to see some of its limitations when we attempt to add more functionality to our robot from the previous lab using FreeRTOS.

Later in the lab, we will transition to the Raspberry Pi. Our specific port of FreeRTOS on the Raspberry Pi is particularly limited in that there is no interactive debugging. In addition, we will need to build our program on a desktop machine and move data back-and-forth via an SD card. FreeRTOS in general supports interactive debuggers and programming over a USB/JTAG link, but in an attempt to limit the platforms we're working on, we've chosen to use this (unofficial and unsupported) port of FreeRTOS for the Raspberry Pi for this lab. Note that there will be a few differences between Arduino FreeRTOS and FreeRTOS on the Pi.

At its heart, FreeRTOS is a set of C libraries and in particular a task scheduler. At every "tick" (set to be 1ms on the Pi and around 15ms on the Arduino) the scheduler throws an interrupt and considers all the tasks "Ready" to run. It runs the highest priority ready task. If there is a tie for highest priority task ready to run, it uses round-robin scheduling to switch between the tasks of that priority. The scheduler can also "Block" tasks and cause them to be removed from the ready list until an event occurs (examples of events include a semaphore being set and a certain amount of time passing). It can also "suspend" tasks and cause them to be unschedulable until explicitly resumed. See the diagram below (taken from the FreeRTOS documentation[4]) for a visual representation.

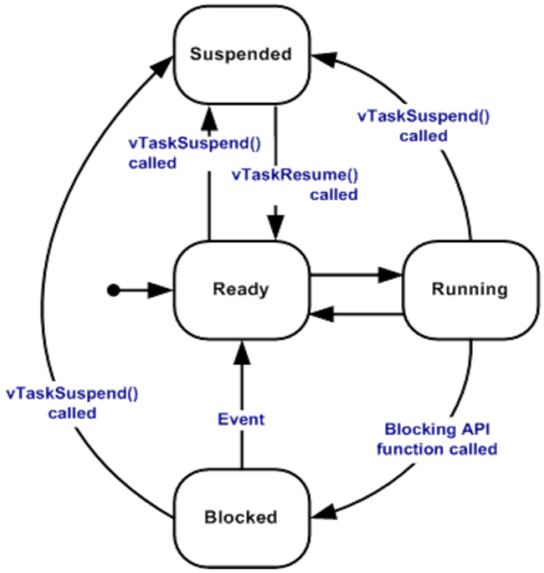
In addition, FreeRTOS provides a number of other features, most notably a set of libraries for accessing the peripherals of a given chip/board. While Arduino FreeRTOS retains the normal Arduino functions for controlling GPIO, a generic API has been ported so that we can control specific GPIO pins when using FreeRTOS on the Pi. Ports to a specific board generally provide a port-specific API for devices on the board (such as LCDs, SPI, etc.)

[2] http://www.freertos.org/a00114.html

[3] https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[4] http://www.freertos.org/RTOS-task-states.html

- **Running**
  - Task is actually executing
- **Ready**
  - Task is ready to execute but a task of equal or higher priority is Running.
- **Blocked**
  - Task is waiting for some event.
    - **Time**: if a task calls vTaskDelay() it will block until the delay period has expired.
    - **Resource**: Tasks can also block waiting for queue and semaphore events.
- **Suspended**
  - Much like blocked, but not waiting for anything.
  - Tasks will only enter or exit the suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.



Scheduling on FreeRTOS

As described above, FreeRTOS provides a scheduling algorithm. You need to create tasks and give them each a priority. This is done by calling the function xTaskCreate(). The two main arguments to xTaskCreate () is a pointer to the function that implements the task and a priority for the task. When using FreeRTOS on the Pi, you will need to call vTaskStartScheduler() after all tasks are created. **In Arduino FreeRTOS, the task scheduler starts automatically after the setup() function completes**. Below is an example (taken from the FreeRTOS documentation) of the scheduler being used.

```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - most small microcontrollers will use
                             much less stack than this. */
                    NULL,    /* We are not using the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
```

Q3. Read through the FreeRTOS documentation (http://www.freertos.org) and answer the following questions.

a. What is the required prototype for all "task" functions?
b. Which has a higher priority, a task with priority "1" or priority "2"?
c. What does the **`vTaskDelay()`** function do? Your answer should include the terms "Ready" and "Blocking".
d. What does the **`vTaskDelayUntil()`** do? How does it differ from **`vTaskDelay()`**?

Q4. Assume all your tasks raise a GPIO pin high when they start being run and lower it when they've finished their work. They then use **`vTaskDelayUntil()`** to allow the task to run for the appropriate fixed amount of time. You have Task "A" running every 5ms at priority 1 and using 1.9ms of CPU time and Task "B" running every 3ms at priority 2 and using 0.9ms of CPU time. Assume the scheduler has no delay.

a. Is the above RM schedulable?
b. Draw a graph of what the GPIO pins of task A and task B would look like starting at time 0 (when both wish to run) and ending at 15ms. Assume that when a task does a DelayUntil the scheduler is run immediately to pick out the next task that should run.
c. Now assume that the GPIO pin goes high when the task is released (first ready to run) rather than when it first starts bring run. Redo part b under this assumption.
d. Redo parts a, b, and c but the tasks take 2.1ms and 1.1 ms each.
e. Discuss the usefulness of your part "b" graphs verse your part "c" graphs. Which would you prefer to see if you were trying to figure out if set of tasks are schedulable under RMS?

Q5. Describe the functionality of **`xSemaphoreTake()`** and **`xSemaphoreGive().`** Also, give an example where one would use a semaphore in an RTOS.

Interrupts provide a problem for the scheduler—the interrupt will run no matter the priority of the current task (unless interrupts are disabled). This leads to no end of problems. The following question gets at one of those problems.

Q6. Say you've got an ISR which has a fairly CPU-intensive task that it needs to run whenever the interrupt occurs—but that task isn't particularly time sensitive. The problem is that the ISR runs until it's finished and can't easily "get out of the way" of a non-interrupt high-priority task (which may be more time sensitive). How would you go about solving this?

C. Miscellaneous issues

Q7. Answer the following:

a. In your own words define the term "watchdog timer".
b. Read https://embedds.com/using-watchdog-timer-in-your-projects/. What is the lowest period you can select for the watchdog timer on an AVR microcontroller?
c. Read the "General" section at https://github.com/feilipu/Arduino_FreeRTOS_Library. The Arduino port of FreeRTOS uses the watchdog timer for the scheduler—every time the watchdog timer goes off, the scheduler is run. So the "tick" time choices are quite limited. What is the default "tick" period?
d. Why do you suppose they are using the watchdog timer for the "tick" rather than a general-purpose timer?

Be aware that lower-cost microprocessors often have quite a bit of variability in their timing. For that reason, there is sometimes an external oscillator needed if you want more accurate timing. So our 15ms or 16ms (notice the two sources disagree) might be even some other value…

D. FreeRTOS Tools for Arduino and RPI5

The FreeRTOS tools for Arduino and the RPI5 are installed on the 473 workstations. If you would like to install them on your PC before the lab (optional), here are some directions.

How to install Arduino FreeRTOS tools to your PC.

Install Arduino. In the Arduino IDE, navigate to the library manager. Look for the FreeRTOS Library (https://github.com/feilipu/Arduino_FreeRTOS_Library) in the library manager tab to make sure it is installed (FreeRTOS by Richard Berry). If not, click install.

How to install RPI5 FreeRTOS tools to your PC

The RPI5 FreeRTOS source and tools will be developed under Ubuntu. The lab workstations have a version of Ubuntu that runs alongside W11 with all the tools included.

You can download the FreeRTOS source files and install the cross compiler to an Ubuntu install on your own machine through the following steps:

If on Windows, open Ubuntu WSL or install WSL if you don't have it. Once logged into Ubuntu, download the cross compiler for your machine. If using x86 Linux or WSL on x86 Windows, run:

```
sudo wget https://gitlab.arm.com/api/v4/projects/tooling%2Fgnu-toolchains-for-arm/packages/generic/gnu-toolchain/15.3.rel1/arm-gnu-toolchain-15.3.rel1-x86_64-aarch64-none-elf.tar.xz

sudo tar -xvf <package name>
```

Install the cross compiler to the proper directory using:

```
sudo mv arm-gnu-toolchain-*-x86_64-aarch64-none-elf /usr/aarch64-none-elf
```

Now you will be set up to compile FreeRTOS for the RPi5 on your machine.

2. Inlab

In this lab you will be learning to use FreeRTOS. At first we will be using Arduino FreeRTOS as a learning tool to understand scheduling with multiple tasks and resource protection using semaphores. The limitations of the Arduino environment and hardware will become apparent when we try to use FreeRTOS to add functionality to our robot from labs 1 and 2.

In the later sections of the lab, we will switch to running FreeRTOS on the Raspberry PI to take advantage of its greater processing and I/O capabilities. Unfortunately, there is no official port to the Raspberry Pi (which we've chosen to use for this lab because we'll be using the Pi in lab 4 also). This causes a few problems, including the fact that we'll need to be doing our programming and compilation on a desktop machine and will move data back-and-forth via an SD card. Sorry about that.

A. Setup: Arduino FreeRTOS,

FreeRTOS is very easy to set up with the Arduino. In the Arduino IDE, navigate to the library manager. Look for the FreeRTOS Library (https://github.com/feilipu/Arduino_FreeRTOS_Library) to make sure it is installed. If not, click install. Note: you can probably also use the default FreeRTOS version found in the standard Arduino libraries, but this lab was tested with the version listed above).

B. "Das blinkenlights"[5] in Arduino FreeRTOS

We will start the lab working with Arduino FreeRTOS to gain familiarity with scheduling and the different subroutines available. As with most boards/systems, it is generally a good idea to first get a light blinking. Take a look at the example code provided by Arduino FreeRTOS in File->Examples->FreeRTOS-> Blink_AnalogRead. Comment out or remove the AnalogRead task and the creation of that task. Run the program and check that your LED is blinking at about the frequency expected. Save this example to your own directory.

Modify the code so that in ***addition*** to toggling the LED, you are also toggling a GPIO pin on the board. You will then use the scope's measurement feature[6] (not just looking at the output) to measure that delay. If you are not working in the lab, you can use the Saleae logic analyzer, though again, don't just estimate by looking at the output, use the measurement tools.

Q1. Briefly answer the following questions:

a. To the nearest millisecond, how long is the period? How close is that to what you expected?
b. If you change the delay from 1000 to other numbers, how close can you get to a ½ Hz frequency (1000ms high and 1000ms low)?
c. From part b, what is the standard deviation (which the scope should be showing you)?
d. From part b and c, talk about the system's accuracy vs. its precision.[7]
e. What do you think is going on? (Look at prelab question Q7…)

[5] http://www.netlingo.com/word/das-blinkenlights.php

[6] You should be aware that you are measuring the signal that is displayed. That sounds obvious, but it means that the level of "zoom" you have can impact your measurements. Also, be sure to "clear statistics" between measurements. Section 14 of https://www.ntecusa.com/docs/75019-97051.pdf is relevant here.

[7] https://en.wikipedia.org/wiki/Accuracy_and_precision describes this idea.

Replace the TaskBlink task with the following code.

```
void TaskBlink(void *pvParameters) {

  (void) pvParameters;

  pinMode(LED_BUILTIN, OUTPUT);

  volatile int i = 0;
  for (;;) // A Task shall never return or exit.
  {
    digitalWrite(LED_BUILTIN, HIGH);
    for(i=0;i<30000;i++);
    digitalWrite(LED_BUILTIN, LOW);
    for(i=0;i<30000;i++);
  }
}
```

Now take measurements of the period of the signal (again use the measurement feature of the scope to get stats on the period of the signal).

Q2. Answer the following questions

a. Why is the variable "i" declared as volatile? What happens when you run the program without the "i" declared as volatile? Why?
b. Try increasing the upper bond of the busy loop above ~32,770. What happens and why? (If you can't figure it out, go to "File ☮ Preferences" and set compiler warnings to "all" and rebuild.)
c. A single increment of "i" seems to take about how long? This number will be important. Note this will vary depending on the type of "i".
d. We are trying to model using CPU time with our busy loop. How would that be different from a call to **`vTaskDelay()`** for the same amount of time?

G1. Write a function called **void `CPU_work(int time)`** which causes the Arduino to use about *time* ms. (So **`CPU_work(2)`** would use 2ms of CPU time.) It should do this with a "for" loop. Make sure your subroutine allows for more than 2^15 – 1 iterations of the busy loop. Modify the code in task1 to:

a. Call **`CPU_work(900)`** and **`CPU_work(100)`** to get a signal with 1s period and a 10% duty cycle.
b. Adjust the CPU_work function as needed to get the period reasonably close to 1 second (say within a couple of milliseconds).

Once done, show your code and the GPIO pin's output on the scope or logic analyzer to your lab instructor.

From here on out, use your CPU_work() function to emulate tasks taking time to do "real" work (rather than delaying, where we'd use the various "delay" functions such as vTaskDelay).

C. Multiple tasks in Arduino FreeRTOS

We're now going to look at having more than one task running and letting the FreeRTOS scheduler manage the scheduling. Each task will run periodically and each will use a certain amount of CPU time. Recall from the prelab and the above section that our "tick" resolution is around 16ms. Read everything up to the first question in this section before proceeding.

- **Task1** should run every about every 85ms and use about 30ms of CPU time (using CPU_work).
- **Task2** should run about every 30ms and use about 10ms of CPU time (again with CPU_work).

Have each task control a different I/O pin so that the pin is set high just before the CPU_work function is started and set low just after it finishes. **Have the task with the lower period have the higher priority** (as is standard for RM scheduling).

Note that you'll need to convert ticks to ms. One way to do so is using portTICK_RATE_MS[8]. Another way is to use the function pdMS_TO_TICKS(). Either way, you have resolution issues (with ~16ms ticks, you won't be able to get exactly the periods desired). Look at the prelab (Q3/4) to determine what function you should use to make the tasks run periodically.

If you get motivated (optionally), try to do this by writing a single task function. You'd write a single task, but use xTaskCreate() twice, using the parameter option. This might be handy later…

Q3. Answer the following questions using the oscilloscope once you have your two tasks running:

a. Measure both the period and +width of both I/O pins over 500 or more samples. Record the mean, min, max and standard deviation of the four measurements.
   - If you are doing this from home, you may not be able to measure this. Instead take a couple of minutes and estimate these values using the Saleae.
b. Explain why one of the tasks is a lot more consistent than the other.
c. Explain the variation seen. Why does the *period* of one of the tasks vary? Why it is both above and below the desired value?
d. Do you believe that both tasks are always completing before their deadline (before they are asked to start again)? How can you be sure?
e. Provide the code written for this section (just cut-and-paste it).

Q2. For this part you will do two demos for the lab instructor.

a. Demonstrate the tasks running as described above. Briefly discuss your answers to Q3 with the lab instructor.
b. Change Task1's CPU utilization to 55ms and run the code. Using the scope or logic analyzer, set things up so that you can convince your lab instructor that both tasks are *always* meeting their deadlines (or not). Or, alternatively, explain why you can't show they are (or are not) meeting their deadlines.

[8] http://www.freertos.org/FreeRTOS_Support_Forum_Archive/December_2006/freertos_What_does_portTICK_RATE_MS_stand_for_1636516.html

D. Semaphores in Arduino FreeRTOS

A problem with the code from part C is that you can't use the scope to clearly see when the lower-priority task first wants to run (its **release time**). That's because the higher-priority task might be using the CPU at that instant, and so the lower-priority task can't drive its GPIO pin high at the moment. This makes it really hard to figure out if a deadline is being missed. We are going to add a high-priority third task to help us out. In particular, our third task will do nothing other than:

- drive the GPIO pin associated with task 1 high
- tell task 1 that it can run (release task 1)

This way we'll be able to see the release time of task 1, as the high-priority task will run so fast that it won't significantly slow task 2. We'll have the high-priority task tell task 1 that task 1 can run using a semaphore. Be aware that having a high-priority task release a low priority task is a fairly common idea in operating systems and is usually seen in the context of interrupts (which we will be doing later in this lab!).

So, we are going to basically have the same situation as you did in part D, but we'll have an additional-high priority task running.

- **Task 1 (T1)** will use 20ms of CPU time but will only run when directed by Task 3.
- **Task 2 (T2)** will have a period of about 30ms and use 10ms of CPU time. Given it's short period, per RMS, it should have a higher priority than task 1.
- **Task 3 (T3)** T3 will have the highest priority. Every 85ms, T3 will drive T1's GPIO pin high and then "wake up" T1. T1 will do its 20ms of "work", drive its pin low, and then wait for T3 to wake it up again. Thus, T1 will have a 85ms period, but that period is being controlled by T3.

The basic idea is that we want to be able to observe T1's release time.

Again, the way T3 will wake up T1 is by using a semaphore. T3 will set the semaphore (in the standard jargon of semaphores it will "give" the semaphore) every 85ms. T1 will be waiting for the semaphore to be set. Once it is set, T1 will do its 20ms of CPU "work" and then wait for the semaphore to be set again. To use semaphores in this context, you will use three functions[9]:

- **`xSemaphoreCreateBinary( void )`**
  - This is a macro that properly creates a semaphore and return a SemaphoreHandle_t object.
- **`xSemaphoreTake(`**`SemaphoreHandle_t` **`xS, TickType_t xBlockTime )`**
  - This macro tries to take the semaphore. If it's not available, it will delay until the semaphore is available or xBlockTime (in ticks) has passed. It will return pdPASS if the semaphore was taken and pdFALSE if not. If you don't want to wake up without the semaphore, set xBlockTime to be fairly large.
- **`xSemaphoreGive(`**`SemaphoreHandle_t` **`xSemaphore )`**
  - Makes the semaphore available for others to take.

Note: You'll need to include <semphr.h> to use semaphore on vanilla FreeRTOS

Hint: see example semaphore usage at File->Examples->FreeRTOS->AnalogRead_DigitalRead. Folks that have had 482 or 570: see https://www.freertos.org/Real-time-embedded-RTOS-mutexes.html to understand why we are using semaphores rather than mutexes.

[9] http://www.freertos.org/a00121.html and related pages (linked from that one) detail these three functions.

G3. Get tasks T1, T2, and T3 working as described above where T1 and T3 are using one IO pin (T3 setting it high and T1 setting it low once CPU_work() has finished). Again, you will do two demonstrations for your lab instructor and answer one question. Try to have your numbers picked for part b and to have thought about part c before you ask your lab instructor to come over.

a. Show your lab instructor the working code you wrote for the above.
b. Modify the CPU_work duration of one of the tasks so that a deadline is being just barely missed. RM scheduling theory should help. Show your lab instructor and explain how you know a deadline is being missed.
c. Explain to your lab instructor what is happening when a deadline is missed. In particular, what is going on when task 3 releases its new task instance when the previous instance of task 1 is still running? Your lab instructor will ask about the semaphore and the pin's value.

Q4. Answer the following questions:

a. Here we are using the semaphore to allow a high-priority task to do some time-critical thing (set the GPIO pin high) and then hand off the rest of the (CPU-intensive but low priority) work to a lower-priority task. Provide a real-world case where doing this might be useful.
b. A more common use of semaphores is the producer-consumer model. In that case, one task might be (for example) reading serial data and putting it into a packet. Once that packet is finished, there is another task that handles the data (maybe the data is from a camera and the other task processes it looking for a certain object). Why would one want to use a semaphore in a situation like that (rather than just having one task do all the work)?

E. Interrupts in Arduino FreeRTOS

Let us briefly look at interrupts. In FreeRTOS interrupts are largely handled using the platform's tools. This does greatly reduce the portability of the code, but frankly if you are dealing with interrupts you've likely got portability problems already. In this setting there are no special flags required to mark that a function is an interrupt handler (ISR). Normally, several steps must be taken to register an ISR with the RTOS. However, Arduino FreeRTOS is quite simple and we will not need to do this for our purposes. If you have extra time, try to work out how to register interrupts on the Raspberry Pi port of FreeRTOS.

The goal of the rest of this section is for you to see how to do deferred interrupts in FreeRTOS and to understand why you might want to use them. As discussed in class, the problem is that a low-priority task might be triggered by an external event. In general, we use interrupts to detect external events (as opposed to polling which wastes CPU time). But the problem is that the ISR will interrupt whatever is running, effectively becoming the highest priority task. This is a problem if there is a task that *should* have a higher priority than the work associated with the interrupt—the decision has been taken out of the hands of the scheduler.

The solution is to have the ISR simply set a semaphore that wakes up a task and then exits the ISR. Now the task that got woken can be scheduled as normal (with whatever the appropriate priority is for that task). In this case the interrupt is said to have deferred the work to a regularly scheduled task.[10]

First, we are going to look at what happens if you don't use a deferred interrupt. Modify your code from the previous section so that when pin 2 is toggled by an external device, you generate an interrupt[11]. Your interrupt should use 100ms of CPU time. Also write a task called "task1" which uses 20ms of CPU time and runs about every 30ms. And we'll use a new version of CPU_work that will toggle a given pin as the work is being done.

```
void CPU_work(int time,int pin)
{
  volatile int i,j,k;
  int mode=0;
  for(i=0;i<time;i++){
    for(j=0;j<8;j++){
      mode=!mode;
      digitalWrite(pin,mode);
      for(k=0;k<100;k++);
    }
   }
}
```

If you are in lab, drive the interrupt using the function generator (be sure you are using a reasonable voltage before connecting to the processor!) generating an interrupt every 1000ms. If you are working from home, you can use the RPi to create this interrupt. Make sure to complete the RPi setup, and then have main.c toggle a GPIO pin at a period of 1000ms. Then use this to trigger the interrupt on the Arduino. Looking at the scope or logic analyzer, capture an instance of the interrupt code jumping in the way of task1.

Q5. Include a screenshot of what you are seeing on the scope and explain what is happening. In particular, does task 1 ever finish on time? Does it ever *not* finish on time?

Now let's try it with a deferred interrupt. Say that we want task1 to have priority over the work being done in the interrupt. So have the function that gets called on the interrupt (the ISR) simply give a semaphore. Have a task2 that has a lower priority than task1 that waits for the semaphore to be given. Task2 should use the 100ms of CPU time (rather than the ISR itself). Again, the idea here is that we have some event we wish to react to. So we use an interrupt. *But* we want the work we do as a reaction to that event to have lower priority. That's something we can't normally do in an ISR—interrupts have the highest priority. So we are handing off the work to a low priority task whenever the event occurs.

Note, there are a few details you have to get right when using semaphores with an ISR:

[10] For example, Linux has "top and bottom halves" for interrupts (https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch10.html). Also called "deferred interrupts" http://www.freertos.org/deferred_interrupt_processing.html, or "deferred procedure calls (DPC)" https://en.wikipedia.org/wiki/Deferred_Procedure_Call.

[11] You will want to use the Arduino function "attachInterrupt". A web search will be helpful.

- When giving a semaphore from within an ISR, you are to use the function **`xSemaphoreGiveFromISR()`** rather than what you've used above.
- You can't use a Mutex-type semaphore when giving a semaphore from an ISR. Instead you must use a binary semaphore. Use the function **`xSemaphoreCreateBinary()`** to create the semaphore.

G4. Using the scope, demonstrate this working code to your lab instructor. Explain what has changed from when the delay code was in the ISR and explain why deferred interrupts could be useful.

E. Pi FreeRTOS

Now that you are familiar with FreeRTOS, we would like to extend the functionality of our simple robot from lab 2. Because Arduino FreeRTOS is limited by the amount of memory available and the drawbacks we observed in part C of this lab, we will work on porting our robot to FreeRTOS on the Raspberry Pi. In addition, we will add some new functionality to utilize the real-time attributes of the operating system. Rember to insulate between the bottom of your Pi and your robot like you did with the Arduino.

Compiling FreeRTOS for the RPI4

Here is a guide to compile FreeRTOS for the RPI4, but you must clone the RPI4 GitHub first. However, it's recommended for labs to use the RPI5 if you can get one.

Compiling FreeRTOS for the RPI5

We are going to compile the FreeRTOS source, a GPIO library, and a main.c using a cross compiler running on Ubuntu Linux. The cross compiler and Ubuntu are installed on the EECS2334 workstations. It is a command line version only and conveniently runs alongside Windows. It is called Windows Subsystem for Linux or WSL. You can run it by typing the following at the command line in Windows:

```
wsl --distribution UbuntuWSL473
```

or simply

```
wsl
```

How to build applications under WSL

Here are the steps to build a simple task toggling GPIO pin 21:

Clone the source code repo (do this in a regular command prompt, not WSL):

```
git clone https://github.com/EECS473/473freeRTOSrpi5.git
cd 473freeRTOSrpi5
```

If you're using Windows (including the EECS2334 workstations), open WSL. If on Linux, open the terminal. If you're on your own machine, make sure you followed the prelab's instructions to install the cross compiler.

Building the Application with the Makefile

The Makefile is a very powerful build tool that uses a cross compiler and source directories to build your project. It is set up to build the RTOS, GPIO and user application source files. The applications we will be running are in a source file named main.c in the src directory. To start it contains a simple single task example that toggles one of the GPIO ports. You can view the contents with:

```
cat main.c
```

Let's try to make this example. In the current directory with the Makefile, run the Makefile by simply typing

```
make clean
make
dir
```

You should see several compile lines executing. When it is complete you should see that it generated a `kernel_2712.img` file in the `application` directory. The `kernel_2712.img` file contains all the compiled RTOS source and application files. It is run during the boot RPI boot process.

Note: You will be unable to compile lab 3 project files out of your N drive. We recommend storing copies of your code in a folder in your N drive and continually updating the main.c in your 473freeRTOSpi5 folder in your C drive.

Editing and Copying the .img file to the Flash Card

You will need to edit the main.c file that will contain your RTOS application, compile and then copy the `application/kernel_2712.img` file to the flash card root directory. Also, find the file `boot/config.txt` in the source code repository. Copy this to the root of the SD card as well. **Your SD card should have 2 files in it: kernel_2712.img and config.txt.**

Make sure to eject the SD card from your file system first before yanking it out of the reader. This prevents any potential corruption of the SD card itself.

Try the Application

Now insert your SD card into an **UNPOWERED** Raspberry Pi. Plug in the Raspberry Pi and wait for it to boot, after a few seconds you should be able to observe GPIO pin 21 toggling every 1ms.

DO NOT INSERT OR REMOVE A SD CARD TO A *POWERED* PI AS THIS CAN DAMAGE THE PI AND POSSIBLY CORRUPT THE SD CARD (reformat required). Also, try to plug and unplug the USB from the host/power supply rather than the Pi (both easier and the host is less likely to suffer damage from wear).

Next, modify the following line from **1ms** to **2ms**. Save the file, go back to the build directory and make again.

From: `const portTickType xFrequency = 1;//1ms task frequency`
To: `const portTickType xFrequency = 2;//2ms task frequency`

Save the file with the **Write Out** and **Exit** commands listed at the bottom of the editor window.
Use the make command to rebuild the project and generate a new IMG file. Try the change and verify that the GPIO pin 21 is now toggling at 2ms.

G5. Demonstrate that you can change GPIO pin 21 to toggle every 2ms as detailed in the how to build an RTOS application example above.

Q6. What are the differences between the Arduino and Pi FreeRTOS GPIO ports? Here is a link to GPIO drivers on the RPI4 (Same as RPI5).

How to use interrupts on the RPI5 FreeRTOS port.

Implementing interrupts is a bit different on the RPI5 then the Arduino. There is an example of a deferred interrupt posted on the 473 FreeRTOS RPI5 Github as deferred_interrupt_example.c.

Try the example and observe the output.

G6. Demonstrate the deferred interrupt example to your instructor.

Q7. How are interrupts different on the Pi than the Arduino.

F. Self-navigating robot in Pi FreeRTOS

Instead of relying on user input over the Bluetooth for navigational control, we would like the robot to maneuver itself independently using external input from a distance sensor. We will be using the HC-SRO4, a simple ultrasonic ranging module; the specification can be found at this website: https://www.mouser.com/ds/2/813/HCSR04-1022824.pdf. We have provided code to interface with this device.

Download partH.c from the lab website. It is currently on the 473 FreeRTOS RPI5 GitHub as robot_control_start_file.c. It is a port of the motor control code in addition to code that interfaces with the distance sensor and returns the distance in system ticks. *You will need to port the LCD code you wrote in the previous lab and complete an implementation of the robot that satisfies the following constraints*. This is often much harder than you might expect and we are making it optional this term and instead replace it with a flashing LED. That being said, the exercise of doing this port is very valuable (it's often easy to overestimate just how much having done the interface before will help when porting to a new processor).

- The highest priority task will check the distance sensor every 50ms and store the distance in system ticks into a global variable. This has been implemented for you.
- The medium priority task will check the distance variable every 100ms and control robot movement. If the distance is below some threshold, the robot should stop, continuously checking the distance variable within separate task iterations to determine when it is free to move forward again. Note that semaphores may be valuable here to protect your global variable.
- The low priority task will alternate between displaying the current direction of motion and the current distance **converted to centimeters** on the LCD. This task will run whenever possible.
    - You may instead flash an LED at a frequency or duty cycle that gives you a sense of distance (for example equal to distance in cm/10 Hz (so at 10 cm, you'd be at 1 Hz). Because of rounding and implementation issues, you may not get the best resolution…

In order to perform quality control on our set of tasks, tie each task to a separate task pin so that we can visualize their runtime and period on the oscilloscope. Drive the task pin high before each task begins its "work" and low once it finishes.

Q7. Show the GSI that your robot can accomplish these tasks in addition to your scope output.

Q8. Turn in a copy of your code as well as a screenshot of your scope showing that runtime and period of each task.

3. Post lab

Q9. Consider the desire to use EDF scheduling. Look over the task scheduler for Raspberry Pi FreeRTOS. How could you use `void vApplicationTickHook( void )` to accomplish this? To answer this question, you'll probably want to start by looking at the task diagram found in the prelab.

Q10. Let's talk about DC motors and capacitors. Read this overview of why we might need capacitors on a DC motor.

a. Explain the purpose of the ceramic capacitor across the motor (in our case we had you put it across the wires from the motors coming into the board).
b. Explain the purpose of having a large electrolytic capacitor across the battery that drives the motor.
c. Why do we want a ceramic capacitor on the motor? Why an electrolytic capacitor across the battery?

Q11. Read prelab 4. What do you need to put on an SD card before you show up to lab 4?