Lab # 09

Introduction to FreeRTOS

Objectives

- Introduce a real-time operating system((RTOS)
- Learn what tasks are and how to create them
- Get to know task states and priorities
- Implement inter-task communication using semaphores

Tools

- Arduino
- Proteus ISIS
- freeRTOS library

Pre Lab

Please read the theoretical background of RTOS.

Suppose you have a robot that is exploring an area. The computer controlling the robot has number of tasks to do: getting sensor input, driving the wheels, and running a navigational program that may have a 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 simple model breaks down in several situations. For example, what if the navigational program is 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.

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. 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 allows 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.

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.

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

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

●**Preemption** - this occurs when a higher priority task is set to run while 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 all of its scheduled tasks. Missing a deadline has different consequences, depending on the type of deadline that was missed.

- ○**hard deadline** - a deadline for which missing it would be a complete system failure (e.g. airbag deployment being late could kill the user)
- ○**soft deadline** – a deadline for which there is a penalty the deadline is missed. These deadlines are usually in place to optimize the system, not to prevent complete failure. (e.g. an MP3 player might create a "pop" if it misses the deadline, but the system would still function)

●**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.

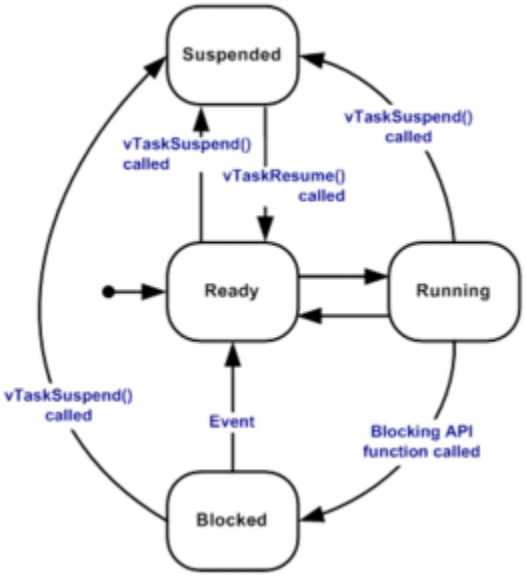
FreeRTOS

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, and is quite heavily used. 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.

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. It can also "suspend" tasks and cause them to be unschedulable until explicitly resumed. See the diagram below (taken from the FreeRTOS documentation) for a visual representation.

- **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.



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 Ports to a specific board generally provide a port-specific API for devices on the board (such as LCDs, SPI, etc.).

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. 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();
```

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) under "Sketch → Include Library" to be sure it is installed.

In-lab task 1: "blinkenlights" 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.

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++);
  }
}
```

In-Lab Task 2: 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. The "tick" resolution is around 16ms.

- 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. 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).

Try to do this by writing a single task function. You'd write a single task, but use xTaskCreate() twice, using the parameter option.

Use two different color leds' to show the output of two independent tasks running in parallel.

Semaphores in Arduino FreeRTOS

A problem with the code from upper part 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.

So, we are going to basically have the same situation as you did in previous part, 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:

vSemaphoreCreateBinary(xSemaphoreHandle xS)

- This is a macro that properly creates a semaphore. If xS isn't NULL the semaphore was created successfully.

xSemaphoreTake(xSemaphoreHandle xS, portTickType 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(xSemaphoreHandle xSemaphore)

- Makes the semaphore available for others to take.

Hint: see example semaphore usage at File->Examples->FreeRTOS->AnalogRead_DigitalRead.

Post-Lab Task 1: Implement the above scenario given in semaphore.

Please take the three tasks running in parallel, Button Input, LED Turn on when button is HIGH and blinking LED with delay of 2 seconds. Here the shared resource that you manage using semaphore is the button state.

Critical Analysis / Conclusion

(By Student about Learning from the Lab)

Lab Assessment				
Pre Lab			**/1**	**/10**
In Lab			**/5**	
Post Lab	**Data Analysis**	**/4**	**/4**	
	Data Presentation	**/4**		
	Writing Style	**/4**		
Instructor Signature and Comments				