ECEN 3002 - RTOS Lab #3 (Tasks)

Derek Wright

Spring 2020

All answers indicated in blue. All steps without a question have been deleted for grader’s convenience.

Procedure:

Part I – See included zip file of the code. Of note here is that the Capsense does not operate as intended when also using the Micrium OS. This is because the capsense relies on timers to sense, and the OS owns the timers once we run it. There is some collision of resources that the OS overrides, and the capsense then gives bad measurements, causing the LEDs to misbehave as a result of the bad state returned by the capsense.

Also of note is that you will not find the uC Probe changes in my code. This is because after many hours of trying to get it to work on both my machine (Mac) and the lab computers (wrong SDKs), I couldn’t get it to work. I used a peer’s uC Probe to complete the lab, but the changes required always resulted in build errors that were unresolvable (files that don’t exist, etc).

Part II – Using Micrium µC-Probe to analyze the system

Note: I am a Mac user and was unable to get Micrium uC Probe running on my machine or the lab computers (as a result of the SDK installed not being compatible). Because of this, I enlisted the support of Jacob Schieffler to answer the questions specific to uC Probe. He was using SDK 2.5, MCU 5.7, Kernel OS 2.6, so there may be discrepancies in some of the OS operations, such as Kernel tasks created.

1. In addition to the tasks that your application created, what other tasks are also running? Describe the purpose of each one (refer to MicriumOS documentation).
   1. Kernel’s Timer Task
      1. This timer task is responsible for controlling the software timers, indicating to the scheduler when other tasks are READY. This is necessary for all of the tasks created by the user in this lab.
      2. From Silicon Labs/Micrium: “The reason the timers operate in a different task than the tick task is the timer callback functions are made from the timer task, so it allows the timer callbacks to run at a different priority than the tick task if desired.”
   2. Kernel’s Stat Task
      1. This task handles run-time statistics like CPU utilization per task, as well as stack info, that are nice to have when using Segger or uC Probe, etc.
   3. Kernel’s Tick Task (not seen in **my** project, only Jacob’s)
      1. The Tick task controls the system tick, which is used for all timing functionality across the system (including timers).
2. Record the CPU Usage and Stack Usage of each task.
   1. Idle – 78.1% CPU, 0% stack (1000 bytes total)
   2. LED -- .14% CPU, 0% stack (1000 bytes total)
   3. Slider – 3.26% CPU, 0% stack (1000 bytes)
   4. Button -- .13% CPU, 0% stack (1000)
   5. Timer Task -- .18% CPU; 39% stack (256 bytes)
   6. Stat – 1.13% CPU; 42% stack (256)
   7. Tick -- 17.04% CPU; 46% stack (256)
3. Determine the period of each task.
   1. Please see part 3 of this Lab writeup since I don’t have access to uC Probe. Segger gave comparable data and more accurately reflects the tasks I created and had running.
4. Save a screenshot of the application, showing the Task(s) tab.

A screenshot of a video game

Description automatically generated

Part III – Using Segger SystemView to analyze the system

4. Save a screenshot of the SystemView application.

A screenshot of a cell phone

Description automatically generated

1. What do you observe about the scheduling of each task? How do interrupts and the priority of other tasks affect scheduling? Does the scheduling of each of the application tasks that you created seem optimal?

The scheduling of each task is handled by its own task, the scheduler! The scheduler has the highest priority and since we have a preemptive OS, it has the power to “interrupt” other tasks at will, and switch to others. This is what is happening with my Idle Task. I never call the OSTimeDly() function in my Idle Task, because I know that I set the Idle task to have the lowest priority and that the OS will properly take control away from it in order to complete the other tasks (like when a timer task tells the scheduler it needs to handle my other three tasks).

As for “seeming optimal,” it’s hard to say. This is because our application is very small and large improvements are hard to identify. I spend 98% of the time in Idle, which is great. For this purpose, the scheduling seems well within our standards of quality. That might change for larger applications, but at that point, improvements will likely be more noticeable.

1. Record the frequency and min/max run time of each task
   1. Scheduler: 60Hz; Min: 45us; Max: 633 us
   2. Task 0x262f: 10 Hz; Min: 0.113ms; Max: 1.18ms
   3. Kernel Timer Task: 10Hz; Min: .151ms; Max: .274ms
   4. Main Start Task: 10Hz; Min: .116ms; Max: .238ms
   5. LED Task: 10Hz; Min: .165ms; Max: .199ms
   6. Capsense Task: 10Hz; Min: .113ms; Max: 1.59ms
   7. Button Task: 10Hz; Min: .111ms; Max: .210ms
   8. My Idle Task: 37Hz; Min: .157ms; Max: **96.3ms (woah!)**

Part IV – Idle Task and Low Energy Mode

1. Start the Energy Profiler and record the average power consumed at nominal conditions (no buttons pressed, slider not touched). Compare this value with the values recorded in Lab 2.

**1.48mA, 4.84mW**

Analysis: Comparing to Lab2, we see that this implementation is right in between the polling method and the interrupt method from last time. The polling method baseline power consumption was 6.45mW, while the interrupt method was 3mW. This makes sense because we haven’t yet tried to enter lower energy modes, so we shouldn’t expect to be as low-power as the interrupt method from before. But because we aren’t polling constantly, and the OS has the power to go to IDLE tasks, we are consuming less power than the polling method!

1. Start the Energy Profiler and record the average power consumed at nominal conditions (no buttons pressed, slider not touched). Compare this value with the value recorded in Step 1.

1.17mA, 3.83mW

So, the average power went down. Yay. That’s what I expect. It still didn’t go down to the interrupt level of low-power consumption, but this is again attributable to the OS now running, keeping tabs on all the tasks that need executing.