ECEN 3002

Derek Wright

Lab #6: ITC

Answers are reported inline and in blue. I have attempted to delete the part numbers that don’t ask for responses, while keeping the numbers that do the same. I cannot guarantee that is the case, though.

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

\*\*\*\*\*NOTE TO THE GRADERS\*\*\*\*

I am one of the people who have a Mac, and because the Lab computers are not equipped to run uC Probe with the right SDK/uC OS versions, I requested the images from a classmate, Jacob Scheiffler. The credit for the image below goes to him, and his implementation varies from mine slightly, so the picture might not reflect my answers as well as they would had I provided my own image. Thanks for your understanding.

1. 
2. Without pressing any buttons or activating the touch slider, which tasks are transitioning to/from the Read state and which tasks are blocked (hint: check Context Switch Counter)?
   1. The Timer Task, the Tick Task, and the Idle Task are transitioning often. The Capsense Task is switching much less than the other mentioned because it only switched every 100ms. It is blocked \*most of the time\*.
   2. The blocked tasks are the LED Task and the Button Task. This is because the Button Task is only awakened on a button press interrupt, which is not happening for this test. Also, the LED Task is only awakened when a message indicated a change of state.
   3. My implementation detected changes in state, and would only post a message to the queue if the state changed. This means that the capsense task would run every 100ms, but would only post a message if the state changed. The button task would post a state change message every time, because every time an interrupt occurs, the state has changed.
3. Press one of the pushbuttons. Which tasks are awakened? Record the number of context switches that occur when a button is pressed and then released.
   1. The Button Task is awakened. This is because the event flag group was posted to and matched the pattern I gave the Button Task to pend() on.
   2. The LED Task is awakened because the button task posts a message to the queue the LED Task pends on.
4. Touch one side of the touch slider? Which tasks are awakened? Record the number of context switches that occur when a button is pressed and then released.
   1. The LED Task is awakened because we have a change in state.
5. Do the number of context switches recorded above match your expectations? If not, check your code to determine why.
   1. They do match my expectations.
6. Save a screenshot of the application, showing the Task(s) tab.
   1. To reiterate, the below image is Jacob Scheiffler’s.

A screenshot of a video game

Description automatically generated

Part III – Using Segger SystemView to analyze the system

1. 
2. Record how often the SliderInput task is run. Does it match the timer period? Save a screenshot that shows the period.
   1. The period is 10Hz, which corresponds to a period of 100ms, as expected. In the image below, in the bottom ‘Contexts’ section, look at the frequency column in the Capsense Task row for confirmation.
   2. A picture containing screenshot

      Description automatically generated
3. Locate when the ButtonInput task is run. Note that by highlighting the task name in the timeline window, left and right arrows will appear for advancing to the previous/next location in the timeline of when the task was run. Does the scheduling of the LedOutput task appear to be synchronized with the ButtonInput task? Explain why. Save a screen shot that shows the scheduling of the ButtonInput task and the LedOutput task.
   1. Yes they do appear synchronized. This is because the LED Task Pend()s on the message queue, and when the Button Task is run, it Post()s to the message queue. This immediately moves the LED Task to the Ready queue and OS\_Reschedule will call it next.
   2. Of note is that this is not unique to the Button Task. The same goes for the Capsense Task when we detect a change in state, which also Post()s to the message queue, with the same results. I have provided a couple images that show this.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Part IV – Idle Task and Low Energy Mode

1. Start the Energy Profiler
2. Record the average power consumed at nominal conditions (no buttons pressed, slider not touched). Compare this value with the values recorded in Lab 2 and Lab 3.
   1. 3.02mW
3. Record the average power consumed when one and both buttons are pressed. Compare this value with the values recorded in Lab 2 and Lab 3.
   1. One button: 1.48mW
   2. Two button: 3.28mW – recall that when both buttons are pressed, the LEDs are off
   3. Because it might be of interest to know the power when both **LEDs** are on, here is that value as well: 6.41mW
   4. Comparing to Lab2 and 3, I see that this method is the best we’ve seen yet. It is comparable in power to the purely interrupt method of Lab2, while utilizing the full capability of the RTOS. In Lab3, the RTOS solution was in between the interrupt and polling methods, but now, we can see marked improvement. Yay!
4. Explain the reason for the difference, if any, in the power measurements between this lab, Lab 2, and Lab 3.
   1. The reason for this performance is that we have moved to an interrupt driven RTOS implementation. The Button Task now only runs when an interrupt happens, not at a consistent interval of 100ms like Lab3. This saves significant CPU utilization. The IDLE task in this implementation is above 99% nominally, and only slightly below 94% with frequent button presses.