Lab 3: Non-pre-emptive Schedulers

Nakseung Choi, 1572578
Kejin Li, 1978130
23-May-2022
Assignment: ECE474 Lab 3

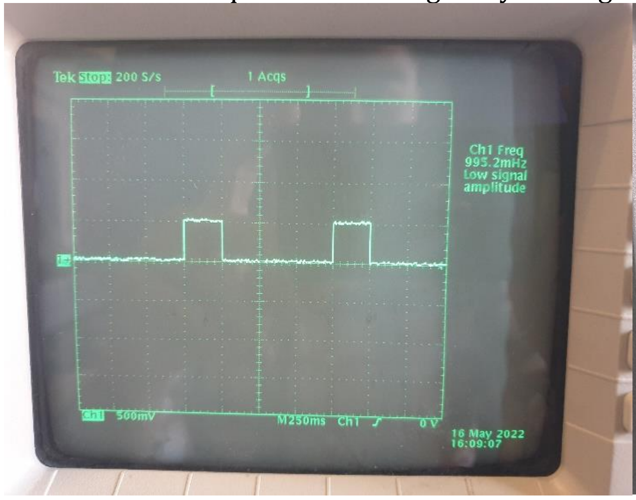
Introduction:

In this lab, we learned how to run several simultaneous tasks using different non-pre-emptive schedulers: a basic Round Robin, Synchronized Round Robin, and Data Driven Scheduler. Each of these non-pre-emptive schedulers has their own characteristic and is employed based on different applications. The difference between pre-emptive and non-pre-emptive schedulers is whether a running process can be interrupted by a high priority process or not. In a non-pre-emptive scheduler, a running process cannot be interrupted by a high priority process whereas it is possible in free-RTOS (pre-emptive scheduling.) Starting this lab, we also learned how to use Doxygen to generate professional documentation automatically from comments, which is widely used in the industry. By doing so, we could effectively record and document our devoted work in an organized manner.

Methods and Techniques:

For demo 1, 2, and 3, an external LED blinks for 250ms and turns off for 750ms. To demonstrate this, we used an oscilloscope to measure the output of the LED. Figure 1 below shows that the signal goes high for 250ms and goes low for 750ms, total of 1000ms. Since the horizontal axis was set to 250ms per block, one block means 250ms.

Next, to verify that the speaker halts to play for 4 seconds after the theme is played once, we also used the oscilloscope to measure the time. Figure 2 shows that the signal does not emit a tone for 4 seconds. In this case, the horizontal axis was set to one second per block, so the signal in the second and third quadrant was regularly coming out for 4 blocks, which means it was for 4 seconds.



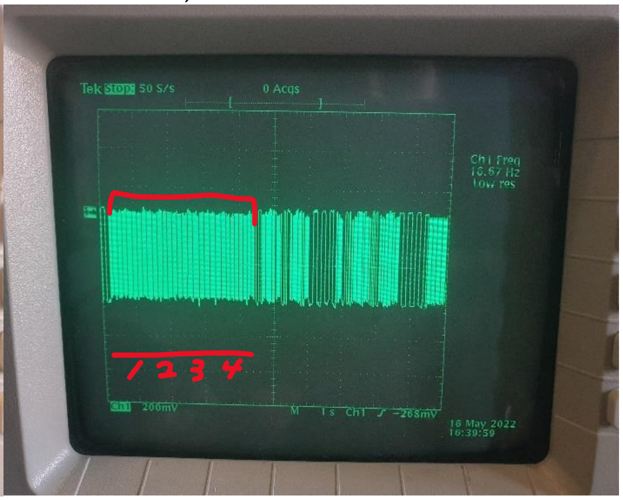


Figure 1. LED flashes for 250ms. Figure 2. The speaker stops for 4 seconds.

We also validated the frequency of each note using an oscilloscope. The speaker generates 293, 329, 261, 130, and 196Hz for 200ms. The figures are shown below:

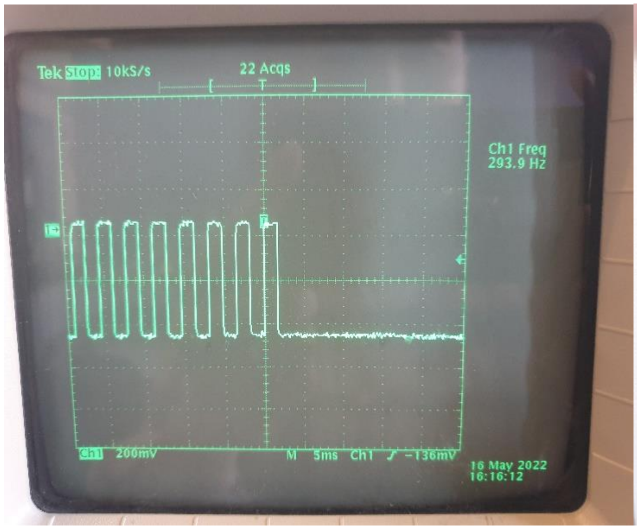


Figure 3. 294Hz

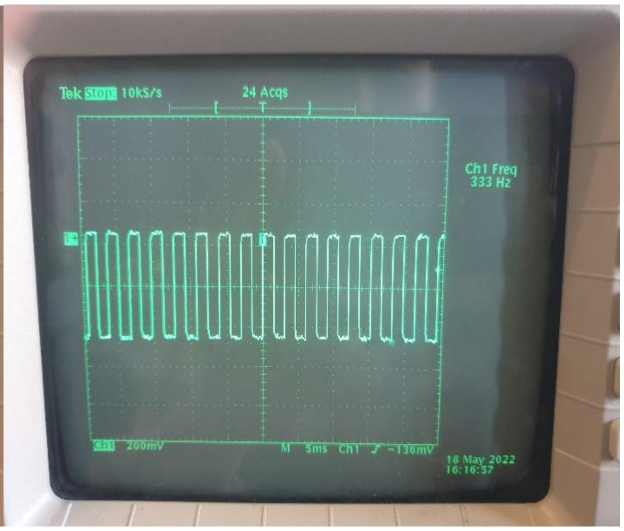


Figure 4. 333Hz

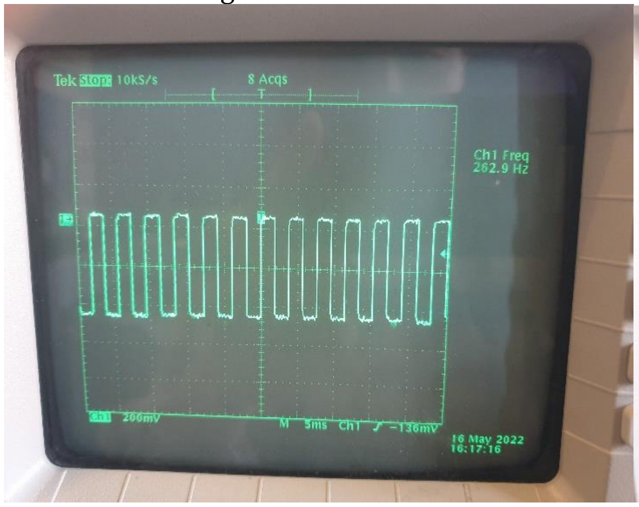


Figure 4. 263Hz

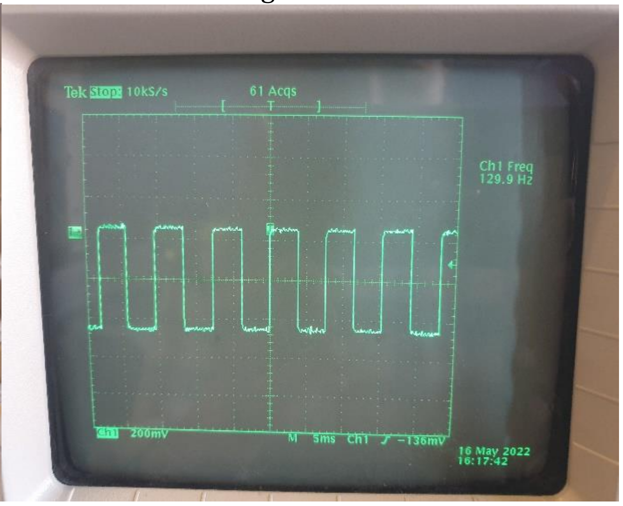


Figure 5. 130Hz

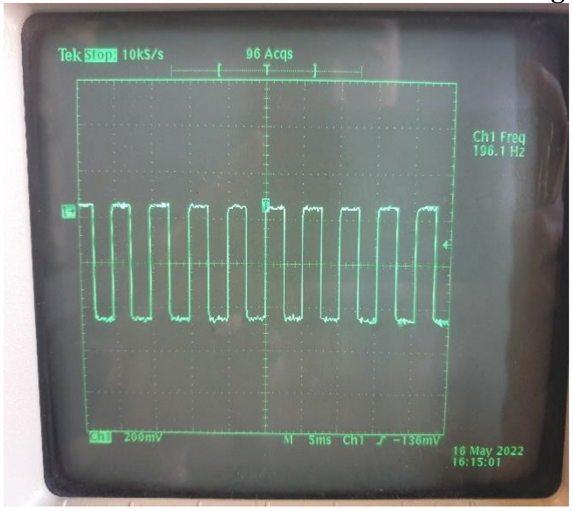


Figure 6. 196Hz.

The oscilloscope was employed again to validate an 8-bit hardware timer generating an interrupt every 2ms. Because PIN 6 was designated for OCR4A, we enabled the pin by DDRH |= 1 << DDH3 to measure the output waveform. Figure 7 below shows that the output of the pin generates a PWM signal for 4ms per cycle.

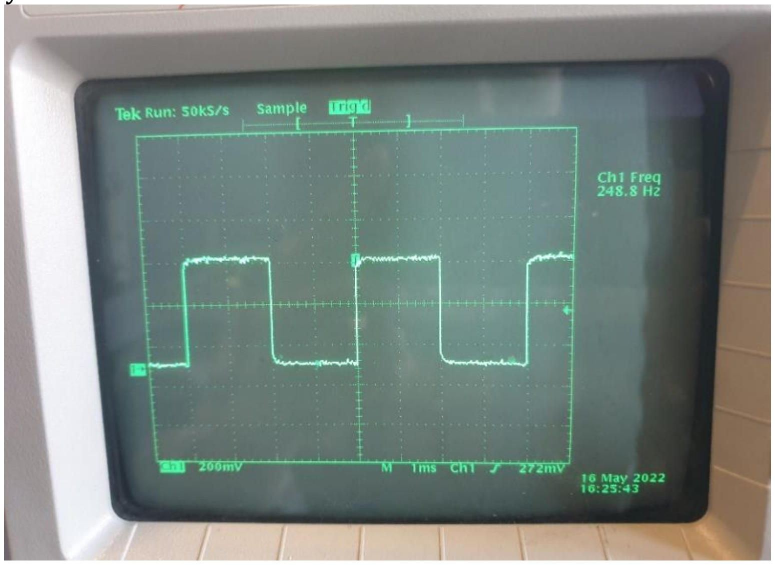


Figure 7. PWM signal for the 8-bit hardware timer. (HIGH for 2ms, LOW for 2ms, total of 4ms)

Using a timer on our mobile devices, we verified the countdown for 3 seconds for demo6. Serial.print functions were widely utilized to verify some of the inputs and outputs going into different functions in the coding-stage.

Experimental Results:

Demo1 was fairly straightforward because we have implemented a simple Round Robin in lab 2. The same CTC was initialized to make tones at different frequencies through firmware programming. The pre-scale was set to 8, but it could be set to a different value, which resulted in the same output. PIN5 was enabled for the output of the speaker because the PIN was designated to the system register. In each task function, static unsigned long time was created and employed for time counting. Loop function calls Task1() and Task2() every 1ms, so the time in the task functions increments by one every milli second. By setting the if statement condition to time == 1* LED_DURATION, the LED turns on for 250ms. It turns off for 750ms due to another if statement which has a condition, time == 0 and time is set back to 0. For Task2, the speaker plays each tone for 200ms as required. Different frequencies were stored in an array called melody. Each frequency goes into a function called set_OC3A_frequency() to make a tone. Two tasks operate concurrently because the loop function was set to call functions every milli second.

For demo2, a SRRI scheduler was employed in replacement for a RR scheduler. Two enumerate statements were also employed for the state of each task and flag. These states are constantly updated as needed for different purposes. Setups for LED, speaker, and interrupt were now located in functions, and they were called in the setup function for simplicity (started taking up too many lines). Each task was initialized in an array called runningTasks, which later was called through a for-loop in the loop function. An 8-bit hardware timer was set to generate an interrupt every 2ms. A ISR function was initialized with TIMER4_COMPA_vect to conduct a simple calculation, which guarantees that tasks repeat after 2 ms. For example, if tasks are performed in 1.5ms, the simple calculation in sFlag == pending performs and takes up 0.5ms such that the tasks are completed within 2ms. An ISR function must be kept simple because it is impossible to debug properly.

schedule_sync function was created and employed to synchronize the timing for tasks' sleeping time and running time, which were stored in an array called sleepTasks and timeTasks. In each task function, instead of a local static int variable called "time" that constantly increments by one to be compared with a defined duration in an if statement condition, a timeTasks array was employed to replace the functionality. The operation of Task1() and Task2() is the same as demo1.

For demo3, a DDS was used to run Task1() and Task2(). A struct is defined as TCB that contains all necessary information about the Tasks, which makes it simpler compared to SRRI. In the setup function, a task_load function is called and stores a task and a name. This task_load function loads up tasks to run. The name was later employed in a function called copy_tcb, which just points to variables and updates deadtasks to runningTasks. The name variable was also employed to start a task through a function called find_dead_task. This function finds a task stored in an array called deadTasks[].name by comparing with the name and returns the deadTasks. In the loop function, a function pointer is going through a for loop and points to loaded tasks to run. For the duration of time to run and sleep, instead of calling a timeTasks array and sleepTasks array in SRRI, runningTasks[currentTasks].time was used to compare with the duration of the time to turn the LED on and off for a certain period of time. Sleep function updates the task's statue to SLEEPING and the time to run or sleep for the tasks such that schedule_sync() constantly updates the runningTasks[currentTasks].sleepytime and runningTasks[currentTasks].time by decrementing by two, every time when loop runs.

Demo4 has an additional task, Task3(), compared with Demo2. This task3 displays count up by one unit every 100ms on the 7-segment LED. All of the other functions from demo2 were kept, and they performed the same functionalities in demo3. An additional task, Task3() was stored in the runningTasks array as 2. Twelve different digital output pins (PIN10 to PIN13 and PIN22 to PIN29) were initialized and employed to run the 7-segment LED. In Task3() function, static int count constantly incremented by one, and the value was converted and was stored in an array called digits[4] digit by digit. A two-dimensional array called seven_seg_counting takes digit[4] and displays the value on the 7-segment LED. How demo4 operates is same as demo2 with an additional task added to the array, runningTasks[2]. In the loop function, a function pointer going through a for loop from 0 to 10 points to every task stored every 2ms. All sleep functions in each task set the state of the task and updates the duration of the time for the task to be run. In schedule_sync function, this time gets updated by decrementing by two every 2ms.

For demo5, a DDS was employed to run Task4, which combined and modifies task2 and task3. When the music was being played, the frequency of the tone was displayed on the 7-segment LED. During the 4 seconds of pause, countdown for 4 seconds was displayed instead. Similar to the operation of demo3, load_up_task function loads up Task4 in the setup function. The start_task functions finds this task and runs it. Going through the loop function, a function pointer constantly points to task4 every 2ms. In task4 function, Task2, Task4_1, and Task4_2 were loaded up, and using several if statements, we set the time to run each task. Task4_1 displays the frequency of the tone being played on the 7-segment LED, and Task4_2 displays countdown from 40, which is equal to 4 seconds of pause.

For demo6, this is similar to Task4 in demo5. Task5 was created as a supervisor that starts and stops other tasks, Task1, Task2, Task4_2, and Task5_1. Task1() flashes an exteral LED. Task2() plays a speaker with different tones at different frequenices. Task4_2 displays countdown from 30, which means 3 seconds. Task5_1() displays a smile face on the 7-segment LED. All sub tasks were loaded up in Task5. Using several if statements, the state and the duration of each task were set in sleep function. Schedule_sync decrements these times until they are equal to 2 to stop. After each task was performed for a certain period of time, all the task array and their state were set to NULL and DEAD in the task_self_quit function to stop running except for Task1.

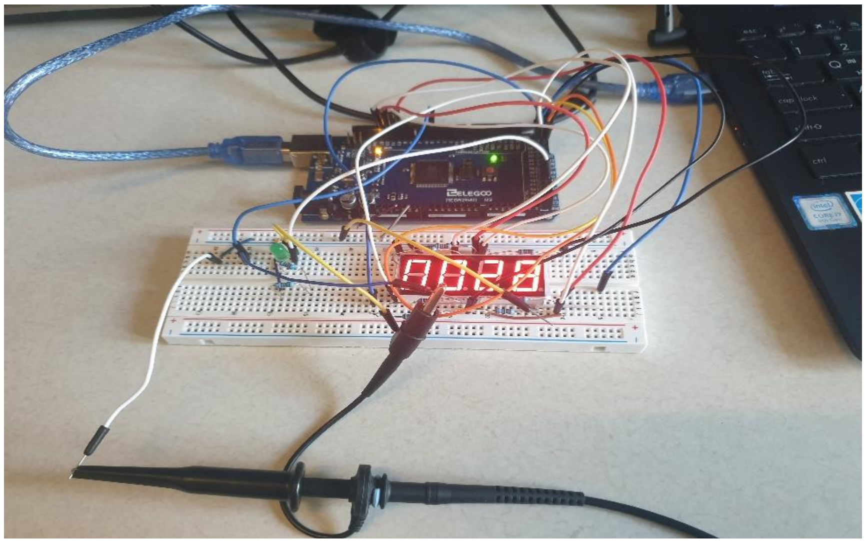


Figure 8. A picture of our hardware.

Code Documentation:

For lab 3, all code documentations have been done thoroughly using Doxygen.

1. Demo1.ino

A simple Round Robin scheduler was implemented to run Task1 and Task2 concurrently. Each Task has a static unsigned long time, which increments by one every milli second when a loop function calls the tasks. The same output compare mode was implemented to make tones at different frequencies. For demo1, we set the pre-scale to 8 and created a function called set_OC3A_freq, which takes a frequency at a time and converts it into a N value, using the following equation:

$$\frac{200{,}000Hz}{2 * target\ frequency} = N$$

Using several if statements, each task performs within the range of the time and the time resets to zero when it reaches the total time (i.e. LED runs for 1ms total, set the time back to zero and repeat.)

2. Demo2.ino

A synchronized Round Robin replaced a simple Round Robin for Demo2. One characteristic of synchronized Round Robin is that a bunch of arrays were created to store tasks, the running time, the sleeping time, and the state of the tasks. The state of tasks was constantly updated as tasks performed. Task1 function was stored in runningTasks[0], Task2 function was stored in runningTasks[1], and schedule_sync was stored in runningTasks[2]. These tasks were called in the loop function by a function pointer and run in 2 milli seconds. An 8-bit hardware timer was implemented to generate an interrupt every 2 milli seconds. In the beginning of the schedule_sync function, a simple calculation was implemented in a while loop with a condition, sFlag == PENDING.

An interrupt Service Routine(ISR) function interrupts and stops the simple calculation by this one line of code setting sFlag to DONE. This ensures that these tasks run within 2 milli seconds. Because the 8-bit hardware timer was set to generate an interrupt every 2 milli seconds, the amount of time that tasks run and sleep should either be incremented or decremented such that these times become synchronized without causing run-time errors.

3. Demo3.ino

Compared to SRRI, tasks were configured using Data-Driven Scheduler (DDS) in a struct called Task Control Block (TCB.) This struct contained all the necessary information about a task. The same 8-bit hardware and ISR were also implemented with a simple calculation in the schedule_sync function to guarantee that tasks run within 2 milli seconds. Tasks were loaded up in the "deadTasks" array and started to perform as they were being updated to runningTasks from deadTasks by a function called copy_tcb. As it goes through the loop function, a function pointer called tasks functions, and they performed their tasks.

4. Demo4.ino

Similar to Demo2, a SRRI scheduler was used for demo4. Thus, the structure is exactly the same as demo2 with an additional task. Using a seven segment LED display, it counts up and displays counting. In the task 3 function, an int variable called count increments by one every time task 3 is called by a function pointer in the loop function. This count variable goes into the convert function with an array called digits. This count variable is employed to calculate 1's, 10's, 100's, and 1000's and stored in the array. These counts stored in the array go into the display_segment function to display counting on the seven segment LED display.

5. Demo5.ino

Again, the structure of demo5 is exactly the same as demo5 because a DDS was implemented to schedule tasks. Task 4 is a combination of task 2 and task3. It is slightly different from the original version. To display the frequency while the music is being played, we created a volatile int called currentfreq to store the current frequency and displayed it on the seven-segment display, using the convert and display_segment function. Similarly, the same "count" variable was employed to decrement 4 seconds by 100 milli seconds and was displayed 4 seconds of count-down on the seven-segment.

6. Demo6.ino

The structure of demo6 is identical to demo 2 and demo5 as a DDS was implemented. This demo has a function that supervises all of the other task functions. It can start and stop the other tasks as needed. Task 1 runs all the time whereas task 2 runs at the start and stops after playing the theme twice. To display 3 seconds of count-down, the same method from demo5 was used. After task 2 runs for one final time, a smile face was displayed for 2 seconds and all of the tasks were set to turn off except for task 1. To be able to stop these tasks, a function called task_self_quit was called at the end of each sub function to set the state of the tasks to DEAD and remove tasks from the running task array.

Overall Performance Summary:

We did not struggle much with the implementation for an 8-bit hardware timer and ISR because we had already read the instructions thoroughly before conducting this lab. One part that was challenging was that the sleeping time and the time counting were supposed to decrement by two each tick because the 8-bit hardware timer generated an interrupt every 2ms. Demo2 was the hardest one to do as we needed some time to get used to populating tasks in a SRRI scheduler. Once we finally made demo2 working with SRRI, the other demos became easier as they required a little bit of modifications from SRRI scheduler. We also struggled with stopping all tasks for demo6. We were aware that we needed a function like halt_me() function, but it was hard to code it incorporating with the state of tasks (DEAD). With much time and effort that we put in this lab, we believe that we achieved all of the learning objectives from this lab.

Teamwork Breakdown:

For Lab 3, both Kejin and I conducted all of the experiments individually for learning purposes. We both were responsible for building architecture, coding, debugging, integrating hardware and software by ourselves. Whenever we had time, we set up a meeting in the lab, discussed, and shared our codes and ideas with each other. This way, we could both have a chance to perform all of the procedures by ourselves and could help each other whenever we got stuck. We are going to stick with this plan for the first part of lab 4 as well. We are planning to have a meeting to discuss what to do for the project this Wednesday, May 23rd.

Discussion and conclusions:

We had a chance to use an ISR and 8-bit hardware timer and set the exact running time for all tasks. In this lab, we set it to 2ms. Compared to a simple Round Robin scheduler, one of the advantages of SRRI over simple RR is that we can set the exact running time to whatever time we want, whereas delay(1) employed in Round Robin does not run tasks exactly within 1ms. This might not be an issue when we deal with non-time sensitive cases, but it would be important to use it for some of the time sensitive hardware, such as aviation, aerospace, and defense technologies. All of the schedulers that we have implemented in this lab were non-pre-emptive schedulers. However, using a SRRI and DDS makes the codes long and difficult. Starting lab 4, we are going to be learning about pre-emptive schedulers (Free-RTOS.) We are looking forward to learning more about it and the other communication protocols for embedded systems.