Lab Assignment 3 **CSE/ECE 474** **Spring 2024**

Prof. Vikram Iyer[1] University of Washington

Learning Objectives

With successful completion of this lab, you will be able to run several simultaneous tasks and

- Implement and explain a round-robin (RR), non-preemptive scheduler
- Implement and explain a RR scheduler using function pointers and timer synchronization, and a sleep() function.
- Implement and explain a scheduler using Task Control Blocks (TCBs) for each task.
- Implement a simple Interrupt Service Routine (ISR) and use it to get hardware precision for your scheduler cycle synchronization.

Turn-In Requirements are included at the bottom of this document.

Equipment

1) Arduino Mega Microcontroller board
2) External Arduino power supply
3) LEDs and 250 Ohm resistors
4) Small 8-Ohm Speaker
5) 4-digit 7-segment LED display

Preparation

Consult the 474 Interrupt Concepts Document and lecture material for important details on scheduling and programming interrupts.

Technical Requirements

In Lab 3 we will implement 5 tasks with different schedulers. To demonstrate mastery of the learning objectives you must integrate the following tasks with specific scheduler types into **Specific Required Configurations (see table below), but not all possible combinations are required.**

In Lab 3 you may use DDR/PORTx manipulation **OR** pinMode()/digitalWrite(). Do not use tone().

Thanks for valuable guidance from Prof. Blake Hannaford

Tasks

1) **Task 1**: Flash an external LED for 250 ms every 1 second. (e.g. ON for 250ms, OFF for 750ms).
2) **Task 2**: Make the speaker emit the beginning of the Mario theme song (see Appendix) or a short theme song of your choice using the hardware timer techniques from Lab 2. If you choose your own please play something different from Mary had a little lamb. Task 2 should play the song and then sleep for 4 seconds, then play the song again. Directly manipulate timers to generate the signal, do not use Arduino's tone() function.
3) **Task 3:** Make the 7-Segment LED display (info below) count up by 1 unit every 100 ms.
4) **Task 4:** After you demo tasks 1-3, Task 4 is a new task that combines and modifies tasks 2 and 3 as follows. While no music is being output, make it so the 7-segment display indicates a countdown in 10ths of a second until the next time the theme is played (check the **HINT below**). During the music, display the frequency (in decimal Hz.) of the current tone on your 7-Segment display. (your code doesn't have to measure the frequency, you have it in your code somewhere).
5) **Task 5**: This task will be a supervisor that starts and stops other tasks. Perform the tasks according to this schedule:
    a) Task 1 runs all the time.
    b) Task 2 runs at the start but **stops** after playing the theme 2 times.
    c) After Task 2, start a **count-down (on the 7-segment display) for 3 seconds and then restarts task 2 (original definition above) for one final time.**
    d) Display a "smile" for 2 seconds (See Appendix B) (code this as a new task to be controlled by Task 5). The 2 second interval begins after completion of part 5c.
    e) Stop all tasks, except "a) Task 1" in this list.

HINT: it may be easier (i.e. cleaner code) to break some tasks up into more than one task, which is encouraged. For example, counting down is logically separate from scanning and updating the 7-segment displays. If you break tasks into multiple parts, please name your tasks with decimals. For example, the 2nd task belonging to "**Task 2"** would be called Task 2.2. (C won't allow the period so use 2p2 or 2_2).

Scheduler Types

We will implement three schedulers for a variable number of tasks. None of the schedulers will be pre-emptive.

1. **Round Robin (RR)** - like the basic Arduino `loop()` function.
   Your scheduler should be a simple loop beginning with:
   ```
   while(1) {
   ```
   (write this within the Arduino loop() function). Then call your tasks in order and repeat. Add a delay loop to cause the overall period P to meet requirements.

Thanks for valuable guidance from Prof. Blake Hannaford

Example RR Scheduler setup:

```
while(1){
    task1();
    task2();
    ...
    taskN();
    // short delay
    for(int i=0; i< delay_value; i++){
        // code that wastes a small amount of time
    }
    // OR you can use delayMicroseconds() instead
}
```

2. **Synchronized Round Robin with ISR (SRRI) -**

2.1. This scheduler will use an ISR, and replace the function calls from the RR scheduler with a pointer or index which should be incremented through:

- an **array** of 10 function pointers (so you can have up to 10 tasks).
- Each of the function pointers in the array shall point to a task except,
- a `Null` pointer shall denote the end of the actual tasks in the task array.
- Each task must have a state belonging to the set
  `{ READY, RUNNING, SLEEPING }`
  You should create an array that stores the state for each task.
- A `sleep_474(int t)` function must be defined that tasks can call while they are running. When a task calls `sleep_474(t)`, it should be stopped and its state should be changed to `SLEEPING`. Also, store the value of `t` (milliseconds) so that you can associate it with the sleeping task. If it is a task's "turn" to run, but it is in the `SLEEPING` state, then do not run it.
- **Every 2 milliseconds,** the waiting time for any `SLEEPING` task must be decremented by 2. (yes, sleep calls will only be accurate to within 2ms.)

If the waiting time for a task goes below 2ms, then its state must change from `SLEEPING` to `READY`.

2.2. Before running the SRRI scheduler, your code must initialize an 8-bit hardware timer to **generate an interrupt for each "tick" (2ms)**. As with any interrupt, you must **write an Interrupt service routine (ISR)** for this interrupt to do something at each clock interrupt. As always, your ISR must be a very small piece of code. Guidance for this lab is to write your ISR to change a volatile global int (let's call it `sFlag`) between the states { `PENDING, DONE` }. The ISR should perform one simple function. Set `sFlag = DONE`. The ISR can also keep track of time by incrementing a time int (or long int) but you might not need that.

Thanks for valuable guidance from Prof. Blake Hannaford

Hint: ISRs can be notoriously tricky to program and debug. That's why the advice to **Keep it Simple!** Specifically, consider a minimalistic setup for initial ISR debugging in which you have only one very simple task plus the ISR running and nothing else. Maybe slow the interrupts down at first (10 per second?). A simple way to test the timer + interrupt is to use the interrupt to toggle an LED. Remember print statements take a significant amount of time to execute and may make it harder to debug if you put these in your ISR. Again, consider instead setting a flag and doing other operations in the loop. After you are confident in the interrupt and ISR, then introduce "real" tasks and increase the rate of interrupts.

2.3. The **last** task in the function pointer list must be a task called `schedule_sync()`. This task must do an infinite loop as long as `sFlag == PENDING`. When `sFLAG == DONE, schedule_sync()` must
   2.3.1. Update remaining sleep times of all sleeping tasks (we set the timer interrupt for 2ms so we know what is deltaT).
   2.3.2. Wake up any sleeping task whose sleep time goes to 0.
   2.3.3. Reset `sFlag` to `PENDING`
   2.3.4. `return`

Now your scheduler will complete exactly one loop every 2ms *unless* your tasks take more than 2ms to return to `loop()` (you should make sure that your tasks run for short bursts so this does not happen).

3. **Data-Driven Scheduler (DDS)** - Define a **Task Control Block** (**TCB**, see lecture notes) for each task. The TCB should be a `struct` containing all information about a task. Specifically include (in addition to the state information from SRRI above).
   3.1. Unique ID code for the task.
   3.2. A string name of the task (up to 20 characters) (Example: "LED Flasher"). `
   3.3. The total number of times the task has been started/restarted.

   Implement the scheduler by creating a list of all the TCBs and looping through them. Start each task or not depending on its Status.
   A new function called `task_self_quit()` must be defined which allows a task to terminate itself by manipulating its TCB, and a similar function called `task_start(TCB *task)` so that a task can start up another task. The scheduler must have a pointer or index which knows which task is running. Thus it knows which task called `task_self_quit()`.
   3.4. Add a new task status: `DEAD`
   3.5. If a task calls `task_self_quit()` it should be removed from the `TCB` list and its status set to `DEAD`. Keep a separate list of the dead tasks.
   3.6. Another task can revive a dead task by calling `task_start( tsk )` where tsk here is a pointer to the `TCB` of a `DEAD` task. The `task_start(tsk)` function should do this by changing tsk's status to `READY`.

Thanks for valuable guidance from Prof. Blake Hannaford

Specific Required Configurations and Demos

Demonstrate the lab by showing the following specific combinations of Tasks and Schedulers

Demo Number	Tasks Running at start	Scheduler Type	Notes or special instructions
1	T1, T2	RR	
2	T1, T2	SRRI	
3	T1, T2	DDS	
4	T1, T2, T3	SRRI	
5	T4	DDS	
6	T5	DDS	(T5 will start and stop additional tasks).

Demo Verbal Questions:

1. What are the advantages and disadvantages, if any, of using interrupts and ISRs as opposed to using flags with “blocking” code (e.g. delay())? Are ISR driven tasks always faster?
2. Why should ISR functions be simple and small?
3. In this lab you used a Timer Interrupt. Which Timer Interrupt did you use? For each of the following Interrupt Definitions, give the relative priority with respect to the Timer Interrupt you used (higher or lower). (HINT: Check the interrupt vector table)
    a. SPI Serial Transfer Complete
    b. RESET
    c. Pin Change Interrupt 2
    d. External Interrupt 7
    e. Timer/Counter(n) overflow (n is the number of the Timer/Counter you used)
4. Why is the SSRI scheduler advantageous with respect to the RR scheduler?
5. Why would you use a TCB?

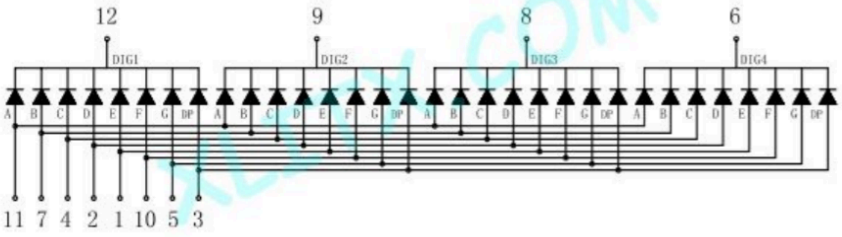
Thanks for valuable guidance from Prof. Blake Hannaford

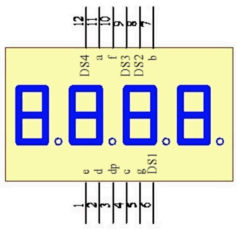
Recommended Procedures

(suggestions for breaking down the lab, step-by-step testing, etc.)

Wiring and driving the 7-segment display

Like the 8x8 display, your 7-segment display is either common-anode common cathode. Here is a typical schematic:





of a common cathode display with 4 digits. Pins 12, 9, 8, and 6 drive the negative side of all segments in a specific digit. You drive the display by scanning the digits and the segments (pins 11, 7,4,2,1,10,5,3) similar to the rows and columns of the 8x8 RAW matrix. It's the same idea as scanning rows and columns, but instead, for each digit, you must light up the segments required to indicate each number.

7-Segment software

There's tons of good info online on the basics of driving your display [Example]. The devil is in the details like getting your wiring right, and importantly, interfacing your LED driver to the required scheduling system. The code listed in most online examples shows you what to do at a low level, but will not work with schedulers. Consider implementing the following functions to aid task building

- Convert an `int` to 4 digits
- Use a number in the range 0-9 to select an array indicating which of the 7 segments light up.
- Activate the output bit for a digit (0-3).
- Activate the 7 segment bits according to the array above.

You can easily debug these building blocks and then it's easier to put them together into a continuous display. As a starting point, light each digit ON for 2ms and OFF for 3ms (5ms per digit). Slower than that can be good for debugging but makes a flickering display.

Thanks for valuable guidance from Prof. Blake Hannaford

Lab 3-Specific Turn-In Requirements

Special Grading Criteria

- See Lab Guidance.

Report: Turn in a PDF report following the document template and the Lab Overview/Guidance CSE474 documents. Include a section detailing each partner's contributions, and be specific. This PDF should be turned in as a standalone document, not in a .zip file.

Demo: Demonstrate all items from "Specific Required Configurations and Demos" and "Demo Verbal Questions" sections below.
Demo your project in the lab with an instructor. Both team members must be present and able to answer questions.

Code: Turn in ALL code in .ino and .c files (if applicable). Submit a zipped folder containing all .ino and .c files, and keep Arduino files in their sketch folder. Comment out (but do not delete) code which might be required for intermediate steps but is not required for the final code. This will help us give you partial credit!

Appendix A: Beginning of Mario theme

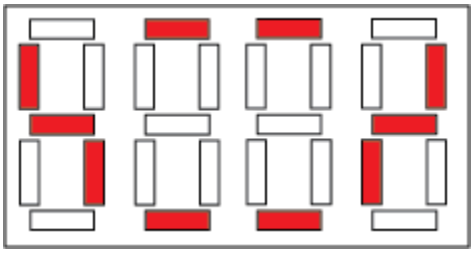
The notes below correspond to the beginning of the Mario theme. By converting these to the correct timer counts you can play the sounds just like Lab 2. [Source with full song]

```
// We recommend a duration of 100 ms per note

#define E 659 // Hz
#define C 523 // Hz
#define G 784 // Hz
#define g 392 // Hz
#define R 0   // Rest, play no sound

int song[] = {E, R, E, R, R, E, R, R, C, R, E, R, R, G, R, R, R, R,
R, g, R};
```

Appendix B: "Smile on LED display" (If you've got a "better" design -- go for it!)



Thanks for valuable guidance from Prof. Blake Hannaford