INTRODUCTION

This lab primarily focuses on implementing multiple concurrent tasks within embedded systems, using an Arduino Mega microcontroller. The process involves understanding scheduling and developing three types of non-preemptive schedulers: Round Robin (RR), Synchronized Round Robin with Interrupt Service Routine (ISR) or SRRI, and Data-Driven Scheduler (DDS). These schedulers manage a range of tasks, each having unique functions.

First, we apply a basic RR scheduler to trigger an external LED to flash periodically while playing the Mario theme song. This function is then reproduced using both SRRI and DDS schedulers. Adding complexity, a third task increases a counter every 100ms and displays that value on a seven-segment display, using the SRRI scheduler.

Next, a task plays the Mario theme song on an external speaker while displaying the frequency of the played notes on a seven-segment display. Finally, a DDS scheduler controls a series of simultaneous events: flashing the LED display, playing the Mario theme for a specific duration, and showing a pattern on the seven-segment display.

Throughout the lab, our focus remains on how interrupts are programmed, their advantages, and potential drawbacks in specific scenarios. We also employ structures like Task Control Blocks (TCBs) to store task-specific information.

METHODS AND TECHNIQUES

In this lab, several methods and techniques were employed to meet the objectives. For the Round Robin (RR) scheduler, a straightforward while loop was used to perpetually cycle through the tasks. The delayMicroseconds() function was utilized to increment different timers at a precise frequency, serving as task-specific timers.

With the Synchronized Round Robin with Interrupt Service Routine (SRRI) schedulers, Interrupt Service Routines (ISR) were used to either toggle the state of a flag or increment a counter. The toggled flag facilitated the incrementing of multiple task-specific counters without risking the omission of an increment, since the if statement, responsible for increasing the counters, would also reset the flag to zero. Conversely, incrementing a counter in the ISR provides a more accurate clock. However, this method may cause some tasks to miss certain counter values, potentially affecting their functionality.

Regarding the Data-Driven Scheduler (DDS), an array of Task Control Block (TCB) structs was implemented, storing various task parameters. Through the use of functions like task_start() and task_quit(), control over task operation was streamlined. In certain instances, an ISR with a flag was used to assist with internal task-specific counters to achieve higher quality results than those without the ISR.

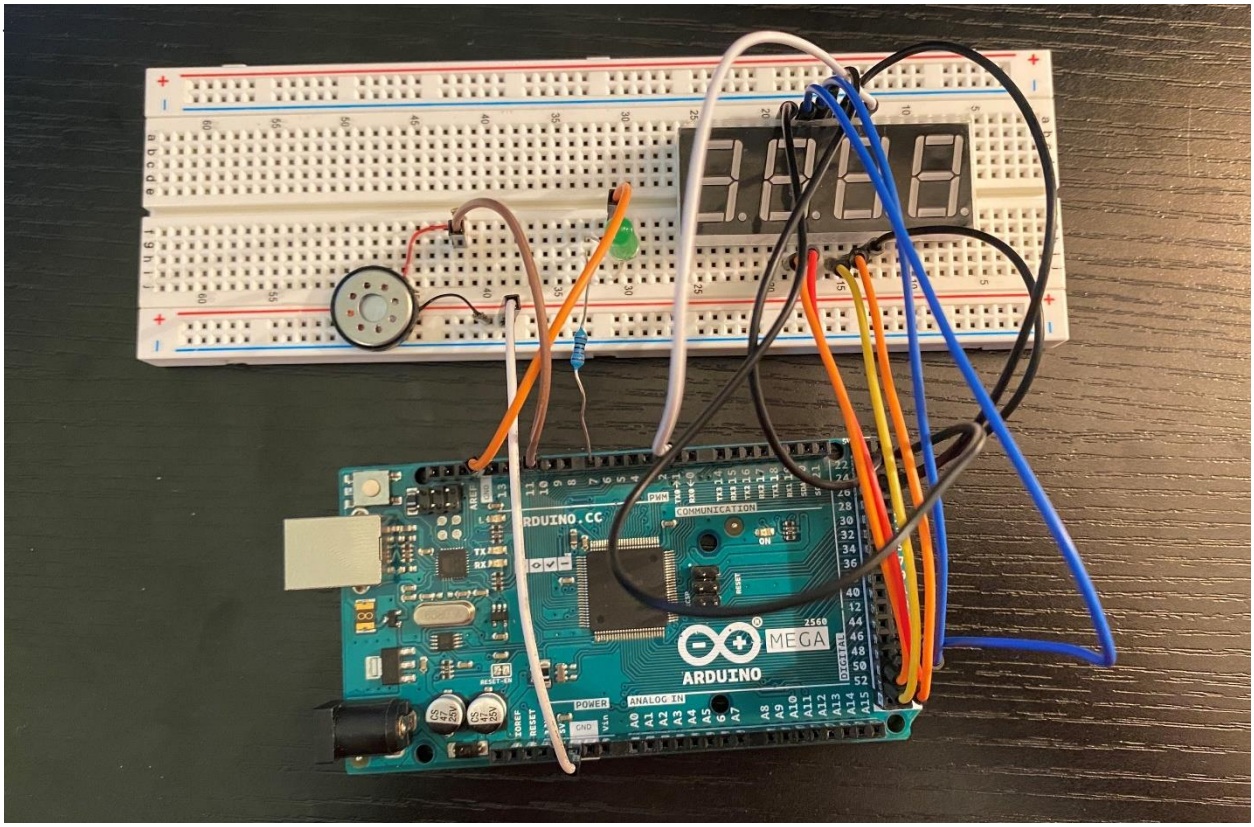

Figure 1 The experimental setup for this lab

EXPERIMENTAL RESULTS

DEMO 1

At the start of the first demonstration, and in all the ones that followed, I first defined several parameters. These were necessary for the proper running of the firmware and to make the code easier to read. Then, I set up a variety of variables such as task-specific timers and state flags. After this setup, I wrote two tasks: flashLED() and playSpeaker().

These tasks were called within an endless while(1) loop found in the void loop() function of the sketch. Inside this while loop, the flashLED() task ran first, then the playSpeaker() task. After these tasks ran, task-specific counters were increased. Finally, I used the delayMicroseconds() function to create a specified frequency at which the while loop iterated.

Upon running this sketch, I could observe that the LED flashed with a period of one second and that the speaker played the recognizable Mario theme song, therefore verify the proper functioning of the RR scheduler.

DEMO 2

In this part of the lab a Synchronized Round Robin with Interrupt Service Routine (SRRI) framework was structured to execute the flashLED() and playSpeaker() tasks. Using the lab lecture's starter code and the general sketch file layout from Professor Makhsous, two tasks: sleep_474() and schedule_sync(), were implemented. These tasks controlled the SLEEP and READY states of tasks and synchronized their operation.

For effective synchronization in schedule_sync(), an Interrupt Service Routine (ISR) was implemented using the Arduino's TIMER4 in Clear Timer on Compare (CTC) mode. The ISR would toggle a

flag every 100 microseconds upon a compare event, thereby acting as a 10KHz clock. In the loop() function of the Arduino, a while loop was created to check if the current task's state was RUNNING. If so, the task would be executed; otherwise, it would be bypassed. Upon completion of this while loop, task-specific timers were incremented, and the firmware looped back to the start of the task array.

Upon running this sketch, I could observe that the LED flashed with a period of one second and that the speaker played the recognizable Mario theme song, therefore verify the proper functioning of the SRRI scheduler.

DEMO 3

In this lab section, a Data-Driven Scheduler (DDS) framework was utilized to concurrently execute the flashLED() and playSpeaker() tasks. Initially, a Task Control Block (TCB) structure was implemented to encapsulate various task parameters. An array was then created to hold these TCB structures, one for each task.

A scheduler was implemented, which differed from the Synchronized Round Robin with Interrupt Service Routine (SRRI). This scheduler verified if the current structure in the array was in a READY or DEAD state, executing it only in the READY state. Two more functions were added: task_start(TCB* task) and task_self_quit(). The task_start(TCB* task) function, when invoked within a task, could activate another task. The task_self_quit() function would terminate the currently running task, but could only be invoked from within the task intended for termination. In the loop() function, only the scheduler was invoked. This scheduler was responsible for executing the tasks and incrementing the various task-specific counters/timers. A TIMER and an Interrupt Service Routine (ISR) were also implemented to enable the tasks to run more efficiently at higher frequencies.

Upon running this sketch, the LED blinked with a period of one second, and the speaker played the Mario theme song. This observation confirmed the proper functioning of the DDS scheduler.

DEMO 4

In this lab segment, a third task, sevSegCounter(), was introduced to control a seven-segment display, making it increment every second. To facilitate this, the Synchronized Round Robin with Interrupt Service Routine (SRRI) structure from the second demo was employed, with the addition of the new sevSegCounter() task to the taskScheduler[] array. Apart from this task addition and the implementation of sevSegCounter(), no other changes were made to the demo 2 code.

Upon running this sketch, the LED flashed every second, the speaker reproduced the Mario theme song, and the seven-segment display incremented by one every 100 milliseconds. These outcomes validated the successful operation of the SRRI scheduler when managing three tasks, as opposed to the two tasks handled in the second demo.

DEMO 5

In this portion of the lab, the Data-Driven Scheduler (DDS) framework was utilized to incorporate a new task. This task was assigned to play the Mario tune and concurrently display the current note's frequency on a seven-segment display. Once the tune concluded, the seven-segment display initiated a countdown from 40, decrementing every 100 milliseconds. Upon reaching zero, the scheduler looped back to replay the tune and display the corresponding note frequencies.

To implement this functionality, I modified the code from Demo 3, substitifying the flashLED() task with a new task, sevSegCounter(). This task executed the aforementioned functionality. Aside from these changes, the code remained consistent with the format presented in Demo 3.

Running this sketch resulted in the speaker playing the Mario theme song, with the seven-segment display showing either the frequency of the current note or counting down from 40 when the tune was not playing. These results confirmed the DDS scheduler's successful management of the assigned tasks.

DEMO 6

In the final stage of this lab, the objective was to utilize a Data-Driven Scheduler (DDS) to operate a specific sequence of tasks. To accomplish this goal, I used the same framework as in Demo 3, with the addition of further task structs to the task array.

Furthermore, I implemented a new task, task5(), responsible for managing the task sequence. To optimize this task management, I restructured the task_self_quit() function into a new function, task_quit(TCB* task). This new function parallels task_start(TCB *task) in its structure but offers the ability to terminate a task from any other task, unlike task_start() which reactivates a task. This adjustment was necessary to enable task5() to efficiently manage other tasks by either activating or terminating them based on specific conditions.

Upon executing this sketch, the Arduino outputted the correct sequence of events, thereby affirming the effective functionality of the DDS scheduler for more intricate schedules compared to the tasks previously presented.

INTENTIONALLY LEFT BLANK

CODE DOCUMENTATION

DEMO 1

```
/*
* University of Washington
* ECE 474,  7/18/23
* Jason Bentley
*
* Lab 3 - DEMO 1
*
* Acknowledgments: LAB 3 starter code from Sep Makhsous
*/

#define LED_PIN  7
#define TONE_PIN 10

//define the notes for the tune
#define NOTE_E 659
#define NOTE_C 523
#define NOTE_G 784
#define NOTE_g 392
#define NOTE_R 0

//100ms -> based on a 100us delay, the length of time each note should play
#define NOTE_TIME  1000

int LED_timer = 0; //counter to store length of time which LED has been in a
particular state
int LED_state = 0; //counter to store the current state of the LED

long tune_timer   = 0; //counter to store length of time which speaker has been
in a particular state
int tune_elem     = 0; //counter to store the the position of the current note in
the mario[] array
int tune_state    = 0; //counter to store the current state of the speaker
int note_timer    = 0; //counter to store the length of time which a note has
been playing

int mario[] = {NOTE_E, NOTE_R, NOTE_E, NOTE_R, NOTE_R, NOTE_E, NOTE_R, NOTE_R,
NOTE_C, NOTE_R,
               NOTE_E, NOTE_R, NOTE_R, NOTE_G, NOTE_R, NOTE_R, NOTE_R, NOTE_R,
NOTE_R, NOTE_g,
               NOTE_R};
```

```
void setup(){
    pinMode(LED_PIN,  OUTPUT);
    pinMode(TONE_PIN,  OUTPUT);
}


void loop() {
  while (1) {
    //flash the external LED
    flashLED();

    //play the mario tuner
    playSpeaker(mario);

    //increment all timers
    tune_timer++;
    note_timer++;
    LED_timer ++;

    //delay
    delayMicroseconds(100);
    }
}


/***********************************************
 * void flashLED()
 *    Arduments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function flashes an external LED for a hardcoded amount of time.
 *    The LED will be on for 250ms and off for 750ms
 *
 *    Acknowledgments: N/A
 */
void flashLED(){
    if(LED_state == LOW && LED_timer == 7500 || LED_timer == 0){
        digitalWrite(LED_PIN, HIGH);
        LED_state = HIGH;
        LED_timer = 0;
    }else if(LED_state == HIGH && LED_timer == 2500){
        digitalWrite(LED_PIN, LOW);
        LED_state = LOW;
```

```
        LED_timer = 0;
    }

    return;
}

/***********************************************
 * void playSpeaker(int[])
 *
 *    Argument - the int array holding the notes of a tune in the
 *               order which they will be played.
 *
 *    Returns - no returns, this is a void function
 *
 *    This function calculates the half-period of a given frequency
 *    in the passed int array. It then itterates through this array
 *    and plays each note for a NOTE_TIME amount of time. After all
 *    of the notes are played there is a hardcoded 4sec pause.
 *
 *    Acknowledgments: N/A
 */
void playSpeaker(int tune[]) {
    //calculate the half period of a frequency
    int period_half = 5000/(tune[tune_elem] + 1);

    //if the speaker is off
    if(tune_state == LOW && tune_timer == period_half && tune_elem != 21){
        digitalWrite(TONE_PIN, HIGH);
        tune_timer = 0;
        tune_state = HIGH;
        return;
    }

    //if the speakers it on
    if(tune_state == HIGH && tune_timer == period_half && tune_elem != 21){
        digitalWrite(TONE_PIN, LOW);
        tune_timer = 0;
        tune_state = LOW;
        return;
    }

    //increment to next note if current note has played for NOTE_TIME amount of
time
    if(note_timer == NOTE_TIME && tune_elem != 21){
        tune_elem++;
```

```
        note_timer = 0;
        tune_timer = 0;
        return;
    }

    //if all notes have been played wait 4sec, then go back to beginning of
tune[] array
    if(tune_elem == 21 && tune_timer == 40000){
        tune_elem = 0;
        tune_timer = 0;
        note_timer = 0;
        return;
    }
}
```

INTENTIONALLY LEFT BLANK

DEMO 2

```
/*
* University of Washington
* ECE 474,  7/18/23
* Jason Bentley
*
* Lab 3 - DEMO 2
*
* Acknowledgments: LAB 3 starter code from Sep Makhsous
*/

// Define necessary variables and constants
#define LED_PIN  7
#define TONE_PIN 10

//define max size of task array
#define MAX_SIZE 10

//define the possible states a task can have
#define READY    0
#define RUNNING  1
#define SLEEPING 2
#define PENDING  4
#define DONE     5

//define the notes for the tune
#define NOTE_E 659
#define NOTE_C 523
#define NOTE_G 784
#define NOTE_g 392
#define NOTE_R 0

//100ms -> based on a 100us delay, the length of time each note should play
#define NOTE_TIME  1000 //100ms

int LED_timer = 0; //counter to store length of time which LED has been in a
particular state
int LED_state = 0; //counter to store the current state of the LED

long tune_timer   = 0; //counter to store length of time which speaker has been
in a particular state
int tune_elem     = 0; //counter to store the the position of the current note in
the mario[] array
int tune_state    = 0; //counter to store the current state of the speaker
```

```
int note_timer    = 0; //counter to store the length of time which a note has
been playing

int timerCounter   = 0; //a global counter acting as a timer
volatile int sFlag = 0; //a flag rasied on interupt
long x = 0;             //a dummy variable for schedule_sync() to do nothing wile
sleeping

int mario[] = {NOTE_E, NOTE_R, NOTE_E, NOTE_R, NOTE_R, NOTE_E, NOTE_R, NOTE_R,
NOTE_C, NOTE_R,
               NOTE_E, NOTE_R, NOTE_R, NOTE_G, NOTE_R, NOTE_R, NOTE_R, NOTE_R,
NOTE_R, NOTE_g,
               NOTE_R};

//initialize a pointer to the array holding the tasks
void (*taskScheduler[MAX_SIZE]) = {0};

int taskSleep[MAX_SIZE]; //Initialize an array for storing sleeping tasks
int taskState[MAX_SIZE]; //Initialize an array for storing non-sleeping tasks
int currentTask = 0;     //an int to hold the position of the current running
task in the array



void setup() {
  // Set up pins as outputs for the speaker and LED
  pinMode(LED_PIN,  OUTPUT);
  pinMode(TONE_PIN,  OUTPUT);

  //populate task array with tasks
  taskScheduler[0] = flashLED;
  taskScheduler[1] = playSpeaker;
  taskScheduler[2] = schedule_sync;
  taskScheduler[3] = NULL;

  //Set up timer and its configurations for ISR (CTC mode, 10KHz, prescaler = 64)
  TCCR4A = 0;
  TCCR4B = (1 << WGM42) | (1 << CS41) | (1 << CS40);
  OCR4A = 25; //100uS interupt, 10Khz
  TIMSK4 |= (1 << OCIE4A);
  sei();
}


void loop() {
```

```
  // Enter the loop when a task is present in the taskScheduler
  while (taskScheduler[currentTask] != NULL) {

    // Run the task if it is not in the SLEEPING state
    if (taskState[currentTask] != SLEEPING){
        function_ptr(taskScheduler[currentTask]);

        // Change the task state to RUNNING
        taskState[currentTask] = RUNNING;
    }

    //increment to next task and timerCounter "clock"
    currentTask++;
    timerCounter++;
  }

  //go back to beginning of task array and increment task-specific timers
  currentTask = 0;
  tune_timer++;
  note_timer++;
  LED_timer ++;
}


/************************************************
 * void flashLED()
 *    Arduments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function flashes an external LED for a hardcoded amount of time.
 *    The LED will be on for 250ms and off for 750ms
 *
 *    Acknowledgments: N/A
 */
void flashLED(void *p){
    if(LED_state == LOW && LED_timer == 7500 || LED_timer == 0){
        digitalWrite(LED_PIN, HIGH);
        LED_state = HIGH;
        LED_timer = 0;
    }else if(LED_state == HIGH && LED_timer == 2500){
        digitalWrite(LED_PIN, LOW);
        LED_state = LOW;
        LED_timer = 0;
    }
```

```
    return;
}


 /***********************************************
 * void function_ptr(function pointer)
 *
 *    Argument 1 - the function being pointed to
 *
 *    Returns - no returns, this is a void function
 *
 *    This function serves as a function pointer to any given task to execute
 *
 *    Acknowledgments: LAB 3 starer code
 */
void function_ptr(void* function()){
    function();
}


/***********************************************
 * void playSpeaker()
 *
 *    Arduments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function calculates the half-period of a given frequency
 *    in the mario[] array. It then itterates through this array
 *    and plays each note for a NOTE_TIME amount of time. After all
 *    of the notes are played there is a hardcoded 4sec pause.
 *
 *    Acknowledgments: N/A
 */
void playSpeaker(void *p) {
  //calculate the half period of a frequency
  int period_half = 5000/(mario[tune_elem] + 1);

  //if the speaker is off
  if(tune_state == LOW && tune_timer == period_half && tune_elem != 21){
    digitalWrite(TONE_PIN, HIGH);
    tune_timer = 0;
    tune_state = HIGH;
    return;
  }
```

```
  //if the speakers it on
  if(tune_state == HIGH && tune_timer == period_half && tune_elem != 21){
    digitalWrite(TONE_PIN, LOW);
    tune_timer = 0;
    tune_state = LOW;
    return;
  }

  //increment to next note if current note has played for NOTE_TIME amount of
time
  if(note_timer == NOTE_TIME && tune_elem != 21){
    tune_elem++;
    note_timer = 0;
    tune_timer = 0;
    return;
  }

  //if all notes have been played wait 4sec, then go back to beginning of tune[]
array
  if(tune_elem == 21 && tune_timer == 40000){
    tune_elem = 0;
    tune_timer = 0;
    note_timer = 0;
    return;
  }

}


/************************************************
 * void sleep_474(int)
 *
 *    Argument - The sleep time in milliseconds
 *
 *    Returns - no returns, this is a void function
 *
 *    Function that controls the sleep time of a task
 *
 *    Acknowledgments: LAB 3 starter code
 */
void sleep_474(int t) {
  // Set the sleep time of the current task
  taskSleep[currentTask] = t;
```

```
  // Change the task state to SLEEPING
  taskState[currentTask] = SLEEPING;
}


/*********************************************
 * void schedule_sync()
 *
 *    Argument - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    Function that synchronizes the schedule and tasks while in the PENDING
state
 *
 *    Acknowledgments: LAB 3 start code
 */
void schedule_sync(void *p) {
  // Enter an infinite loop when sFlag is PENDING
  while(sFlag == PENDING){
    x += 1;
  }
  // Update the sleep time of the task
  sleep_474(timerCounter);
  taskSleep[currentTask] -= 2;
  // Wake up any sleeping task if the sleep time is negative
  if (taskSleep[currentTask] < 0){
    taskState[currentTask] = READY;
  }
  // Reset the sFlag to PENDING
  sFlag = PENDING;
}


/**
 * Interrupt Service Routine for TIMER0_COMPA_vect
 * Sets the flag to DONE
 */
ISR(TIMER4_COMPA_vect) {
  sFlag = DONE;
  TIFR4 |= (1 << OCF4A);
}
```

DEMO 3

```
/*
* University of Washington
* ECE 474,  7/18/23
* Jason Bentley
*
* Lab 3 - DEMO 3
*
* Acknowledgments: LAB 3 starter code from Sep Makhsous
*/

// Define necessary variables and constants
#define LED_PIN  7
#define TONE_PIN 10

//define max size of task struct array
#define MAX_PROCESSES 2

//define the possible states a task can have
#define READY 1
#define DEAD  0

//define the notes for the tune
#define NOTE_E 659
#define NOTE_C 523
#define NOTE_G 784
#define NOTE_g 392
#define NOTE_R 0

//define specefic elements in the deadTaskList that will hold respective tasks
#define REVIVE_FLASH_LED_TASK    &deadTaskList[0]
#define REVIVE_PLAY_SPEAKER_TASK &deadTaskList[1]

//define specefic elements in the processList that will hold respective tasks
#define KILL_FLASH_LED_TASK    &processList[0]
#define KILL_PLAY_SPEAKER_TASK &processList[1]

//100ms -> based on a 100us delay, the length of time each note should play
#define NOTE_TIME   1000 //100ms


int LED_timer = 0; //counter to store length of time which LED has been in a
particular state
int LED_state = 0; //counter to store the current state of the LED
```

```
long tune_timer   = 0; //counter to store length of time which speaker has been
in a particular state
int tune_elem     = 0; //counter to store the the position of the current note in
the mario[] array
int tune_state    = 0; //counter to store the current state of the speaker
int note_timer    = 0; //counter to store the length of time which a note has
been playing

int sFlag       = 1; //a flag rasied on interupt
int currentTask = 0; //an int to hold the position of the current running task in
the array

int mario[] = {NOTE_E, NOTE_R, NOTE_E, NOTE_R, NOTE_R, NOTE_E, NOTE_R, NOTE_R,
NOTE_C, NOTE_R,
               NOTE_E, NOTE_R, NOTE_R, NOTE_G, NOTE_R, NOTE_R, NOTE_R, NOTE_R,
NOTE_R, NOTE_g,
               NOTE_R};


//A struct that defines the template of characteristics for one task
struct TCB {
  int pid;           //integer of the process ID
  void* function;    //the function of the process
  int state;         //state of the process
  char name;         //name of the struct
  int taskRunAmount; //integer to keep track of the times it has run
} TCB_struct;        // name given to the TCB struct

static struct TCB processList[MAX_PROCESSES]; //struct array to control
READY/RUNNING tasks
static struct TCB deadTaskList[MAX_PROCESSES]; //struct array to control DEAD
tasks


void setup() {
  // Set up pins as outputs for the speaker and LED
  pinMode(LED_PIN,  OUTPUT);
  pinMode(TONE_PIN, OUTPUT);

  // Initialize characteristics of tasks in the processList array
  processList[0].pid          = 0;
  processList[0].function     = flashLED;
  processList[0].state        = READY;
  processList[0].name         = 'a';
  processList[0].taskRunAmount = 0;
```

```
  processList[1].pid           = 1;
  processList[1].function      = playSpeaker;
  processList[1].state         = READY;
  processList[1].name          = 'b';
  processList[1].taskRunAmount = 0;

//Set up timer and its configurations for ISR (CTC mode, 10KHz, prescaler = 64)
  TCCR4A  = 0;
  TCCR4B  = (1 << WGM42) | (1 << CS41) | (1 << CS40);
  OCR4A   = 25; //100uS interupt, 10Khz
  TIMSK4 |= (1 << OCIE4A);
  sei();
}


void loop() {
    scheduler();
}

/************************************************
 * void flashLED()
 *    Arduments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function flashes an external LED for a hardcoded amount of time.
 *    The LED will be on for 250ms and off for 750ms
 *
 *    Acknowledgments: N/A
 */
void flashLED(){
    if(LED_state == LOW && LED_timer == 7500){
        digitalWrite(LED_PIN, HIGH);
        LED_state = HIGH;
        LED_timer = 0;
    }else if(LED_state == HIGH && LED_timer == 2500){
        digitalWrite(LED_PIN, LOW);
        LED_state = LOW;
        LED_timer = 0;
    }
    return;
}
```

```
/*********************************************
 * void playSpeaker()
 *
 *    Arduments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function calculates the half-period of a given frequency
 *    in the mario[] array. It then itterates through this array
 *    and plays each note for a NOTE_TIME amount of time. After all
 *    of the notes are played there is a hardcoded 4sec pause.
 *
 *    Acknowledgments: N/A
 */
void playSpeaker() {
  //calculate the half period of a frequency
  int period_half = 5000/(mario[tune_elem] + 1);

  if(tune_timer == period_half && tune_elem != 21){
    //if the speaker is off
    if(tune_state == LOW){
      digitalWrite(TONE_PIN, HIGH);
      tune_timer = 0;
      tune_state = HIGH;

    //if the speakers it on
    }else if(tune_state == HIGH){
      digitalWrite(TONE_PIN, LOW);
      tune_timer = 0;
      tune_state = LOW;
    }
  }

  //increment to next note if current note has played for NOTE_TIME amount of
time
  if(note_timer == NOTE_TIME && tune_elem != 21){
    tune_elem++;
    note_timer = 0;
    tune_timer = 0;

  //if all notes have been played wait 4sec, then go back to beginning of tune[]
array
  }else if(tune_elem == 21 && tune_timer == 40000){
    tune_elem = 0;
    tune_timer = 0;
```

```
    note_timer = 0;
  }
}


 /*********************************************
 * void function_ptr(function pointer)
 *
 *    Argument 1 - the function being pointed to
 *
 *    Returns - no returns, this is a void function
 *
 *    This function serves as a function pointer to any given task to execute
 *
 *    Acknowledgments: LAB 3 starter code
 */
void function_ptr(void* function()){
    function();
}


  /*********************************************
 * void task_self_quit()
 *
 *    Arguments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function allows a task to terminate itself by manipulating its TCB
 *    A task can call this function to change its own status to DEAD and remove
itself
 *    from the tcbList.
 *
 *    Acknowledgments: LAB 3 starter code
 */
void task_self_quit(){
  //set the state of the current task to DEAD
  processList[currentTask].state = DEAD;

  //if the current task is flashLED move teh task to deadTaskList
  //and replace all values in processList[0] with NULL
  if (processList[currentTask].pid == 0){
    deadTaskList[0] = *KILL_FLASH_LED_TASK;
     memset(&processList[currentTask], 0, sizeof(processList[currentTask]));
    return;
```

```
  }

  //if the current task is playSpeajer move the task to deadTaskList
  //and replace all values in processList[1] with NULL
  if (processList[currentTask].pid == 1){
    deadTaskList[1] = *KILL_PLAY_SPEAKER_TASK;
     memset(&processList[currentTask], 0, sizeof(processList[currentTask]));
    return;
  }
}


  /*********************************************
 * void task_start(TCB*)
 *
 *    Arguments - pointer to task control block struct
 *
 *    Returns - no returns, this is a void function
 *
 *    This function allows a task to start up another task
 *    A task can call this function to change the status of a DEAD task to READY,
 *    effectively "reviving" the task.
 *
 *    Acknowledgments: LAB 3 starter code
 */
void task_start(TCB* task) {
  //change teh DEAD task's state to ready and increment raskRunAmount
  task->state = READY;
  task->taskRunAmount++;

  //if the task is flashLED move the task to processList[0]
  //and replace all values in deadTaskList[0] with NULL
  if (task == REVIVE_FLASH_LED_TASK){
    processList[0] = *task;
    memset(REVIVE_FLASH_LED_TASK, 0, sizeof(deadTaskList[0]));
    return;
  }

  //if thetask is flashLED move the task to processList[0]
  //and replace all values in deadTaskList[0] with NULL
  if (task == REVIVE_PLAY_SPEAKER_TASK){
    processList[1] = *task;
    memset(REVIVE_PLAY_SPEAKER_TASK, 0, sizeof(deadTaskList[1]));
    return;
  }
```

```
}


   /*********************************************
 * void scheduler()
 *
 *    Arguments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function allows for tasks to be processed and executed
 *    This function will be called in the loop() function. It should check the
status of
 *    each task in the tcbList and decide which one to run next, updating
currentTaskIndex
 *    accordingly.
 *
 *    Acknowledgments: LAB 3 starter code
 */
void scheduler(){
  // resets currentTask to beginning of processList
  if ((currentTask == MAX_PROCESSES)){
    currentTask = 0;
  }

  // when task is in ready state, execute
  if (processList[currentTask].state != 0) {
    function_ptr(processList[currentTask].function);
  }

  // increment currentTask to traverse processList
  currentTask++;

  //on interupt increment task-internal counters
  if (sFlag == 1){
    LED_timer++;
    tune_timer++;
    note_timer++;
    sFlag = 0;
  }
}

/**
 * Interrupt Service Routine for TIMER0_COMPA_vect
 * Sets the flag to DONE
```

```
 */
ISR(TIMER4_COMPA_vect) {
  sFlag = 1;
  TIFR4 |= (1 << OCF4A);
}
```

INTENTIONALLY LEFT BLANK

DEMO 4

```
/*
* University of Washington
* ECE 474,  7/18/23
* Jason Bentley
*
* Lab 3 - DEMO 4
*
* Acknowledgments: LAB 3 starter code from Sep Makhsous
*/

//include the SevSeg library for controlling the 7-segment display
#include "SevSeg.h"

// Define necessary variables and constants
#define LED_PIN  7
#define TONE_PIN 10

//define max size of task array
#define MAX_SIZE 10

//define the possible states a task can have
#define READY    0
#define RUNNING  1
#define SLEEPING 2
#define PENDING  4
#define DONE     5

//define the notes for the tune
#define NOTE_E 659
#define NOTE_C 523
#define NOTE_G 784
#define NOTE_g 392
#define NOTE_R 0

//100ms -> based on a 100us delay, the length of time each note should play
#define NOTE_TIME  1000 //100ms

int LED_timer = 0; //counter to store length of time which LED has been in a
particular state
int LED_state = 0; //counter to store the current state of the LED

long tune_timer   = 0; //counter to store length of time which speaker has been
in a particular state
```

```
int tune_elem     = 0; //counter to store the the position of the current note in
the mario[] array
int tune_state    = 0; //counter to store the current state of the speaker
int note_timer    = 0; //counter to store the length of time which a note has
been playing

long sevSeg_counter  = 0; //a counter for the sevSeg task to divide the 100us
clock
int  sevSeg_number   = 0; //the current number being displayed on the sevSeg
display

int timerCounter   = 0; //a global counter acting as a timer
volatile int sFlag = 0; //a flag rasied on interupt
long x = 0;             //a dummy variable for schedule_sync() to do nothing wile
sleeping

int mario[] = {NOTE_E, NOTE_R, NOTE_E, NOTE_R, NOTE_R, NOTE_E, NOTE_R, NOTE_R,
NOTE_C, NOTE_R,
               NOTE_E, NOTE_R, NOTE_R, NOTE_G, NOTE_R, NOTE_R, NOTE_R, NOTE_R,
NOTE_R, NOTE_g,
               NOTE_R};

//initialize a pointer to the array holding the tasks
void (*taskScheduler[MAX_SIZE]) = {0};


int taskSleep[MAX_SIZE]; //Initialize an array for storing sleeping tasks
int taskState[MAX_SIZE]; //Initialize an array for storing non-sleeping tasks
int currentTask = 0;     //an int to hold the position of the current running
task in the array

SevSeg sevseg; //Initialize a sevseg object


void setup() {
  // Set up pins as outputs for the speaker and LED
  pinMode(LED_PIN,  OUTPUT);
  pinMode(TONE_PIN,  OUTPUT);

  //setup the sevseg intilization variables
  byte numDigits = 4;
  byte digitPins[] = {23, 22, 24, 25}; //Digits: 1,2,3,4 <--put one resistor (ex:
220 Ohms, or 330 Ohms, etc, on each digit pin)
  byte segmentPins[] = {47, 2, 52, 51, 50, 49, 53}; //Segments:
A,B,C,D,E,F,G,Period
```

```
  bool resistorsOnSegments = false; // 'false' means resistors are on digit pins
  byte hardwareConfig = COMMON_CATHODE; // See README.md for options
  bool updateWithDelays = false; // Default 'false' is Recommended
  bool leadingZeros = false; // Use 'true' if you'd like to keep the leading
zeros
  bool disableDecPoint = true; // Use 'true' if your decimal point doesn't exist
or isn't connected

  //initilaize the sevseg diplay
  sevseg.begin(hardwareConfig, numDigits, digitPins, segmentPins,
resistorsOnSegments,
  updateWithDelays, leadingZeros, disableDecPoint);

  //populate task array with tasks
  taskScheduler[0] = flashLED;
  taskScheduler[1] = playSpeaker;
  taskScheduler[2] = sevSegCounter;
  taskScheduler[3] = schedule_sync;
  taskScheduler[4] = NULL;

  //Set up timer and its configurations for ISR (CTC mode, 10KHz, prescaler = 64)
  TCCR4A  = 0;
  TCCR4B  = (1 << WGM42) | (1 << CS41) | (1 << CS40);
  OCR4A   = 25;
  TIMSK4 |= (1 << OCIE4A);
  sei();
}


void loop() {
  // Enter the loop when a task is present in the taskScheduler
  while (taskScheduler[currentTask] != NULL) {

    // Run the task if it is not in the SLEEPING state
    if (taskState[currentTask] != SLEEPING){
        function_ptr(taskScheduler[currentTask]);

        // Change the task state to RUNNING
        taskState[currentTask] = RUNNING;
    }

    //increment to next task and timerCounter "clock"
    currentTask++;
    timerCounter++;
  }
```

```
  //go back to beginning of task array and increment task-specific timers
  currentTask = 0;
  tune_timer++;
  note_timer++;
  LED_timer ++;
  sevSeg_counter++;
}


/***********************************************
 * void flashLED()
 *    Arduments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function flashes an external LED for a hardcoded amount of time.
 *    The LED will be on for 250ms and off for 750ms
 *
 *    Acknowledgments: N/A
 */
void flashLED(void *p){
    if(LED_state == LOW && LED_timer == 7500 || LED_timer == 0){
        digitalWrite(LED_PIN, HIGH);
        LED_state = HIGH;
        LED_timer = 0;
    }else if(LED_state == HIGH && LED_timer == 2500){
        digitalWrite(LED_PIN, LOW);
        LED_state = LOW;
        LED_timer = 0;
    }
    return;
}


 /***********************************************
 * void function_ptr(function pointer)
 *
 *    Argument 1 - the function being pointed to
 *
 *    Returns - no returns, this is a void function
 *
 *    This function serves as a function pointer to any given task to execute
 *
 *    Acknowledgments: LAB 3 starer code
```

```
 */
void function_ptr(void* function()){
    function();
}


/***********************************************
 * void playSpeaker()
 *
 *    Arduments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function calculates the half-period of a given frequency
 *    in the mario[] array. It then itterates through this array
 *    and plays each note for a NOTE_TIME amount of time. After all
 *    of the notes are played there is a hardcoded 4sec pause.
 *
 *    Acknowledgments: N/A
 */
void playSpeaker(void *p) {
  //calculate the half period of a frequency
  int period_half = 5000/(mario[tune_elem] + 1);

  //if the speaker is off
  if(tune_state == LOW && tune_timer == period_half && tune_elem != 21){
    digitalWrite(TONE_PIN, HIGH);
    tune_timer = 0;
    tune_state = HIGH;
    return;
  }

  //if the speakers it on
  if(tune_state == HIGH && tune_timer == period_half && tune_elem != 21){
    digitalWrite(TONE_PIN, LOW);
    tune_timer = 0;
    tune_state = LOW;
    return;
  }

  //increment to next note if current note has played for NOTE_TIME amount of
time
  if(note_timer == NOTE_TIME && tune_elem != 21){
    tune_elem++;
    note_timer = 0;
```

```
    tune_timer = 0;
    return;
  }

  //if all notes have been played wait 4sec, then go back to beginning of tune[]
array
  if(tune_elem == 21 && tune_timer == 40000){
    tune_elem = 0;
    tune_timer = 0;
    note_timer = 0;
    return;
  }
}


/************************************************
 * void sleep_474(int)
 *
 *    Argument - The sleep time in milliseconds
 *
 *    Returns - no returns, this is a void function
 *
 *    Function that controls the sleep time of a task
 *
 *    Acknowledgments: LAB 3 starter code
 */
void sleep_474(int t) {
  // Set the sleep time of the current task
  taskSleep[currentTask] = t;
  // Change the task state to SLEEPING
  taskState[currentTask] = SLEEPING;
}
/************************************************
 * void schedule_sync()
 *
 *    Argument - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    Function that synchronizes the schedule and tasks while in the PENDING
state
 *
 *    Acknowledgments: LAB 3 start code
 */
void schedule_sync(void *p) {
```

```
  // Enter an infinite loop when sFlag is PENDING
  while(sFlag == PENDING){
    x += 1;
  }
  // Update the sleep time of the task
  sleep_474(timerCounter);
  taskSleep[currentTask] -= 2;
  // Wake up any sleeping task if the sleep time is negative
  if (taskSleep[currentTask] < 0){
    taskState[currentTask] = READY;
  }
  // Reset the sFlag to PENDING
  sFlag = PENDING;
}
/***********************************************
 * void sevSegCounter()
 *
 *    Argument - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This tasks uses the sevSeg_counter counter to increment an int
 *    every 100ms and then display that int on the sevseg display.
 *
 *    Acknowledgments: LAB 3 start code
 */
void sevSegCounter(void* p){
    //every 100ms increment the display
    if (sevSeg_counter == 1000){
        sevSeg_counter = 0;
        sevSeg_number++;
        sevseg.setNumber(sevSeg_number, 1);
    }
    sevseg.refreshDisplay();
}
/**
 * Interrupt Service Routine for TIMER0_COMPA_vect
 * Sets the flag to DONE
 */
ISR(TIMER4_COMPA_vect) {
  sFlag = DONE;
  x=0;
  TIFR4 |= (1 << OCF4A);
}
```

DEMO 5

```
/*
* University of Washington
* ECE 474,  7/18/23
* Jason Bentley
*
* Lab 3 - DEMO 5
*
* Acknowledgments: LAB 3 starter code from Sep Makhsous
*/

//include the SevSeg library for controlling the 7-segment display
#include "SevSeg.h"

// Define necessary variables and constants
#define LED_PIN  7
#define TONE_PIN 10

//define max size of task struct array
#define MAX_PROCESSES 2

//define the possible states a task can have
#define READY 1
#define DEAD  2

//define the notes for the tune
#define NOTE_E 659
#define NOTE_C 523
#define NOTE_G 784
#define NOTE_g 392
#define NOTE_R 0

//define specefic elements in the deadTaskList that will hold respective tasks
#define REVIVE_SEVSEG_TASK    &deadTaskList[0]
#define REVIVE_PLAY_SPEAKER_TASK &deadTaskList[1]

//define specefic elements in the processList that will hold respective tasks
#define KILL_SEVSEG_TASK    &processList[0]
#define KILL_PLAY_SPEAKER_TASK &processList[1]

//100ms -> based on experimentation, the length of time each note should play
#define NOTE_TIME   333 //100ms

long tune_timer   = 0; //counter to store length of time which speaker has been
in a particular state
```

```
int tune_elem     = 0; //counter to store the the position of the current note in
the mario[] array
int tune_state    = 0; //counter to store the current state of the speaker
int note_timer    = 0; //counter to store the length of time which a note has
been playing

int currentTask   = 0; //an int to hold the position of the current running task
in the array

int sevSeg_Counter = 0; //a counter to hold the value from which sevseg display
will decrement from
                        //value is set in sevseg task

int mario[] = {NOTE_E, NOTE_R, NOTE_E, NOTE_R, NOTE_R, NOTE_E, NOTE_R, NOTE_R,
NOTE_C, NOTE_R,
               NOTE_E, NOTE_R, NOTE_R, NOTE_G, NOTE_R, NOTE_R, NOTE_R, NOTE_R,
NOTE_R, NOTE_g,
               NOTE_R};


//A struct that defines the template of characteristics for one task
struct TCB {
  int pid;           //integer of the process ID
  void* function;    //the function of the process
  int state;         //state of the process
  char name;         //name of the struct
  int taskRunAmount; //integer to keep track of the times it has run
} TCB_struct;        // name given to the TCB struct

static struct TCB processList[MAX_PROCESSES]; //struct array to control
READY/RUNNING tasks
static struct TCB deadTaskList[MAX_PROCESSES]; //struct array to control DEAD
tasks

SevSeg sevseg; //Initialize a sevseg object


void setup() {
  // Initialize the required pins as output
  pinMode(LED_PIN,  OUTPUT);
  pinMode(TONE_PIN, OUTPUT);

  //setup the sevseg intilization variables
  byte numDigits = 4;
```

```
  byte digitPins[] = {23, 22, 24, 25}; //Digits: 1,2,3,4 <--put one resistor (ex:
220 Ohms, or 330 Ohms, etc, on each digit pin)
  byte segmentPins[] = {47, 2, 52, 51, 50, 49, 53}; //Segments:
A,B,C,D,E,F,G,Period
  bool resistorsOnSegments = false; // 'false' means resistors are on digit pins
  byte hardwareConfig = COMMON_CATHODE; // See README.md for options
  bool updateWithDelays = false; // Default 'false' is Recommended
  bool leadingZeros = false; // Use 'true' if you'd like to keep the leading
zeros
  bool disableDecPoint = true; // Use 'true' if your decimal point doesn't exist
or isn't connected

  //initilaize the sevseg diplay
  sevseg.begin(hardwareConfig, numDigits, digitPins, segmentPins,
resistorsOnSegments,
  updateWithDelays, leadingZeros, disableDecPoint);

  // Initialize characteristics of tasks in your processList array
  processList[0].pid           = 0;
  processList[0].function      = sevSegCounter;
  processList[0].state         = READY;
  processList[0].name          = 'a';
  processList[0].taskRunAmount = 0;

  processList[1].pid           = 1;
  processList[1].function      = playSpeaker;
  processList[1].state         = READY;
  processList[1].name          = 'b';
  processList[1].taskRunAmount = 0;
}


void loop() {
    scheduler();
}


/*********************************************
 * void playSpeaker()
 *
 *    Arduments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function calculates the half-period of a given frequency
```

```
 *    in the mario[] array. It then itterates through this array
 *    and plays each note for a NOTE_TIME amount of time. After all
 *    of the notes are played there is a hardcoded 4sec pause.
 *
 *    Acknowledgments: N/A
 */
void playSpeaker() {
  //calculate the half period of a frequency
  int period_half = 5000/(mario[tune_elem] + 1);

  //if the speaker is off
  if(tune_state == LOW && tune_timer == period_half && tune_elem != 21 &&
note_timer != NOTE_TIME){
    digitalWrite(TONE_PIN, HIGH);
    tune_timer = 0;
    tune_state = HIGH;
  }

  //if the speakers it on
  else if(tune_state == HIGH && tune_timer == period_half && tune_elem != 21 &&
note_timer != NOTE_TIME){
    digitalWrite(TONE_PIN, LOW);
    tune_timer = 0;
    tune_state = LOW;
  }

  //increment to next note if current note has played for NOTE_TIME amount of
time
  else if(note_timer == NOTE_TIME && tune_elem != 21){
    tune_elem++;
    note_timer = 0;
    tune_timer = 0;
  }

  //if all notes have been played wait 4sec, then go back to beginning of tune[]
array
  else if(tune_elem == 21 && tune_timer == 40000){
    tune_elem = 0;
    tune_timer = 0;
    note_timer = 0;
  }

  //kill the current task
  task_self_quit();
```

```
  //if the next ask is DEAD, revive it
  if(processList[0].function == 0){
    task_start(REVIVE_SEVSEG_TASK);
  }

  //increment internal timers
  tune_timer++;
  note_timer++;
}


/************************************************
 * void function_ptr(function pointer)
 *
 *    Argument 1 - the function being pointed to
 *
 *    Returns - no returns, this is a void function
 *
 *    This function serves as a function pointer to any given task to execute
 *
 *    Acknowledgments: LAB 3 starter code
 */
void function_ptr(void* function()){
    function();
}


/************************************************
 * void task_self_quit()
 *
 *    Arguments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function allows a task to terminate itself by manipulating its TCB
 *    A task can call this function to change its own status to DEAD and remove
itself
 *    from the tcbList.
 *
 *    Acknowledgments: LAB 3 starter code
 */
void task_self_quit(){
  //set the state of the current task to DEAD
  processList[currentTask].state = DEAD;
```

```
  //if the current task is sevseg move the task to deadTaskList
  //and replace all values in processList[0] with NULL
  if (processList[currentTask].pid == 0){
    deadTaskList[0] = *KILL_SEVSEG_TASK;
    memset(&processList[0], 0, sizeof(processList[0]));
    return;
  }

  //if the current task is playSpeajer move the task to deadTaskList
  //and replace all values in processList[1] with NULL
  if (processList[currentTask].pid == 1){
    deadTaskList[1] = *KILL_PLAY_SPEAKER_TASK;
    memset(&processList[1], 0, sizeof(processList[currentTask]));
    return;
  }
}


/***********************************************
 * void task_start(TCB*)
 *
 *    Arguments - pointer to task control block struct
 *
 *    Returns - no returns, this is a void function
 *
 *    This function allows a task to start up another task
 *    A task can call this function to change the status of a DEAD task to READY,
 *    effectively "reviving" the task.
 *
 *    Acknowledgments: LAB 3 starter code
 */
void task_start(TCB* task) {
  //change teh DEAD task's state to ready and increment taskRunAmount
  task->state = READY;
  task->taskRunAmount++;

  //if the task is sevseg move the task to processList[0]
  //and replace all values in deadTaskList[0] with NULL
  if (task == REVIVE_SEVSEG_TASK){
    processList[0] = *task;
    memset(REVIVE_SEVSEG_TASK, 0, sizeof(deadTaskList[0]));
    return;
  }

  //if the task is playSpeaker move the task to processList[0]
```

```
  //and replace all values in deadTaskList[0] with NULL
  if (task == REVIVE_PLAY_SPEAKER_TASK){
    processList[1] = *task;
    memset(REVIVE_PLAY_SPEAKER_TASK, 0, sizeof(deadTaskList[1]));
    return;
  }
}


/*********************************************
 * void scheduler()
 *
 *    Arguments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function allows for tasks to be processed and executed
 *    This function will be called in the loop() function. It should check the
status of
 *    each task in the tcbList and decide which one to run next, updating
currentTaskIndex
 *    accordingly.
 *
 *    Acknowledgments: LAB 3 starter code
 */
void scheduler(){
  //increment to the next task
  currentTask++;

  // resets currentTask to beginning of processList
  if ((currentTask == MAX_PROCESSES)){
    currentTask = 0;
  }

  // otherwise sets task to new task
  // when task is in ready state, execute
  if (processList[currentTask].state == READY) {
    function_ptr(processList[currentTask].function);
  }
}


/*********************************************
 * void sevSegCounter()
 *
```

```
 *    Argument - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    If there is a note playing then this task will display the
 *    frequency of that note on the sevseg counter. If there is no note
 *    playing then this tasks uses the tune timer to decrement the sevSeg_Counter
 *    every 100ms, and then displays this value on the sevseg display. Once
 *    sevSeg_counter reaches zero, the tune will play again.
 */
void sevSegCounter(){
  //if the tune has NOT completed playing yet
  if (tune_elem != 21){
    sevSeg_Counter = 41;
    sevseg.setNumber(mario[tune_elem], 1); //display the frequency of the current
note being played

  //if the tune HAS completed playing
  } else if (tune_elem == 21){
    if (tune_timer == 0){
      sevseg.setNumber(sevSeg_Counter, 1);
    }else if (tune_timer % 1000 == 0) {    //every 100ms decrement sevSeg_Counter
and display new value
      sevSeg_Counter--;
      sevseg.setNumber(sevSeg_Counter, 1);
    }
  }

  sevseg.refreshDisplay();

  //kill the current task
  task_self_quit();

  //if the next task is DEAD, revive it
  if(processList[1].function == 0){
    task_start(REVIVE_PLAY_SPEAKER_TASK);
  }

}
```

DEMO 6

```
/*
* University of Washington
* ECE 474,  7/18/23
* Jason Bentley
*
* Lab 3 - DEMO 6
*
* Acknowledgments: LAB 3 starter code from Sep Makhsous
*/

//include the SevSeg library for controlling the 7-segment display
#include "SevSeg.h"

// Define necessary variables and constants
#define LED_PIN  7
#define TONE_PIN 10

//define max size of task struct array
#define MAX_PROCESSES 5
#define READY 1
#define DEAD  2

//define the notes for the tune
#define NOTE_E 659
#define NOTE_C 523
#define NOTE_G 784
#define NOTE_g 392
#define NOTE_R 0

//define specefic elements in the deadTaskList that will hold respective tasks
#define REVIVE_SEVSEG_TASK       &deadTaskList[1]
#define REVIVE_PLAY_SPEAKER_TASK &deadTaskList[2]
#define REVIVE_SEVSEGSMILE_TASK  &deadTaskList[3]
#define REVIVE_FLASHLED_TASK     &deadTaskList[4]

//define specefic elements in the processList that will hold respective tasks
#define KILL_SEVSEG_TASK       &processList[1]
#define KILL_PLAY_SPEAKER_TASK &processList[2]
#define KILL_SEVSEGSMILE_TASK  &processList[3]
#define KILL_FLASHLED_TASK     &processList[4]

//100ms -> based on a 100us delay, the length of time each note should play
#define NOTE_TIME   1000 //100ms
```

```
int LED_timer = 0; //counter to store length of time which LED has been in a
particular state
int LED_state = 0; //counter to store the current state of the LED

long tune_timer   = 0; //counter to store length of time which speaker has been
in a particular state
int tune_elem     = 0; //counter to store the the position of the current note in
the mario[] array
int tune_state    = 0; //counter to store the current state of the speaker
int note_timer    = 0; //counter to store the length of time which a note has
been playing
int times_played  = 0; //counter to hold the number of times the tune has been
played

int currentTask = 0; //an int to hold the position of the current running task in
the array
int sFlag       = 1; //a flag used to store current current step of task5()
int tFlag       = 0; //a flag raised on interupt

int sevSeg_Counter      = 30; //a counter to hold the value from which sevseg
display will decrement from
int sevSeg_timer        = 0;  //an internal counter used in both sevseg tasks
(smile and count)

int mario[] = {NOTE_E, NOTE_R, NOTE_E, NOTE_R, NOTE_R, NOTE_E, NOTE_R, NOTE_R,
NOTE_C, NOTE_R,
              NOTE_E, NOTE_R, NOTE_R, NOTE_G, NOTE_R, NOTE_R, NOTE_R, NOTE_R,
NOTE_R, NOTE_g,
              NOTE_R};

uint8_t smile[4] ={0x64, 0x9, 0x9, 0x52}; //an array to hold the smile pattern
for the sevseg display

//A struct that defines the template of characteristics for one task
struct TCB {
  int pid;           //integer of the process ID
  void* function;    //the function of the process
  int state;         //state of the process
  char name;         //name of the struct
  int taskRunAmount; //integer to keep track of the times it has run
} TCB_struct;        // name given to the TCB struct

static struct TCB processList[MAX_PROCESSES]; //struct array to control
READY/RUNNING tasks
```

```
static struct TCB deadTaskList[MAX_PROCESSES]; //struct array to control DEAD
tasks

SevSeg sevseg; //Initialize a sevseg object


void setup() {
  // Initialize the required pins as output
  pinMode(LED_PIN,  OUTPUT);
  pinMode(TONE_PIN, OUTPUT);

  //setup the sevseg intilization variables
  byte numDigits = 4;
  byte digitPins[] = {23, 22, 24, 25}; //Digits: 1,2,3,4 <--put one resistor (ex:
220 Ohms, or 330 Ohms, etc, on each digit pin)
  byte segmentPins[] = {47, 2, 52, 51, 50, 49, 53}; //Segments:
A,B,C,D,E,F,G,Period
  bool resistorsOnSegments = false; // 'false' means resistors are on digit pins
  byte hardwareConfig = COMMON_CATHODE; // See README.md for options
  bool updateWithDelays = false; // Default 'false' is Recommended
  bool leadingZeros = false; // Use 'true' if you'd like to keep the leading
zeros
  bool disableDecPoint = true; // Use 'true' if your decimal point doesn't exist
or isn't connected

  //initilaize the sevseg diplay
  sevseg.begin(hardwareConfig, numDigits, digitPins, segmentPins,
resistorsOnSegments,
  updateWithDelays, leadingZeros, disableDecPoint);

  // Initialize characteristics of tasks in your processList array
  int j = 0;

  processList[j].pid           = 0;
  processList[j].function      = task5;
  processList[j].state         = READY;
  processList[j].name          = 'e';
  processList[j].taskRunAmount = 0;
  j++;

  processList[j].pid           = 1;
  processList[j].function      = sevSegCounter;
  processList[j].state         = READY;
  processList[j].name          = 'a';
  processList[j].taskRunAmount = 0;
```

```
  j++;

  processList[j].pid           = 2;
  processList[j].function      = playSpeaker;
  processList[j].state         = READY;
  processList[j].name          = 'b';
  processList[j].taskRunAmount = 0;
  j++;

  processList[j].pid           = 3;
  processList[j].function      = sevSegSmile;
  processList[j].state         = READY;
  processList[j].name          = 'c';
  processList[j].taskRunAmount = 0;
  j++;

  processList[j].pid           = 4;
  processList[j].function      = flashLED;
  processList[j].state         = READY;
  processList[j].name          = 'd';
  processList[j].taskRunAmount = 0;

  //Set up timer and its configurations for ISR (CTC mode, 10KHz, prescaler = 64)
  TCCR4A = 0;
  TCCR4B = (1 << WGM42) | (1 << CS41) | (1 << CS40);
  OCR4A = 25;
  TIMSK4 |= (1 << OCIE4A);
  sei();
}


void loop() {
    scheduler();
}

/*************************************************
 * void flashLED()
 *    Arduments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function flashes an external LED for a hardcoded amount of time.
 *    The LED will be on for 250ms and off for 750ms
 *
 *    Acknowledgments: N/A
```

```
 */
void flashLED(){
  if(LED_state == LOW && LED_timer == 7500){
    digitalWrite(LED_PIN, HIGH);
    LED_state = HIGH;
    LED_timer = 0;
  }else if(LED_state == HIGH && LED_timer == 2500){
    digitalWrite(LED_PIN, LOW);
    LED_state = LOW;
    LED_timer = 0;
  }
  return;
}


/************************************************
 * void playSpeaker()
 *
 *    Arduments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function calculates the half-period of a given frequency
 *    in the mario[] array. It then itterates through this array
 *    and plays each note for a NOTE_TIME amount of time. After all
 *    of the notes are played there is a hardcoded 4sec pause.
 *
 *    Acknowledgments: N/A
 */
void playSpeaker() {
  //calculate the half period of a frequency
  int period_half = 5000/(mario[tune_elem] + 1);

  //if the speaker is off
  if(tune_state == LOW && tune_timer == period_half && tune_elem != 21 &&
note_timer != NOTE_TIME){
    digitalWrite(TONE_PIN, HIGH);
    tune_timer = 0;
    tune_state = HIGH;
  }

  //if the speakers it on
  if(tune_state == HIGH && tune_timer == period_half && tune_elem != 21 &&
note_timer != NOTE_TIME){
    digitalWrite(TONE_PIN, LOW);
```

```
    tune_timer = 0;
    tune_state = LOW;
  }

  //increment to next note if current note has played for NOTE_TIME amount of
time
  if(note_timer == NOTE_TIME && tune_elem != 21){
    tune_elem++;
    note_timer = 0;
    tune_timer = 0;
  }

  //if all notes have been played wait 4sec, then go back to beginning of tune[]
array
  if(tune_elem == 21 && tune_timer == 40000){
    tune_elem = 0;
    tune_timer = 0;
    note_timer = 0;
    times_played++;
  }
}


 /*************************************************
 * void function_ptr(function pointer)
 *
 *    Argument 1 - the function being pointed to
 *
 *    Returns - no returns, this is a void function
 *
 *    This function serves as a function pointer to any given task to execute
 *
 *    Acknowledgments: LAB 3 starer code
 */
void function_ptr(void* function()){
  function();
}


/*************************************************
 * void task_start(TCB*)
 *
 *    Arguments - pointer to task control block struct
 *
 *    Returns - no returns, this is a void function
```

```
 *
 *    This function allows a task to terminate another task
 *    A task can call this function to change the status of a READY task to DEAD,
 *    effectively "killing" the task.
 *
 *    Acknowledgments: LAB 3 starter code
 */
void task_quit(TCB* task){
  //change the READY task's state to DEAD
  task ->state = DEAD;

  //if the current task pid = 1 move the task to deadTaskList[1]
  //and replace all values in processList[1] with NULL
  if (task->pid == 1){
    deadTaskList[1] = *task;
    memset(KILL_SEVSEG_TASK, 0, sizeof(processList[1]));
    return;
  }

  //if the current task pid = 2 move the task to deadTaskList[2]
  //and replace all values in processList[2] with NULL
  if (task -> pid == 2){
    deadTaskList[2] = *task;
    memset(KILL_PLAY_SPEAKER_TASK, 0, sizeof(processList[2]));
    return;
  }

  //if the current task pid = 3 move the task to deadTaskList[3]
  //and replace all values in processList[3] with NULL
  if (task -> pid == 3){
    deadTaskList[3] = *task;
    memset(KILL_SEVSEGSMILE_TASK, 0, sizeof(processList[3]));
    return;
  }

  //if the current task pid = 4 move the task to deadTaskList[4]
  //and replace all values in processList[4] with NULL
  if (task -> pid == 4){
    deadTaskList[4] = *task;
    memset(KILL_FLASHLED_TASK, 0, sizeof(processList[4]));
    return;
  }
}

/*************************************************
```

```
 * void task_start(TCB*)
 *
 *    Arguments - pointer to task control block struct
 *
 *    Returns - no returns, this is a void function
 *
 *    This function allows a task to start up another task
 *    A task can call this function to change the status of a DEAD task to READY,
 *    effectively "reviving" the task.
 *
 *    Acknowledgments: LAB 3 starter code
 */
void task_start(TCB* task) {
  //change the DEAD task's state to ready and increment taskRunAmount
  task->state = READY;
  task->taskRunAmount++;

  //if the task is sevSegCounter move the task to processList[1]
  //and replace all values in deadTaskList[2] with NULL
  if (task == REVIVE_SEVSEG_TASK){
    processList[1] = *task;
    memset(REVIVE_SEVSEG_TASK, 0, sizeof(deadTaskList[1]));
    return;
  }

  //if the task is playSpeaker move the task to processList[2]
  //and replace all values in deadTaskList[3] with NULL
  if (task == REVIVE_PLAY_SPEAKER_TASK){
    processList[2] = *task;
    memset(REVIVE_PLAY_SPEAKER_TASK, 0, sizeof(deadTaskList[2]));
    return;
  }

  //if the task is sevSegSmile move the task to processList[3]
  //and replace all values in deadTaskList[3] with NULL
  if (task == REVIVE_SEVSEGSMILE_TASK){
    processList[3] = *task;
    memset(REVIVE_SEVSEGSMILE_TASK, 0, sizeof(deadTaskList[3]));
    return;
  }

  //if the task is flashLED move the task to processList[4]
  //and replace all values in deadTaskList[4] with NULL
  if (task == REVIVE_FLASHLED_TASK){
    processList[4] = *task;
```

```
    memset(REVIVE_FLASHLED_TASK, 0, sizeof(deadTaskList[4]));
    return;
  }
}


/***********************************************
 * void scheduler()
 *
 *    Arguments - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This function allows for tasks to be processed and executed
 *    This function will be called in the loop() function. It should check the
status of
 *    each task in the tcbList and decide which one to run next, updating
currentTaskIndex
 *    accordingly.
 *
 *    Acknowledgments: LAB 3 starter code
 */
void scheduler(){
  // resets currentTask to beginning of processList
  if ((currentTask == MAX_PROCESSES)){
    currentTask = 0;
  }

  // otherwise sets task to new task
  // when task is in ready state, execute
  if (processList[currentTask].state != 0) {
    function_ptr(processList[currentTask].function);
  }

  //increment to the next task
  currentTask++;

  //on interupt increent all internal counter and reset flag
  if (tFlag == 1){
    LED_timer++;
    sevSeg_timer++;
    tune_timer++;
    note_timer++;
    tFlag = 0;
  }
```

```
}


/***********************************************
 * void sevSegCounter()
 *
 *    Argument - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This tasks uses the sevSeg_timer to decrement the sevSeg_Counter
 *    every 100ms, and then displays this value on the sevseg display.
 */
void sevSegCounter(){
  //decrement sevSeg_Counter every 100ms
  if (sevSeg_timer == 1000 && sevSeg_Counter > 0){
    sevseg.setNumber(sevSeg_Counter, 1);
    sevSeg_Counter--;
    sevSeg_timer = 0;
    }

  sevseg.refreshDisplay();

}

/***********************************************
 * void sevSegSmile()
 *
 *    Argument - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This task resets the sevSeg_timer to zero and uses the
 *    sevseg library to display a smile pattern on the display
 */
void sevSegSmile(){
  //reset sevSeg_timer if it is over 2 seconds
  if(sevSeg_timer > 20000){
    sevSeg_timer = 0;
  }
    sevseg.setSegments(smile);
    sevseg.refreshDisplay();
}
```

```
/*********************************************
 * void task5()
 *
 *    Argument - no arguments
 *
 *    Returns - no returns, this is a void function
 *
 *    This task controls the other tasks through a sequence of
 *    if statements, timers/counters, and the sFlag variable.
 */
void task5(){
  //initiate only the LED task and the playSpeaker task
  if (sFlag == 1){
    task_quit(KILL_SEVSEG_TASK);
    task_quit(KILL_SEVSEGSMILE_TASK);
    sFlag = 2;

  //After playing the tune two "kill" the speaker task
  //and initiate the sevSegCounter task
  } else if (times_played == 2 && sFlag == 2){
    task_quit(KILL_PLAY_SPEAKER_TASK);
    task_start(REVIVE_SEVSEG_TASK);
    sFlag = 3;

  //After the sevSegCounter task has counted down to zero "kill"
  //the task and "revive" the playSpeaker task
  }else if (sevSeg_Counter < 1 && sFlag == 3 && times_played == 2){
    task_quit(KILL_SEVSEG_TASK);
    tune_elem = 0;
    tune_timer = 0;
    note_timer = 0;
    sevseg.blank();
    task_start(REVIVE_PLAY_SPEAKER_TASK);
    sFlag = 4;

  //After one play through of the tune "kill" the playSpeaker task
  //and 'revive' the sevSegSmile task
  } else if (times_played == 3 && sFlag == 4){
    task_start(REVIVE_SEVSEGSMILE_TASK);
    task_quit(KILL_PLAY_SPEAKER_TASK);
    sFlag = 5;

  //After two seconds 'kill' the sevSegSmile task
  } else if (sevSeg_timer == 20000 && sFlag == 5){
    task_quit(KILL_SEVSEGSMILE_TASK);
```

```
    sevseg.blank();
    sFlag = 6;

  //Do nothing forever
  } else if (sFlag == 6){
      return;
  }
}


ISR(TIMER4_COMPA_vect) {
  tFlag = 1;
  TIFR4 |= (1 << OCF4A);
}
```

OVERALL PERFORMANCE SUMMARY

The lab demonstrations were conducted successfully, with all code effectively functioning on the Arduino. Nonetheless, an unusual behavior was noted between the different schedulers, particularly regarding the frequency of the notes in the Mario theme song. Specifically, when utilizing the SRRI scheduler, the tune appeared to play at lower frequencies. This behavior likely stems from an unintentional delay in my code, though the exact source of this delay remains unclear.

Interestingly, with the DDS scheduler architecture, although some delay still exists, it's far less noticeable. I attribute this delay to the extensive amount of code involved in implementing DDS. Despite this complexity, I lean towards a DDS scheduler over an SRRI due to the greater control it affords.

In relation to the learning objectives, I am confident in my successful implementation of the Round-Robin (RR), Synchronized Round Robin with Interrupts (SRRI), and Data-Driven Scheduler (DDS) using while loops, task arrays, and Task Control Block (TCB) structs, respectively. I also managed to effectively implement an Interrupt Service Routine (ISR) for enhancing the precision of my tasks and firmware. Furthermore, I demonstrated my understanding of the data structures and algorithms involved in the schedulers examined in this lab by modifying the starter code to add functionality to the provided tasks.

TEAM BREAKDOWN

I am the only member on my team and did all the work.

DICUSSION AND CONCLUSION

The most demanding part of this lab was initially configuring the Data-Driven Scheduler (DDS). Understanding the workings of the Task Control Block (TCB) struct, particularly in relation to the task_self_quit() and task_start(TCB* task) functions, posed a significant challenge. After several hours of working through the code, I was able to grasp the underlying algorithm controlling the DDS scheduler. With this understanding, I managed to modify the task_self_quit() function, allowing it to terminate a task from a different task rather than just the current one. This modification, which I am particularly

proud of, facilitated the implementation of a straightforward task5() for managing the sequence of events in DEMO 6.

Overall, I consider this lab a rewarding experience. Specifically, I am pleased with my newly acquired understanding of interrupts and schedulers. Given my role as a hardware engineer at a startup, where I am currently involved in a design requiring firmware development, these concepts are particularly valuable. Hence, I expect to apply the knowledge gained from this lab in my work.

To summarize, this lab provided a useful exploration into various scheduler architectures. I succeeded in implementing a Round-Robin (RR), Synchronized Round Robin with Interrupts (SRRI), and Data-Driven Scheduler (DDS), along with an Interrupt Service Routine (ISR). Despite the lab's complexity, the skills and knowledge acquired will undoubtedly be beneficial in my future role as an electronics developer.