Lab 3 Report

Authors:

Arvin Nusalim, 2276767
Samantha Reksosamudra, 2276717

Date: Feb 20, 2024

EE/CSE 474 Winter 2024

Table of Contents

Introduction **2**
System Design and Implementation **2**
Code Structure **4**
Discussion **14**
Suggestions **14**
Conclusions **15**
References **15**
Appendices **16**
Doxygen 16
GitHub Release 16
Total Number of hours Spent 16

Introduction

The learning objectives in this project include implementing different types of schedulers to execute multiple tasks at different periods of time. The scheduler types include (1) a round-robin (RR), (2) a synchronized RR with interrupt service routine (SRRI), (3) and a data-driven scheduler (DDS). The primary programming challenges in this project revolve around implementing the hardware timer interrupts for SRRI. This involved understanding of the Arduino timer specifications and minimizing any redundant task calls (to avoid complicated debugging issues) . Another challenge was synchronizing the tasks using Task Control Blocks (TCBs) in the DDS. This required an understanding of each tasks' state control and manipulation using arrays and flags.

System Design and Implementation

This section covers the design and implementation of different schedulers controlling different amounts of tasks. Each task has their own period of sleep and start time which will be handled by the schedulers. There are three different tasks that are used:

- Task 1 : flashing an external LED with a period of 1s (i.e. 250ms on, 750ms off)
- Task 2 : playing a song using a passive buzzer (i.e. Mario theme song)
- Task 3: : count-up/down using a 4-digit 7-segment display each 100ms

Part 1: RR Scheduler

The system design focuses on achieving multiple task execution using a round-robin; a non-preemptive scheduler. So each task is executed to completion in the order that they are called. For example, Task 1 and 2 have their own prototype functions, called `task1()` and `task2()` respectively. The main objective of this part is to flash the external LED while playing a song at the same time. This is achievable by using a while-loop in the main `loop()` function which calls `task1()` and `task2()` over and over. This design prompts the Arduino to complete each task in the order that they are called. Also, a small delay is required at the end of the `loop()` to meet the end period requirements.

Part 2: SRRI Scheduler

The system design focuses on achieving efficient task management through a SRRI scheduler. The implementation began with configuring a hardware timer on the Arduino to generate periodic interrupts, establishing a foundation for synchronized task execution. Similar to Part 1, there are prototype functions to handle each task. However in this case, the tasks are managed by a state machine that transitions between READY, RUNNING, and SLEEPING states based on system timing and task requirements. The main `loop()` function loops through an array of

function pointers and will execute the task according to their state and sleeping time. If a task is scheduled and READY, then it will be executed.

Part 3: DD Scheduler

The system design uses a Task Control Block (TCB) struct to store each task and extra information, such as the task name. During the setup step, the TCBs for each task are initialized and the main `loop()` function loops through an array of function pointers and will execute the task according to their state and sleeping time. Similar to Part 2, the tasks are managed by a state machine but there is also a new state called DEAD. Tasks will terminate themselves (i.e. set their state to DEAD) and start other tasks with the `task_self_quit()` and `task_start()` function which modifies a task's TCB. However, tasks such as flashing LEDs do not need to be terminated (i.e. quit themselves) since it is constantly going back and forth between on and off. Switching the states from READY and SLEEPING is more efficient and requires less delay.

Part 4: Hardware Setup

There are three main components: (1) an external LED with 330 ohms resistor, (2) a passive buzzer, (3) and a 4-digit 7-segment display. Figure 1 shows an example setup and wiring of the circuit used to demonstrate the task scheduler in this project.

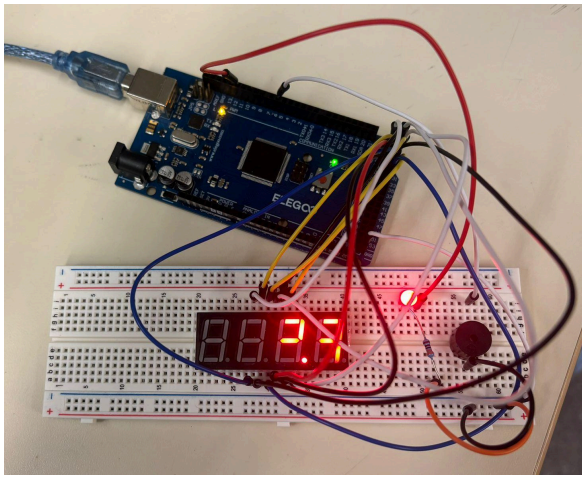

Fig.1 Hardware setup for tasks 1, 2, and 3

The pinouts for the 7-segment display are shown in Figure 2. All pinouts must be connected to the Arduino Digital I/O pins. Pins 11, 10, 7, 5, 4, 3, 2, and 1 will be displayed as the segments of each digit, while the other pins (i.e. 12, 9, 8, and 6) will control which digit is to write a value to.

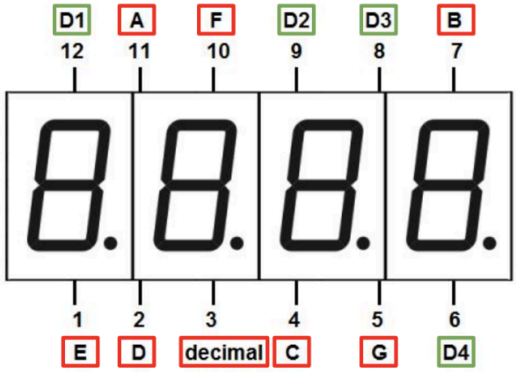

Fig.2 Hardware setup for tasks 1, 2, and 3

Code Structure

Demo 1: RR Scheduler (Task 1 and Task 2)

For the setup, digital pins for the external LED and buzzer are initialized using pinMode() as shown in Figure 3. Pin 12 and 6 are used for the external LED and buzzer consecutively. Timer 4 is also used to set the prescaler to 8 and toggle the buzzer.

```
void setup() {
  // Initialize external LED
  pinMode(12, OUTPUT);

  // Initialize passive buzzer
  pinMode(6, OUTPUT);

  // Setup timer
  TCCR4B = 0;
  TCCR4A = 0;
  TCCR4B |= (1 << WGM42) | (1 << CS41);    // set to CTC mode and prescaler 8
  TCCR4A |= (1 << COM4A0);                // toggle (buzzer) pin when TIMER4_ON_BIT reaches MAX

}
```

Fig.3 the setup for RR Scheduler

A while-loop is utilized within the Arduino loop to implement the Round Robin Scheduler as shown in Figure 4. Tasks (i.e. task1 and task2) are called within the while-loop, with a short delay to fulfill the requirements for the scheduler.

```
void loop() {
  // RR scheduler
  while (1) {
    task1();
    task2();

    // short delay
    delayMicroseconds(10);
  }
}
```

Fig.4 Round Robin Scheduler

Figure 5 shows the code block used to flash the external LED (connected to digital pin 12 of the Arduino) in a period of 1000ms. This code block controls the LED such that it is on for 250ms, and off for 750ms.

```
void task1() {
  if (millis() % 1000 < 250) {
    digitalWrite(12, HIGH);
  } else {
    digitalWrite(12, LOW);
  }
}
```

Fig.5 Task1 function

Figure 6 shows the code block used to play an entire song. It uses a global variable start to keep track of the timestamp, so the function knows when to play the song, to rest (i.e. play nothing), and reset once it finishes its rest. Meanwhile, Figure 7 and 8 shows the helper function used to help Task 2. Figure 7 shows a code block to play a note in a song, and increment the counter to access the next note once 100ms has passed. And Figure 8 shows a code block of another helper function to play a note by setting the OCR4A value based on a note's frequency.

```
// Play song for 2.1 seconds, rest for 6.1 seconds, repeat
void task2() {
  // Play the song
  if (millis() - start < 2100) {
    task2_2(melody[counter]);
  }
  // Delay for 4 seconds after the song ends
  else if (millis() - start < 6100) {
    OCR4A = 0;
  }
  // Else, reset counter and timestamp
  else {
    counter = -1;
    start = millis();
    task2_counter++;
  }
}
```

Fig.6 Task2 function (plays song)

```
// Play each note in the array. Will result in playing the whole song
void task2_2(int freq) {
  if (millis() - temp > 100) {
    counter++;
    temp = millis();
  } else {
    task2_1(melody[counter]);
  }
}
```

Fig.7 Task2_2 function (controls a note in the song)

```
// Play a tone given a frequency
void task2_1(int freq) {
  if (freq == 0) {
    OCR4A = 0;
  } else {
    OCR4A = F_CPU / (2 * prescaler * freq);
  }
}
```

Fig.8 Task2_1 function (plays a note)

Demo 2: SRRI Scheduler (Task 1 and Task 2)

Figure 9 shows the setup process for an SRRI scheduler. To utilize the ISR, Timer 1 was used since the interrupt needs to be 32 bits. Variables such as task_index, sFlag, and j were initialized to set the specific tasks, their initial sleep times, and states, as well as their order. After initializing all the tasks, the last task is filled as a NULL pointer to mark the end of the task initialization. Meanwhile, Figure 10 shows a code block to initialize the interrupt service routine (ISR) using Timer 1.

```
// Timer 1 configuration for system timing/interrupts.
OCR1A = CLK_FREQ / (500 * 2 * CLK_SCALE) - 1; // Calculate compare match value for a 2ms period.
TCCR1B = (1<<WGM12) | (1<<CS10) | (1<<CS12); // Set Timer 1 to CTC mode with a prescaler of 1024.
TIMSK1 |= (1<<OCIE1A); // Enable Timer 1 Compare Match A Interrupt.

sei(); // Enable global interrupts for ISR handling.

// Initialize task management variables and setup initial task states and sleep times.
task_index = 0; // Start processing tasks from the beginning of the taskArray.
sFlag = PENDING; // Initial ISR cycle status set to pending.

// Setup taskArray with specific tasks, their initial sleep times, and states.
int j = 0; // Task array index for initialization.
// task1 and task2 is initially set to READY to start execution without delay.
taskScheduler[j] = &task1; taskSleep[j] = 0; taskState[j] = READY; j++;
taskScheduler[j] = &task2; taskSleep[j] = 0; taskState[j] = READY; j++;
// schedule_sync is always READY to adjust task states and sleep times based on the ISR signal.
taskScheduler[j] = &schedule_sync; taskSleep[j] = 0; taskState[j] = READY; j++;
taskScheduler[j] = NULL; // Mark the end of task initialization with a NULL pointer.
```

Fig.9 The setup for SRRI in addition to the RR setup

```
// ISR for Timer 1 Compare Match A, triggered every 2ms based on timer setup.
ISR(TIMER1_COMPA_vect) {
  sFlag = DONE; // Signal that the ISR cycle is complete, allowing schedule_sync to update tasks.
}
```

Fig.10 ISR for Timer 1

Figure 11 shows a code block in the main `loop()` function of an SRRI scheduler. The loop cycles through all the tasks in the taskScheduler array (which holds function pointers to each task) and changes the state to RUNNING if the task is in READY state, before executing the task. Once we reached the maximum number of tasks or reached the NULL pointer, then we reset our `task_index` to start from the first task (i.e. task_index = 0) again.

```
void loop() {
  // Main loop cycles through tasks, checking their state and executing READY tasks.
  if (taskState[task_index] == READY) {
    taskState[task_index] = RUNNING;
    (*taskScheduler[task_index])(); // Execute the current READY task.
  }
  task_index++; // Move to the next task in the array.
  // Reset task_index to cycle through tasks continuously.
  if (task_index >= N_MAX_TASKS || taskScheduler[task_index] == NULL) task_index = 0;
}
```

Fig.11 Main loop() function for SRRI

Figure 12 shows the `schedule_sync()` function where it synchronizes each task by controlling the current task's sleeping time. The function keeps on decreasing the sleep time of the tasks by

2ms each loop (ISR triggered every 2ms) and when a certain task's sleep time reaches 0, it will change state from SLEEPING to READY.

```
void schedule_sync() {
  // Wait for ISR to signal that the current cycle is done.
  while (sFlag != DONE) {}
  for (int i = 0; taskScheduler[i] != NULL; i++) {
    // Decrement sleeptime for SLEEPING tasks and update their state to READY if sleeptime has elapsed.
    taskSleep[i] -= 2; // Each cycle represents a 2ms decrement.
    if (taskSleep[i] <= 0) {
      taskSleep[i] = 0;
      taskState[i] = READY; // Task is now READY for execution.
    }
  }
  task_index = -1; // Reset task_index for the next round, ensuring the first task is processed next.
  sFlag = PENDING; // Reset sFlag for the next ISR cycle.
}
```

Fig.12 schedule_sync function

Figure 13 shows a code block of the sleep() function, where it changes the task's state to SLEEPING and adds the sleep time of the tasks.

```
void sleep_474(int t) {
  // Set the current task's state to SLEEPING and assign a sleeptime.
  taskState[task_index] = SLEEPING;
  taskSleep[task_index] = t;
}
```

Fig.13 sleep function

Since SRRI uses a hardware timer interrupt to start Task 2 instead of using a round-robin scheduler, task2() needed to be modified. Instead of using millis() and start for the timestamp, the new function used the sleep() function to handle the task's state and sleeping time for rest. So after the program finished playing the song until the last note, it will go into the else statement where the sleep() function changes its state into SLEEPING for 480ms before resetting the counter to 0 (to play the first note again once Task 2 is READY).

```
// Play song for till the end
void task2() {
  // Play the song
  if (counter < (sizeof(melody) / sizeof(melody[0]))) {
    task2_2(melody[counter]);
  }
  // Else, SLEEP for 4 sec, reset counter and timestamp
  else {
    sleep_474(480); // sleep for 4 sec
    counter = 0;
  }

}
```

Fig.14 SRRI's task2 slight change from the RR's task2

Demo 3: DD Scheduler (Task 1 and Task 2)

Figure 15 shows a structure of a TCB that can hold information about a task's details. The structure can hold different information depending on what a programmer needs. For example, a unique task ID for each task, or a counter to keep track of how many times a task was called. The DD scheduler includes several key parts such as the Task Management, responsible for scheduling executing tasks, and State Management, which handles the state transitions of tasks. There are two arrays which help to keep track of all the tasks in the program, `taskList[]`, , which acts like a registry of all tasks, and `deadList[]`, which holds the tasks that have been terminated (i.e. in a DEAD state).

```
int task_delays[] = {1000, 1000, 1}; // Array defining delays for tasks in milliseconds. Adjust these values to change the blinking rate.

// Structure defining a Task Control Block (TCB), which holds information about each task.
typedef struct TCBstruct {
  int taskID; // Unique identifier for the task within the task list.
  char taskName[20]; // Descriptive name for the task. Useful for debugging and readability.
  int numCalls; // Counts how many times the task has been called. Useful for monitoring task activity.
  unsigned short int state; // Current state of the task (e.g., READY, SLEEP, DEAD).
  void (*functionptr)(); // Pointer to the function that the task needs to execute.
  unsigned int sleeptime; // Time in milliseconds the task needs to wait before next execution.
} task;

task tasklist[N_MAX_TASKS]; // Array to hold tasks. Acts as the Task Control Block (TCB) registry.
int deadlist[N_MAX_TASKS]; // Array to keep track of tasks that have been terminated.
int time_count; // Counter used to simulate time passing.
int quit_flag; // A flag indicating whether a task has requested to terminate itself.
```

Fig.15 TCB structure

There are two helper functions called `task_self_quit()` and `task_start()` which helps the state transition. Rather than changing the states of each task, the helper functions managed the states using flags and the `deadList[]` array. The code block is shown in Figure 16 and 17, respectively.

```
// Function to terminate a task from within itself.
void task_self_quit() {
  quit_flag = 1; // Set quit flag to indicate a task wishes to terminate.
}
```

Fig.16 The function to terminate a task

```
// Function to restart a terminated task.
void task_start(int taskID) {
  if (deadlist[taskID] != -1) { // Check if task is in the deadlist.
    deadlist[taskID] = -1; // Remove task from deadlist.
    initTask(taskID, tasklist[taskID].taskName, tasklist[taskID].functionptr, STATE_READY, 0); // Reinitialize the task.
  }
}
```

Fig.17 The function to restart a terminated task

For the setup function, digital pins for the external LED and buzzer are initialized using pinMode() as shown in Figure x. The initTask() function is used to set up information for each tasks, such as the taskID, task name, pointer to the function, task state and sleep time.

```
void setup() {
  // Initializes digital pins connected to LEDs as outputs. Sets the initial state of LEDs to OFF for safety.
  pinMode(LED_PIN_TASK1, OUTPUT);
  // Initialize passive buzzer
  pinMode(BUZZER, OUTPUT);

  // Setup timer
  TCCR4B = 0;
  TCCR4A = 0;
  TCCR4B |= (1 << WGM42) | (1 << CS41);    // set to CTC mode and prescaler 8
  TCCR4A |= (1 << COM4A0);                // toggle (buzzer) pin when TIMER4_ON_BIT reaches MAX

  time_count = 0; // Reset time counter.
  quit_flag = 0; // Ensure quit flag is reset at start.

  // Initialize the task list with tasks for toggling LEDs. Each task is assigned a function, initial state, and sleep time.
  // The pattern is to have one task for turning an LED on and another for turning it off, creating a blinking effect.
  initTask(0, "LED1_on", &task1_on, STATE_READY, 0);
  initTask(1, "LED1_off", &task1_off, STATE_SLEEP, 250);
  // initTask(1, "LED1_off", &task1_off, STATE_SLEEP, 250); // LED2 blinks with a period of 500 ms (250 ms on, 250 ms off).
  initTask(2, "Play_song", &task2, STATE_READY, 0);
}
```

Fig.18 The function to restart a terminated task

In the main loop() function (as shown in Figure 19), each task inside the taskList[] array are called every iteration. It will execute a task depending on its current state. For example, the for-loop will change a task's state from SLEEP to READY if the sleep time is 0. So the task will run the next time it is called, while a terminated task (which has a DEAD state) will never run unless it is reinitialized.

```
void loop() {
  // Main loop iterates over the tasklist, executing tasks based on their state and sleep time.
  for (int i = 0; tasklist[i].functionptr != NULL; i++) {
    // Check if task is ready and sleep time has elapsed.
    if (tasklist[i].state == STATE_READY && tasklist[i].sleeptime == 0) {
      tasklist[i].functionptr(); // Execute the task's function.
      tasklist[i].numCalls++; // Increment the call count for monitoring.
      tasklist[i].state = STATE_SLEEP; // Set task to sleep state.
      tasklist[i].sleeptime = task_delays[i % 3]; // Assign sleep time from the predefined delays array.
    } else if (tasklist[i].state == STATE_SLEEP) {
      // If task is sleeping, decrement its sleep time.
      tasklist[i].sleeptime--;
      if (tasklist[i].sleeptime <= 0) {
        // Once sleep time is up, set task state to ready for next execution.
        tasklist[i].state = STATE_READY;
      }
    }

    if (counter == (sizeof(melody) / sizeof(melody[0]))) {
      tasklist[2].sleeptime = 4000;
      counter = 0;
    }
    // Handle task self-termination logic.
    if (quit_flag != 0) {
      tasklist[i].state = STATE_DEAD;
      deadlist[i] = tasklist[i].taskID;
      quit_flag = 0; // Reset quit flag after handling.
    }
  }
  increment(); // Increment the simulated time counter.
}
```

Fig.19 Main loop function for DDS

The initask function is used in the setup to initialize all the tasks that are going to be used.

```
// Utility function to initialize tasks.
void initTask(int id, const char* name, void (*func)(), unsigned short int initialState, unsigned int initialSleep) {
  tasklist[id].taskID = id;
  strncpy(tasklist[id].taskName, name, sizeof(tasklist[id].taskName) - 1); // Ensure name is null-terminated.
  tasklist[id].functionptr = func;
  tasklist[id].state = initialState;
  tasklist[id].sleeptime = initialSleep;
  tasklist[id].numCalls = 0;
}
```

Fig.20 initTask function for DDS

Demo 4: SRRI Scheduler (Task 1, Task 2, and Task3)

Demo 4 is similar to demo 2 as it uses ISR as the scheduler and adds another task (task3) when initializing the task. The tasks' initialization is put inside the setup function.

```
// Setup taskArray with specific tasks, their initial sleep times, and states.
int j = 0; // Task array index for initialization.
taskScheduler[j] = &task1; taskSleep[j] = 0; taskState[j] = READY; j++;
taskScheduler[j] = &task2; taskSleep[j] = 0; taskState[j] = READY; j++;
taskScheduler[j] = &task3; taskSleep[j] = 0; taskState[j] = READY; j++;
// schedule_sync is always READY to adjust task states and sleep times based on the ISR signal.
taskScheduler[j] = &schedule_sync; taskSleep[j] = 0; taskState[j] = READY; j++;
taskScheduler[j] = NULL; // Mark the end of task initialization with a NULL pointer.
```

Fig.21 initialization task for demo 4

To set the 7-segments display and connect it to the board, there are several parameters that need to be observed such as numDigits, digitalPins, segmentPins, and hardwareConfig. When connecting to the board, the pins are also not in a neat order. Look at fig.2 for the correct pins order.

```
// Setup 7-segment
byte numDigits = 4;
byte digitPins[] = {22, 23, 24, 25};
byte segmentPins[] = {26, 27, 28, 29, 30, 31, 32, 33};
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 = false; // Use 'true' if your decimal point doesn't exist or isn't connected

sevseg.begin(hardwareConfig, numDigits, digitPins, segmentPins, resistorsOnSegments,
updateWithDelays, leadingZeros, disableDecPoint);
sevseg.setBrightness(90);
```

Fig.22 setup for the 7-segments display

Task3 is a counter that counts each deciSeconds and displays it into the 7- segments display. The deciSeconds increase each 100ms and when it reaches 10000 (1000 seconds), the value reset to 0.

```
void task3() {
  static unsigned long timer = millis();
  static int deciSeconds = 0;

  if (millis() - timer >= 100) {
    timer += 100;
    deciSeconds++; // 100 milliSeconds is equal to 1 deciSecond

    if (deciSeconds == 10000) { // Reset to 0 after counting for 1000 seconds.
      deciSeconds=0;
    }
    sevseg.setNumber(deciSeconds, 1);
  }

  sevseg.refreshDisplay(); // Must run repeatedly

}
```

Fig.23 deciSeconds counter

Demo 5: DD Scheduler (Task 4)

The initializations are set in the setup function for task2 (song) and task3 (countdown 7-segments display). The sleep time for task2 is set to 4000 because there is a 4000ms countdown shown in the 7-segments display before the song starts playing.

```
initTask(0, "Play_T3", &task3, STATE_READY, 0);
initTask(1, "Play_T2", &task2, STATE_SLEEP, 4000);
```

Fig.24 tasks initialization for demo 5

This code is put inside the loop function to change the sleep time of task2 when reaches the end of the song and reset the counter to 0 and the deciSeconds to 40 for another 4 seconds countdown.

```
if (counter == (sizeof(melody) / sizeof(melody[0]))) {
  tasklist[1].sleeptime = 4000;
  counter = 0;
  deciSeconds = 40;
}
```

Fig.25 change the sleep time of task2 into 4000 when it reach the end of the song

The task3 function is used as a countdown in deciSeconds and when it reaches 0, it displays the frequency of the song playing.

```
void task3() {
  static unsigned long timer = time_count;
  if (time_count - timer >= 100) {
    // timer += 100;
    timer = time_count;
    if (deciSeconds > 0) {
      Serial.println(deciSeconds);
      deciSeconds--;
      sevseg.setNumber(deciSeconds, 1);
    } else {
      sevseg.setNumber(melody[counter], 0);
    }
  }
  sevseg.refreshDisplay(); // Must run repeatedly

}
```

Fig.26 the countdown function and displaying it in the 7-segments display

Demo 6: DD Scheduler (Task 5)

The initializations are set in the setup function for task1_on and task1_off (LED), task2 (song), task3 (countdown 7-segments display), and task5 (smile display).

```
initTask(0, "Play_T1_on", &task1_on, STATE_READY, 0);
initTask(1, "Play_T1_off", &task1_off, STATE_SLEEP, 250);
initTask(2, "Play_T2", &task2, STATE_READY, 0);
initTask(3, "Play_COUNTDOWN", &task3, STATE_DEAD, 0);
initTask(4, "Play_smile", &task5_1, STATE_DEAD, 0);
```

Fig.27 tasks initialization for demo 5

This code is put inside the loop function to keep count of the task. When the counter reaches 2, task2 has played twice so the state can be set to dead and start task3 (3 seconds countdown). When the counter reaches 3, restart task2. Finally, when the counter reaches 4, call display smile in 7-segments display for 2 seconds.

```
if (task2_counter == 2) {
  tasklist[2].state = STATE_DEAD;
  task_start(3);
}
else if (task2_counter == 3) {
  task_start(2);
  smilie_timer = time_count;
}
else if ((task2_counter == 4) && time_count - smilie_timer <= 2000) {
  tasklist[2].state = STATE_DEAD;
  task_start(4);
}
```

Fig.28 change the sleep time of task2 into 4000 when it reach the end of the song

Task5 function display the smile for 2 second and set the state into dead.

```
// smilie
void task5_1() {
  if (time_count - smilie_timer == 2000) {
    sevseg.blank();
    tasklist[4].state = STATE_DEAD;
  } else {
    sevseg.setSegments(smile);
  }
  sevseg.refreshDisplay(); // Must run repeatedly
}
```

Fig.29 the smile in the 7-segments display

Discussion

In the implementation of this project, there were a myriad of technical challenges that tested problem-solving skills and collaborative abilities. One significant difficulty was configuring the correct Timer/Counter for the hardware interrupt timing and adjusting the sleep time so all the tasks were synchronized. Work was divided accordingly to ensure swift and efficient progress, such that one member focuses on code structure and debugging, while another member focuses on code documentation and testing. Given that there were limited ways to debug an ISR using Timer/Counter, this challenge needed a lot of trial and error attempts, coupled with teamwork effort during testing. It turned out that Timer 1 was more appropriate to use in this project compared to Timer 0, because the interrupt needs to be in 32-bits instead of 8-bits.

Before diving into the main part of this project, which is schedulers, breaking down each task one-by-one into manageable tasks helped each team member to understand the underlying behavior of each task on a hardware component. Regular work sessions provided a platform for discussing progress, addressing challenges, and coordinating efforts, ensuring that both members remained informed and engaged throughout the project. Peer code reviews played a crucial role in maintaining code quality and sharing knowledge among team members. For example, ensuring that there are function prototypes in the beginning of each sketch code explained what functions were used, and adding short comments on each key code line worked as a small note for anyone (i.e. team members or TA) to review what they are doing.

Suggestions

For this project, integrating another aspect where students can explore using a preemptive scheduler would greatly enhance the system's adaptability and responsiveness to real-time

requirements. Non-preemptive schedulers are great for basic knowledge and beginner-friendly practice, but it is hardly used again in the real world. Also, a point-system motivation would be a great incentive for students to explore more efficient ways to finish this project. For example, extra credits can be given for students who put in effort to solve power management issues, or time-constraint tasks, especially for battery-operated IoT devices. These technical improvements could be a great skill that can be applied on a myriad of other projects; including more advanced ones.

Conclusions

This project successfully implemented non-preemptive schedulers, such as round-robin, a synchronized round-robin with ISR, and a data-driven scheduler, to perform multiple tasks at different periods or points of time on an Arduino Mega board. The project revolved around demonstrating efficient task management and synchronization using hardware interrupts and state management. Achieving this objective has greatly enhanced our understanding of real-time operating systems and embedded systems. Besides our final project, we are excited to implement our skills and experience on future projects which can showcase our practical system design, code optimization, and debugging skills.

References

Arduino. "Timer Interrupts on Arduino." Arduino Official Documentation, 2022. https://www.arduino.cc.

ChatGPT. 2024. https://chat.openai.com.

Appendices

Doxygen

https://github.com/sreksosamudra/Intro-to-Embedded-Systems-Lab/tree/main/lab3/doxygen_html

GitHub Release

https://github.com/sreksosamudra/Intro-to-Embedded-Systems-Lab/releases/tag/lab_3

Total Number of hours Spent

12 hours