VIETNAM NATIONAL UNIVERSITY
HO CHI MINH UNIVERSITY OF TECHNOLOGY
COMPUTER SCIENCE & ENGINEERING FACULTY

EXPERIMENT REPORT

CO3054
EMBEDDED SYSTEM

Teacher: Vũ Khánh Hưng

Student 1:	Nguyễn Thanh Hiền	ID: 2111203
Student 2:	Nguyễn Nhật Khải	ID: 2111506
Student 3:	Phạm Văn Nhật Vũ	ID: 2110676
Student 4:	Nguyễn Thanh Liêm	ID: 2111637

TP. Hồ Chí Minh, tháng 12 - 2024

TABLE OF CONTENT

OVERVIEW

Source code

https://github.com/phamvannhatvu/EmbeddedSystem

Work delegation

Full name	Student ID	Task
Nguyễn Thanh Hiền	2111203	Chap 1, 2, 3
Nguyễn Nhật Khải	2111506	Chap 6
Phạm Văn Nhật Vũ	2110676	Chap 5, 8
Nguyễn Thanh Liêm	2111637	Chap 4, 7

Chapter 1: General Purpose Input Output

Exercise 1

Write a program to control LED3 to turn on for 2 seconds, then turn off for 4 seconds. This process repeats indefinitely.

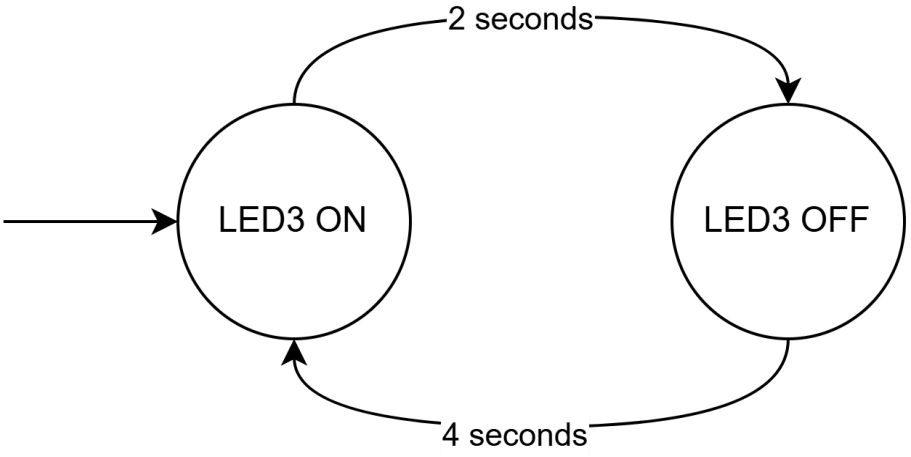
1. Requirement Analysis

Description: This exercise can use `delay()` to maintain states before switching. Initially, the LED would turn ON and `delay()` for 2 seconds. Then, the LED would turn OFF and `delay()` for 4 seconds. The process would repeats indefinitely.

LED3 behaviour:

- ON: 2 seconds.
- OFF: 4 seconds

2. System Design



3. System Implementation

Initialization: N/A - Keep the program as-is

Main program logic:

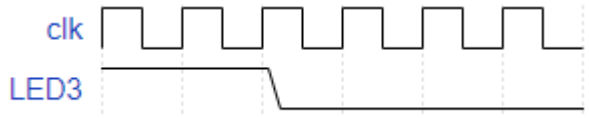
The program uses a consequence approach with 2 phases

1. **ON State**: The LED is turned ON by setting the GPIO pin to HIGH. A delay of 2 seconds is introduced using `HAL_Delay(2000)`
2. **OFF State**: The LED is turned OFF by setting the GPIO pin to LOW. A delay of 4 seconds is introduced using `HAL_Delay(4000)`

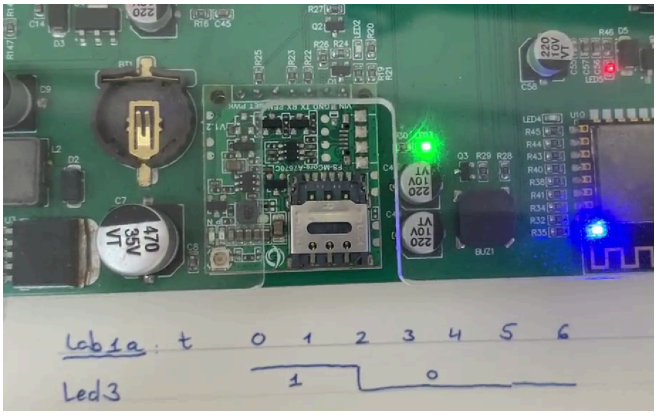
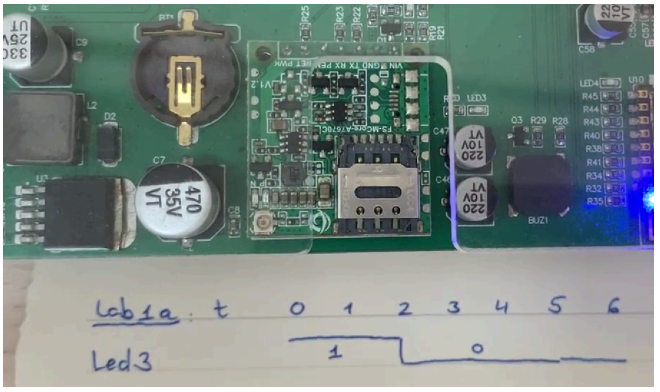
The program continuously alternates between the ON and OFF states in an infinite loop, ensuring that the LED blinks according to the specified timing.

4. Result

Simulation description: Assuming each clock cycle is 1 seconds then, a cycle of our LED3 is 6 seconds (2 seconds ON and 4 seconds OFF).



Test case

• **Target:** LED3 (Green) • **Output:** ON • **Duration:** 2 seconds	
• **Target:** LED3 (Green) • **Output:** ON • **Duration:** 4 seconds	

Source code: https://github.com/phamvannhatvu/EmbeddedSystem/tree/main/Lab01/Lab1_Ex1
Video demo:
https://drive.google.com/file/d/1MnYRQ-JuEz6pRTGpR2oZE91BHGPdG-q6/view?usp=sharing

Exercise 2

Rewrite the program from Exercise 1 but only use a single delay statement at the end of the while loop. Hint: use a counter variable and a variable to store the state of the light.

1. Requirement Analysis

Similar to Lab1 - Exercise 1.

2. System Design

Similar to Lab1 - Exercise 1.

3. System Implementation

Initialization:

Create variables to keep track of the current time slice and state:

- `cycleCounter`: Count up after every to keep track of the current state time.
- `currentState`: Save the main state of the LED.

Main program logic:

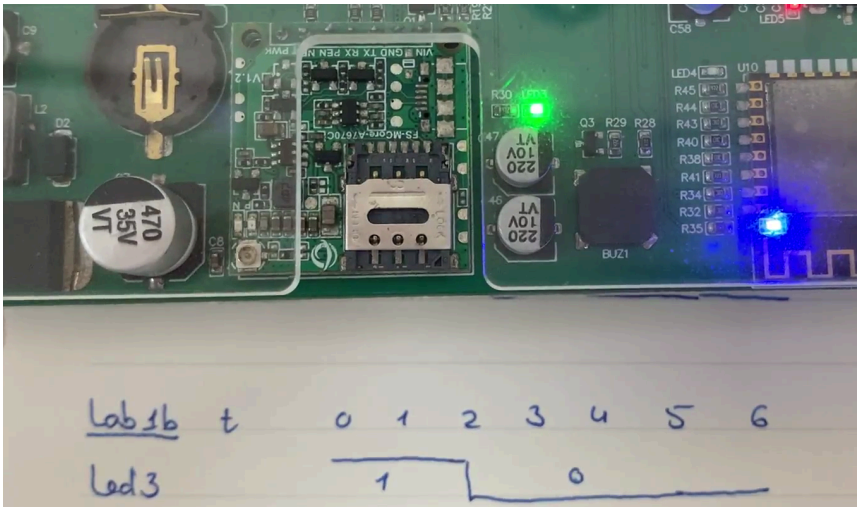
The program uses a consequence approach with 2 phases

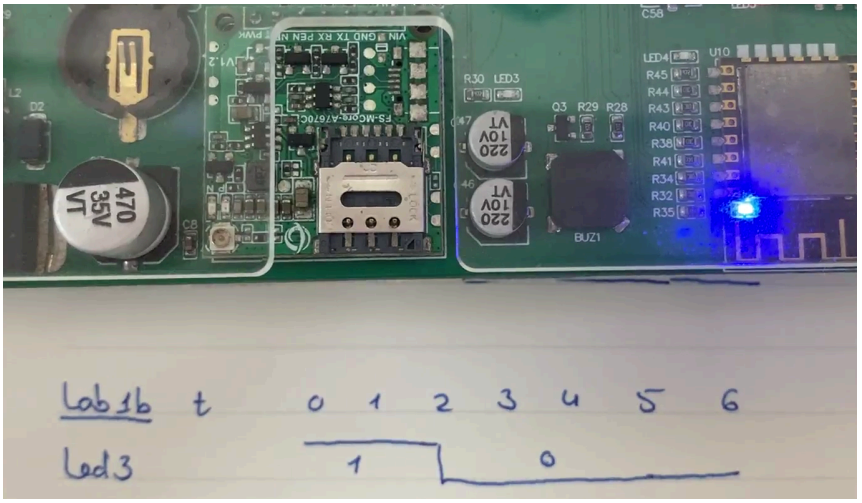
1. **ON State**: The LED is turned ON by setting the GPIO pin to HIGH. When the reached `cycleCounter` 2 cycles (~2 seconds), then the `currentState` would change into OFF State.
2. **OFF State**: The LED is turned OFF by setting the GPIO pin to HIGH. When the rea`HAL_Delay(1000)`ched `cycleCounter` 4 cycles (~4 seconds), then the `currentState` would change into ON State.

The program continuously alternates between the ON and OFF states in an infinite loop, with a `HAL_Delay(1000)` at the end of the state display.

4. Result

Test case

• **Target:** LED3 (Green) • **Output:** ON • **Duration:** 2 seconds	

• **Target:** LED3 (Green) • **Output:** ON • **Duration:** 4 seconds	

Source code: https://github.com/phamvannhatvu/EmbeddedSystem/tree/main/Lab01/Lab1_Ex2
Video demo:
https://drive.google.com/file/d/1r_4sG4POIfOL0Wpbi-GviObTcfvXugw6/view?usp=drive_link

Exercise 3

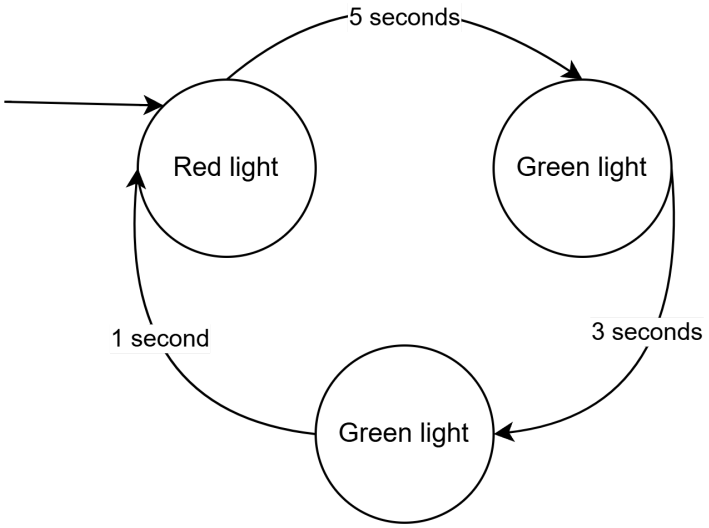
Use LED3 and the LEDs at output Y0, Y1 to simulate traffic light signals. Assume each LED represents a signal on a traffic light, with the following durations: the red light lasts for 5 seconds, the green light lasts for 3 seconds, and the yellow light lasts for 1 second. Only use a single delay statement in the program.

1. Requirement Analysis

The traffic light has 3 light states, at each state, only 1 LED can turn on. The process would repeat indefinitely, with:

- Red light (5 seconds): LED3
- Green light (3 seconds): LED_Y0
- Yellow light (1 second): LED_Y1

2. System Design



3. System Implementation

Initialization:

Create variables to keep track of the current time slice and state:

- `cycleCounter`: Count up after every to keep track of the current state time.
- `currentState`: Save the main state of the LED.

Main program logic:

The program uses a **finite state machine** approach with two states:

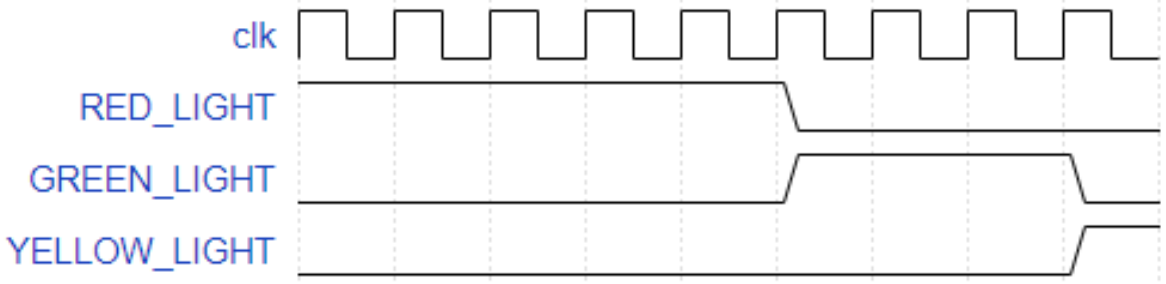
1. **Red light State ()**: The LED is turned ON by setting the GPIO pin to HIGH. When the reached `cycleCounter` 5 cycles (~5 seconds), then the `currentState` would change into Green State.
2. **Green light State**: The LED is turned OFF by setting the GPIO pin to LOW. When the reached `cycleCounter` 3 cycles (~3 seconds), then the `currentState` would change into Yellow State.
3. **Yellow light State**: The LED is turned OFF by setting the GPIO pin to LOW. When the reached `cycleCounter` 1 cycle (~1 second), then the `currentState` would change into Red State.

The program continuously alternates between the color states in an infinite loop, with a `HAL_Delay(1000)` at the end of the state display.

4. Result

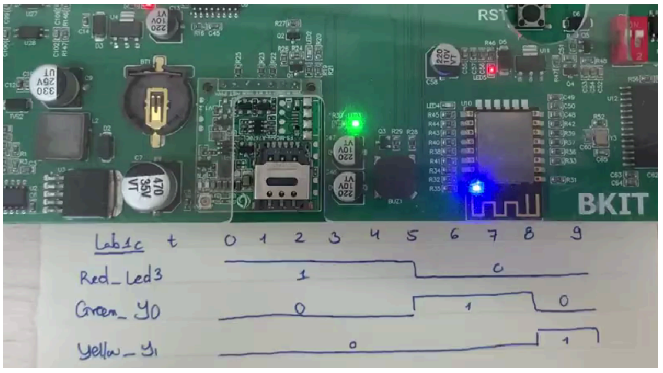
Simulation desrciption: Assuming each clock cycle is 1 seconds then, a cycle of our traffic light is 9 seconds.

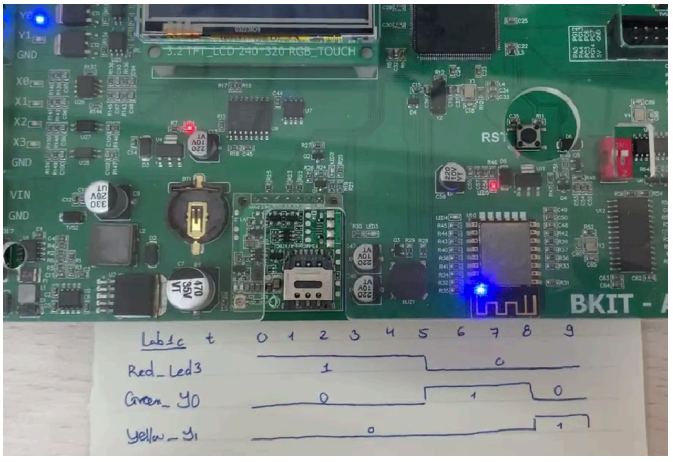
- First, the red light would turns on for 5 seconds, with all other LEDs turn off.
- Next, the green light would turn on for 3 seconds, with all other LEDs turn off.
- Finally, the yellow light would turn on for 1 second, with all other LEDs turn off.

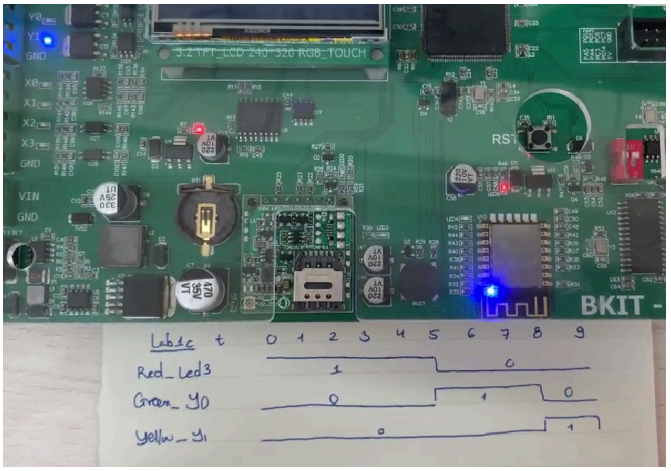


Test case:

State	Description	Evidence

Red light	• **Input:** None
	• **Output**:
	  ◦ LED3: ON
	  ◦ LED_Y0: OFF
	  ◦ LED_Y1: OFF
	• **Duration**: 5 seconds
		

Green light	• **Input**: None
	• **Output**:
	  ◦ LED3 (Green): OFF
	  ◦ LED_Y0: ON
	  ◦ LED_Y1: OFF
	• **Duration**: 3 seconds
		

Yellow light	• **Input**: None
	• **Output**:
	  ◦ LED3 (Green): OFF
	  ◦ LED_Y0: OFF
	  ◦ LED_Y1: ON
	• **Duration**: 1 seconds
		

Source code: https://github.com/phamvannhatvu/EmbeddedSystem/tree/main/Lab01/Lab1_Ex3
Video demo:
https://drive.google.com/file/d/1XZGldNno0-5ksX2sLhHJ9uY7Yy4j7iY5/view?usp=sharing

Chapter 2: Timer Interrupt & LED Scanning

Exercise 1

Using software timers and the functions provided, create the following LED effects:

- **LED3:** Blinks on and off every 2 seconds.
- **LED_Y0:** Turns on for 2 seconds and off for 4 seconds (repeats).
- **LED_Y1:** Turns on for 5 seconds and off for 1 second (repeats).

1. Requirement Analysis

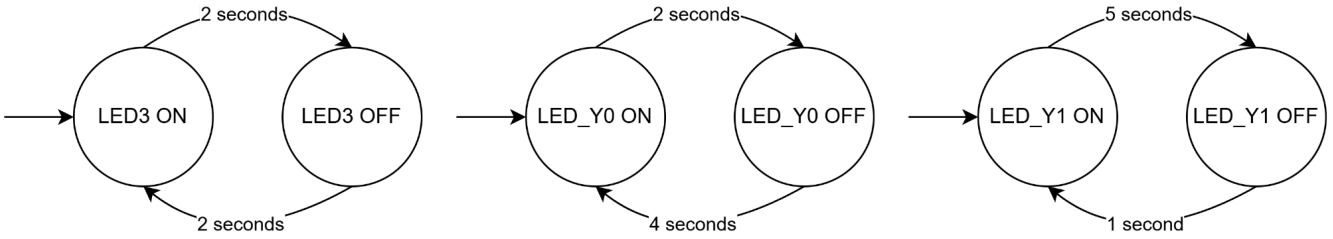
LED3: The LED blinks with an ON and OFF duration of 2 second each, creating a complete cycle of 4 seconds. The cycle repeats continuously.

LED_Y0: The LED stays ON for 2 seconds, then OFF for 4 seconds, creating a complete cycle of 6 seconds. The cycle repeats continuously.

LED_Y1: The LED stays ON for 5 seconds, then OFF for 1 second, creating a complete cycle of 6 seconds. The cycle repeats continuously.

The system complete cycle is the longest cycle of all LEDs, then it would be 6 seconds / cycle.

2. System Design



Each of the LED would run independently, with all started at the same time. The software timer cycle's duration is 50ms, therefore, we would have to convert the duration of each LED:

- 1 second = 1000 / 50 = 20 (cycles)
- 2 seconds = 2000 / 50 = 40 (cycles)
- 4 seconds = 4000 / 50 = 80 (cycles)
- 5 seconds = 5000 / 50 = 100 (cycles)

3. System Implementation

Initialization:

Create variables to keep track of the current time slice and state:

- `count_led_debug`: Count the number of software timer cycles (50ms) for LED3.
- `count_led_y0`: Count the number of software timer cycles (50ms) for LED_Y0.
- `count_led_y1`: Count the number of software timer cycles (50ms) for LED_Y1.

LED states:

- LED3: Initialize the LED with ON state.
- LED_Y0: Initialize the LED with ON state.
- LED_Y1: Initialize the LED with ON state.

Main program logic:

LED3 states:

- ON state: The LED is ON, changes to OFF state after `count_led_debug == 40`
- OFF state: The LED is OFF, changes to ON state after `count_led_debug == 40`

LED_Y0 states:

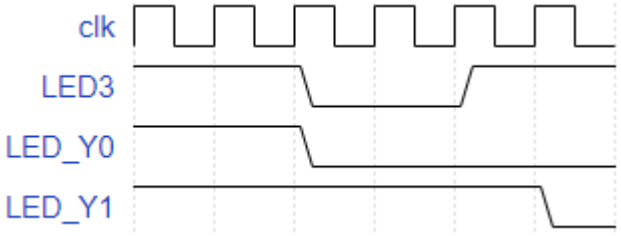
- ON state: The LED is ON, changes to OFF state after `count_led_y0 == 40`
- OFF state: The LED is OFF, changes to ON state after `count_led_y0 == 0`

LED_Y2 states:

- ON state: The LED is ON, changes to OFF state after `count_led_y1 == 100`
- OFF state: The LED is OFF, changes to ON state after `count_led_y1 == 0`

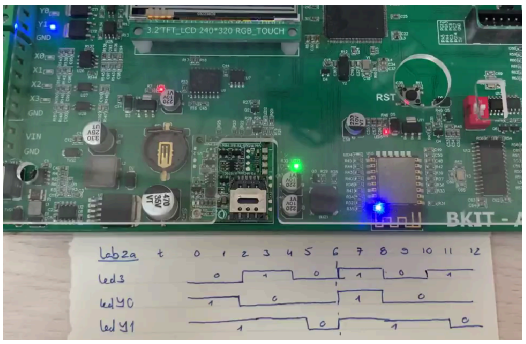
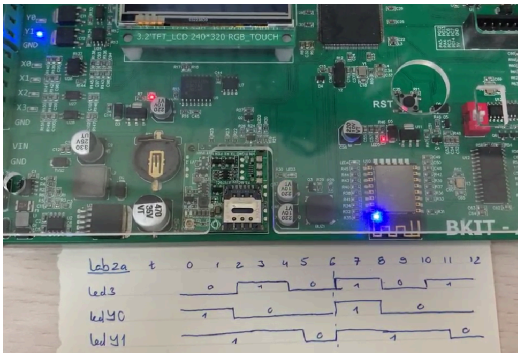
4. *Result*

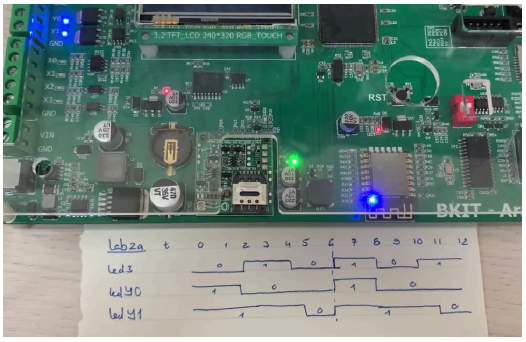
Simulation description:

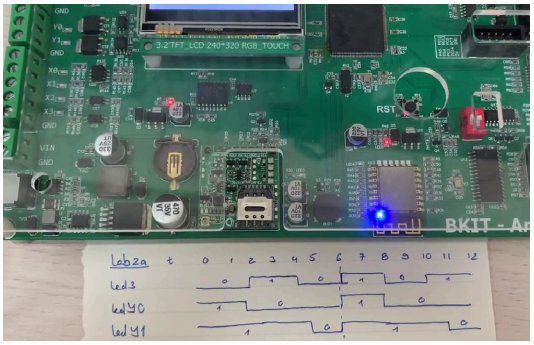


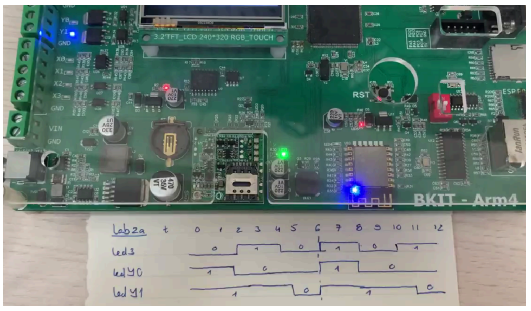
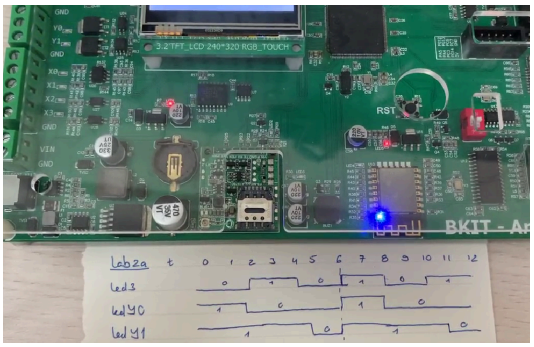
Test case:

LED	Description	Evidence

LED3	• **Input:** None
	• **Output**:
	  ◦ ON: 2 second ≈ 40 `count_led_debug`
	  ◦ OFF: 2 second ≈ 40 `count_led_debug`)
		
		 

LED_Y0	• **Input**: None
	• **Output**:
	  ◦ ON: 2 seconds ≈ 40 `count_led_Y0`
	  ◦ OFF: 4 seconds ≈ 80 `count_led_Y0`
		

LED_Y1	• **Input**: None
	• **Output**:
	  ◦ ON: 5 seconds ≈ 100 `count_led_Y1`
	  ◦ OFF: 1 second ≈ 20 `count_led_Y1`
		
		 

Source code: https://github.com/phamvannhatvu/EmbeddedSystem/tree/main/Lab02/Lab2_Ex1
Video demo:
https://drive.google.com/file/d/1GmuPKfhnnXYjW9g4uCTMUNoLFWqh4hd2/view?usp=sharing

Exercise 2

Reimplement the traffic light exercise from Lab 1 using a timer.

1. Requirement Analysis

Similar to Lab 1 - Exercise 3.

2. System Design

Similar to Lab 1 - Exercise 3.

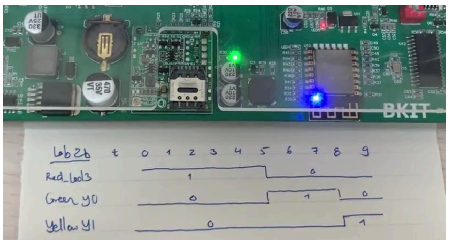
3. System Implementation

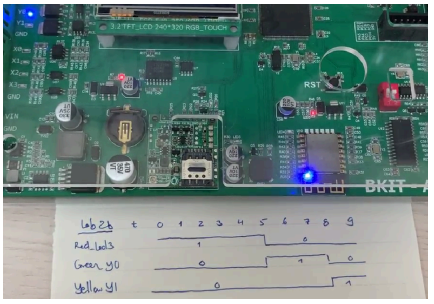
Similar to Lab 1 - Exercise 3. But we would only call the functions after every `flag_timer2 == 0`.

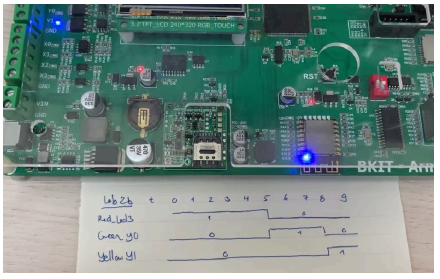
4. Result

Test case:

State	Description	Evidence

Red light	• **Input:** None
	• **Output**:
	  ◦ LED3: ON
	  ◦ LED_Y0: OFF
	  ◦ LED_Y1: OFF
	• **Duration**: 5 seconds
		

Green light	• **Input**: None
	• **Output**:
	  ◦ LED3 (Green): OFF
	  ◦ LED_Y0: ON
	  ◦ LED_Y1: OFF
	• **Duration**: 3 seconds
		

Yellow light	• **Input**: None
	• **Output**:
	  ◦ LED3 (Green): OFF
	  ◦ LED_Y0: OFF
	  ◦ LED_Y1: ON
	• **Duration**: 1 seconds
		

Source code: https://github.com/phamvannhatvu/EmbeddedSystem/tree/main/Lab02/Lab2_Ex2

Video demo:
https://drive.google.com/file/d/1GmuPKfhnnXYjW9g4uCTMUNoLFWqh4hd2/view?usp=sharing

Exercise 3

Change the scanning frequency of the clock LED to 1Hz, 25Hz, and 100Hz.

1. Requirement Analysis

Assuming that the LED is displaying the number 1547, which has 4 digits. With the scanning frequency as specified, we would have to convert it into the equivalent software timer clock rate. We have the software timer frequency = 1KHz.

2. System Design

f_{scan}	**LED ON duration**	N_{LED0}	N_{LED1}	N_{LED2}	N_{LED3}
1 Hz	$f_{LED} = 1 * 4 = 4\ (Hz)$ $\Rightarrow T_{ON} = 1 / 4 = 250(ms)$	250	250	250	250
25 Hz	$f_{LED} = 25 * 4 = 100\ (Hz)$ $\Rightarrow T_{ON} = 1 / 100 = 10(ms)$	10	10	10	10
100 Hz	$f_{LED} = 100 * 4 = 400\ (Hz)$ $\Rightarrow T_{ON} = 1 / 400 = 2.5(ms)$	3	3	2	2

3. System Implementation

Initialization:

Configure multiple clock frequency for each test cases:

- `TIMER_CYCLE_1Hz (1000 / (1 * 4))`
- `TIMER_CYCLE_25Hz (1000 / (25 * 4))`
- `TIMER_CYCLE_100Hz (1000 / (100 * 4))`

Display the number “1547” by saving it to `led_seg_number`

Create variable to count up the number of cycle needed before state transition:

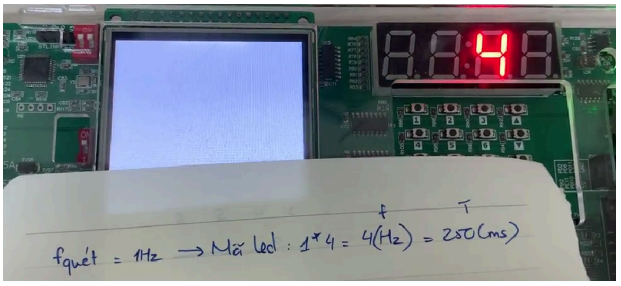
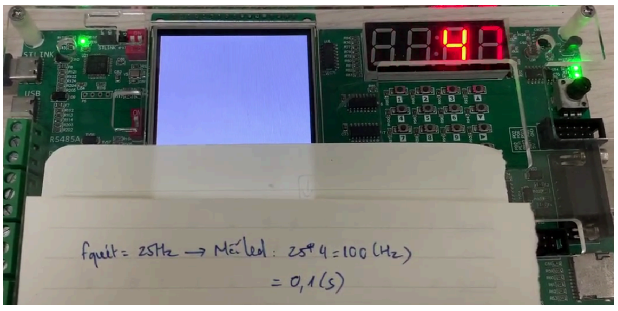
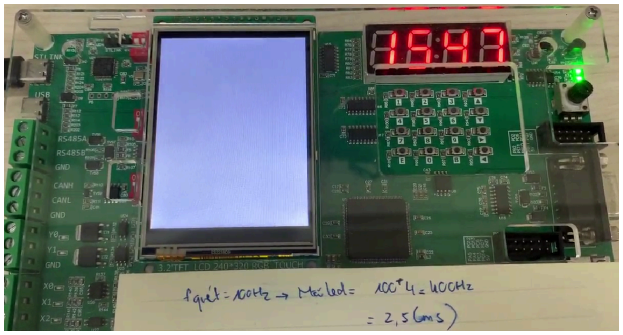
- `timer_modify_count` == 0: `led7_Scan()`

Main program logic:

Call the function `test_7seg()` after every 50ms to run the program.

4. Result

Test case:

Scanning frequency	Description	Evidence
1 Hz	• **Input:** None • **Output**: ○ LED 7 segments scan with 1 Hz	
25 Hz	• **Input:** None • **Output**: ○ LED 7 segments scan with 25 Hz	
100 Hz	• **Input:** None • **Output**: ○ LED 7 segments scan with 100 Hz	

Source code: https://github.com/phamvannhatvu/EmbeddedSystem/tree/main/Lab02/Lab2_Ex3

Video Demo:

$f = 1\ Hz$	$f = 25\ Hz$	$f = 100\ Hz$
https://drive.google.com/file/d/1aFQb53LIFXeJzbIVjafZ3ZTOUoYayBgv/view?usp=sharing	https://drive.google.com/file/d/1oBH_p7LZz44IPCxSId33jDT9eD-5RNQ7/view?usp=drive_link	https://drive.google.com/file/d/1nn5qFxyg2afeh_Gh-wkTeTKMJq8-wnQt/view?usp=drive_link

Exercise 4

Use the clock LEDs to simulate a digital clock, where the first two digits display the hours, and the next two digits display the minutes. The time will update in real-time. The colon in the middle of the clock will blink at a frequency of 2Hz.

1. Requirement Analysis

LED Behavior:

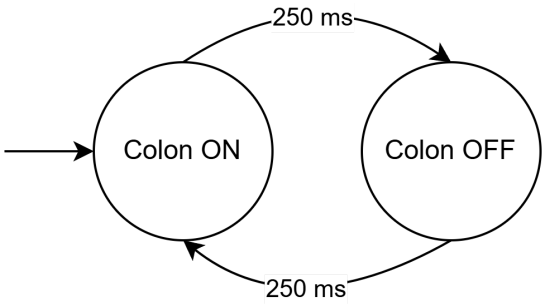
- **Hour Display (First 2 Digits)**: The first two digits on the clock will display the hours in real-time.
- **Minute Display (Last 2 Digits)**: The last two digits will display the minutes in real-time.

Colon Blinking: The colon in the middle of the clock will blink at a frequency of 2Hz, turning ON and OFF every 0.5 seconds.

Time Update: The time on the clock updates in real-time, reflecting actual hours and minutes.

2. System Design

Only the colon would have a cycle ON/OFF, which we can define in finite state machine. With the LED 7 segment clock, it would call `led7_Scan()` every 1ms.



To increase by 1 minute, then we would have to wait 60 seconds ≈ 60k (N_{cycle}).

LED	Range	Display value
LED0 (hour)	[0:6]	$hour \ / \ 10$
LED1 (hour)	[0:9]	$hour \ \% \ 10$
LED2 (minute)	[0:5]	$minute \ / \ 10$
LED3 (minute)	[0:9]	$minute \ \% \ 10$

3. System Implementation

Initialization:

COLON:

Configure number of cycles for colon: `COLON_CYCLE_2Hz (1000 / (2))`

Save the colon current status to toggle every 0.5s in `colon_status`

Call the function inside software timer every 500ms (`colon_blink` would increase by 1 every 1ms): `colon_blink == 0: led7_SetColon(colon_status)`

LED 7 SEGMENTS:

Create variable for real-time clock display:

- `cycle_counter`: Keep track of the current time to increase second after 1000ms (~20 cycle)

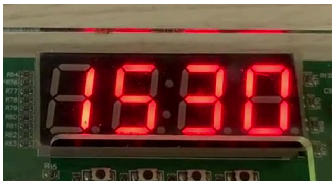
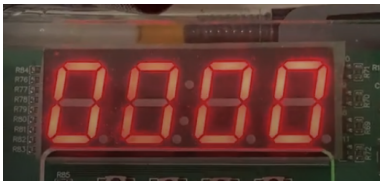
- `hour`: Save the hour to display
- `minute`: Save the minute to display
- `second`: Save the second to display

Main program logic:

- `cycle_counter` reaches 1s, we would increase the `second` variable within the range of [1, 60]
- `second` reaches 60s, we would reset the `second` variable and increase the `minute` variable within the range of [1, 60]
- `minute` reaches 60m, we would reset the `minute` variable and increase `hour`variable the within the range of [1, 24]
- `hour` reaches 24h, we would reset the entire clock.

4. Result

Test case

Test case	Before	After	Video demo
Increase minute			https://drive.google.com/file/d/1RpdCMM98fbXsSuo1kBu48d9R0YdOdlG3/view?usp=sharing
Increase hour			https://drive.google.com/file/d/16i0JSvlcp11Um5MHWrs8vyWpQ9YABEuM/view?usp=drive_link
Time overflow			https://drive.google.com/file/d/1IAb1fw7zcB1VeSDoSPLwxOUZZ19RZdud/view?usp=drive_link

Source code: https://github.com/phamvannhatvu/EmbeddedSystem/tree/main/Lab02/Lab2_Ex4

Exercise 5

Display 4 different numbers on the clock LED, creating an effect where these numbers shift to the right every 1 second.

1. Requirement Analysis

LED Display:

- The program must utilize a 4-digit 7-segment LED display.
- Each digit should display a unique number (e.g., "1", "2", "3", "4").

Shifting Effect:

- Every 1 second, all displayed numbers should shift one position to the right.
- The rightmost digit should "wrap around" to the leftmost position (e.g., "1234" → "4123" → "3412").

Timing:

- Timing should be accurate to 1 second for each shift.
- The timing must not block other processes (preferably using a timer interrupt or software timer).

2. System Design

- Display Numbers: Show "1547" on the LED initially.
- Shifting Effect: Numbers shift right every second, with wrap-around (e.g., "1234" → "4123").
- Every 1 second, trigger an event, rotate the array of digits right.
- Display Update: Update the LED display by sending the new digit values sequentially to each position.

3. System Implementation

Initialization:

Initialize a variable to represent the 4-digit number to display: `led_seg_number = 1547`

Define a counter to manage the shifting effect: `count_led_shift` (shift after 20 cycles ~ 1 second)'

Main program logic:

Digit Display Logic:

```
// Use led7_SetDigit() to display each digit:
led7_SetDigit(led_seg_number / 1000, 0, 0);   // Most significant digit
led7_SetDigit((led_seg_number / 100) % 10, 1, 0); // Second digit
led7_SetDigit((led_seg_number / 10) % 10, 2, 0);  // Third digit
led7_SetDigit(led_seg_number % 10, 3, 0);   // Least significant digit
```

Shifting Effect Logic:

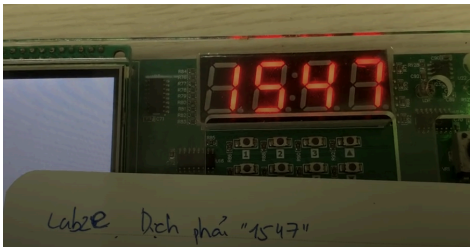
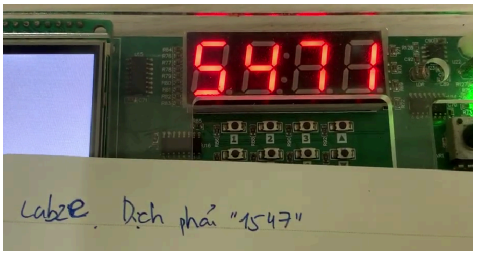
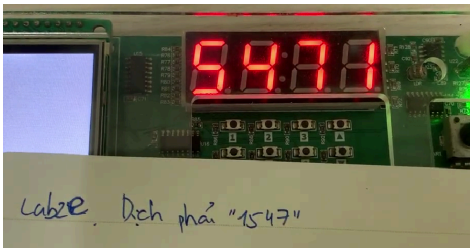
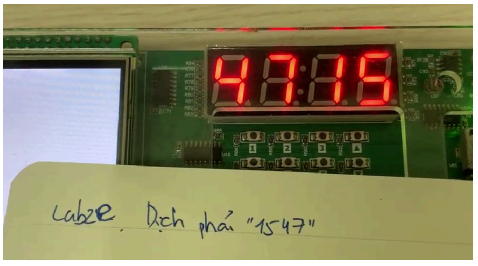
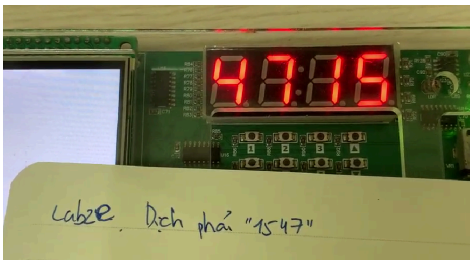
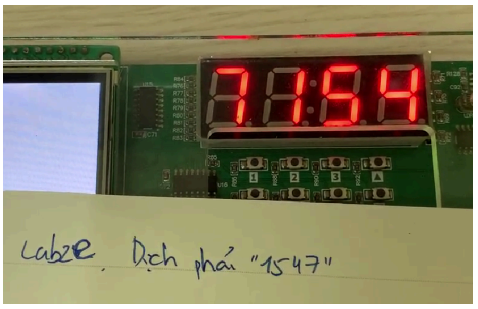
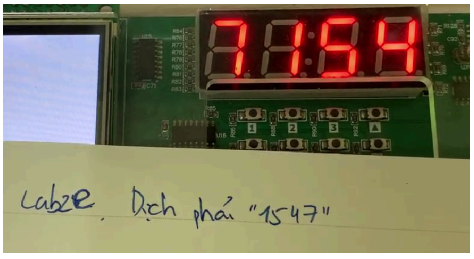
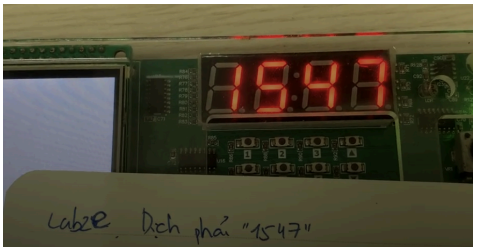
```
count_led_shift = (count_led_shift + 1) % 20; // Increment and reset every 20 cycles
if (count_led_shift != 0) return; // Only shift on the 20th cycle
```

Perform the right shift with wrapping logic:

```
led_seg_number = (led_seg_number % 10) * 1000 + (led_seg_number / 10);
```

4. *Result*

Simulation result:

Test case	Before	After
Displays “1547” then shifts to “5471”		
Displays “5471” then shifts to “4715”		
Displays “4715” then shifts to “7154”		
Displays “7154” then shifts to “1547”		

Source code: https://github.com/phamvannhatvu/EmbeddedSystem/tree/main/Lab02/Lab2_Ex5
Video demo:
https://drive.google.com/file/d/1KOPSi51Hb3tdTZW89PKFvh76ROjD2Zn2/view?usp=drive_link

Chapter 3: LCD & Button Matrix

Build a state machine and implement a traffic light system at an intersection with the following features:

- The application will have 6 traffic lights corresponding to 2 roadways (2 green lights, 2 red lights, and 2 yellow lights). The traffic lights will be simulated on the LCD screen.
- The application will have 3 buttons to:
  - Select the mode.
  - Adjust the cycle times of the lights.
  - Confirm the adjusted settings.
- The application will have at least 4 modes, controlled by the first button. Mode 1 is the NORMAL mode, while Modes 2, 3, and 4 are MODIFICATION modes. The first button will be used to toggle between the modes. The mode will change from 1 to 4 and then back to 1.

Mode 1 - NORMAL: The traffic lights operate normally.
Mode 2 - Adjust the Red Light Cycle:

- The red light blinks at a frequency of 2Hz.
- The adjusted value is displayed on the LCD screen.
- The current mode is displayed on the LCD screen.
- The second button is used to increase the red light cycle value.
- The red light cycle value is within the range of 1 to 99.
- The third button is used to confirm the selected value.

Mode 3 - Adjust the Green Light Cycle: Similar to Mode 2.
Mode 4 - Adjust the Yellow Light Cycle: Similar to Mode 2.

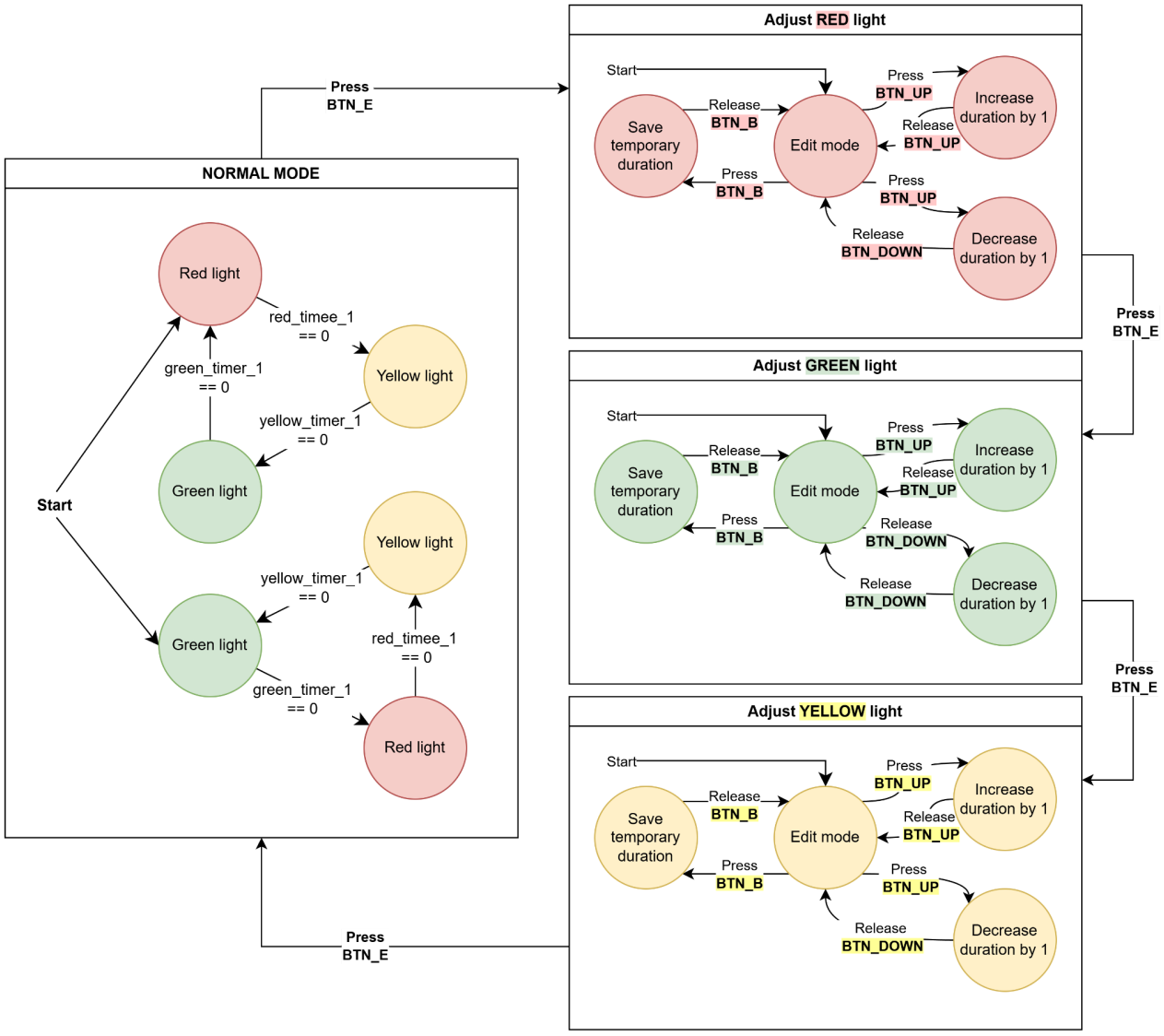
1. Requirement Analysis

Feature	Description

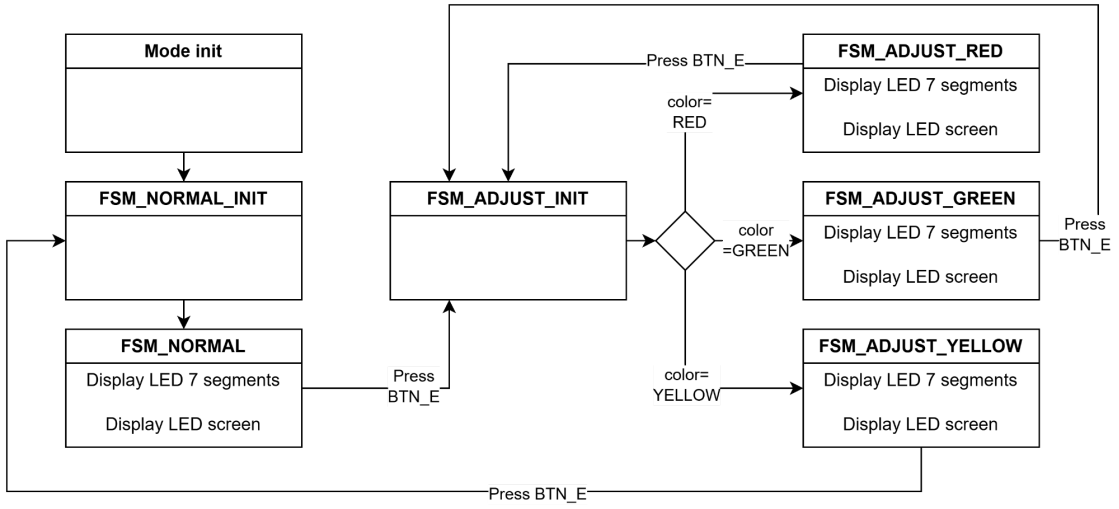
NORMAL	**<u>INPUT</u>:**
	• Press BTN_E: Change to “ADJUST_RED”
	<u>OUTPUT</u>:
	Initial state:
	• Traffic light 1: Initialize with RED light
	• Traffic light 2: Initialize with GREEN light
	Mechanism:
	• **Red light state:** Only red light is ON, other LEDs are OFF (duration = max_time[RED_LIGHT]. When timeout, traffic light changes to Green light state.
	• **Green light state:** Only green light is ON, other LEDs are OFF (duration = max_time[GREEN_LIGHT]. When timeout, traffic light changes to Yellow light state.
	• **Yellow light state:** Only yellow light is ON, other LEDs are OFF (duration = max_time[YELLOW_LIGHT]. When timeout, traffic light changes to Red light state.
	LED screen:
	• ***Title:*** “Normal mode”
	• ***Traffic light 1 (Left)***
	  ◦ Circle: Displays the color of traffic light 1.
	  ◦ Number: Displays the remaining time of traffic light 1.
	• ***Traffic light 2 (Right)***

	◦ Circle: Displays the color of traffic light 2. ◦ Number: Displays the remaining time of traffic light 2. **LED 7 segments:** • Traffic light 1 (Left): Displays the remaining time of traffic light 1 • Traffic light 2 (Right): Displays the remaining time of traffic light 2
ADJUST_RED	**INPUT:** • Press BTN_E: Change to "ADJUST_GREEN" • Press BTN_UP: Increase the max_time[RED_LIGHT] by 1. • Press BTN_B: Save adjusted value to max_time[RED_LIGHT] **OUTPUT:** **LED screen:** • ***Title:*** "Modify RED" • ***Traffic light - Adjust RED:*** ◦ Circle***:*** Displays the color RED on the left. ◦ Number***:*** Displays the current adjusted time on the right. **LED 7 segments:** • Display the current adjusted time with value between [1, 99]
ADJUST_GREEN	**INPUT:** • Press BTN_E: Change to "ADJUST_YELLOW". • Press BTN_UP: Increase the max_time[GREEN_LIGHT] by 1. • Press BTN_B: Save adjusted value to max_time[GREEN_LIGHT] **OUTPUT:** **LED screen:** • ***Title:*** "Modify GREEN" • ***Traffic light - Adjust GREEN:*** ◦ Circle***:*** Displays the color GREEN on the left. ◦ Number***:*** Displays the current adjusted time on the right. **LED 7 segments:** • Display the current adjusted time with value between [1, 99]
ADJUST_YELLOW	**INPUT:** • Press BTN_E: Change to "NORMAL". • Press BTN_UP: Increase the max_time[YELLOW] by 1. • Press BTN_B: Save adjusted value to max_time[YELLOW_LIGHT] **OUTPUT:** **LED screen:** • ***Title:*** "Modify YELLOW" • ***Traffic light - Adjust YELLOW:*** ◦ Circle***:*** Displays the color YELLOW on the left. ◦ Number***:*** Displays the current adjusted time on the right. **LED 7 segments:** • Display the current adjusted time with value between [1, 99]

2. System Design

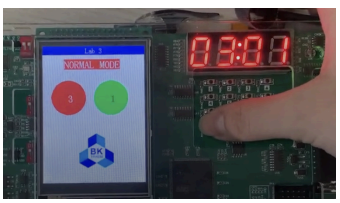
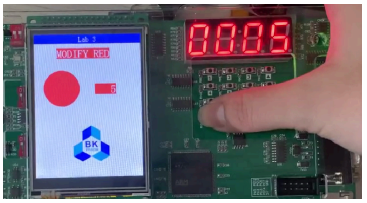
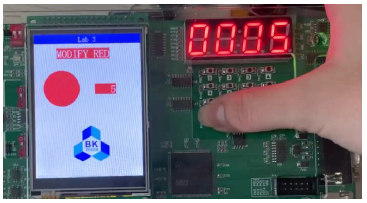
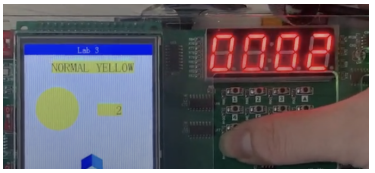
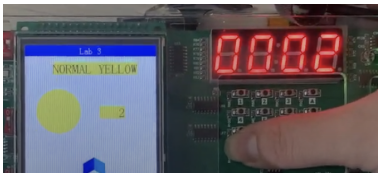
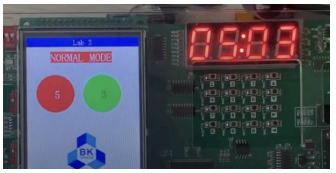
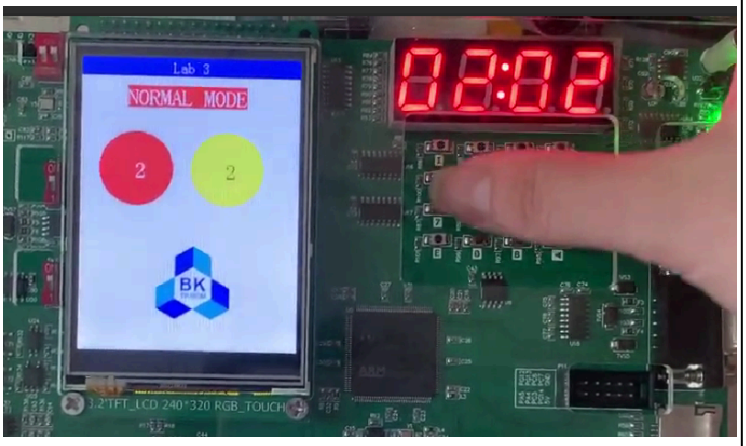
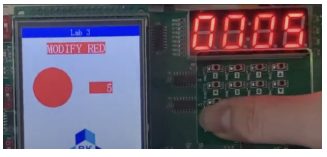
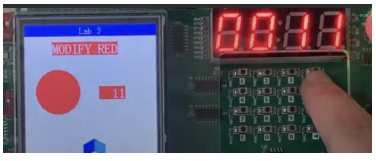
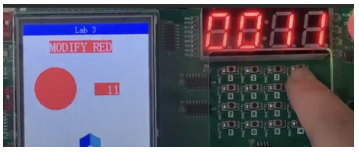
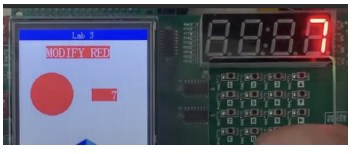


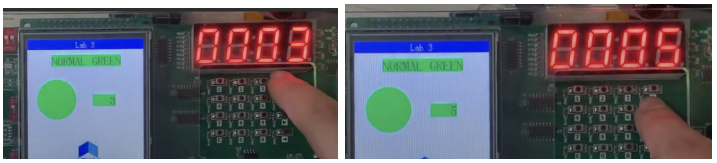
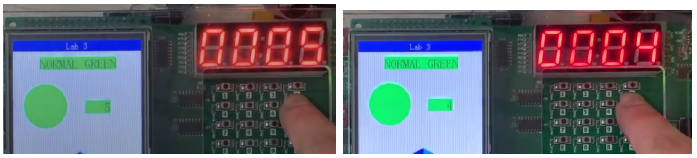
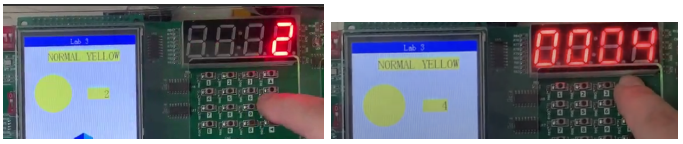
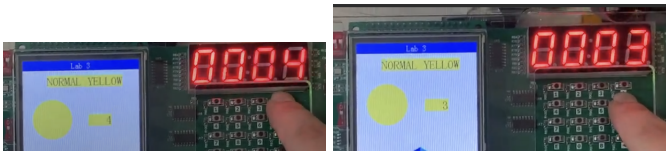
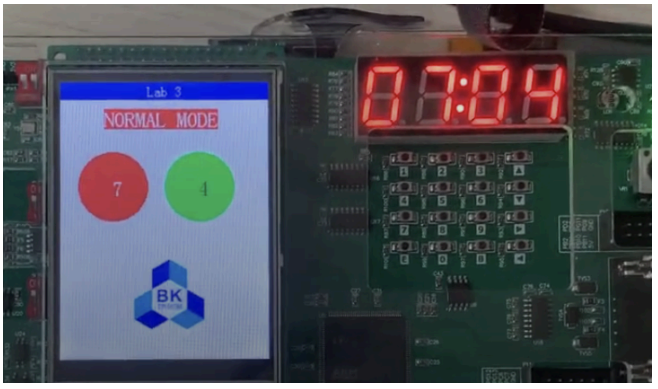
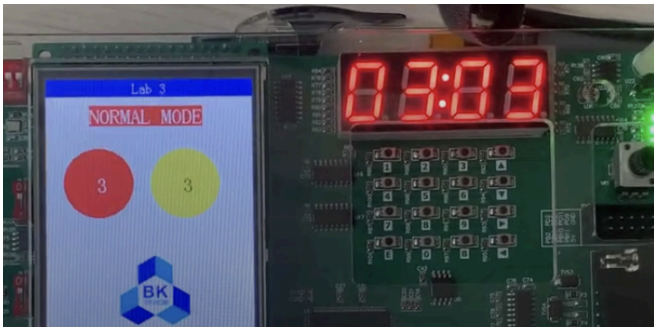
3. System Implementation



4. *Result*

Test cases

Feature	Description	Evidence
Switch mode	NORMAL MODE → ADJUST RED	 
Switch mode	ADJUST RED → ADJUST GREEN	 
Switch mode	ADJUST GREEN → ADJUST YELLOW	 
Switch mode	ADJUST YELLOW → NORMAL MODE	 
NORMAL MODE - Inital (Red: 5, Green 3, Yellow: 2)	Display LED screen: - Title: NORMAL MODE - Traffic light: left & right - Number: left & right Display LED 7 segments: - Traffic light: left & right	
Adjust RED - Increase	From 5 → 11	 
Adjust RED - Decrease	From 11 → 7, and “Save”	 

Adjust GREEN - Increase	From 3 → 5	
Adjust GREEN - Decrease	From 5 → 4, and "Save"	
Adjust YELLOW - Increase	From 2 → 4	
Adjust YELLOW - Decrease	From 4 → 3, and "Save"	
NORMAL MODE - After adjust	Display LED screen: - Title: NORMAL MODE - Traffic light: left & right - Number: left & right Display LED 7 segments: - Traffic light: left & right	 

Source code: https://github.com/phamvannhatvu/EmbeddedSystem/tree/main/Lab03
Video demo:
https://drive.google.com/file/d/12mdQ0tD-u2HQkqn0F6qO53z2az4MkPrR/view?usp=drive_link

Chapter 4: Real-time clock

1. Requirement Analysis

ID	Feature	Description
1	Time view mode	The LCD screen displays information such as day, date, month, year, hour, minute, second in a reasonable format.
2	Time adjustment mode	Time adjustment mode: Allows users to adjust the parameters of the watch. At this time, the watch screen will stop running, when adjusting a parameter, that parameter will flash at a frequency of 2Hz on the screen. There are 2 buttons that will be used in this mode: * Parameter adjustment button (suggested to use the "up arrow" button): when pressed, the parameter will increase by 1, if held for 2 seconds, the parameter will increase every 200ms. * Parameter saving button (suggested to use the "E" button): press to save the parameter and move to the next parameter.
3	Timer mode	Adjust similarly to the time adjustment mode. When the watch reaches the scheduled time, display the alarm effect on the LCD screen.

2. System Design

The fsm() function (Finite State Machine) is the control center of the entire system. It manages the operating modes of the electronic clock including:

- Time viewing mode (modeNormal).
- Time adjustment mode (modeModify).
- Timer mode (modeTimer).

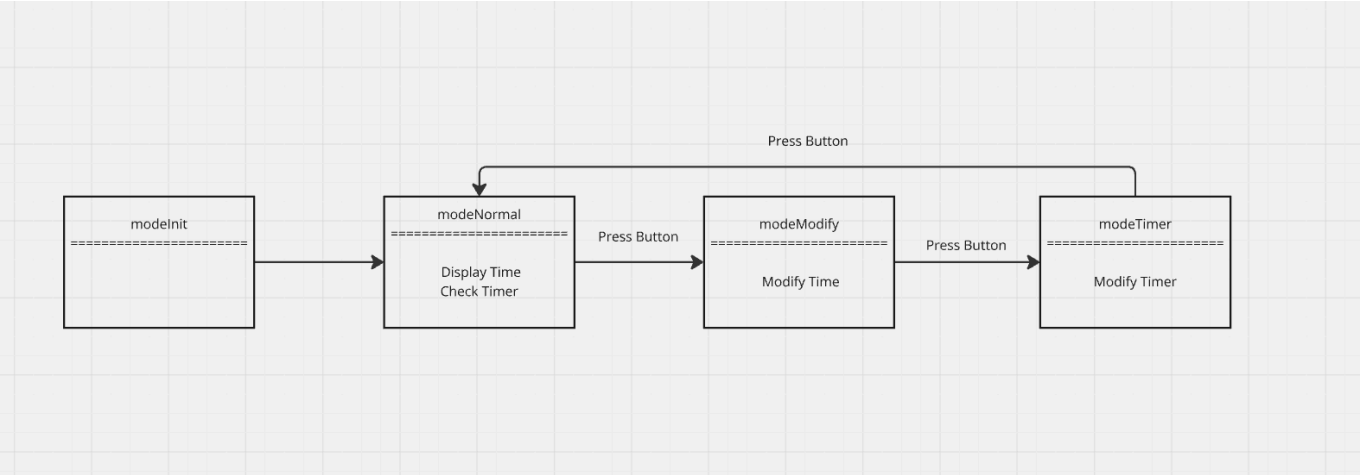
General structure of the fsm() function:

FSM is implemented as a switch-case, in which:

- currentMode is the current state variable.
- Based on the value of currentMode, the system will execute the corresponding functions:
    - Call the processing functions (for example: displayTime(), modifyTime()).
    - Check the button status to switch to another mode.

3. System Implementation

a. Build the fsm() function



- Case modeInit
  - This is the initialization state. When the system starts, the initial mode is set to modeInit. After initialization, the system automatically switches to clock mode (modeNormal).

- Case modeNormal

  - Display real time (hour, minute, second, day, month, year) on LCD screen, and then check and compare current time with timer time to give alarm (if any).
  - Allow user to switch to time setting mode (modeModify) by pressing mode switch button.
  - displayTime(): This function displays all time information on LCD screen (hour, minute, second, day, day, month, year).
  - button_count[0] == 1: Check if mode switch button is pressed or not. If so, the state is changed to modeModify.

- Case modeModify
  - Allows the user to edit the time parameters
  - When the mode switch button is pressed, the system switches to timer mode (modeTimer).
  - modifyTime(): This is the function to edit the time
  - button_count[0] == 1: If the mode switch button is pressed, the system switches to timer mode.

- Case modeTimer
  - Allows the user to edit the timer time and when the timer time is reached, displays an alarm.
  - The user can press the button to return to the time view mode.
  - modifyTimer(): This function is similar to modifyTime(), but is specifically for editing the timer time.
  - button_count[0] == 1: If the mode switch button is pressed, the system returns to the time view mode.

b. <u>Build the modifyTimer() function</u>

The modifyTimer() function focuses on changing the timer time set by the user, which is stored in the timer[] array.

The purpose of modifyTimer():

- Edit the timer time components, including: hours and minutes (or can expand other components such as seconds, days, months if needed).
- The parameter being edited will flash to show the editing status.
- Two buttons are used:
  - "Up arrow" button: Increase the current parameter value.

○ "E" (Enter) button: Save the current parameter and move on to adjusting the next parameter.
● Once completed, the timer time is saved locally in the timer[] array.
The modifyTimer() logic
The modifyTimer() function works in the following steps:
● Flashing the parameter being edited:
○ The current parameter (hour, minute, etc.) will be cleared and displayed again periodically (based on the count_modify counter variable).
○ This creates a flashing effect on the LCD to signal that the user is editing that parameter.
● Incrementing the parameter value when pressing the "up arrow" button:
○ When the "up arrow" button is pressed, the value of the current parameter in the timer[] array is increased.
○ The checkTimer() function is called to check and ensure that the value is valid (e.g. the hour does not exceed 23, the minute does not exceed 59).
● Saving the parameter and moving to the next parameter when the "E" button is pressed:
○ When the "E" button is pressed, the current parameter is saved to the timer[] array.
○ Move to the next parameter (if any). If all parameters have been adjusted, return to the first parameter.

c. Build the modifyTime() function

The modifyTime() function in the source code has the function of editing the timer time, similar to the modifyTimer() function (used to adjust the current time). However, this function is used in the time adjustment mode (modeModify). It allows the user to edit time parameters (hour, minute, second, day, month, year, day) with operations such as:

● Increase the parameter value.
● Save the current parameter and move to the next parameter.
● Flash the parameter being edited to display the status.
General logic of the function:
● Current time parameters are temporarily saved in the temp_time[] array.
● The parameter being edited is determined by the temp variable.
● When editing:
○ The parameter is increased when the "up arrow" button is pressed.
○ The value is checked to ensure validity (for example, the hour does not exceed 24).
● When the "E" button is pressed, the current parameter is saved to the DS3231 and moves to the next parameter.

4. Result

- Video demonstration for these exercises is stored here.
- The source for this exercise is stored here.

Chapter 5: Universal Asynchronous Receiver - Transmitter

1. Requirement Analysis

ID	Feature	Description
1	Ring buffer for UART received data	- Use a ring buffer to store the data received in the UART Receive Interrupt Service Routine. - Set a flag and process the received data in the main thread.
2	Remote timer update through RS-232	Enhance the digital clock application in Lab 4: - Add a mode for updating time via RS232 communication with a computer. - The system will sequentially request and update the hour, minute, and second values. - Once all values are received, they are stored in the DS3231 real-time clock (RTC) module. For example, in the updating hour phase, the sequence is: 1. LCD display “Updating hours” 2. The kit will send **request** “hours” to the PC 3. The PC will **response** a string containings the hour needed to be updated 4. System save the hour value and progress to updating minutes phases

2. System Design

2.1. Ring buffer

- We design a ring buffer with the following attributes:
  - **length** is the number of elements in the buffer.
  - **head** represents the **oldest** element of the buffer.
  - **tail** represents the **newest** element of the buffer.
  - **data** contains the elements of the buffer.
- And define 3 main functions for manipulating the ring buffer
  - **ringBufferPush()** adds a new element to the buffer.
  - **ringBufferPop()** removes the **oldest** element from the buffer and returns its value.
  - **ringBufferPeek()** returns the value of the **oldest** element.

2.2. Finite State machine for remote update values for DS3231 RTC module

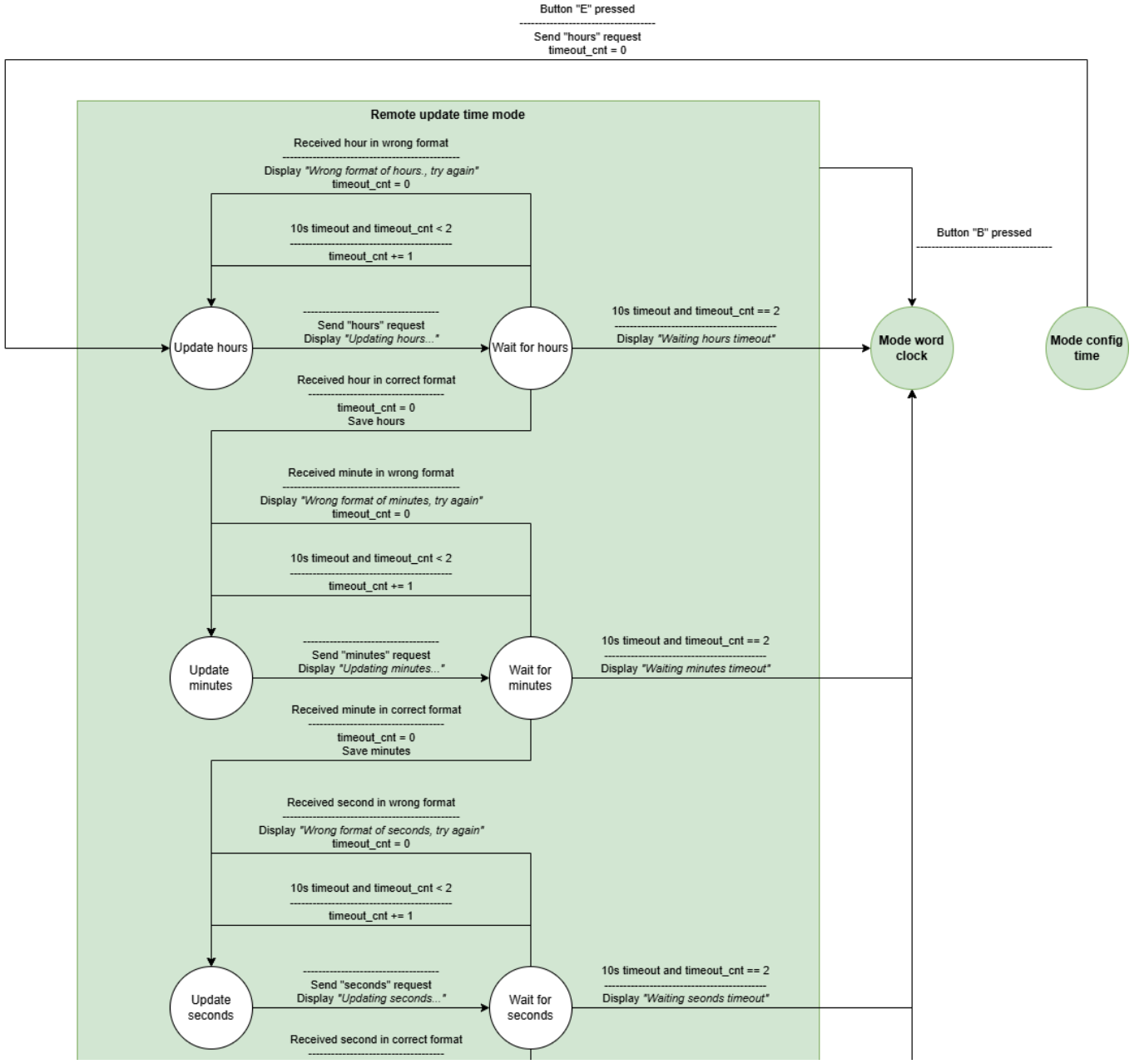


Figure … Finite State Machine of Exercise 5.2 and 5.3 (1/3).

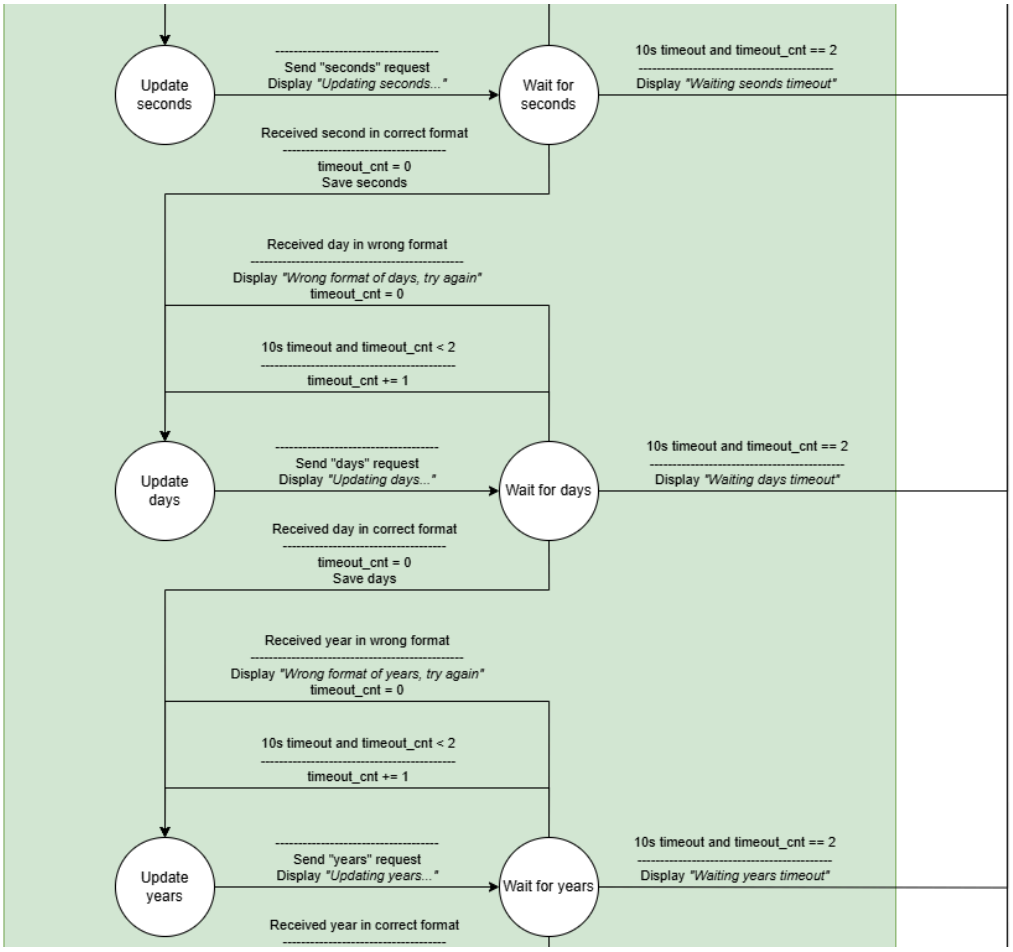


Figure … Finite State Machine of Exercise 5.2 and 5.3 (2/3).

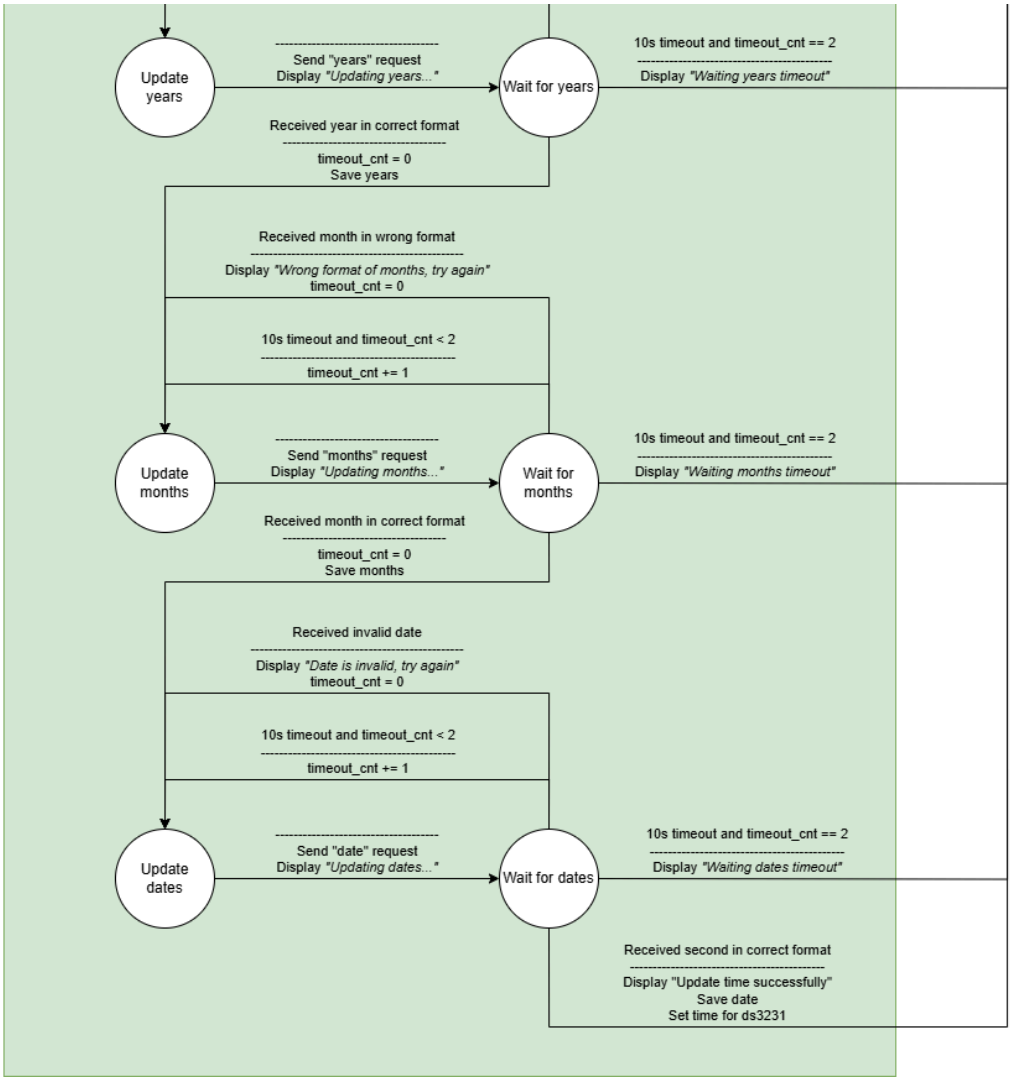


Figure … Finite State Machine of Exercise 5.2 and 5.3 (3/3).

For each value need to be updated, we design two corresponding states

- The ***"Update"*** states are used to send request through the RS232 port to the PC
- The ***"Wait for"*** states are used to
  - Check for the receive flag from the UART ring buffer.
  - Stores, processes data received from UART and checks for its validity with the corresponding value of the DS3231 RTC Module.
  - Checks for timeout (10s) for each update request, manages the number of timeouts and returns the system state to the ***Mode_word_clock*** state when the maximum number of timeouts is exceeded.
- When all responses for every value of the RTC module have been received, we update these values and return the system state to the ***Mode_word_clock*** state

3. System Implementation

3.1. Ring buffer

We implemented the ring buffer as follows:

- **data:** A 1D array with a fixed size defined as RING_BUFFER_MAX_SIZE.
- **head:** The index in the data array representing the **oldest** element in the buffer.
- **tail:** The index in the data array representing the position **where the next new element** will be added (newest element + 1).
- **ringBufferPush()** function adds the specified value to the tail index of the data array and increments tail by one, wrapping around if necessary.

- **ringBufferPop()** function removes the value at the head index, retrieves it, and increments head by one, wrapping around if necessary.
- **ringBufferPeek()** function retrieves the value at the head index without modifying head

3.2. Finite State machine for remote update values for DS3231 RTC module

- We introduced a new ***Mode_remote_config_time*** state to the finite state machine described in Chapter 4.
  - The system transitions to this mode when the **'E'** button is pressed while the system is in the ***Mode_config_time*** state.
  - The system transitions back to the ***Mode_word_clock*** state in the following cases:
    - The user presses the **'B'** button to skip the remote update.
    - Three timeouts (10 seconds each) occur during the update of any value.
    - All values are successfully updated.
- A software timer is used to monitor timeouts.
- All state-switching mechanisms for this feature are implemented using the finite state machine described in Part 2.

4. Results

- Video demonstration for these exercises is stored here.
- The source for this exercise is stored here.

Chapter 6: ADC - PWM

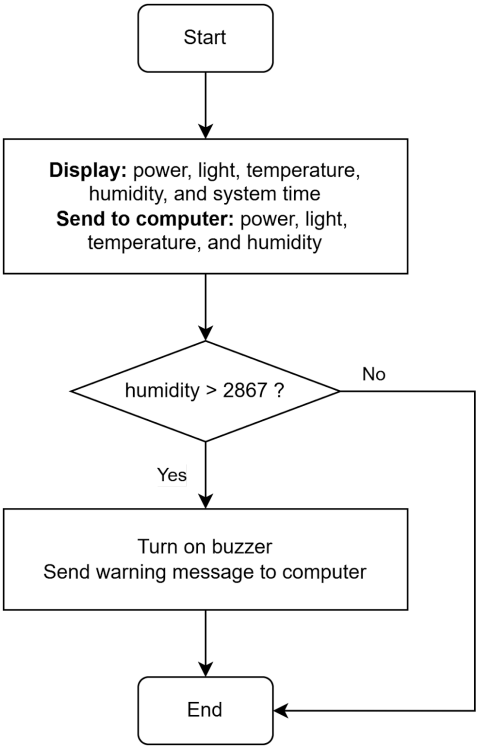
1. Requirement Analysis

ID	Feature	Description
1	Measurement	Display measured parameters on the screen including: - Power consumption - Light - Temperature - Humidity - System time The measured data will be sent to the computer via UART. When the humidity exceeds the threshold (>70%), the system enters the warning state, specifically: - Buzzer will alarm every 1s. - Warning messages will be sent to the computer via UART every 1s.
2	Line chart drawing	Mandatory requirement: - Draw a line chart showing power consumption over time. - Sampling period is 15s. Additional constraints: - Displays the 9 most recent measured values.

2. System Design

2.1. Measurement

- The formula for calculating power is P = UI, where P represents the power in watts, U is the voltage in volts, and I is the current in amperes.
- With the measured ADC values ranging from 0 to 4095, the humidity threshold of 70% of the maximum value is determined to be approximately 2867.
- The flowchart for such a system is described below.



2.2. Line chart drawing

- Use a queue structure to store measured power consumption values, with parameters including:
  - *arr*: one-dimensional array used to store values.
  - *front:* index that refers to the oldest value in the queue.
  - *rear:* index where the next value is ready to be written, that is, one unit after the newest value in the queue.
- And the implemented functions include:
  - *init*: initialize the queue.
  - *insert*: add a new value to the queue, in two cases:
    - If the queue is not full, it is a normal enqueue action.
    - Conversely, if the queue is full, it simultaneously dequeues the oldest value and enqueues the newest value.
- Also, the following parameters are defined to support better chart display:
  - *HEIGHT*: height of the chart, in pixels.
  - *MIN*: minimum value of power consumption, in milliwatts.
  - *MAX*: maximum value of power consumption, in milliwatts.

3. System Implementation

3.1 Measurement

- Calculating power consumption in milliwatts: *sensor_GetVoltage* * *sensor_GetCurrent* * 0.001, where *sensor_GetVoltage*, *sensor_GetCurrent* are functions in the template library.
- The light, temperature, and humidity parameters can be obtained directly by calling the *sensor_GetLight*, *sensor_GetTemperature*, and *sensor_GetPotentiometer* functions from the template library, respectively.
- Initialization of the environment for system time and UART is done similarly to Lab 4 and Lab 5, so it is not presented again.

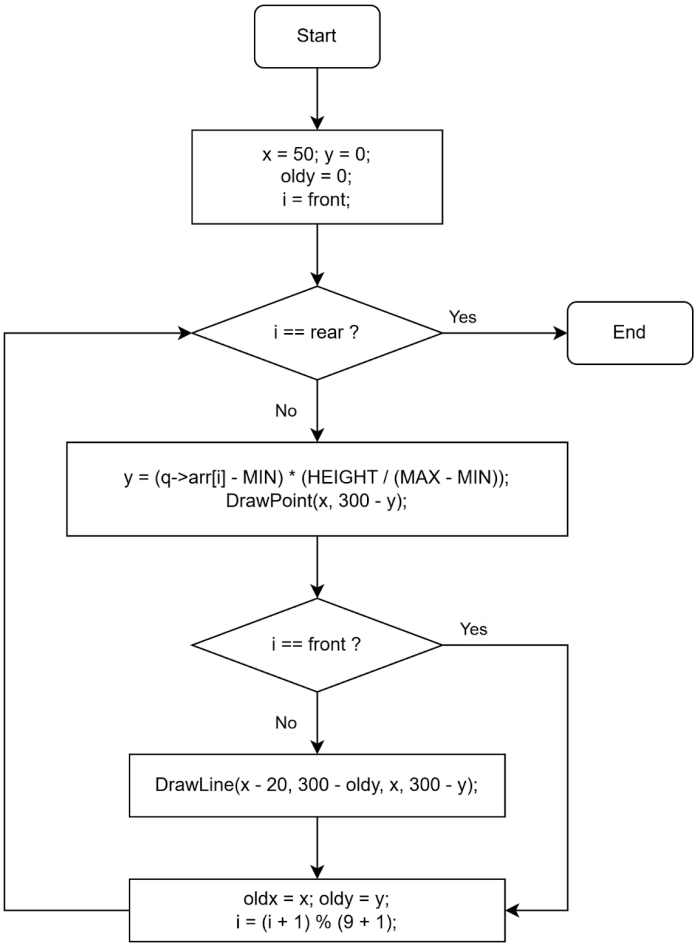
3.2 Line chart drawing

- First of all, draw the coordinate axes with the horizontal axis representing time and the vertical axis representing power consumption as follows.

```
lcd_DrawLine(40, 300, 40, 300 - HEIGHT, WHITE);
```

```
    lcd_DrawLine(40, 300, 220, 300, WHITE);
    lcd_ShowIntNum(10, 300 - 10, MIN, 2, WHITE, BLACK, 16);
    lcd_ShowIntNum(10, 300 - HEIGHT - 10, MAX, 2, WHITE, BLACK, 16);
```

- To ensure aesthetics, the 9 power consumption values will be represented at the x coordinate as follows: 50, 70, …, 210. With the definition of the queue as presented, it is clear that the closer the power consumption values are to the front, the older they are, resulting in their x-time coordinates being smaller.
- Meanwhile, determining the y coordinates for the power consumption values is calculated using the formula: y = (value - MIN) * (HEIGHT / (MAX - MIN)), where HEIGHT, MAX, MIN, are parameters defined in the System Design section.
- In summary, the process of drawing the chart at each sampling is as shown in the flowchart below.

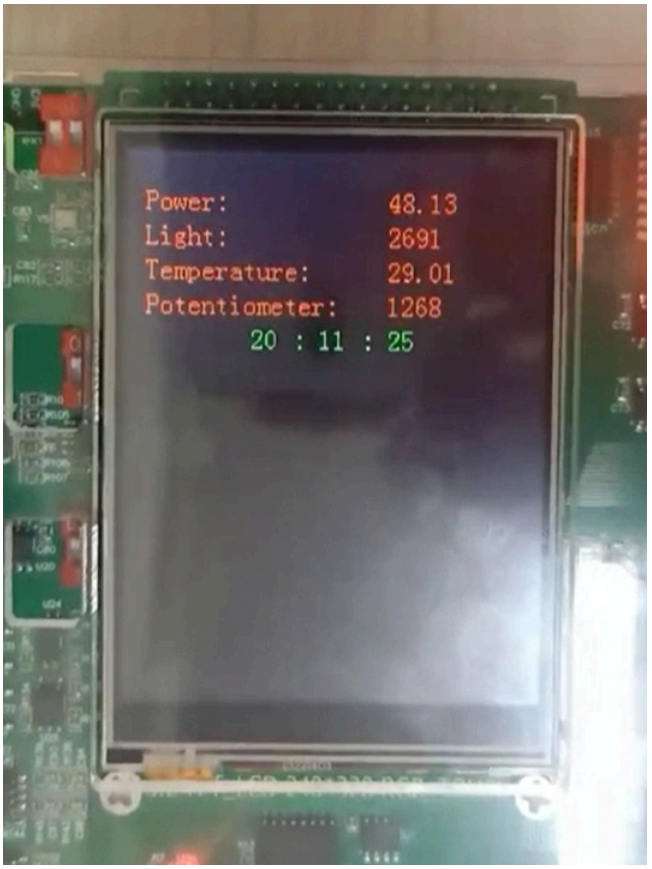


- Note that the use of 300 - y and 300 - oldy as in the flowchart is to adjust the position of y to match the coordinate axis drawn from the previous step.

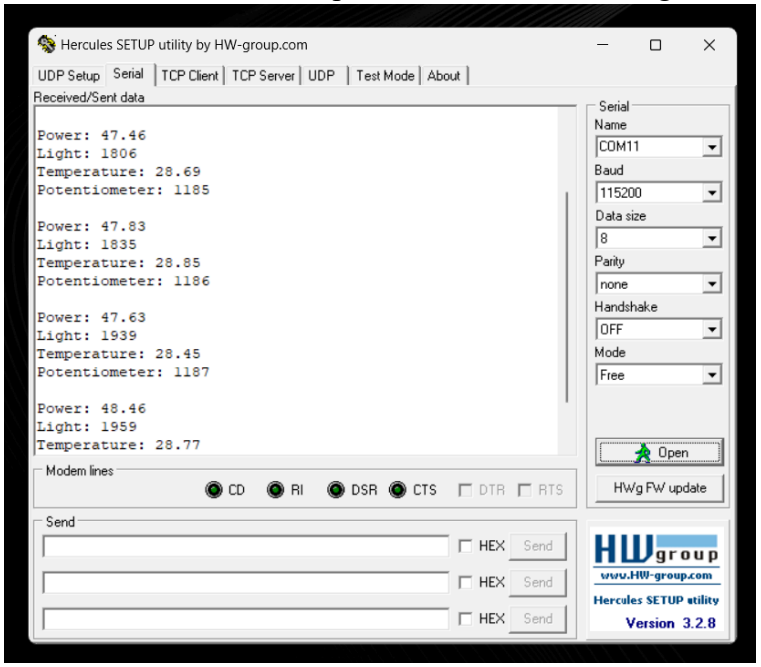
4. Results

4.1. Measurement

- The results displayed on the kit are saved here.
- A sample result is shown below:

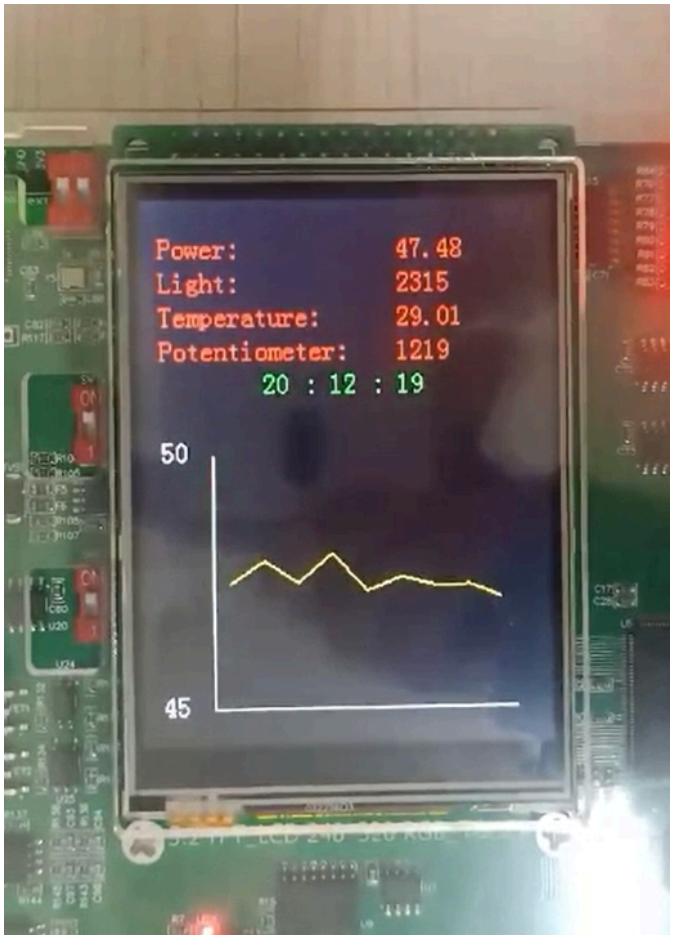


- The result received via UART on the computer is shown as the image below:



4.2 Line chart drawing

- The results displayed on the kit are saved here.
- A sample result is shown below:


Power: 47.48
Light: 2315
Temperature: 29.01
Potentiometer: 1219
20 : 12 : 19
50
45

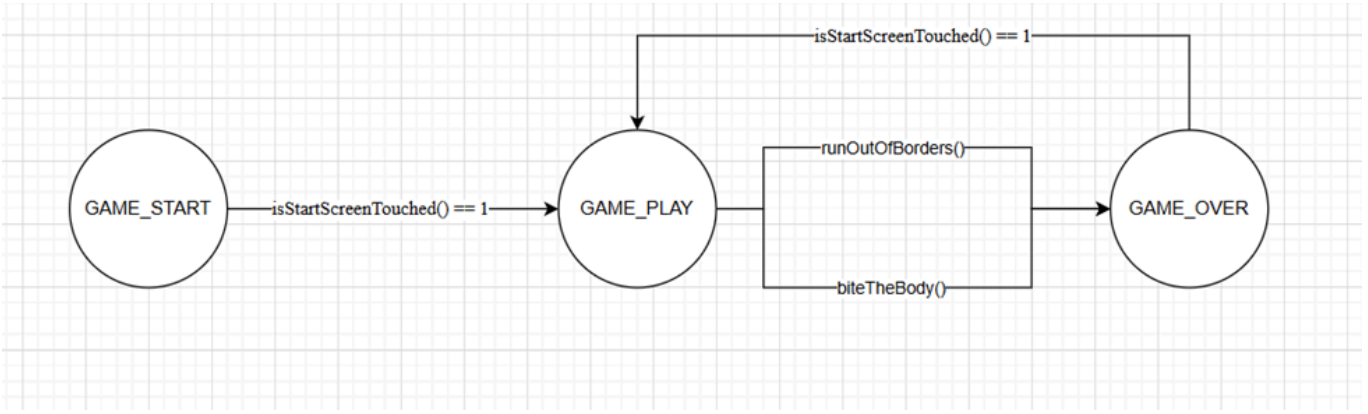
Chapter 7: LCD Touch

1. Requirement Analysis

Realize the classic snake game on LCD. In which, all interactions with the user will be done on the touch screen, for example:

- Touch the "Start" button on the screen to start the game.
- Touch the navigation buttons on the screen to control the snake.

2. System Design



Overview of the flow of activities

1. Start: The game starts with the GAME_START state, displaying the start screen.
2. Play: When the player touches the screen, the game enters the GAME_PLAY state. In this state, the game continuously processes input from the player and updates the snake's movements.
3. End: If the snake collides with a wall or its body, the game enters the GAME_OVER state, displaying the end screen. The player can restart the game.

3. System Implementation

- GAME_START State
  - Purpose: Display the start screen and wait for the player to start the game.
  - Main actions:
    - Call initializeButtons() to set up the controls.
    - Display the start screen interface via displayStartScreen().
    - Check if the screen is touched or not using the isStartScreenTouched() function.
  - State transition conditions:
    - If the player touches the screen:
      - The state changes to GAME_PLAY.
      - Call initializeGame() to initialize the necessary parameters for the game.
- GAME_PLAY State
  - Purpose: Controls logic while the game is in progress, including handling input, moving the snake, and updating the game state.
  - Main actions:

    - Button handling: If button_read_flag is enabled, call handleInput() to handle the snake's direction, then reset the button timer (setTimer_button(5)).

    - Handle snake movement:

• If snake_move_flag is enabled: Calculate the next position of the snake's head based on the direction (UP, DOWN, LEFT, RIGHT).
• Check the condition on the next cell: Collide with a wall or the snake itself: End the game (transition to GAME_OVER).
• Eat a fruit (value 2 on the grid):
• Increase the score (score++).
• Call advanceSnakeHead() to add a new element to the snake's head.
• Generate a new fruit using generateFruit().
• Normal movement:
• Call advanceSnakeHead() to update the snake's head.
• Call removeSnakeTail() to remove the tail.
• Redraw the screen using renderScreen().
• Reset the snake's movement timer (setTimer_snake(500)).

• GAME_OVER state
○ Purpose: Displays the end screen and allows the player to restart the game.
○ Main actions:
• Displays the start screen via displayStartScreen().
○ Transition conditions:
• If the player touches the screen:
• The state changes to GAME_PLAY.
• Calls initializeGame() to reset the game state (including the grid and snake positions).
• Calls initializeGame() to reset the game state (including the grid and snake positions).

4. *Results*

- Video demonstration for these exercises is stored here.
- The source for this exercise is stored here.

Chapter 8: ESP8266 - Wifi

1. Requirement Analysis

ID	Feature	Description
1	Temperature reading	Measure and log the temperature from the kit every 30 seconds.
2	Temperature display	Display the collected temperature data on the Adafruit dashboard in graph form.

2. System Design

2.1. Architecture Design

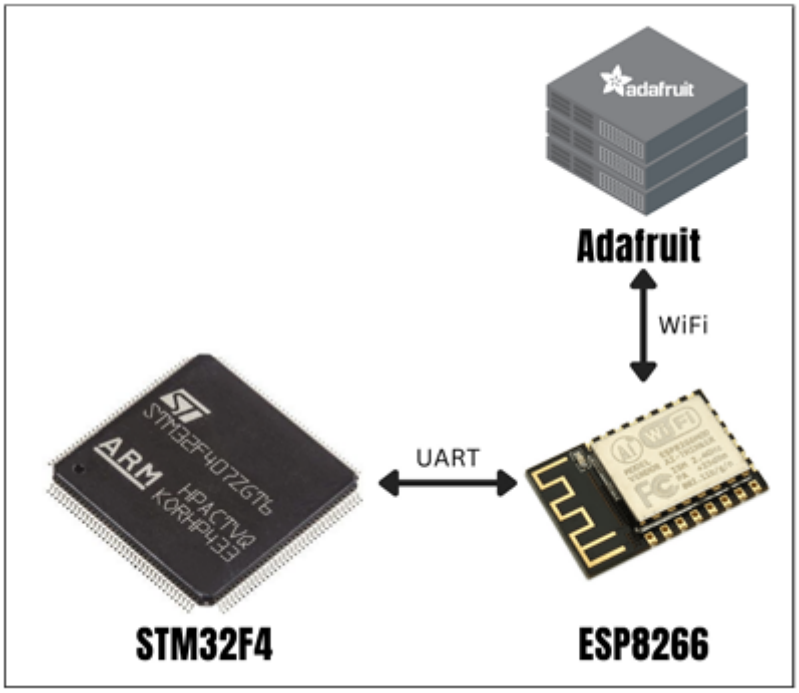


Figure … Architecture Design for chapter 8.

- The STM32 microcontroller measures environmental temperature using a temperature sensor.
- The collected temperature data is then transmitted to the ESP8266 through UART.
- The ESP8266 publishes it to the Adafruit server through MQTT protocol.

2.2. Finite State Machine

2.2.1 STM32's finite state machine

The STM32 must wait for the ESP to complete its initialization (connecting to Wi-Fi and the Adafruit MQTT server) before starting communication. This behavior can be modeled using the FSM below.

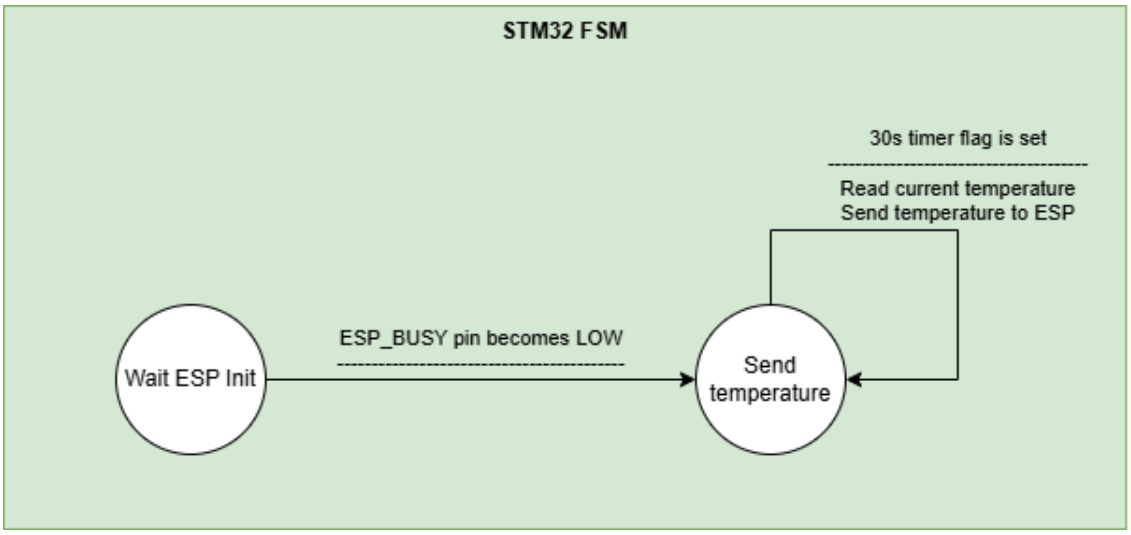


Figure … Finite State Machine for STM32 in chapter 8

2.2.2. ESP8266's finite state machine

The main responsibility of the ESP8266 is to receive temperature data from the STM32 via UART and publish it to the Adafruit MQTT server. This behavior can be modeled using the FSM below.

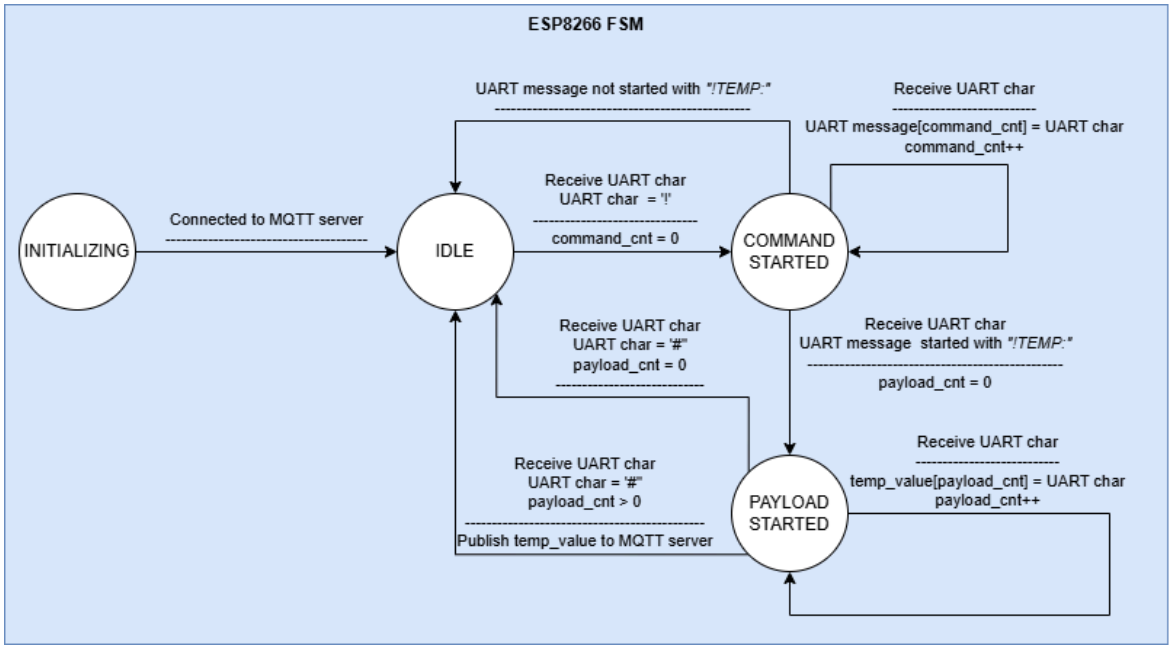


Figure … Finite State Machine for EPS8266 in chapter 8

- The ***INITIALIZING*** state represents the system's process of connecting to the Wi-Fi network and the MQTT server.
- Once initialization is complete, the system transitions to the ***IDLE*** state, where it waits for new data from the STM32.
- When new data is received, the system enters the ***COMMAND_STARTED*** state to verify if the data follows the expected format for temperature readings, i.e., ***!TEMP:<temperature>#***
    - If the command is in the correct format, the system moves to the ***PAYLOAD_STARTED*** state to read and store the payload (temperature) sent by the STM32 until the # character is received, indicating the end of the message. Once the message is complete, the ESP8266 publishes the stored data to the MQTT server and returns to the ***IDLE*** state.
    - Otherwise, if the command is not in the correct format, the system immediately returns to the ***IDLE*** state.

3. System Implementation

3.1. STM32

- The 30-second interval for reading the temperature and sending it to the ESP32 via UART is managed using a software timer.
- The temperature is read using the **sensor_init()** and **sensor_read()** functions described in Chapter 6. The float temperature value is then converted to a string, formatted into the predefined structure ***(!TEMP:<temperature>#)***, and sent to the ESP8266 via UART using the **uart_init_esp()** and **uart_EspSendBytes()** functions.
- All state-switching mechanisms are implemented using the finite state machine outlined in Part 2.

3.2. ESP8266

- We use the following libraries:
  - **ESP8266WiFi** for connecting to Wi-Fi.
  - **Adafruit_MQTT** to establish a connection and publish messages to the MQTT server.
  - **Serial** to process UART-received data.
- All state-switching mechanisms are implemented using the finite state machine outlined in Part 2.

4. Results

- We attempted to establish UART communication between the STM32 and ESP8266 multiple times but were unsuccessful. To troubleshoot, we decided to test the UART interface on the board separately: First, we sent the temperature data to the RS232 port and used the Hercules terminal to verify the received data. When new data appeared on the Hercules terminal, we copied it and used the Arduino Serial Monitor to transmit this data to the ESP8266 via UART.
- By performing these tests, all data were sent successfully, leading us to suspect issues with the UART connection on the BKIT Arm4 board.
- The video of our experiment is store here, divided into four main panels:
  1. The Arduino Serial Monitor.
  2. The Hercules terminal.
  3. The LCD on the board (to display the read temperature and verify it against the temperature shown on the Hercules terminal).
  4. The Adafruit dashboard (to confirm the correct data was sent to the server).
- The source code of the STM32 is stored here.
- The source code of the ESP8266 is stored here.