### Embedded Systems Group Work

Smart Lightning System

Members:

22/00105 Don’EL Mageni

20/02539 Casper Wandera

Silvester Oluoch 19/04131

Raymond Kipkogei 19/02816

### 1. Problem Identification (10%)

**Objective**: Design a system to monitor and count the number of lights on in a smart home setup where each of the 8 rooms is equipped with a light.

#### Literature Review:

**Existing Solutions**:

1. **Philips Hue**: A widely used smart lighting system that allows control of individual lights through a mobile app. However, it does not inherently provide a real-time count of lights that are on.
2. **Lutron Caseta**: A smart lighting control system known for its reliability and integration with smart home ecosystems. It offers robust control features but lacks a straightforward, real-time light status monitoring and counting system.
3. **SmartThings**: A versatile home automation platform that can integrate various smart devices, including lights. While it supports monitoring and control, it does not provide a simple way to count the number of active lights in real-time without complex configurations.

**Need for Improvement**:

* **Real-Time Monitoring**: Many existing systems do not provide real-time updates on the status of all lights in a straightforward manner.
* **Cost Efficiency**: High-end smart lighting systems can be expensive, highlighting a need for a more cost-effective solution.
* **User-Friendly Interface**: Existing solutions often require extensive setup and user interaction through apps, which can be cumbersome.

#### Scope and Objectives:

**Scope**:

* **Real-Time Status Monitoring**: The system should continuously monitor the status of lights in 8 rooms.
* **Count of Active Lights**: The system should count the number of lights that are currently on and store this count in an easily accessible format.
* **User Notification**: Potential for future expansion to notify users when lights are on, for energy management.

**Objectives**:

1. **Accuracy**: Ensure the system accurately counts the number of lights that are on.
2. **Efficiency**: The system should perform this monitoring with minimal power consumption.
3. **Real-Time Updates**: The count of lights should be updated in real-time.
4. **Cost-Effective**: Use affordable components and tools to implement the system.

By identifying the problem and outlining the need for an innovative solution, we can proceed to design and implement a smart home lighting system that addresses these gaps in current offerings. This system will offer real-time monitoring and counting of active lights, providing a practical and efficient solution for smart home energy management.

### 2. System Design (20%)

**Functional Requirements (10%)**:

**Inputs**:

* 1. An 8-bit unsigned character A where each bit represents the status of a room's light (1 for on, 0 for off).

**Outputs**:

* 1. An 8-bit unsigned character B that holds the count of lights that are currently on.

**Key Processes**:

* 1. Continuously monitor the status of lights.
  2. Count the number of bits set to 1 in the 8-bit character A.
  3. Update the count in B in real-time.

**Non-Functional Requirements (10%)**:

**Performance**:

* 1. The system should update the light count in B at least every second.

**Reliability**:

* 1. The system must be highly reliable with minimal error tolerance in counting the lights.

**Usability**:

* 1. The system should be straightforward for users to understand and interact with.
  2. The status and count should be easily interpretable.

**Constraints**:

* 1. **Power Consumption**: The system should operate with minimal power usage to be efficient for a smart home setup.
  2. **Cost**: The system should be designed to be cost-effective, using affordable components.
  3. **Environmental Conditions**: The system should operate reliably under typical indoor environmental conditions.

### System Design Breakdown

#### Inputs

* **A (8-bit unsigned character)**:
  + Each bit in A represents a room's light status:
    - A0 to A7 correspond to the status of rooms 1 to 8, respectively.
    - Bit value 1 indicates the light is on.
    - Bit value 0 indicates the light is off.

#### Outputs

* **B (8-bit unsigned character)**:
  + Represents the number of lights that are currently on (0 to 8).

#### Processes

1. **Monitoring**:
   * The system will continuously read the input A to determine the status of each room's light.
2. **Counting**:
   * The system will use a loop to check each bit in A.
   * For each bit that is set to 1, it will increment a counter.
3. **Updating**:
   * The system will update the value of B with the count of 1s in A.
   * This ensures B always holds the real-time count of lights that are on.

### Functional Flow

1. **Initialization**:
   * Set B to 0.
2. **Main Loop**:
   * Initialize a counter to 0.
   * Iterate over each bit in A.
     + Check if the bit is set to 1.
     + Increment the counter for each bit set to 1.
   * Update B with the counter value.
   * Repeat the process to ensure real-time monitoring and updating.

### Example Scenario

**Initial State**:

* A = 0b01010101 (binary representation of 85 in decimal)

**Room Light Status**:

* Room 1: On
* Room 2: Off
* Room 3: On
* Room 4: Off
* Room 5: On
* Room 6: Off
* Room 7: On
* Room 8: Off

**Output**:

* B = 4 (4 lights are on)

**Dynamic Update**: If the status changes to A = 0b11110000 (binary representation of 240 in decimal)

**Room Light Status**:

* Room 1: On
* Room 2: On
* Room 3: On
* Room 4: On
* Room 5: Off
* Room 6: Off
* Room 7: Off
* Room 8: Off

**Output**:

* B = 4 (4 lights are on)

### 3. State Machine Design using RIBS (20%)

**Objective**: Create a graphical representation of the synchronous state machines (synchSMs) using RIBS, ensuring the state machine accurately represents the logic and transitions required to meet the functional requirements of the smart home lighting system.

#### State Machine Design

**States**:

1. **s1 (Monitoring State)**: Continuously checks the status of the lights.
2. **s2 (Counting State)**: Counts the number of lights that are currently on and updates the count in B.

**Transitions**:

* From **s1** to **s2**: When the status of A is read.
* From **s2** to **s1**: After updating the count in B.

**State Machine Diagram**:

plaintext

Copy code

+-----------------+

| |

| s1 |

| Monitoring |

| |

+--------+--------+

|

v

+--------+--------+

| |

| s2 |

| Counting |

| |

+--------+--------+

|

v

+--------+--------+

| |

| s1 |

| Monitoring |

| |

+-----------------+

#### State Machine Logic in RIBS

**State** s1 **(Monitoring State)**:

* + Action: Continuously monitor the status of lights by reading the value of A.

**State** s2 **(Counting State)**:

* + Action: Count the number of bits set to 1 in A and update the count in B.
  + Transition: After updating B, transition back to s1.

### Generated C Code for State Machine

c

#include "rims.h"

/\* Define user variables and functions for this state machine here. \*/unsigned char value = 0;unsigned char SM1\_Clk;

void TimerISR() {

SM1\_Clk = 1;

}

enum SM1\_States { SM1\_s1, SM1\_s2 } SM1\_State;

TickFct\_State\_machine\_1() {

switch(SM1\_State) { // Transitions

case -1:

SM1\_State = SM1\_s1;

break;

case SM1\_s1:

if (1) { // Always transition to counting state

SM1\_State = SM1\_s2;

}

break;

case SM1\_s2:

if (1) { // Always transition back to monitoring state

SM1\_State = SM1\_s1;

}

break;

default:

SM1\_State = SM1\_s1;

} // Transitions

switch(SM1\_State) { // State actions

case SM1\_s1:

break;

case SM1\_s2:

value = 0;

for (unsigned char i = 0; i < 8; i++) {

if ((A & (0x01 << i)) != 0) {

value++;

}

}

B = value;

break;

default: // ADD default behaviour below

break;

} // State actions

}

int main() {

const unsigned int periodState\_machine\_1 = 1000; // 1000 ms default

TimerSet(periodState\_machine\_1);

TimerOn();

SM1\_State = -1; // Initial state

B = 0; // Init outputs

while(1) {

TickFct\_State\_machine\_1();

while(!SM1\_Clk);

SM1\_Clk = 0;

} // while (1)

} // Main

#### Explanation of Code

**Initialization**:

* + TimerISR(): ISR for the timer to manage the state machine clock.
  + enum SM1\_States { SM1\_s1, SM1\_s2 } SM1\_State: Enumeration of states.
  + TickFct\_State\_machine\_1(): Tick function to handle state transitions and actions.

**State Transitions**:

* + case -1: Initial state transition to SM1\_s1.
  + case SM1\_s1: Transition to SM1\_s2 to count the bits.
  + case SM1\_s2: Transition back to SM1\_s1 after counting.

**State Actions**:

* + case SM1\_s1: No action, just monitoring.
  + case SM1\_s2: Count the number of lights on by checking each bit in A and updating B.

### 4. Implementation and Coding using RIMS (20%)

**C Code for Embedded System:**

Below is the provided C code using RIMS for the state machine. You’ll need to compile, execute, and debug this code to ensure it functions as expected.

c

#include "rims.h"

/\* Define user variables and functions for this state machine here. \*/unsigned char A = 0; // Input value for light statusunsigned char B = 0; // Output value for count of lights onunsigned char SM1\_Clk;

void TimerISR() {

SM1\_Clk = 1;

}

enum SM1\_States { SM1\_s1, SM1\_s2 } SM1\_State;

void TickFct\_State\_machine\_1() {

switch(SM1\_State) { // Transitions

case -1:

SM1\_State = SM1\_s1;

break;

case SM1\_s1:

SM1\_State = SM1\_s2;

break;

case SM1\_s2:

SM1\_State = SM1\_s1;

break;

default:

SM1\_State = SM1\_s1;

} // Transitions

switch(SM1\_State) { // State actions

case SM1\_s1:

// No action, just monitoring

break;

case SM1\_s2:

// Count the number of bits set to 1 in A and update B

{

unsigned char value = 0;

for (unsigned char i = 0; i < 8; i++) {

if ((A & (0x01 << i)) != 0) {

value++;

}

}

B = value;

}

break;

default:

break;

} // State actions

}

int main() {

const unsigned int periodState\_machine\_1 = 1000; // 1000 ms default

TimerSet(periodState\_machine\_1);

TimerOn();

SM1\_State = -1; // Initial state

B = 0; // Init outputs

while(1) {

TickFct\_State\_machine\_1();

while(!SM1\_Clk);

SM1\_Clk = 0;

} // while (1)

} // Main

**Compile, Execute, and Debug:**

1. **Compile the Code:** Use your embedded system’s compiler to compile the code. Make sure there are no syntax errors or missing libraries.
2. **Execute the Code:** Upload the compiled code to your embedded system or simulator.
3. **Debug:** Use debugging tools to step through the code, set breakpoints, and check variable values. Ensure that the state transitions and outputs are as expected.

**Integrate RIBS Code:**

* **Combine State Machine Code:** Ensure the code generated by RIBS is correctly integrated into your main system code. You may need to adjust the state transitions or variable names to match your overall system design.

### 5. Timing Analysis using RITS (10%)

**Generate Timing Diagrams:**

1. **Use RITS Tool:** Import your system code into RITS to generate timing diagrams. Focus on critical sections, such as state transitions and output updates.

**Analyze Timing Diagrams:**

* **Check Update Frequency:** Ensure the state machine updates every second as required.
* **Verify Real-Time Performance:** Confirm that transitions between states and updates to B occur within the expected time frame.

**Document Timing Issues:**

* **Identify Issues:** Note any discrepancies between the expected and actual timing.
* **Resolution:** Describe how you resolved any issues, such as adjusting timer intervals or optimizing code for performance.

### 6. Testing and Validation (10%)

**Develop Test Cases:**

**Functional Tests:** Test the system with different inputs to ensure it correctly counts the number of lights that are on. For example:

* + **Test Case 1:** A = 0b11111111 (All lights on, expect B = 8)
  + **Test Case 2:** A = 0b00000000 (No lights on, expect B = 0)
  + **Test Case 3:** A = 0b10101010 (Four lights on, expect B = 4)

**Performance Tests:** Verify that the system meets performance criteria, such as update frequency and response time.

**Conduct Rigorous Testing:**

* **Run Tests:** Execute the test cases and check if the system behaves as expected.
* **Fix Defects:** Address any issues identified during testing, such as incorrect counts or timing problems.

**Validate Against Requirements:**

* **Check Requirements:** Ensure the system meets all specified functional and non-functional requirements, including accuracy, efficiency, and user interface.