**T.R**
**ESKISEHIR OSMANGAZI UNIVERSITY**
**DEPARTMENT OF ELECTRICAL-ELECTRONICS ENGINEERING**
**AND**
**DEPARTMENT OF COMPUTER ENGINEERING**

# INTRODUCTION TO MICROCOMPUTERS TERM PROJECT

*Smart Home System*

*152120211096, Abdullah Eren COŞKUN, Computer Engineering*
*152120221053, Meryem DİNÇ, Computer Engineering*
*151220212126, Buse BAYRAMLI, Electrical and Electronics Engineering*
*151220222101, Yağız SÜRÜCÜ, Electrical and Electronics Engineering*
*151220222104, Furkan ARSLAN, Electrical and Electronics Engineering*

**January 2026**

## 1. Introduction

The objective of this project is to design and implement a complete "Home Automation System" using the PIC16F877A microcontroller, demonstrating the concepts of embedded system design and real-time control. The system is divided into two major functional subsystems to facilitate convenience and energy efficiency at home. The first one, the "Home Air Conditioner System" (Board #1), regulates the temperature inside the house through heaters, coolers, and fans. This is achieved by taking input from the user through a keypad and/ or real-time data from temperature sensors. A second subsystem, the "Curtain Control System" (Board #2), automatically opens and closes window curtains using a stepper motor based on the ambient conditions captured by Light Dependent Resistors (LDR) sensors.

The project combines low-level hardware control with high-level software management. All embedded logic is prepared in the Assembly language and simulated on PICSimLab, enabling precise and advanced control over hardware peripherals and interrupts. It allows remote monitoring and control through the PC application by interacting with the microcontrollers via the UART serial communication protocol. This report describes the design methodology, which includes the communication API and hardware architecture. The modular structure in software enables smooth interaction between the user interface and embedded hardware.

# 2. Design

## 2.1 Home Air Conditioner System

Board 1 is a Smart Temperature Control System based on the PIC16F877A microcontroller that monitors ambient temperature through an ADC sensor and automatically controls a heater or cooler to maintain a user-defined target temperature within the strict range of 10.0°C to 50.0°C. The system features a multiplexed 4-digit 7-segment display that rotates between showing the target temperature, ambient temperature, and fan speed, while a 4x4 matrix keypad allows users to manually input new target values by pressing 'A' to start, entering digits, using '*' for the decimal point, and confirming with '#'. Additionally, the system supports remote monitoring and control through UART serial communication at 9600 baud, enabling external applications to read sensor values and set the target temperature via a defined command protocol.

### 1. UART Module

The UART Module enables serial communication between the PIC16F877A microcontroller and a PC application. It operates at 9600 baud rate with 8 data bits, no parity, and 1 stop bit. RC6 is used for TX (transmit) and RC7 for RX (receive). The module supports 7 commands: 5 GET commands for reading temperature and fan values, and 2 SET commands for controlling the target temperature remotely.

#### UART_INIT

This function initializes the UART peripheral for 9600 baud communication at 4MHz oscillator. It switches to Bank 1 and loads 25 into SPBRG for the baud rate divisor. It configures TXSTA with 0x24 (BRGH=1, TXEN=1, async mode). Switching back to Bank 0, it sets RCSTA to 0x90 (SPEN=1, CREN=1) to enable the serial port and continuous receive mode.

#### UART_CHECK_DATA

This function polls the UART receive flag and processes incoming commands. It checks PIR1 bit 5 (RCIF) to see if data arrived. It handles overrun errors (OERR) by calling UART_ERR_OERR, and framing errors (FERR) by calling UART_ERR_FERR. For valid data, it reads RCREG into uart_rx_data. The function decodes commands: 11xxxxxx goes to UART_CMD_SET_INT, 10xxxxxx goes to UART_CMD_SET_FRAC, 0x01-0x05 go to respective GET handlers.

#### UART_ERR_OERR

This function handles UART overrun error by toggling the CREN bit. It clears RCSTA bit 4 to disable receiver, then sets it again to re-enable, clearing the error condition.

**UART_ERR_FERR**

This function handles UART framing error by reading and discarding the corrupted byte from RCREG.

**UART_CMD_SET_INT**

This function handles the SET Target Integer command (11xxxxxx format). It masks the lower 6 bits using AND 0x3F and stores in uart_temp_int. It sets bit 0 of update_flag to indicate an integer value is pending, waiting for the fractional part to complete the update.

**UART_CMD_SET_FRAC**

This function handles the SET Target Fractional command (10xxxxxx format). It masks the lower 6 bits and stores in uart_temp_frac. If update_flag bit 0 is not set (no pending integer), it returns without action. Otherwise, it validates the complete value: rejects if uart_temp_int < 10 or > 50, and if uart_temp_int = 50, rejects any non-zero fractional part. Valid values are committed to temp_target_int and temp_target_frac, then TEMP_UPDATE_LOGIC is called.

**UART_CMD_GET_TARGET_FRAC**

This function responds to the GET Target Fractional command (0x01). It loads temp_target_frac into W and calls UART_SEND_DATA to transmit the fractional part of the target temperature.

**UART_CMD_GET_TARGET_INT**

This function responds to the GET Target Integer command (0x02). It loads temp_target_int into W and calls UART_SEND_DATA to transmit the integer part of the target temperature.

**UART_CMD_GET_AMBIENT_FRAC**

This function responds to the GET Ambient Fractional command (0x03). Since the ADC provides only integer values, it transmits 0 as the fractional part.

**UART_CMD_GET_AMBIENT_INT**

This function responds to the GET Ambient Integer command (0x04). It loads temp_ambient (current ambient temperature) into W and calls UART_SEND_DATA.

**UART_CMD_GET_FAN**

This function responds to the GET Fan Speed command (0x05). It loads fan_status into W and calls UART_SEND_DATA to transmit the current fan speed (0 or 5).

**UART_SEND_DATA**

This function transmits a single byte over UART. It switches to Bank 1 and waits for TXSTA bit 1 (TRMT) to be set, indicating the transmit buffer is empty. It then switches to Bank 0 and writes the byte to TXREG.

**2. Keypad Module**

The Keypad Module interfaces with a 4x4 matrix keypad for manual temperature input. The keypad is connected with rows on RB0-RB3 (outputs) and columns on RB4-RB7 (inputs with pull-ups enabled). It provides a complete input interface: press 'A' to start, enter digits, '*' for decimal point, and '#' to confirm.

**Key Constants**

The module uses predefined constants for special keys: KEY_A = 0xA0 (160), KEY_STAR = 0xB0 (176), KEY_HASH = 0xC0 (192), KEY_NONE = 0xFF (255). Numeric keys 0-9 retain their face values, while B=11, C=12, D=13.

**KEYPAD_CHECK_MAIN**

This function is the main keypad handler called from the main loop. It calls KEYPAD_SCAN to detect any pressed key. If key_last_pressed is not 0xFF (a key was pressed), it calls KEYPAD_PROCESS_KEY to process it, then clears key_last_pressed to 0xFF.

**KEYPAD_SCAN**

This function scans the 4x4 keypad matrix using row scanning technique. It sets key_last_pressed to 0xFF initially. For each row (RB0-RB3), it drives that row LOW while keeping others HIGH, calls KEYPAD_DEBOUNCE for signal settling, then reads columns RB4-RB7. A LOW on any column indicates a key press at that intersection. The function maps keys: Row 1 = 1,2,3,A; Row 2 = 4,5,6,B; Row 3 = 7,8,9,C; Row 4 = *,0,#,D with codes 0-15.

**KEYPAD_ASSIGN_1 through KEYPAD_ASSIGN_9, KEYPAD_ASSIGN_0**

These are numeric key detection handlers. Each stores its corresponding digit value (0-9) in key_last_pressed, calls KEYPAD_WAIT_RELEASE for debouncing and key release detection, then returns.

**KEYPAD_ASSIGN_A, KEYPAD_ASSIGN_B, KEYPAD_ASSIGN_C, KEYPAD_ASSIGN_D**

These handle letter keys. KEYPAD_ASSIGN_A stores KEY_A (0xA0), KEYPAD_ASSIGN_B stores 11, KEYPAD_ASSIGN_C stores 12, KEYPAD_ASSIGN_D stores 13. Each calls KEYPAD_WAIT_RELEASE before returning.

**KEYPAD_ASSIGN_S (Star), KEYPAD_ASSIGN_H (Hash)**

KEYPAD_ASSIGN_S handles the '*' key by storing KEY_STAR (0xB0). KEYPAD_ASSIGN_H handles the '#' key by storing KEY_HASH (0xC0). Both call KEYPAD_WAIT_RELEASE for debouncing.

**KEYPAD_ASSIGN_0 through KEYPAD_ASSIGN_D**

These are individual key detection handlers. Each stores its corresponding key code in key_last_pressed, calls KEYPAD_WAIT_RELEASE for debouncing and key release detection, then returns. The key codes are as follows:

Numeric keys (0-9): Store their face values (0-9)

'A' key: Stores KEY_A (0xA0 = 160)

'B' key: Stores 11

'C' key: Stores 12

'D' key: Stores 13

'*' key: Stores KEY_STAR (0xB0 = 176)

'#' key: Stores KEY_HASH (0xC0 = 192)

**KEYPAD_WAIT_RELEASE**

This function waits for key release while maintaining display refresh. It loops 48 times, refreshing all 4 display digits each iteration to prevent flicker during the wait. After the display loop, it sets all PORTB rows HIGH and continuously checks columns RB4-RB7 until all read HIGH (no key pressed), ensuring complete debounce.

**KEYPAD_DEBOUNCE**

This function provides a minimal stabilization delay (8 NOPs in a loop) after changing row outputs, allowing the column lines to settle before reading. This prevents false readings from electrical noise.

**KEYPAD_PROCESS_KEY**

This function routes detected keys based on current input state. It checks for 'A' (KEY_A = 0xA0) to start new input, '#' (KEY_HASH = 0xC0) to confirm, '*' (KEY_STAR = 0xB0) for decimal point. For numeric keys (0-9), it only processes if menu_step > 0 (input in progress), routing to KEYPAD_INPUT_1, KEYPAD_INPUT_2, or KEYPAD_INPUT_3 based on current step.

**KEYPAD_PROC_A**

This function handles 'A' key press. It simply sets menu_flag to 1, which the main loop detects to initialize a new input sequence.

**KEYPAD_RESET_MENU**

This function initializes the input state machine when menu_flag is detected. It sets menu_step to 1 (waiting for first digit), clears input_tens, input_ones, input_decimal, and clears the flag.

**KEYPAD_INPUT_1**

This function saves the first digit (tens place). It stores key_last_pressed in input_tens, clears input_ones and input_decimal, and advances menu_step to 2.

**KEYPAD_INPUT_2****

This function saves the second digit (ones place). It stores key_last_pressed in input_ones and advances menu_step to 3 (waiting for '*' or '#').

**KEYPAD_PROC_STAR**

This function handles '*' key for decimal point entry. It only acts in step 3, advancing menu_step to 4 to accept the decimal digit.

**KEYPAD_INPUT_3**

This function saves the third digit (decimal place). It stores key_last_pressed in input_decimal and advances menu_step to 5 (ready to confirm).

**KEYPAD_PROC_CONFIRM**

This function handles '#' key to confirm input. In step 3, it jumps to KEYPAD_MODE_FULL (no decimal entered). In step 5, it jumps to TEMP_INPUT_LIMIT_CHECK. Other steps are ignored.

**3. Seven-Segment Display Module**

The Seven-Segment Display Module drives a 4-digit multiplexed 7-segment LED display. The digit select lines are connected to RC0-RC3, and segment data to PORTD. The display shows temperature values with one decimal place (format: XX.X) and rotates through target temperature, ambient temperature, and fan speed.

**DISPLAY_GET_SEGMENT**

This function converts a numeric value (0-10) to a 7-segment display pattern. It uses a series of DECF and compare operations to match the input with each digit. Return values are: 0=0x3F, 1=0x06, 2=0x5B, 3=0x4F, 4=0x66, 5=0x6D, 6=0x7D, 7=0x07, 8=0x7F, 9=0x6F, 10=0x40 (dash for input mode), else 0x00 for invalid input.

**DISPLAY_PROCESS_DATA**

This function refreshes all 4 digits of the display using multiplexing technique. It loops 5 times (display_scan_loop) for visibility. For each digit: loads the value (disp_decimal2, disp_decimal1, disp_ones, disp_tens), calls DISPLAY_GET_SEGMENT for the pattern, outputs to PORTD, enables the digit select (RC0-RC3), calls DISPLAY_MUX_DELAY, then disables the digit. The ones digit (digit 2) has 0x80 ORed to enable the decimal point. After display refresh, it handles auto-cycling: every 100 loops, increments display_mode, wrapping from 2 back to 0.

**DISPLAY_MUX_DELAY**

This function provides the multiplexing delay between digit switches. It loads 195 into delay_counter and decrements with NOP until zero, providing approximately 975µs per digit at 4MHz.

**DISPLAY_PREP_TARGET**

This function prepares target temperature for display. It loads temp_target_int and calls DISPLAY_CONVERT_BCD to extract tens and ones digits. It copies temp_target_frac to disp_decimal1 and clears disp_decimal2.

**DISPLAY_PREP_AMBIENT**

This function prepares ambient temperature for display. It loads temp_ambient into W, calls DISPLAY_CONVERT_BCD, and clears both decimal digit positions.

**DISPLAY_PREP_FAN**

This function prepares fan speed for display. It loads fan_status and calls DISPLAY_CONVERT_BCD, clearing both decimal positions.

**DISPLAY_CONVERT_BCD**

This function converts a binary value (0-99) in W to separate tens and ones digits using successive subtraction.

**DISPLAY_SHOW_MENU**

This function displays input mode values. In step 1, it shows dashes (code 10) on all digits indicating ready for input. In other steps, it displays the entered input_tens, input_ones, input_decimal values in the appropriate display positions.

**Display Mode Rotation**

This logic in DISPLAY_PROCESS_DATA rotates the display mode every 100 cycles when not in input mode. It increments mode_timer and when it equals 100, resets it and increments display_mode. Mode 0=target, 1=ambient, 2=fan, wrapping back to 0 at mode 3.

**4. Temperature Sensor (ADC) Module**

The Temperature Sensor Module reads ambient temperature from an analog sensor connected to AN0. The ADC is configured for 10-bit operation in right-justified mode, but only the lower

8 bits (ADRESL) are read, effectively using bits 0-7 of the 10-bit result. The result is then scaled by division by 2, yielding a value in the range 0-127.

**SYSTEM_INIT**

This function initializes all I/O ports and the ADC. It configures ADCON0 with 0x81 (ADC on, channel 0, Fosc/8) and switches to Bank 1 to configure ADCON1 with 0x8E (AN0 as analog, right justified result). It sets TRISA bit 0 for RA0 analog input, TRISB with 0xF0 for RB0-3 outputs (keypad rows) and RB4-7 inputs (keypad columns), TRISC with 0x80 (RC7 input for UART RX), and TRISD to 0x00 (all outputs for display). OPTION_REG bit 7 is cleared to enable PORTB pull-ups. Back in Bank 0, it clears PORTC and PORTD, sets PORTB to 0xFF.

**ADC_READ_AMBIENT**

This function reads the temperature sensor value from ADC channel AN0. It sets ADCON0 bit 2 (GO/DONE) to start conversion, then loops until GO/DONE clears indicating conversion complete. It switches to Bank 1 to read only ADRESL (lower 8 bits of the 10-bit result), ignoring ADRESH (upper 2 bits). This effectively reads bits 0-7 of the ADC result and stores it in temp_ambient. It then switches back to Bank 0 and divides the value by 2 using RRF instruction, storing the result back in temp_ambient. The final value ranges from 0-127, with the actual temperature mapping dependent on the sensor characteristics.

**5. Temperature Control (Heater/Cooler) Module**

The Temperature Control Module maintains the ambient temperature at the user-specified target by controlling a heater and cooler. The heater is connected to RC5 and the cooler/fan to RC4. The control logic uses hysteresis-based on/off control to prevent rapid switching when temperature is near the target.

**HYSTERESIS_IDLE_CHECK**

This function handles the idle state when HVAC is not active. It compares temp_ambient with temp_target_total. If ambient is less than target, it jumps to TEMP_HEATER_ON to start heating. If ambient is greater than target, it jumps to TEMP_COOLER_ON to start cooling. If equal, it jumps to TEMP_ALL_OFF to keep both off.

**HYSTERESIS_HEATING_CHECK**

This function manages the heating state using a hysteresis mechanism. It computes a hysteresis temperature (hyst_temp) as one degree below the target temperature (hyst_temp = temp_target_total - 1). The comparison is performed by subtracting hyst_temp from temp_ambient and evaluating the carry flag (STATUS register, bit 0). If the carry flag is set, indicating that the ambient temperature has reached or exceeded the threshold (temp_ambient

≥ target - 1), execution branches to TEMP_STOP_HEATING to deactivate the heater and prevent temperature overshoot. Otherwise, the system continues heating via TEMP_HEATER_CONTINUE. This 1-degree hysteresis effectively prevents rapid on/off switching of the heater.

**HYSTERESIS_COOLING_CHECK**

This function handles the cooling state. It compares ambient with target temperature. If ambient is less than or equal to target, cooling stops. Otherwise, cooling continues until target is reached.

**TEMP_HEATER_ON**

This function activates the heater to warm the room. It sets hvac_state to 1 to indicate heating mode, then falls through to TEMP_HEATER_CONTINUE.

**TEMP_HEATER_CONTINUE**

This function maintains heating operation. It sets PORTC bit 5 (heater ON) and clears PORTC bit 4 (cooler OFF), then continues to display mode selection.

**TEMP_COOLER_ON**

This function activates the cooler to reduce temperature. It sets hvac_state to 2 to indicate cooling mode, then falls through to TEMP_COOLER_CONTINUE.

**TEMP_COOLER_CONTINUE**

This function maintains cooling operation. It clears PORTC bit 5 (heater OFF) and sets PORTC bit 4 (cooler ON). It also decrements temp_ambient by 1 to simulate cooling effect.

**TEMP_STOP_HEATING, TEMP_STOP_COOLING**

These labels share the same code path with TEMP_ALL_OFF. When heating needs to stop (ambient reached target-1) or cooling needs to stop (ambient reached target), control flows here.

**TEMP_ALL_OFF**

This function turns off both heating and cooling when target is reached or when transitioning from active HVAC states. It clears hvac_state to 0 and clears both PORTC bits 4 and 5, disabling both heater and cooler outputs. This label is shared with TEMP_STOP_HEATING and TEMP_STOP_COOLING.

**FAN_STATUS_CHECK**

This function determines fan speed based on cooler state. It checks PORTC bit 4: if set (cooler ON), fan_status is set to 5 indicating fan running; otherwise, fan_status is cleared to 0 indicating fan off.

## 6. Input Validation Module

The Input Validation Module ensures all target temperature inputs (from keypad or UART) are within the valid range of 10.0°C to 50.0°C. This prevents dangerous or unrealistic temperature settings and provides a consistent user experience.

**TEMP_INPUT_LIMIT_CHECK**

This function validates keypad input before applying. It checks: if input_tens equals 0, input is rejected (less than 10). If input_tens is greater than or equal to 6, input is rejected (60 or higher). If input_tens equals 5 and either input_ones or input_decimal is greater than 0, input is rejected (exceeds 50.0). Valid inputs proceed to TEMP_CALC_TARGET.

**TEMP_CALC_TARGET**

This function converts validated digits to target values. It calculates temp_target_int = input_tens * 10 + input_ones using shift-and-add multiplication (RLF for x2, then x4, add original for x5, RLF again for x10). It copies input_decimal to temp_target_frac and calls TEMP_UPDATE_LOGIC to apply the new target.

**TEMP_INPUT_REJECT**

This function handles invalid input by simply clearing menu_step to return to normal display mode without updating any target values.

**UART Validation in UART_CMD_SET_FRAC**

This section validates UART SET commands. It checks uart_temp_int against 10 (minimum) and 51 (maximum). If uart_temp_int equals 50 and uart_temp_frac is greater than 0, the value is rejected. Valid values proceed to UART_COMMIT.

**TEMP_UPDATE_LOGIC**

This function updates temp_target_total, which is used by the control logic, based on temp_target_int and temp_target_frac. Initially, temp_target_int is copied directly into temp_target_total. To perform rounding, the function first checks whether temp_target_int equals 50 using the XORLW instruction. If this condition is met, the function returns immediately without applying rounding, thereby preventing the target temperature from exceeding the defined maximum limit.

If the integer part is below the maximum, the function evaluates the fractional component by executing SUBLW 4 on temp_target_frac and examining the carry flag. When the fractional value is 5 or greater (indicated by a cleared carry flag when 4 − frac < 0), temp_target_total is incremented to achieve rounding up. For instance, a target temperature of 25.5 °C is rounded to 26 °C for control purposes.

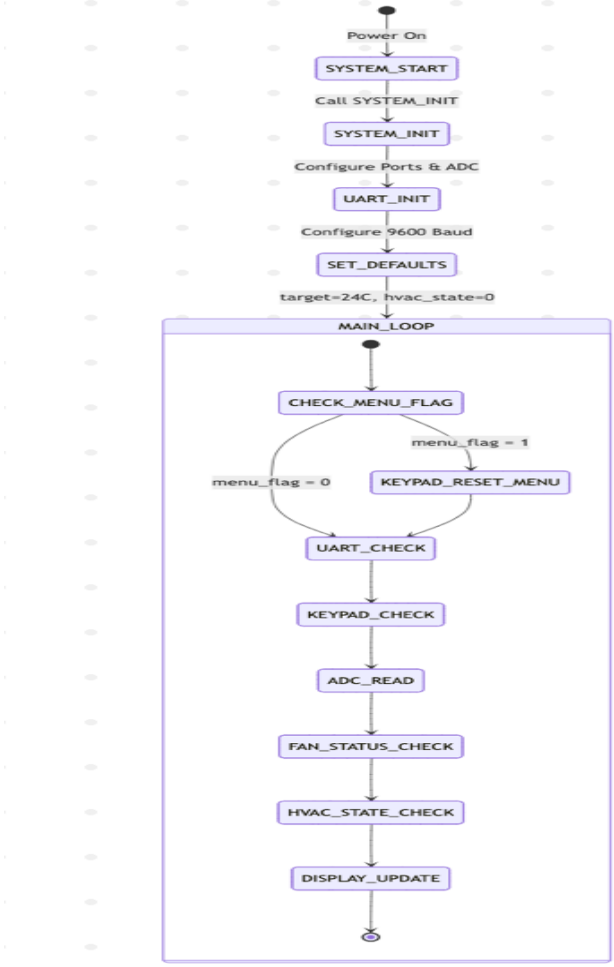
**UART_ABORT**

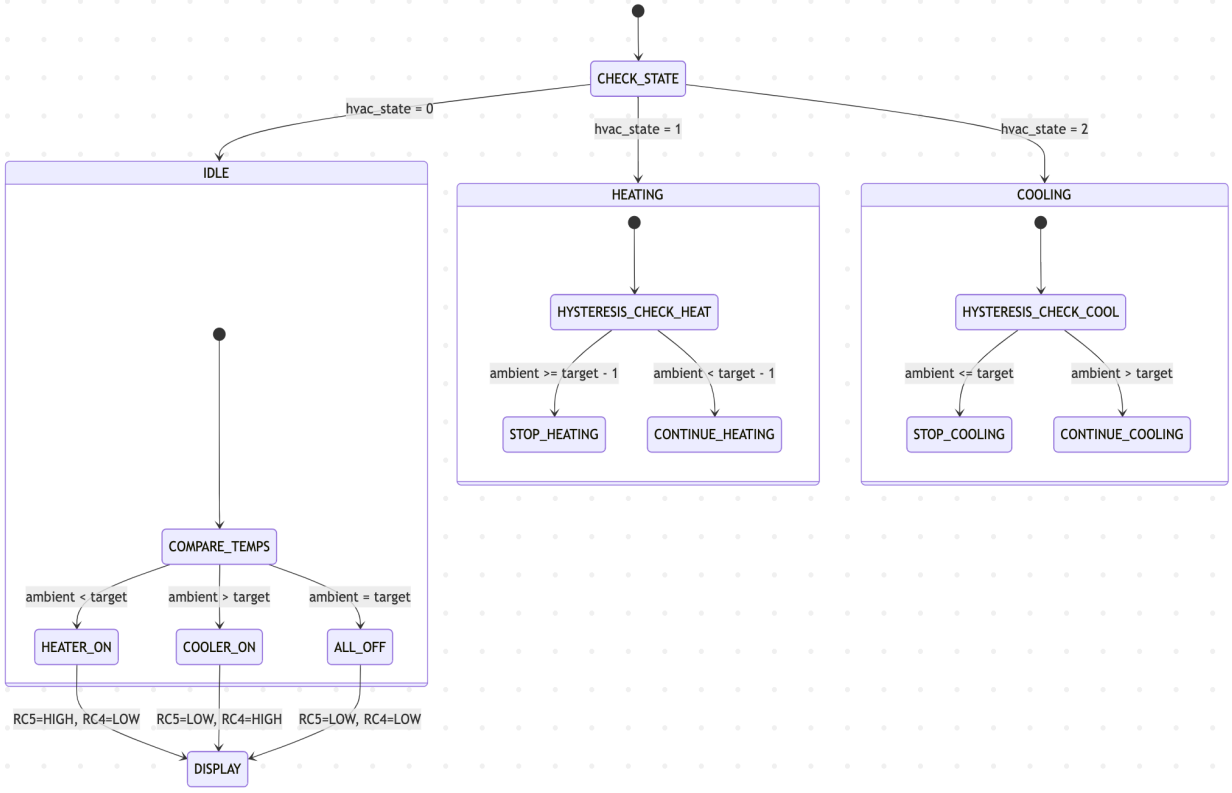
This function handles rejected UART input by clearing update_flag bit 0 and returning without modifying target values.

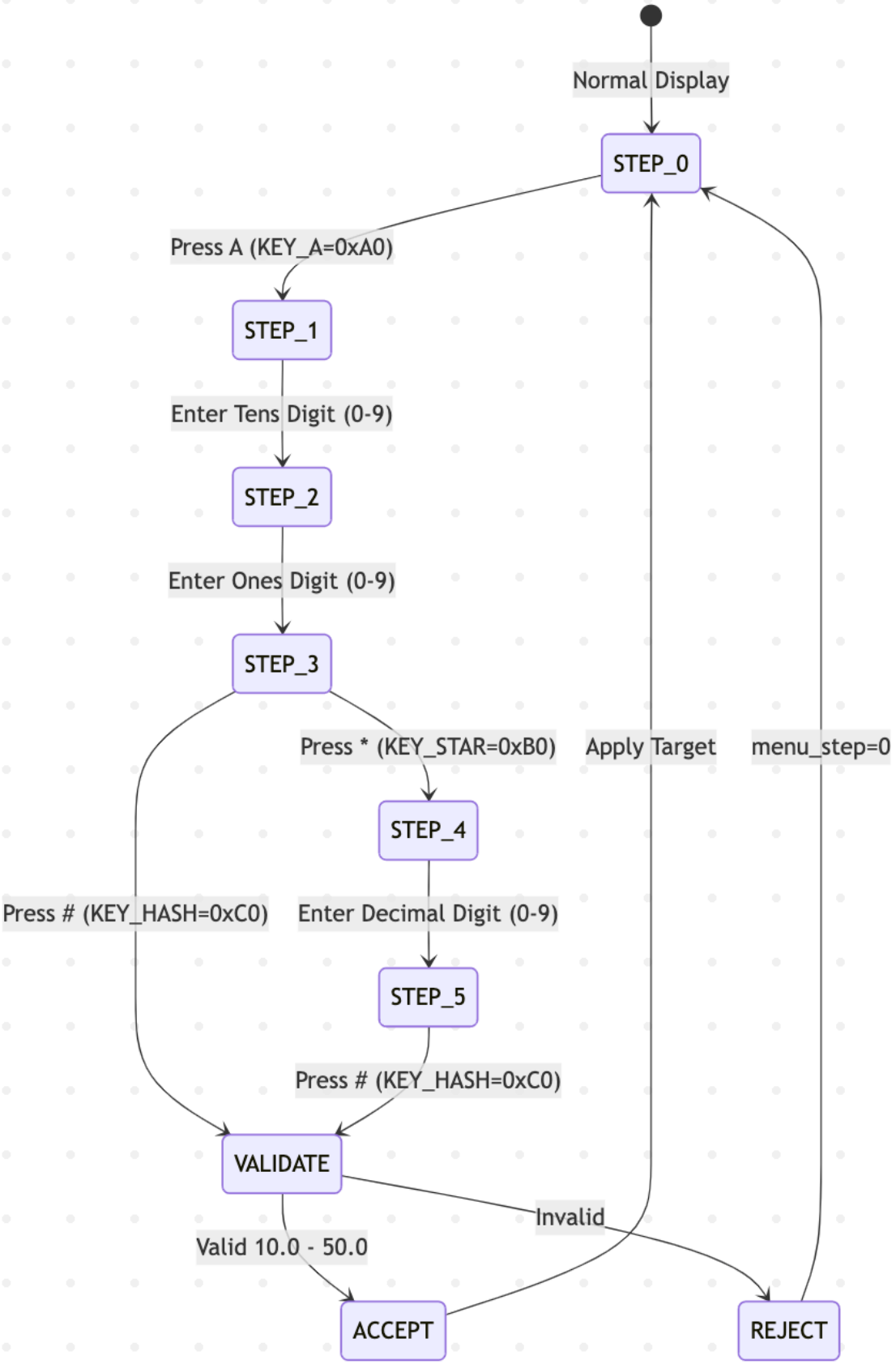
PIN ASSIGNMENTS:

| Pin | Port | Function | Direction | Description |
|---|---|---|---|---|
| 2 | RA0 | Temperature Sensor | Input | ADC Channel 0 (AN0) |
| 33 | RB0 | Keypad Row 1 | Output | Active LOW scanning |
| 34 | RB1 | Keypad Row 2 | Output | Active LOW scanning |
| 35 | RB2 | Keypad Row 3 | Output | Active LOW scanning |
| 36 | RB3 | Keypad Row 4 | Output | Active LOW scanning |
| 37 | RB4 | Keypad Column 1 | Input | Internal pull-up enabled |
| 38 | RB5 | Keypad Column 2 | Input | Internal pull-up enabled |
| 39 | RB6 | Keypad Column 3 | Input | Internal pull-up enabled |
| 40 | RB7 | Keypad Column 4 | Input | Internal pull-up enabled |
| 15 | RC0 | 7-Segment Digit 1 | Output | Tens digit select |
| 16 | RC1 | 7-Segment Digit 2 | Output | Ones digit (with DP) |
| 17 | RC2 | 7-Segment Digit 3 | Output | Decimal digit 1 |
| 18 | RC3 | 7-Segment Digit 4 | Output | Decimal digit 2 |
| 23 | RC4 | Cooler/Fan Control | Output | HIGH = Cooler Active |
| 24 | RC5 | Heater Control | Output | HIGH = Heater Active |
| 25 | RC6 | UART TX | Output | Serial transmit 9600 |
| 26 | RC7 | UART RX | Input | Serial receive 9600 |
| 19-22 | RD0-3 | Segment a-d | Output | Lower nibble pattern |
| 27-30 | RD4-7 | Segment e-g, DP | Output | Upper nibble pattern |

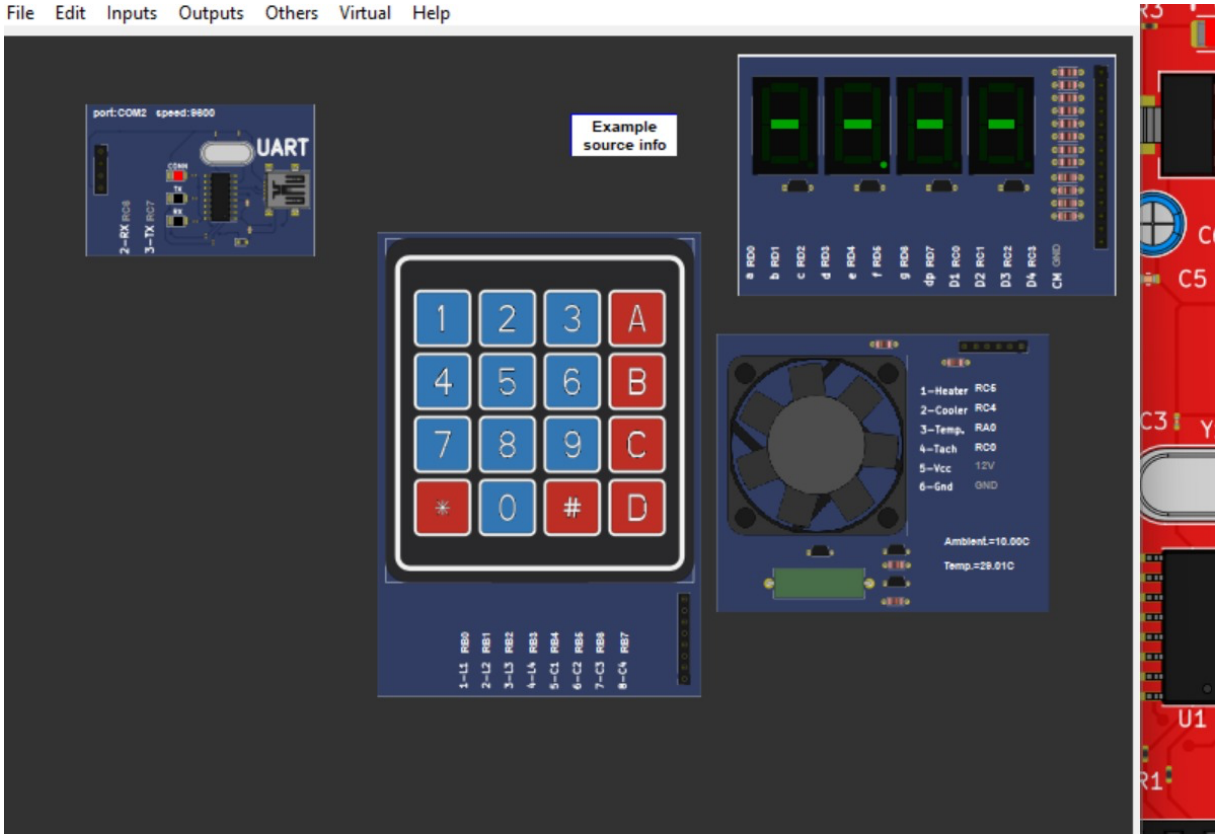
*Pin Assignments for Board #1*


Power On
SYSTEM_START
Call SYSTEM_INIT
SYSTEM_INIT
Configure Ports & ADC
UART_INIT
Configure 9600 Baud
SET_DEFAULTS
target=24C, hvac_state=0
MAIN_LOOP
CHECK_MENU_FLAG
menu_flag = 1
menu_flag = 0
KEYPAD_RESET_MENU
UART_CHECK
KEYPAD_CHECK
ADC_READ
FAN_STATUS_CHECK
HVAC_STATE_CHECK
DISPLAY_UPDATE

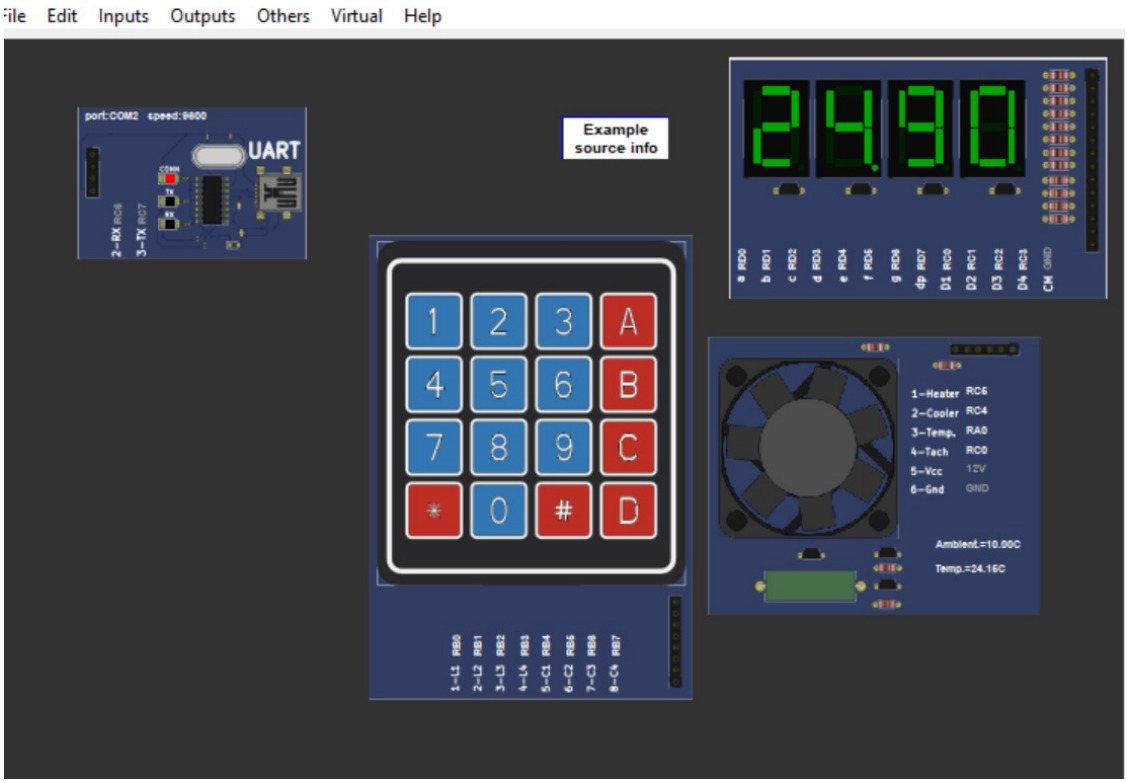

CHECK_STATE
hvac_state = 0
hvac_state = 1
hvac_state = 2
IDLE
HEATING
COOLING
HYSTERESIS_CHECK_HEAT
ambient >= target - 1
ambient < target - 1
STOP_HEATING
CONTINUE_HEATING
HYSTERESIS_CHECK_COOL
ambient <= target
ambient > target
STOP_COOLING
CONTINUE_COOLING
COMPARE_TEMPS
ambient < target
ambient > target
ambient = target
HEATER_ON
COOLER_ON
ALL_OFF
RC5=HIGH, RC4=LOW
RC5=LOW, RC4=HIGH
RC5=LOW, RC4=LOW
DISPLAY


Normal Display
STEP_0
Press A (KEY_A=0xA0)
STEP_1
Enter Tens Digit (0-9)
STEP_2
Enter Ones Digit (0-9)
STEP_3
Press * (KEY_STAR=0xB0)
Apply Target
menu_step=0
STEP_4
Press # (KEY_HASH=0xC0)
Enter Decimal Digit (0-9)
STEP_5
Press # (KEY_HASH=0xC0)
VALIDATE
Invalid
Valid 10.0 - 50.0
ACCEPT
REJECT

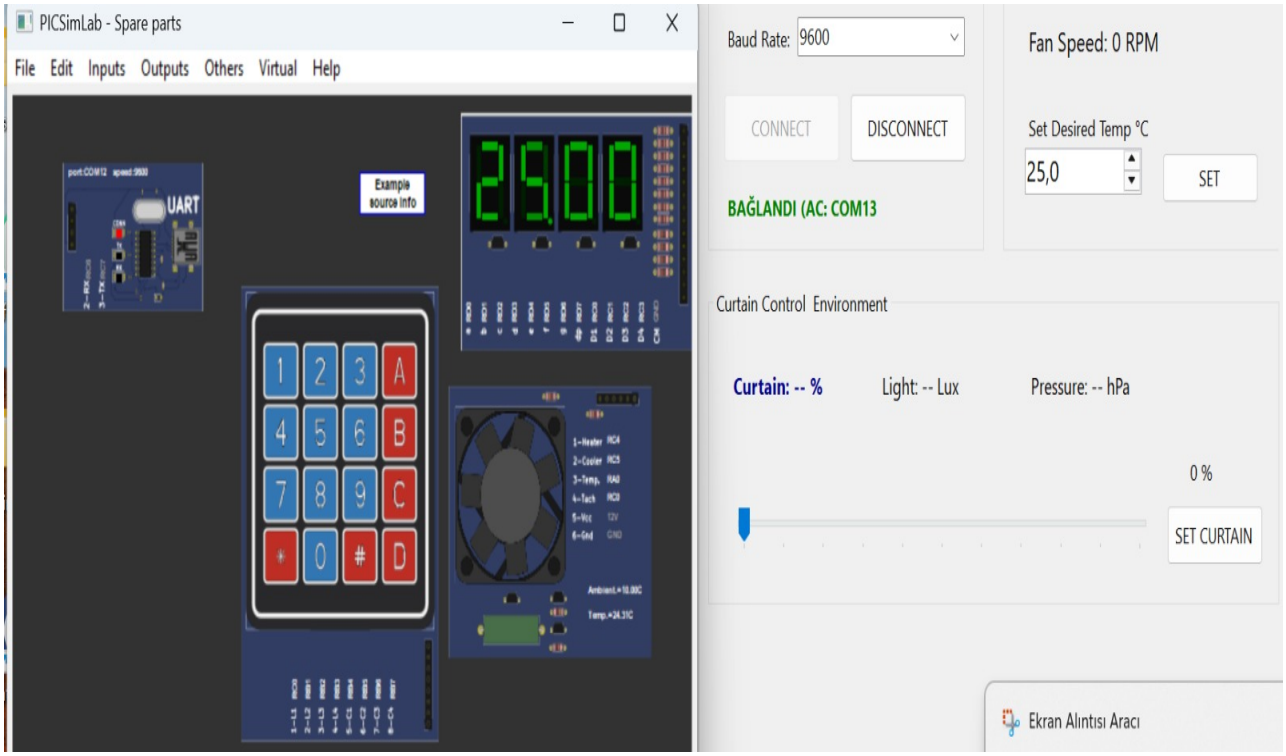
When 'A' is pressed on the keypad;



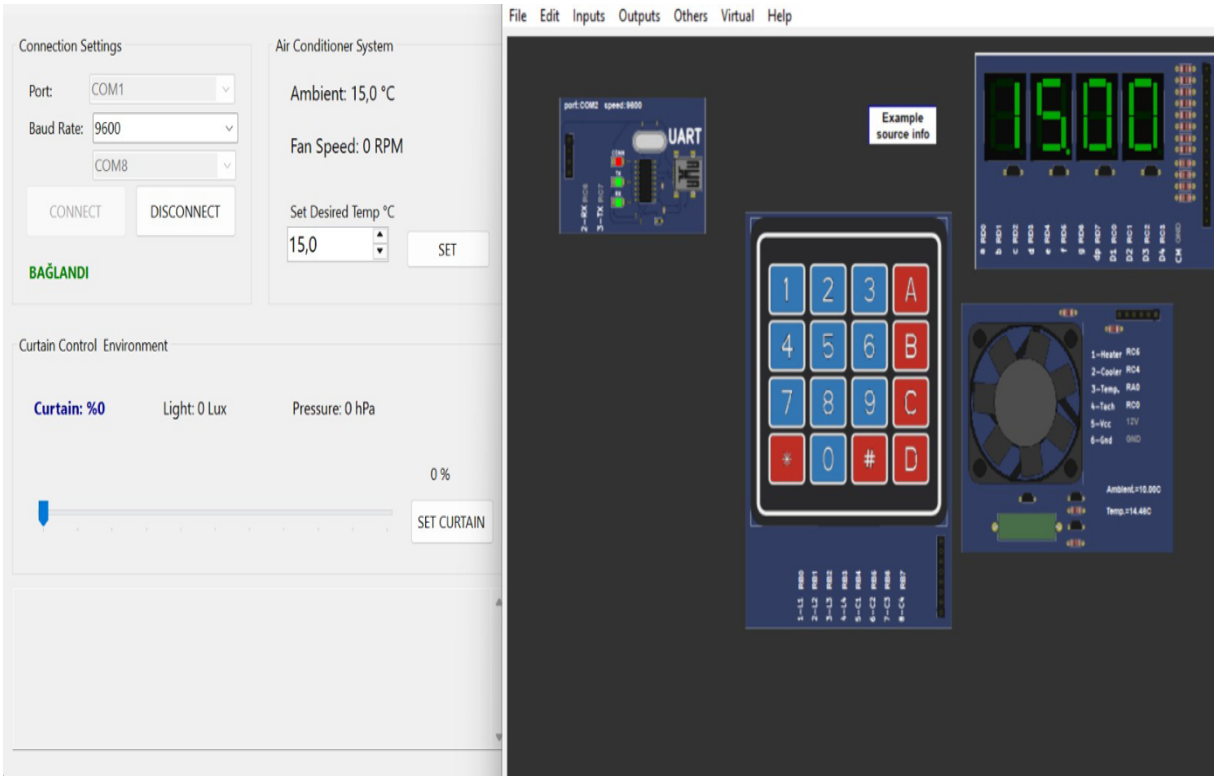

24.90 degrees;



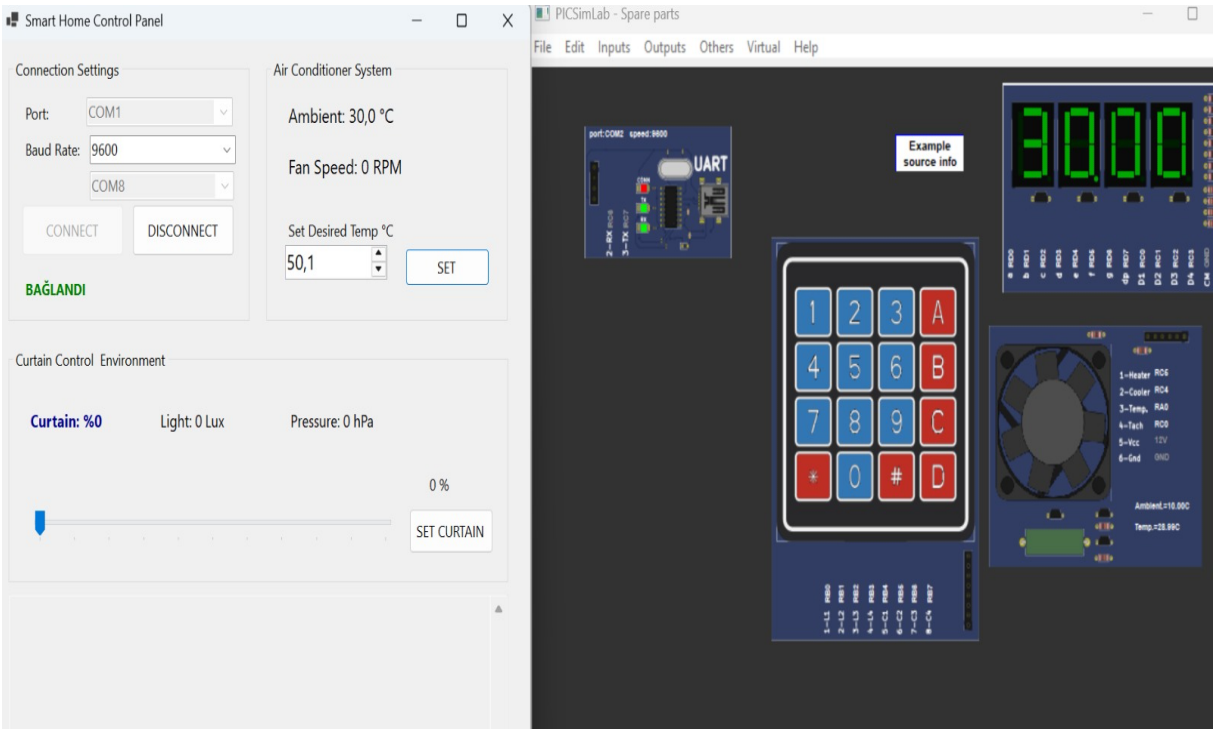
25 degrees entered from the user panel;



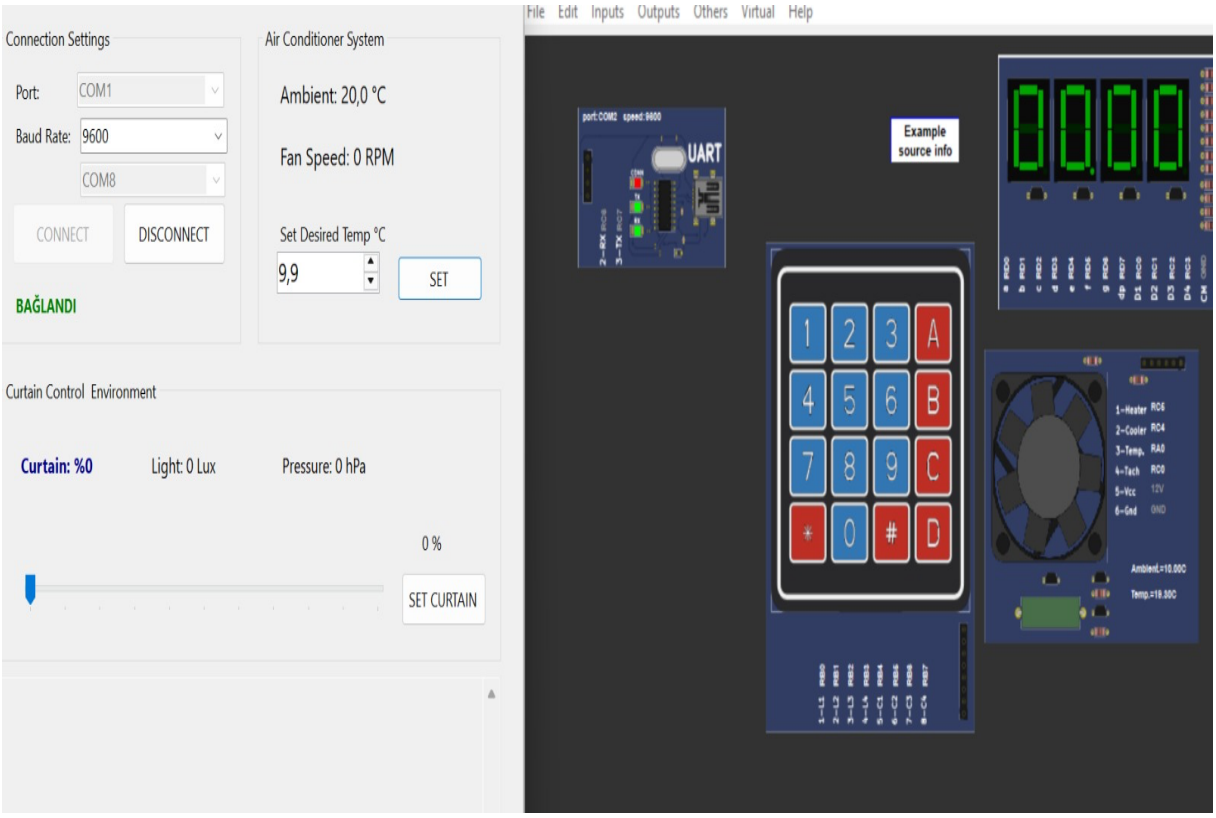

15 degrees entered from the user panel;



When an attempt is made to set the temperature above 50, it remains at the current temperature.




If an attempt is made to set the temperature below 10, the system rejects it and remains at the current temperature.



## 2.2 Curtain Control System Design

The primary task of Board #2 is controlling the opening and closing of curtains in a domestic setting in order to enhance comfort. The Curtain Control System is comprised of a stepper motor controller module, an LDR light sensor module, a BMP180 temperature, and pressure sensors module, a rotary potentiometer module, an LCD display module, as well as a UART communication module. The system enables the user to control it either automatically or through user-oriented manual controls. Board #2 is designed employing the PIC16F877A microcontroller and simulated within the PICSimLab simulator. It can also connect with a computer application via the UART communication process, enabling distant controlling capabilities. By forming this configuration, the curtain controlling system is implemented as an intelligent, environment-interacting component within the home automation system.

### 2.2.1 Modules and Functions

**Step Motor Control Module**

It is a control unit that operates the step motor that opens and closes the curtain. It compares the current position and the desired position of the curtain and determines the motor rotation direction and the number of steps needed, thus enabling the curtain to move towards the desired position.

**STEP_TABLE**

This is a lookup table that returns the coil energization pattern for each of the four motor phases. It uses a computed GOTO technique where the phase index in W is added to PCL to jump to the correct RETLW instruction. Phase 0 returns 0b00000001 (Coil A), Phase 1 returns 0b00000010 (Coil B), Phase 2 returns 0b00000100 (Coil C), and Phase 3 returns 0b00001000 (Coil D). The table is placed at address 0x0100 to ensure proper PCLATH handling.

**MOTOR_CONTROL**

This is the main motor control decision logic. It loads CURTAIN_TARGET into W and subtracts CURTAIN_CURRENT to determine if movement is needed. If the Zero flag is set (target equals current), no movement is required and execution jumps to UART_PROCESS. Otherwise, it checks the Carry flag: if clear, the target is greater than current (close curtain, go to MOTOR_FORWARD), if set, the target is less than current (open curtain, go to MOTOR_REVERSE).

**MOTOR_FORWARD**

This function moves the motor in the forward direction to close the curtain. It increments STEP_COUNTER and checks if 10 steps (STEPS_PER_PERCENT) have been completed. If so, it clears STEP_COUNTER, increments both CURTAIN_CURRENT and CURTAIN_POS_INT to track the new position, then proceeds to execute the step. This ensures position tracking is updated every 10 motor steps, corresponding to 1% movement.

**MOTOR_REVERSE**

This function moves the motor in the reverse direction to open the curtain. Similar to MOTOR_FORWARD, it increments STEP_COUNTER and checks for 10-step completion. When reached, it clears STEP_COUNTER and decrements both CURTAIN_CURRENT and CURTAIN_POS_INT. This provides the counterclockwise rotation needed to open the curtain.

**EXECUTE_STEP_FWD**

This function advances the motor phase in the forward direction. It increments MOTOR_PHASE and applies a mask of 0x03 to keep the value within 0-3 range, implementing the circular sequence 0→1→2→3→0. Then it falls through to APPLY_STEP.

**EXECUTE_STEP_REV**

This function advances the motor phase in the reverse direction. It decrements MOTOR_PHASE and applies a mask of 0x03. Due to the masking, decrementing from 0 wraps to 3, implementing the reverse sequence 0→3→2→1→0.

**APPLY_STEP**

This function outputs the current phase pattern to the motor. It sets PCLATH to the high byte of STEP_TABLE address, loads MOTOR_PHASE into W, and calls STEP_TABLE to get the coil pattern. The returned pattern is written to PORTB to energize the appropriate motor coil. PCLATH is cleared after the call and DELAY_MOTOR is called to allow the motor to settle before the next step.

**DELAY_MOTOR**

This function provides the timing delay between motor steps, controlling motor speed. It loads 8 into DELAY_CNT_Y and repeatedly calls DELAY_LONG (8 times), producing approximately 4ms total delay. This determines the stepper motor rotation speed - slower delay means slower motor movement.

**LDR Light Sensor Module**

This module calculates the light strength in the room through the use of the LDR sensor. This light strength is measured in the form of an analog signal. If the light strength is below a set limit, then the curtains are automatically closed.

**ADC_READ_LDR**

This function reads the light sensor value from ADC channel AN0. It configures ADCON0 by clearing CHS2, CHS1, and CHS0 bits to select channel 0, then sets ADON to enable the ADC. After calling DELAY_20US for acquisition time, it sets the GO/DONE bit to start conversion. The function waits in a loop until GO/DONE clears (conversion complete), then reads ADRESH. Importantly, it subtracts the result from 255 (SUBLW 255) to invert the reading so that higher values represent brighter light and lower values represent darkness.

**ADC_TO_PERCENT**

This function converts an 8-bit ADC value (0-255) to a percentage (0-100). Direct division is complex on PIC, so it uses an approximation algorithm. The function uses alternating

subtraction of 3 and 2 in a pattern (3,2,3,2,3) which averages 2.6 per count, close to the ideal 2.55 (255/100). For each successful subtraction, it increments MATH_TEMP (the percentage result). Special cases handle zero input (return 0) and maximum input >=250 (return 100). The result is capped at 100 to prevent overflow.

**Night Mode Logic**
Located in MAIN_LOOP, this logic implements the automatic night mode. It loads LIGHT_THRESHOLD (40) into W and subtracts from LIGHT_INT. By checking the Carry flag, it determines if light is below threshold. If Carry is clear (light < 40%), it sets CURTAIN_TARGET to 100, meaning fully closed. If Carry is set (light >= 40%), it proceeds to use the potentiometer value as the target instead.

**BMP180 Temperature and Pressure Sensor Module**
In this module, the module measures the temperature and air pressure outside using the BMP180 sensor. The communication method that the sensor uses to connect to the microcontroller is the I2C protocol. These measurements will be stored and displayed on the LCD display module. This module exists as a comment line in the hex code, but it wasn't used on the board because the BMP180 doesn't work in PICSimlab.

**BMP180_READ**
This function would read temperature and pressure from the BMP180 sensor. It sends the I2C start condition, writes the BMP180 address (0xEE), writes the control register address (0xF4), writes the temperature read command (0x2E), then sends stop. After waiting 10ms for conversion, it reads the result by sending start, address, register 0xF6, restart, read address (0xEF), then reads the data byte into TEMP_INT.

**I2C_START**
This function generates an I2C Start condition by setting the SEN bit in SSPCON2. It waits in a loop until SEN clears, indicating the Start condition has been transmitted on the bus.

**I2C_STOP**
This function generates an I2C Stop condition by setting the PEN bit in SSPCON2. It waits until PEN clears, indicating the Stop condition has been completed and the bus is released.

**I2C_RESTART**
This function generates a Repeated Start condition for switching from write to read mode without releasing the bus. It sets the RSEN bit in SSPCON2 and waits for completion.

**I2C_WRITE**
This function transmits a byte on the I2C bus. It loads the byte into SSPBUF, then waits for the BF (Buffer Full) flag in SSPSTAT to clear, indicating transmission complete. It checks ACKSTAT in SSPCON2 to verify the slave acknowledged the byte.

**I2C_READ**
This function receives a byte from the I2C bus. It sets RCEN in SSPCON2 to enable receive mode, waits until RCEN clears (receive complete), then reads the received byte from SSPBUF into I2C_BUFFER.

**I2C_ACK**

This function sends an Acknowledge bit to the slave to indicate more bytes are expected. It clears ACKDT (ACK data = 0), then sets ACKEN to initiate the acknowledge sequence, waiting for completion.

**I2C_NACK**

This function sends a Not Acknowledge bit to indicate the last byte has been received. It sets ACKDT (NACK data = 1), then sets ACKEN to transmit the NACK, waiting for completion.

**I2C_ERROR**

This function handles I2C communication errors. It calls I2C_STOP to release the bus and returns. Additional error handling or retry logic could be implemented here.

**Rotary Potentiometer Module**

The module makes it possible for the user to determine the curtain's position manually. The value extracted from the rotary potentiometer is converted into a percentage, which serves as the target position of the curtain.

**ADC_READ_POT**

This function reads the potentiometer value from ADC channel AN1. It configures ADCON0 by clearing CHS2 and CHS1, then setting CHS0 to select channel 1. ADON is set to enable the ADC. After the acquisition delay (DELAY_20US), it sets GO/DONE to start conversion and waits for completion. The function returns the raw ADRESH value (0-255) without inversion, unlike the LDR reading.

**DELAY_20US**

This function provides approximately 20 microseconds of delay required for ADC acquisition time. It loads 10 into DELAY_CNT_X and decrements in a loop. At 20MHz with 4 cycles per instruction, this produces approximately 10 * 4 * 200ns = 8µs per iteration, totaling about 20µs with overhead.

**Target Setting Logic**

In the MAIN_LOOP during AUTO mode, after checking that SYSTEM_MODE bit 0 is clear (not in manual mode), the code calls ADC_READ_POT to get the potentiometer value. It then calls ADC_TO_PERCENT to convert to 0-100%, storing the result in CURTAIN_TARGET_INT and clearing CURTAIN_TARGET_FRAC.

**SKIP_POT_TARGET**

This label marks the jump target for skipping potentiometer target setting when in Manual mode. When SYSTEM_MODE bit 0 is set (UART control active), the BTFSC instruction jumps here, bypassing the potentiometer reading and preserving the UART-set target.

**LCD Display Module**

The module is designed to display system details for the user. The exterior temperature, pressure, light intensity, and status of the curtains are displayed on the user's LCD display. These details are easily viewable by the user.

**LCD_INIT**

This function initializes the LCD in 4-bit mode following the HD44780 initialization sequence. It first calls DELAY_LONG to wait more than 40ms after power-up. Then it sends 0x03 three times (function set attempts) with delays between each, followed by 0x02 to switch to 4-bit mode. After establishing 4-bit mode, it sends 0x28 (4-bit, 2 lines, 5x7 font), 0x0C (display on, cursor off), 0x01 (clear display), and 0x06 (entry mode: increment, no shift).

**LCD_SEND_CMD**
This function sends a command byte to the LCD. It stores the byte in LCD_TEMP and clears RE0 (RS=0 for command mode). Using SWAPF, it extracts the high nibble first, masks with 0x0F, and sends via LCD_NIBBLE. Then it sends the low nibble directly. A short delay follows to allow the LCD to process the command.

**LCD_SEND_DATA**
This function sends a data byte (character) to the LCD for display. It stores the byte in LCD_TEMP and sets RE0 (RS=1 for data mode). Like LCD_SEND_CMD, it sends the high nibble first, then the low nibble, each through LCD_NIBBLE. A short delay follows.

**LCD_NIBBLE**
This function sends a 4-bit nibble to the LCD data pins. It stores the nibble in MATH_TEMP, preserves the upper nibble of PORTD by ANDing with 0xF0, then ORs the data into the lower nibble. It generates an enable pulse by setting RE1 high, executing two NOPs for timing, then clearing RE1. This latches the data into the LCD.

**LCD_UPDATE**
This function refreshes all dynamic values on the LCD display. It only executes when LCD_TIMER reaches LCD_UPDATE_RATE, implementing a periodic refresh. For Line 1, it positions at 0x80 and writes: sign character (+), temperature integer (2 digits via DISPLAY_2DIGIT), decimal point, temperature fractional, degree symbol (0xDF), 'C', spaces, '1', '0', pressure integer (2 digits), 'h', 'P', 'a'. For Line 2, it positions at 0xC0 and writes light level (3 digits via DISPLAY_3DIGIT), decimal, fractional, then at 0xC8 writes curtain position similarly.

**DISPLAY_2DIGIT**
This function converts a number from 0-99 into two ASCII digit characters and sends them to the LCD. It stores the input in NUM_TEMP and clears DIGIT_10. It repeatedly subtracts 10 from NUM_TEMP, incrementing DIGIT_10 each time until the result would be negative. This extracts the tens digit by counting how many times 10 fits. It then displays the tens digit by adding '0' to DIGIT_10 and calling LCD_SEND_DATA, followed by the units digit (remaining NUM_TEMP + '0').

**DISPLAY_3DIGIT**
This function converts a number from 0-999 into three ASCII digit characters. Similar to DISPLAY_2DIGIT, it uses repeated subtraction but first extracts hundreds by subtracting 100, then tens by subtracting 10. It maintains DIGIT_100 and DIGIT_10 as counters, with NUM_TEMP holding the final units. Each digit is converted to ASCII by adding '0' and sent to the LCD in order: hundreds, tens, units.

**UART Communication Module**
It is responsible for communicating with the PC application via the serial communication connection between Board #2 and the PC application. It will send system data to the PC application and receive control commands from the user.

**ISR_HANDLER**

This is the Interrupt Service Routine that handles UART receive interrupts. When a byte arrives via UART, this routine is automatically called. It first saves the context by storing the W register in W_SAVE and the STATUS register in STATUS_SAVE using the SWAPF instruction to avoid affecting status flags. Then it switches to Bank 0 for peripheral register access. The routine checks for overrun errors (OERR) in RCSTA register and clears them by toggling the CREN bit. If no error exists and data has arrived (RCIF flag set), it reads the received byte from RCREG into UART_RX_DATA and sets the UART_CMD_READY flag for the main loop to process. Finally, it restores the context using double SWAPF technique and returns with RETFIE.

**UART_CMD_HANDLER**
This function parses the received command byte and routes it to the appropriate handler. It first checks if the command is a SET command by examining the top 2 bits using the COMMAND_MASK (0xC0). If the top 2 bits are "11" (0xC0-0xFF), it jumps to SET_CURTAIN_INTEGER. If the top 2 bits are "10" (0x80-0xBF), it jumps to SET_CURTAIN_FRACTIONAL. For GET commands, it compares the received byte against each command code (0x01 through 0x08) using XOR operations, jumping to the corresponding SEND function when a match is found. If command 0x09 is received, it switches to AUTO mode.

**UART_TRANSMIT**
This function sends a single byte over the UART interface. It saves the byte to transmit in MATH_TEMP, then switches to Bank 1 to check the TRMT bit in TXSTA. The TRMT bit indicates whether the transmit shift register is empty. The function waits in a loop until TRMT is set, then switches back to Bank 0, loads the byte from MATH_TEMP, and writes it to TXREG to initiate transmission.

**SET_CURTAIN_INTEGER**
This function handles the SET Curtain Integer command (11xxxxxx format). It extracts the lower 6 bits of the received command using AND with 0x3F, giving a value from 0 to 63. It then scales this value to 0-100% by multiplying by 2 using the RLF (rotate left through carry) instruction. If the result exceeds 100, it caps the value at 100. The scaled value is stored in both CURTAIN_TARGET_INT and CURTAIN_TARGET. Finally, it sets bit 0 of SYSTEM_MODE to switch to Manual mode, preventing the potentiometer from overriding the UART-set target.

**SET_CURTAIN_FRACTIONAL**
This function handles the SET Curtain Fractional command (10xxxxxx format). It extracts the lower 6 bits using the VALUE_MASK (0x3F) and stores the result directly in CURTAIN_TARGET_FRAC for the fractional part of the curtain position.

**SET_AUTO_MODE**
This function handles command 0x09 which returns the system to automatic mode. It clears bit 0 of SYSTEM_MODE, allowing the potentiometer to control the curtain target position again instead of UART commands.

**SEND_CURTAIN_FRAC**
This function responds to the GET Curtain Fractional command (0x01). It loads the current curtain fractional position from CURTAIN_POS_FRAC into W and calls UART_TRANSMIT to send it to the PC.

**SEND_CURTAIN_INT**
This function responds to the GET Curtain Integer command (0x02). It loads the current curtain integer position from CURTAIN_POS_INT into W and calls UART_TRANSMIT to send it to the PC.

**SEND_TEMP_FRAC**
This function responds to the GET Temperature Fractional command (0x03). It loads the temperature fractional part from TEMP_FRAC and transmits it via UART.

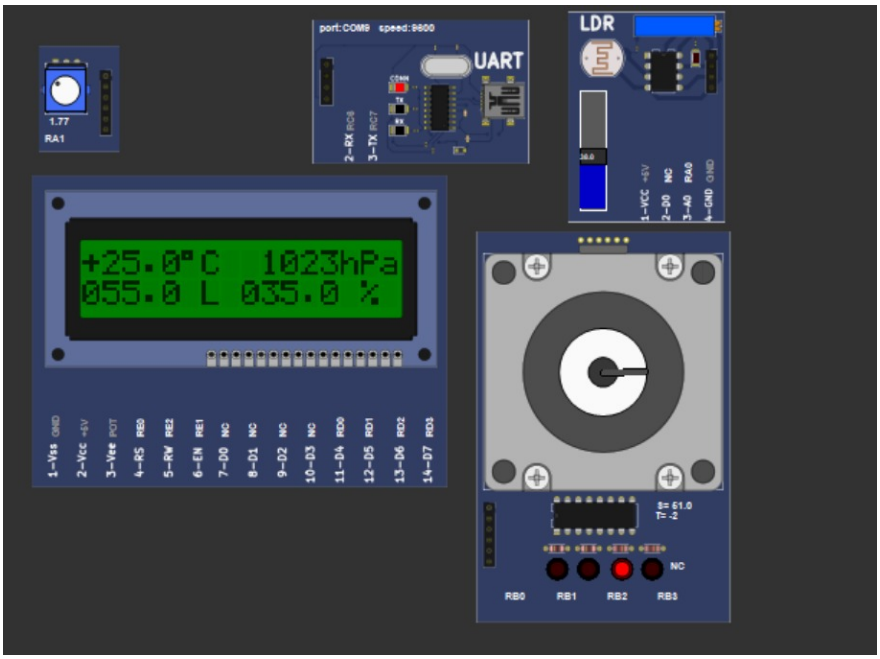
**SEND_TEMP_INT**
This function responds to the GET Temperature Integer command (0x04). It loads the temperature integer part from TEMP_INT and transmits it via UART.

**SEND_PRESS_FRAC**
This function responds to the GET Pressure Fractional command (0x05). It loads the pressure fractional part from PRESS_FRAC and transmits it via UART.

**SEND_PRESS_INT**
This function responds to the GET Pressure Integer command (0x06). It loads the pressure integer part from PRESS_INT and transmits it via UART.

**SEND_LIGHT_FRAC**
This function responds to the GET Light Fractional command (0x07). It loads the light level fractional part from LIGHT_FRAC and transmits it via UART.

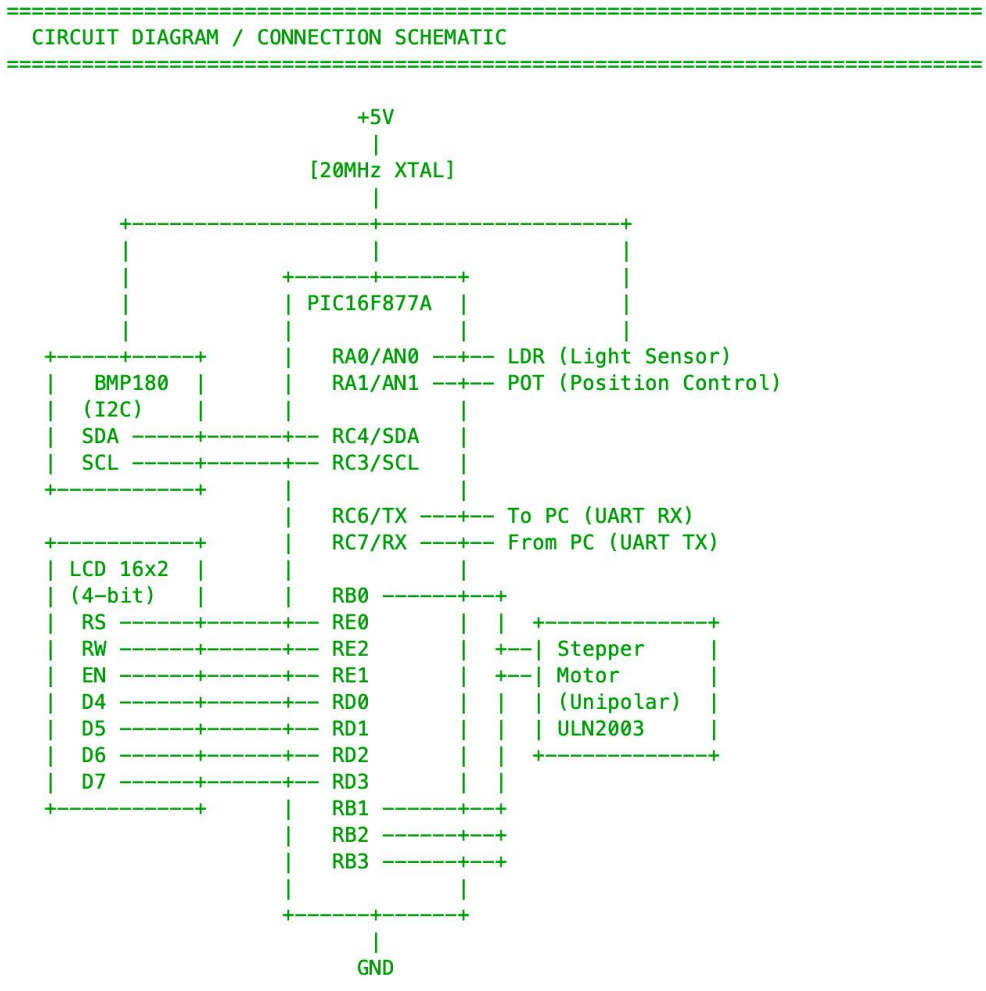
**SEND_LIGHT_INT**
This function responds to the GET Light Integer command (0x08). It loads the light level integer part from LIGHT_INT and transmits it via UART.



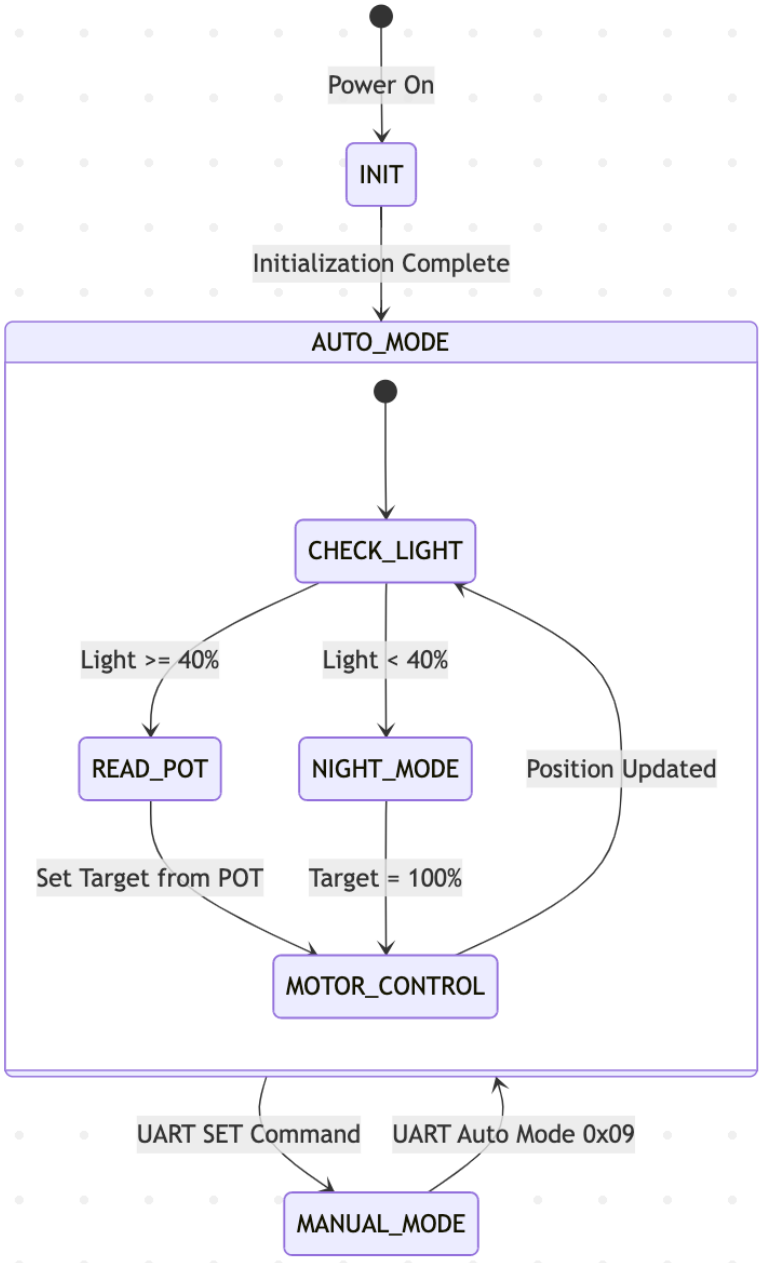

*Board #2 (without BMP180 module)*

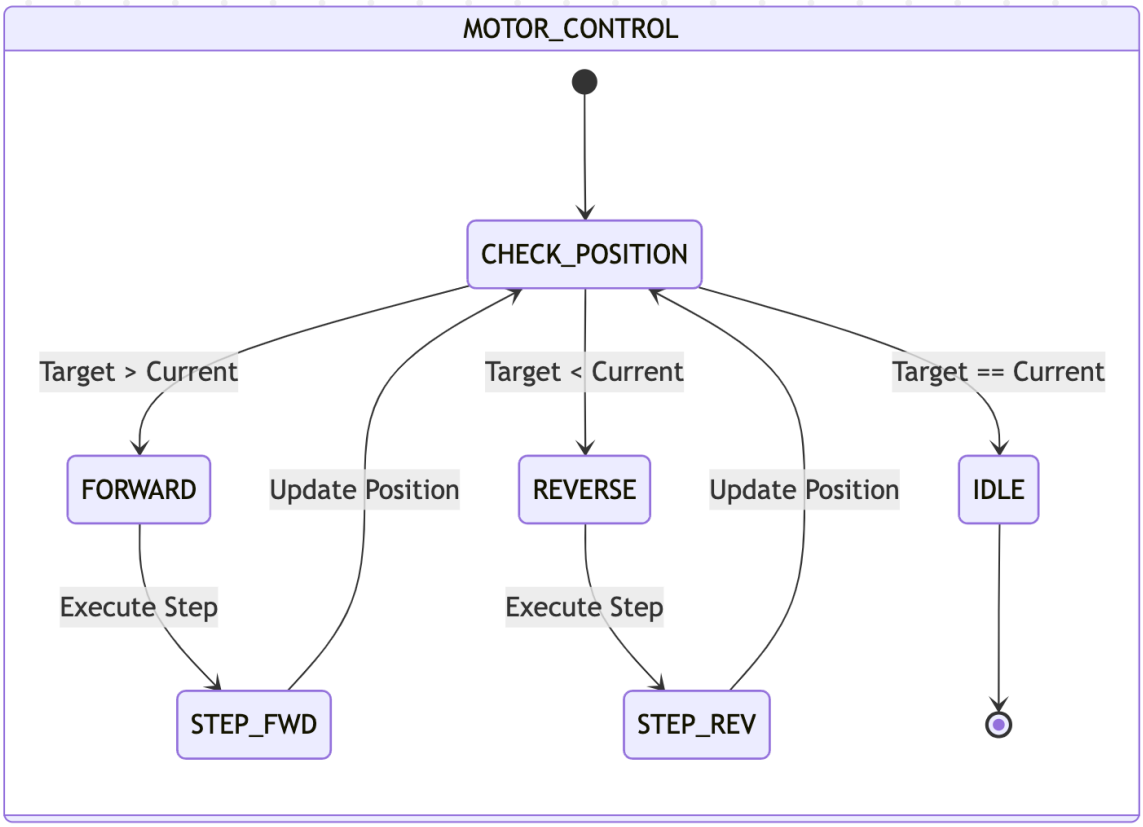
```
================================================================================
 PIN ASSIGNMENT TABLE
================================================================================

 PIN     | PORT  | FUNCTION              | DIRECTION | NOTES
---------|-------|-----------------------|-----------|-----------------
 2       | RA0   | LDR (Light Sensor)    | Input     | Analog AN0
 3       | RA1   | POT (Position)        | Input     | Analog AN1
 33      | RB0   | Motor Phase A         | Output    | To ULN2003
 34      | RB1   | Motor Phase B         | Output    | To ULN2003
 35      | RB2   | Motor Phase C         | Output    | To ULN2003
 36      | RB3   | Motor Phase D         | Output    | To ULN2003
 18      | RC3   | I2C SCL               | Bidir     | 100kHz clock
 23      | RC4   | I2C SDA               | Bidir     | Data line
 25      | RC6   | UART TX               | Output    | 9600 baud (→ IO UART P2-RX)
 26      | RC7   | UART RX               | Input     | 9600 baud (← IO UART P3-TX)
 19      | RD0   | LCD D4                | Output    | 4-bit data
 20      | RD1   | LCD D5                | Output    | 4-bit data
 21      | RD2   | LCD D6                | Output    | 4-bit data
 22      | RD3   | LCD D7                | Output    | 4-bit data
 8       | RE0   | LCD RS                | Output    | Register Select
 9       | RE1   | LCD EN                | Output    | Enable pulse
 10      | RE2   | LCD RW                | Output    | Always LOW (0)
```

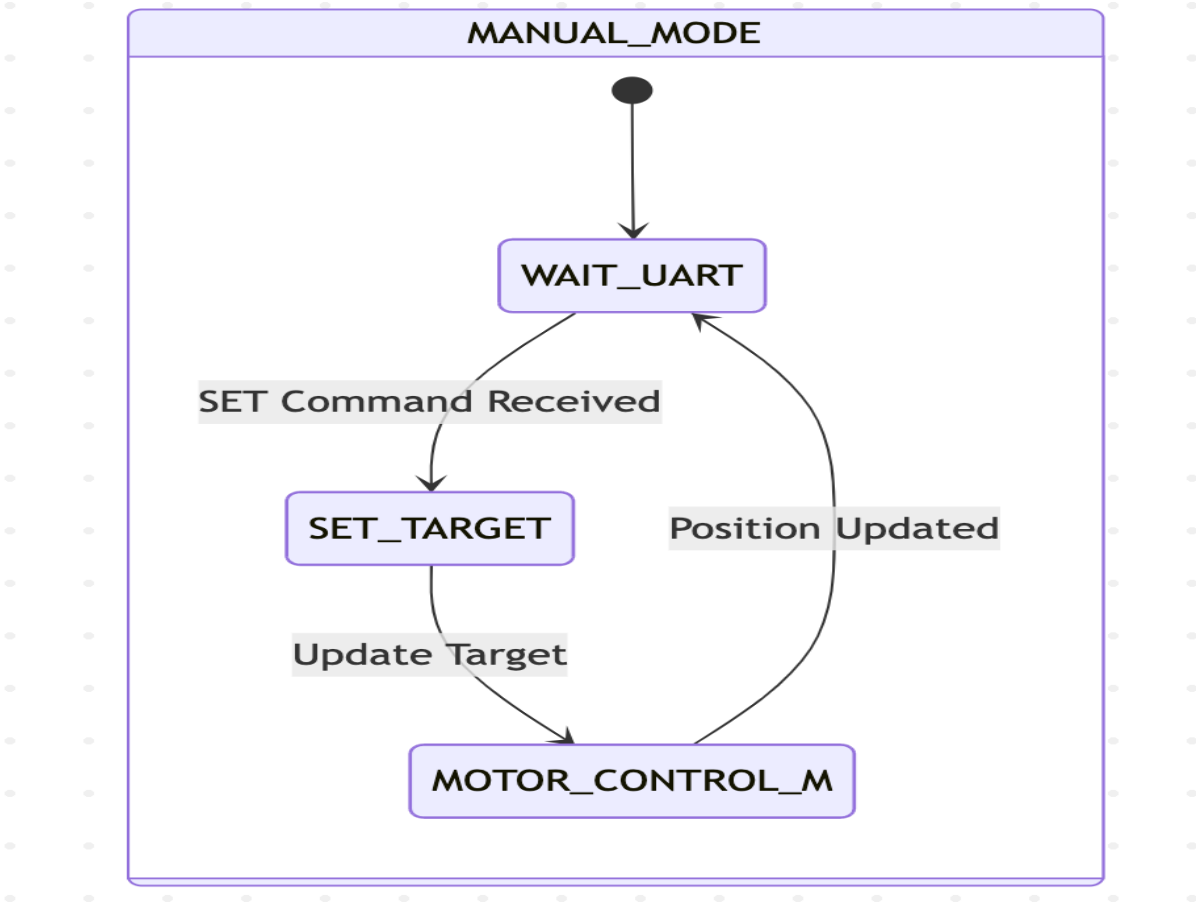
*Pin Assignments for Board #2*




*Circuit Diagram for Board #2*


Power On
INIT
Initialization Complete
AUTO_MODE
CHECK_LIGHT
Light >= 40%
Light < 40%
READ_POT
NIGHT_MODE
Position Updated
Set Target from POT
Target = 100%
MOTOR_CONTROL
UART SET Command
UART Auto Mode 0x09
MANUAL_MODE

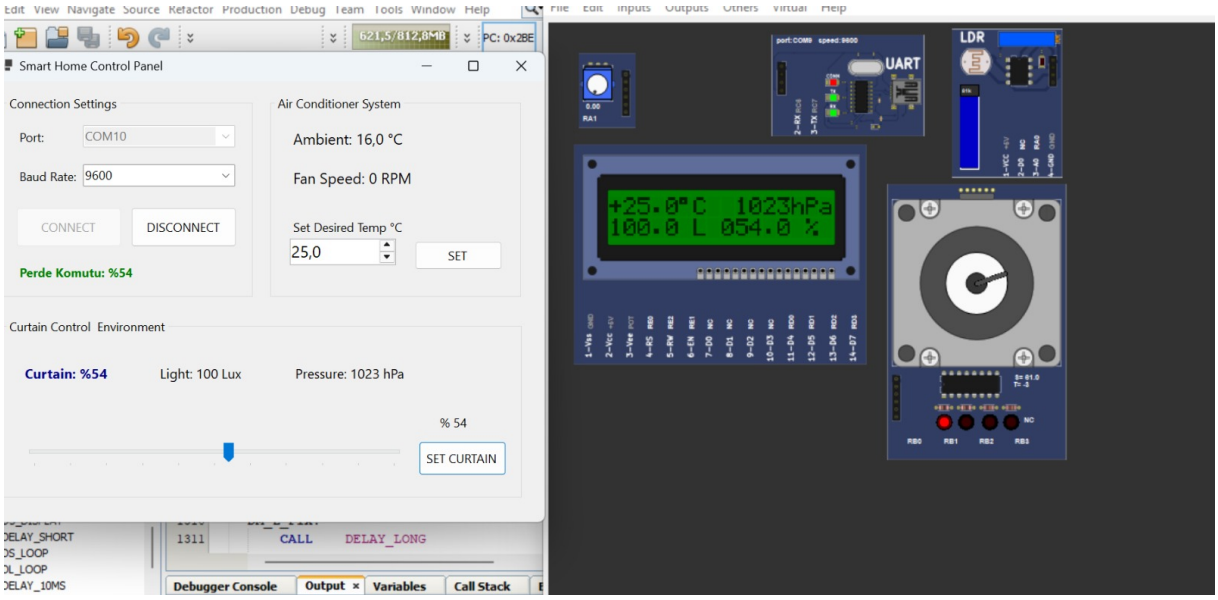

MOTOR_CONTROL
CHECK_POSITION
Target > Current
Target < Current
Target == Current
FORWARD
Update Position
REVERSE
Update Position
IDLE
Execute Step
Execute Step
STEP_FWD
STEP_REV

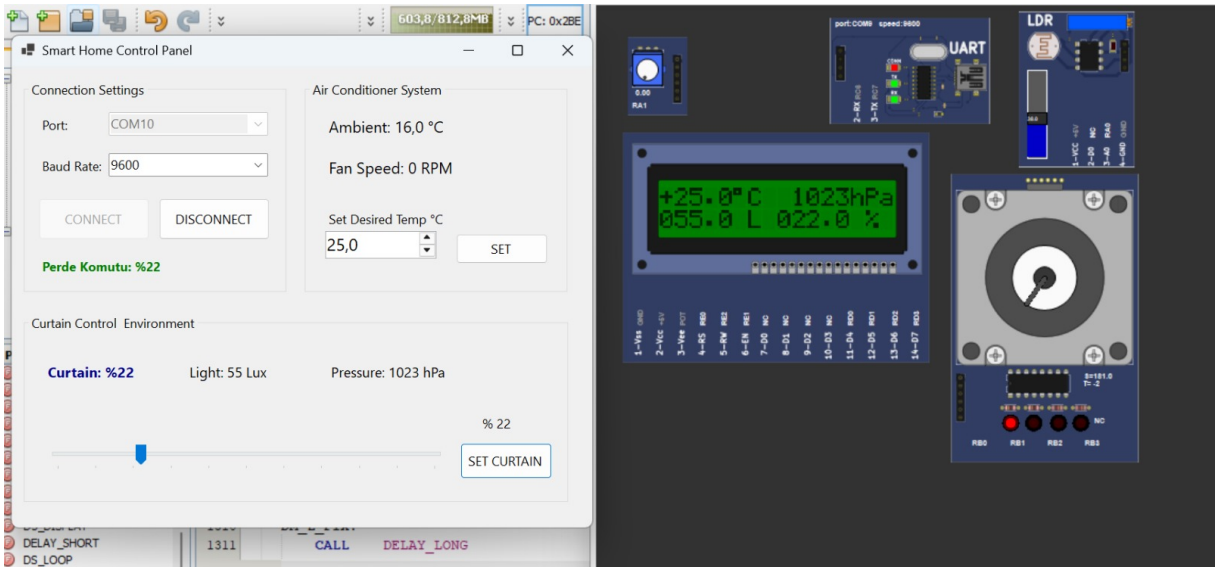



| State | Description |
|---|---|
| INIT | System initialization: Configure ports, ADC, UART ,I2C, LCD |
| AUTO_MODE | Automatic control based on POT and LDR sensors |
| MANUAL_MODE | Remote control via UART commands from PC |
| CHECK_LIGHT | Read LDR sensor, compare with 40% threshold |
| NIGHT_MODE | Light <40%, automatically close curtain to 100% |
| READ_POT | Read potentiometer for target position (0-100%) |
| MOTOR_CONTROL | Move stepper motor toward target position |
| FORWARD | Motor rotating CW (closing curtain) |
| REVERSE | Motor rotating CWW (opening curtain) |
| IDLE | Target reached, no motor movement |

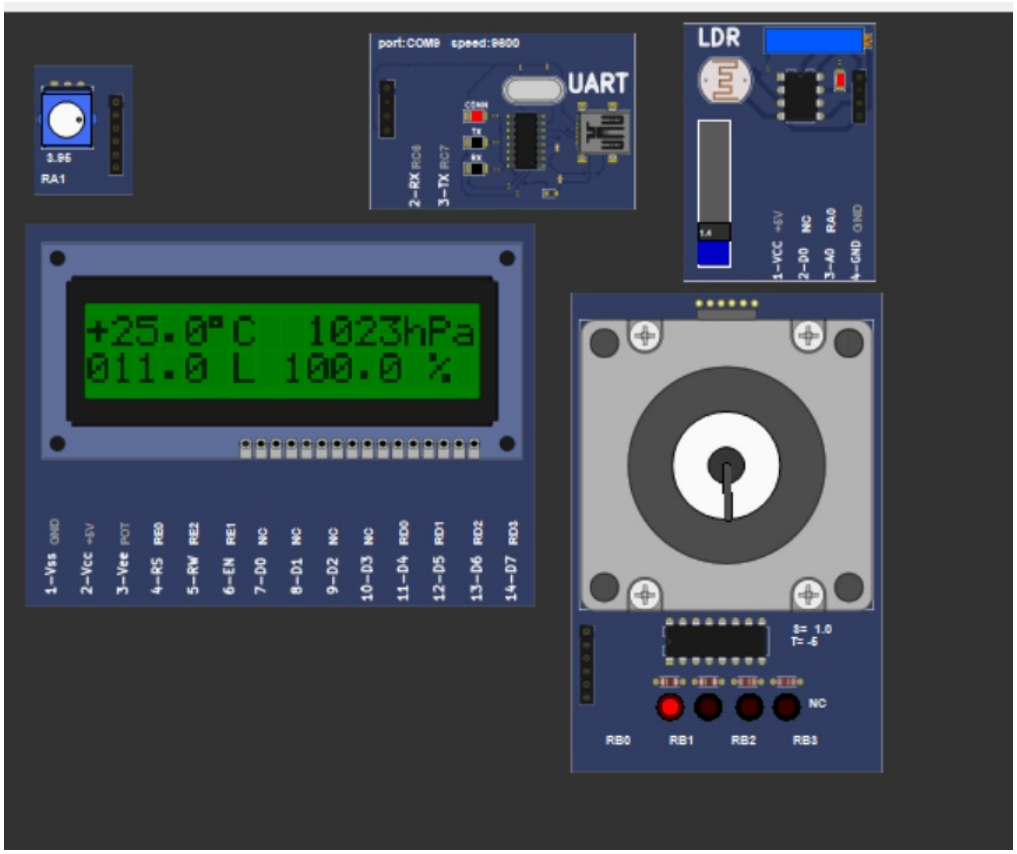
| From | To | Condition |
|---|---|---|
| INIT | AUTO_MODE | LCD initialized, all peripherals ready |
| AUTO_MODE | MANUAL_MODE | UART SET command received (0xC0-0xFF) |
| MANUAL_MODE | AUTO_MODE | UART command 0x09 received |
| CHECK_LIGHT | NIGHT_MODE | LIGHT_INT < 40 |
| CHECK_LIGHT | READ_POT | LIGHT_INT>=40 |
| CHECK_POSITION | FORWARD | CURTAIN_TARGET > CURTAIN_CURRENT |
| CHECK_POSITION | REVERSE | CURTAIN_TARGET < CURTAIN_CURRENT |
| CHECK_POSITION | IDLE | CURTAIN_TARGET == CURTAIN_CURRENT |

From the panel the user controls the percentage of the curtain;

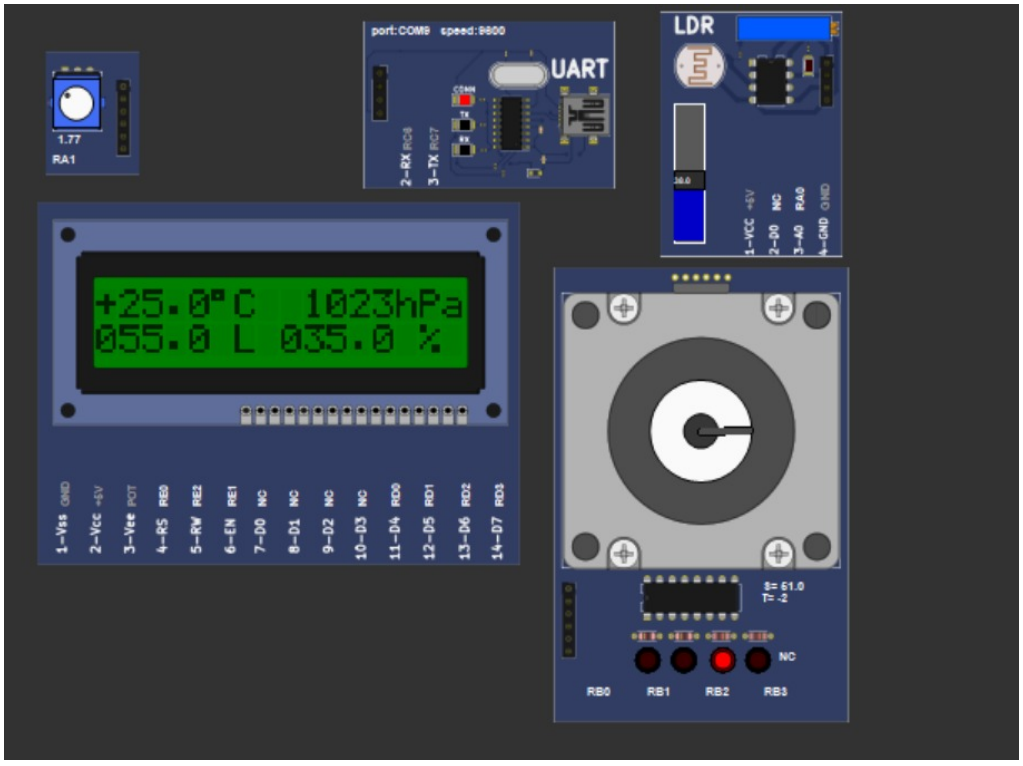


Night Mode (Lux=11, L<40)

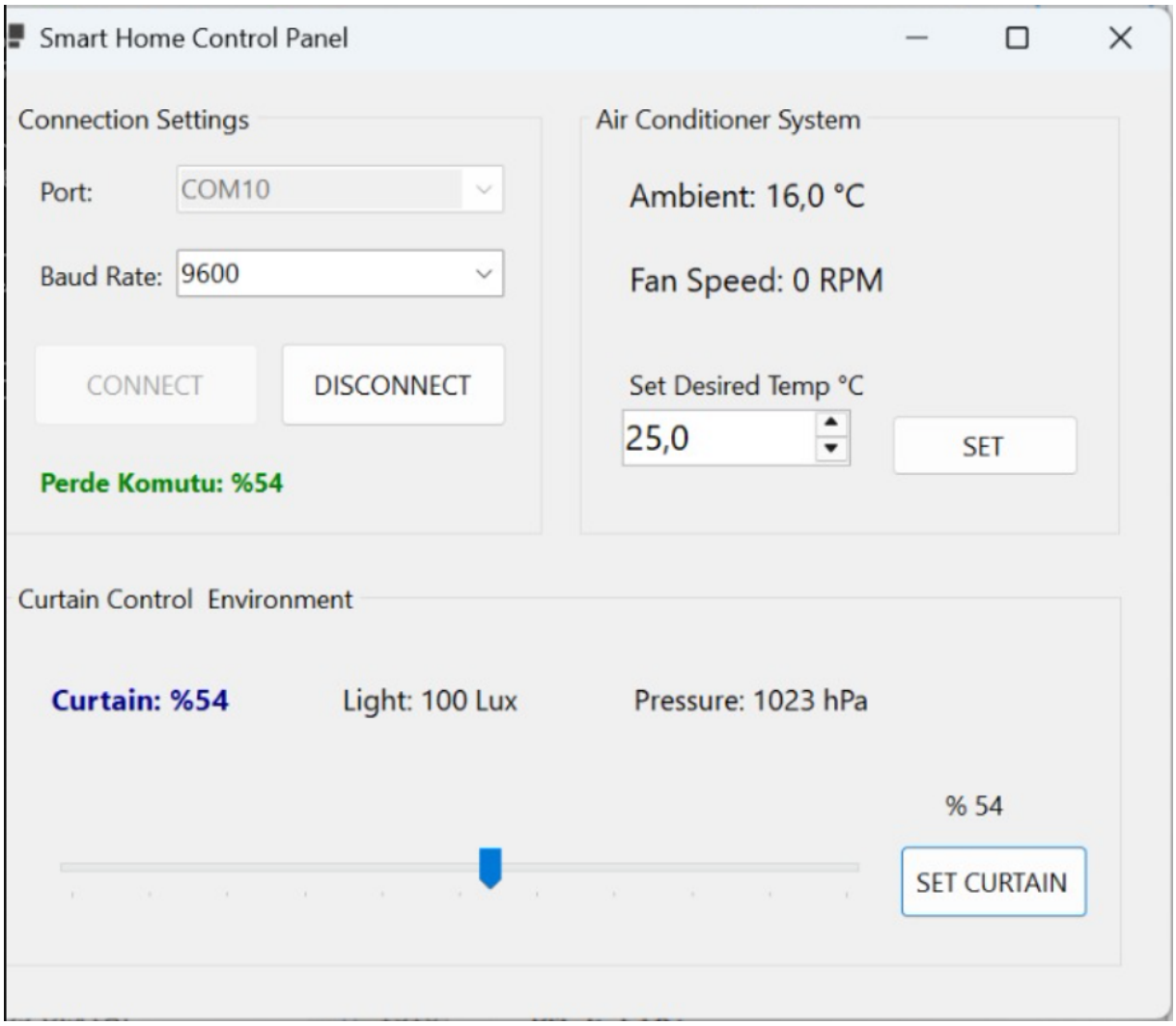



Daytime Mode (Lux=55, L>40)

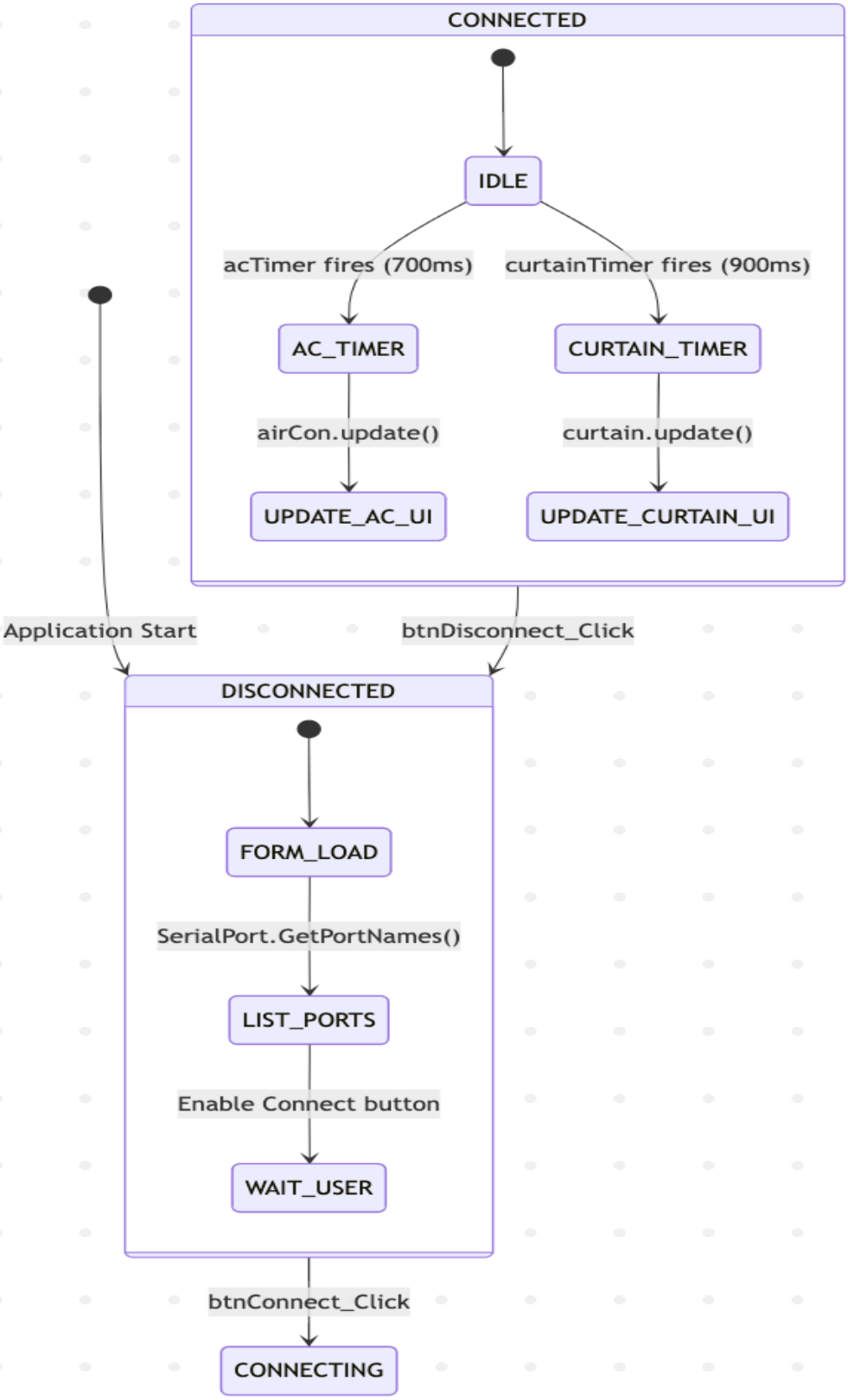


**API Purpose and Functionality**

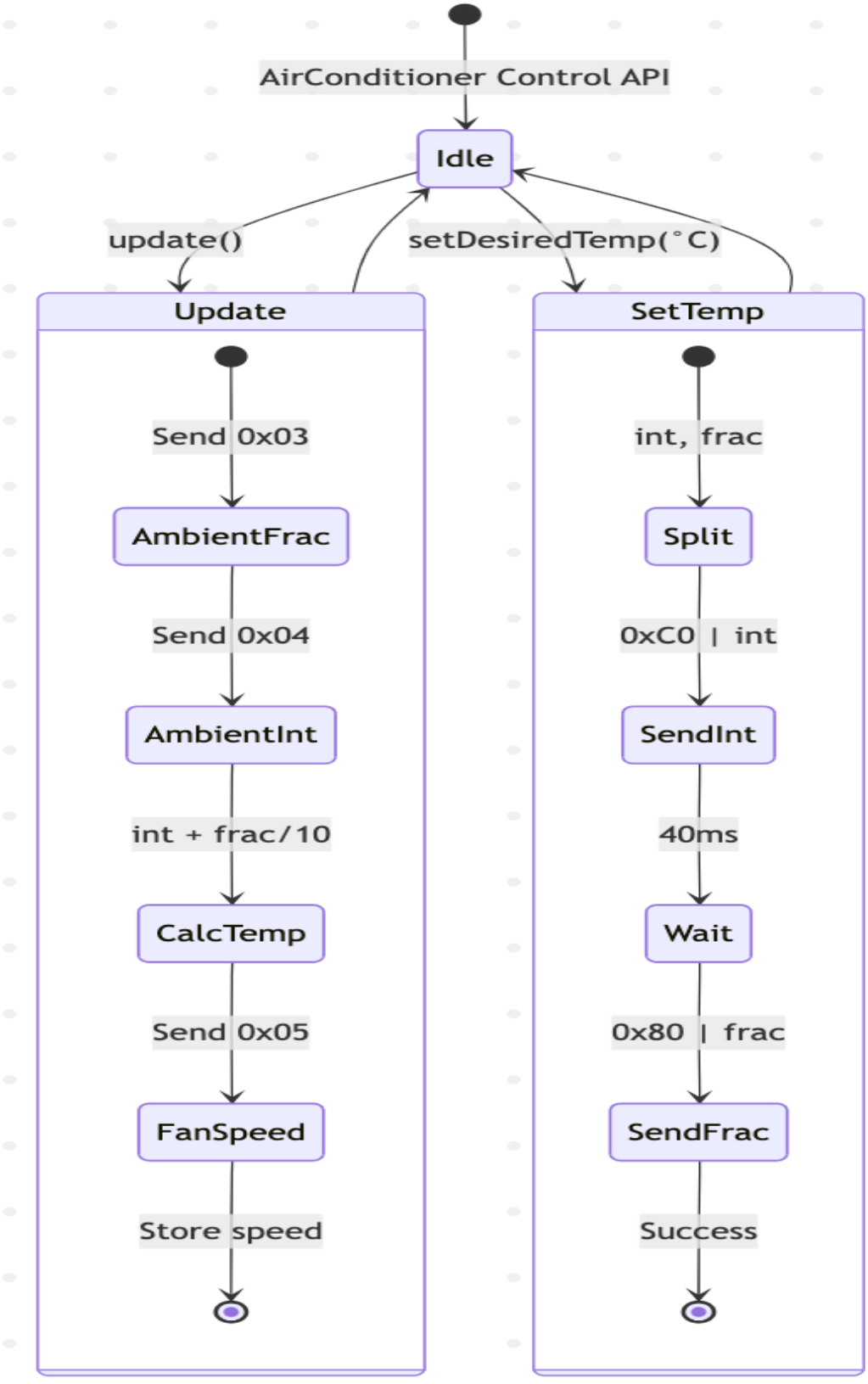
The Smart Home System API is a C# Windows Forms application that serves as the central communication bridge between the user interface and the PIC16F877A microcontroller boards via UART serial communication at 9600 baud. The API enables real-time monitoring and remote control of two subsystems: the Temperature Control System (Board 1) and the Curtain Control System (Board 2). For Board 1, the API reads ambient temperature using GET commands (0x03 for fractional, 0x04 for integer) and fan speed (0x05) and sets new target temperatures within the validated range of 10.0°C to 50.0°C using SET commands (0xC0|value for integer, 0x80|value for fractional). For Board 2, the API monitors curtain position (0x01, 0x02), light level (0x07, 0x08), temperature (0x03, 0x04), and pressure (0x05, 0x06) through GET commands, while allowing users to set curtain positions (0-100%) using the same SET protocol with automatic 0-50 scaling. The application provides a graphical dashboard with real-time value displays, NumericUpDown for temperature setting, TrackBar for curtain position control, and timer-based polling for continuous sensor updates, creating a seamless smart home control experience.



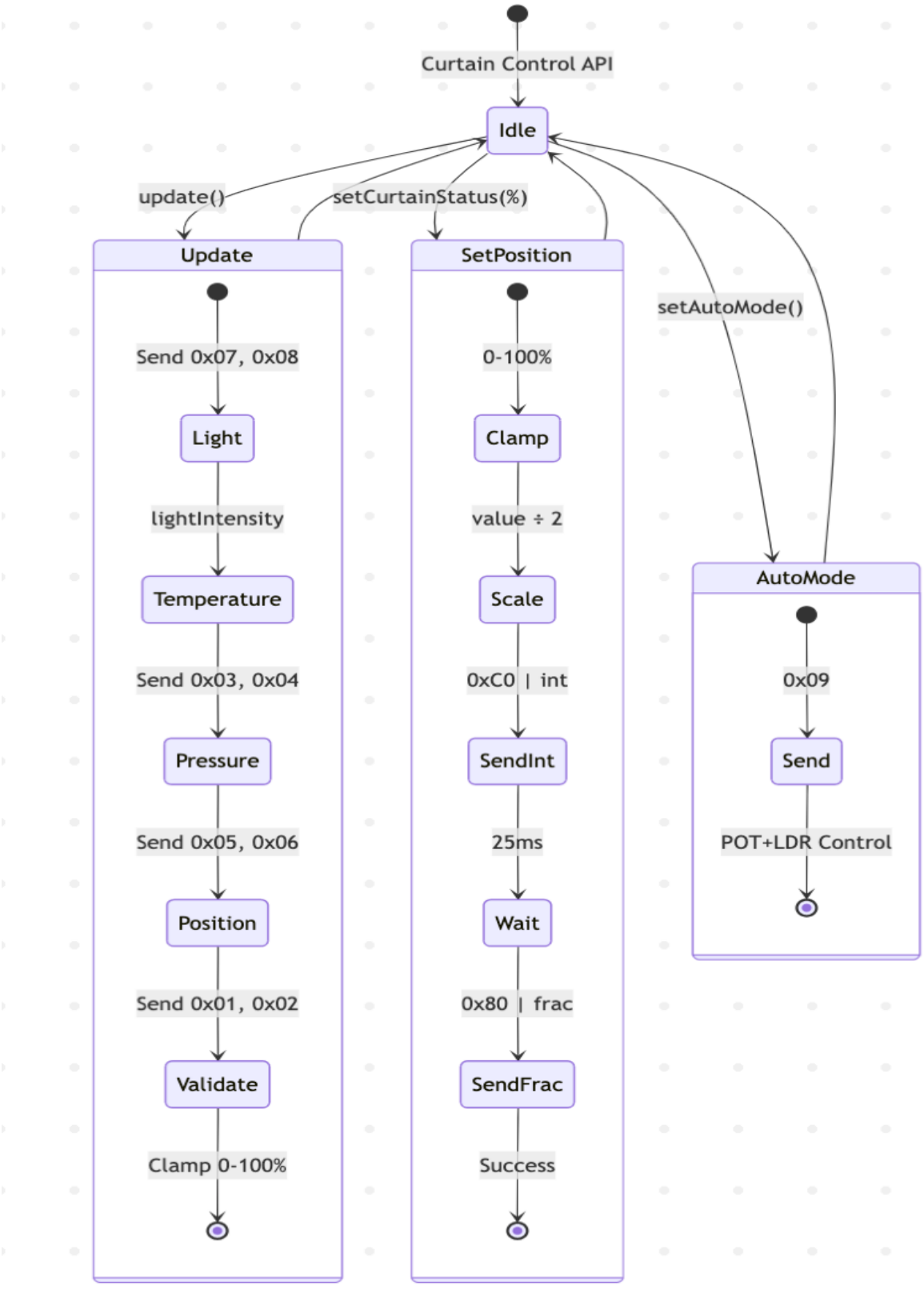

*Smart Home Control Panel*



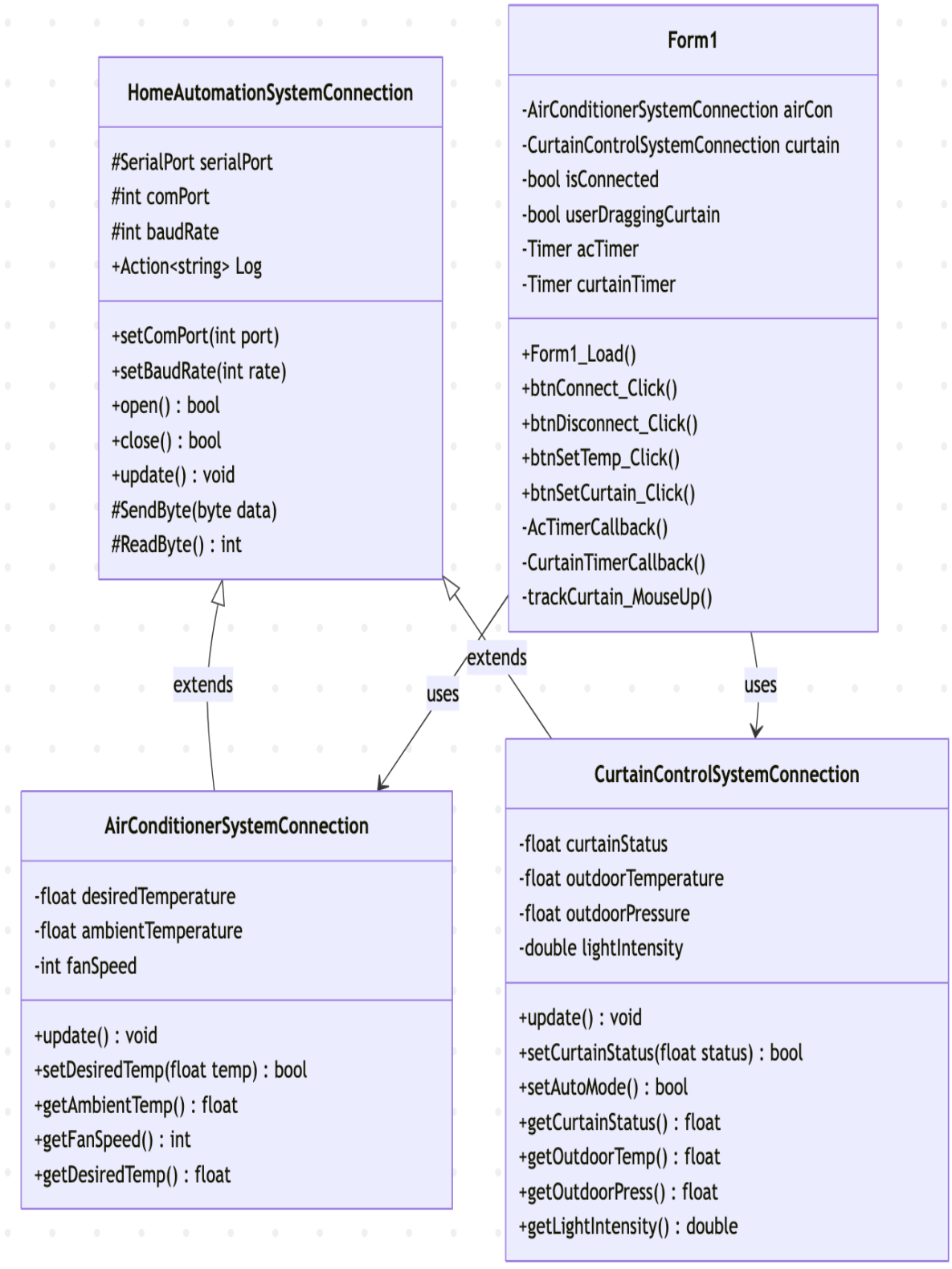

*Application Main State Diagram*



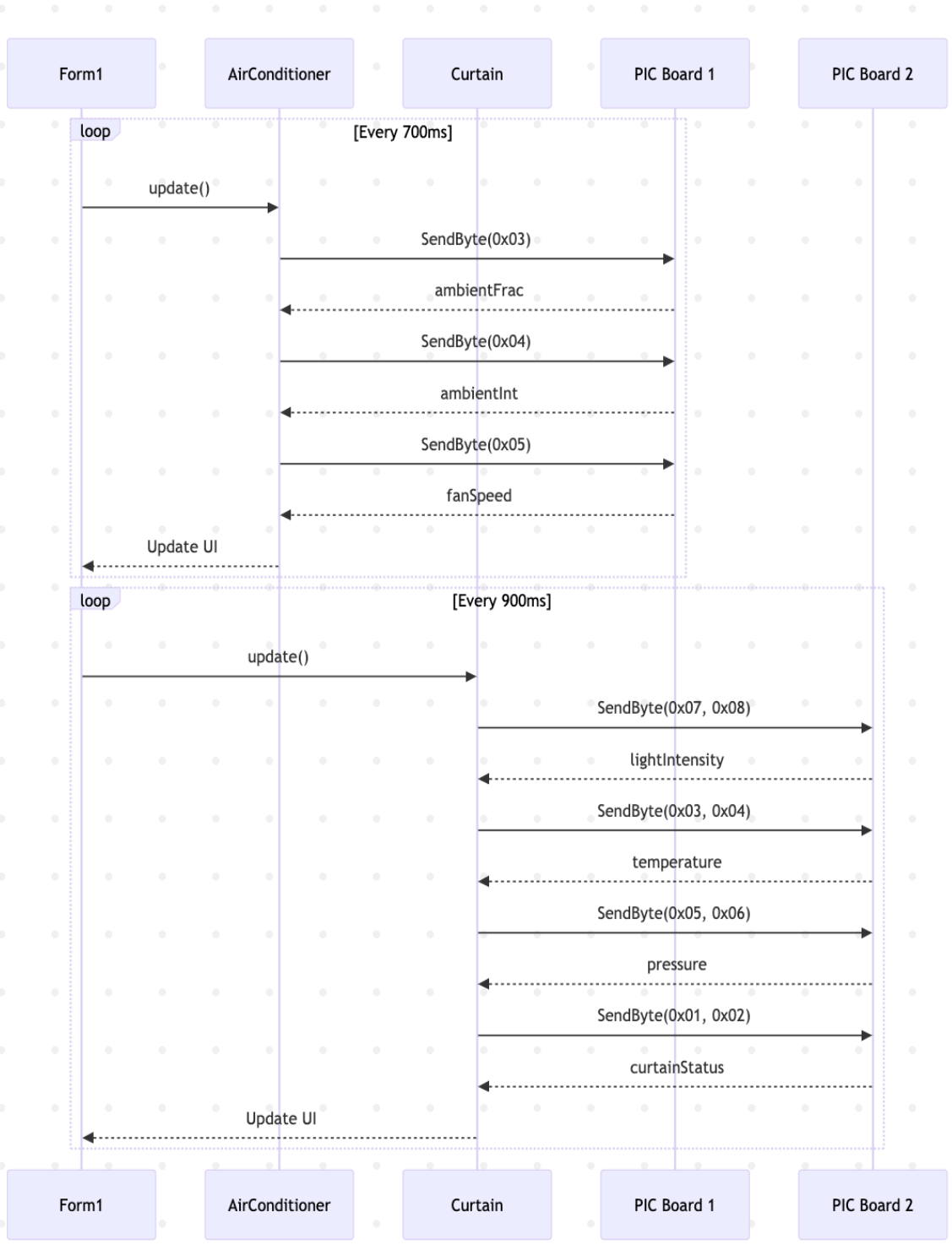

*Air Conditioner Control API - State Diagram*



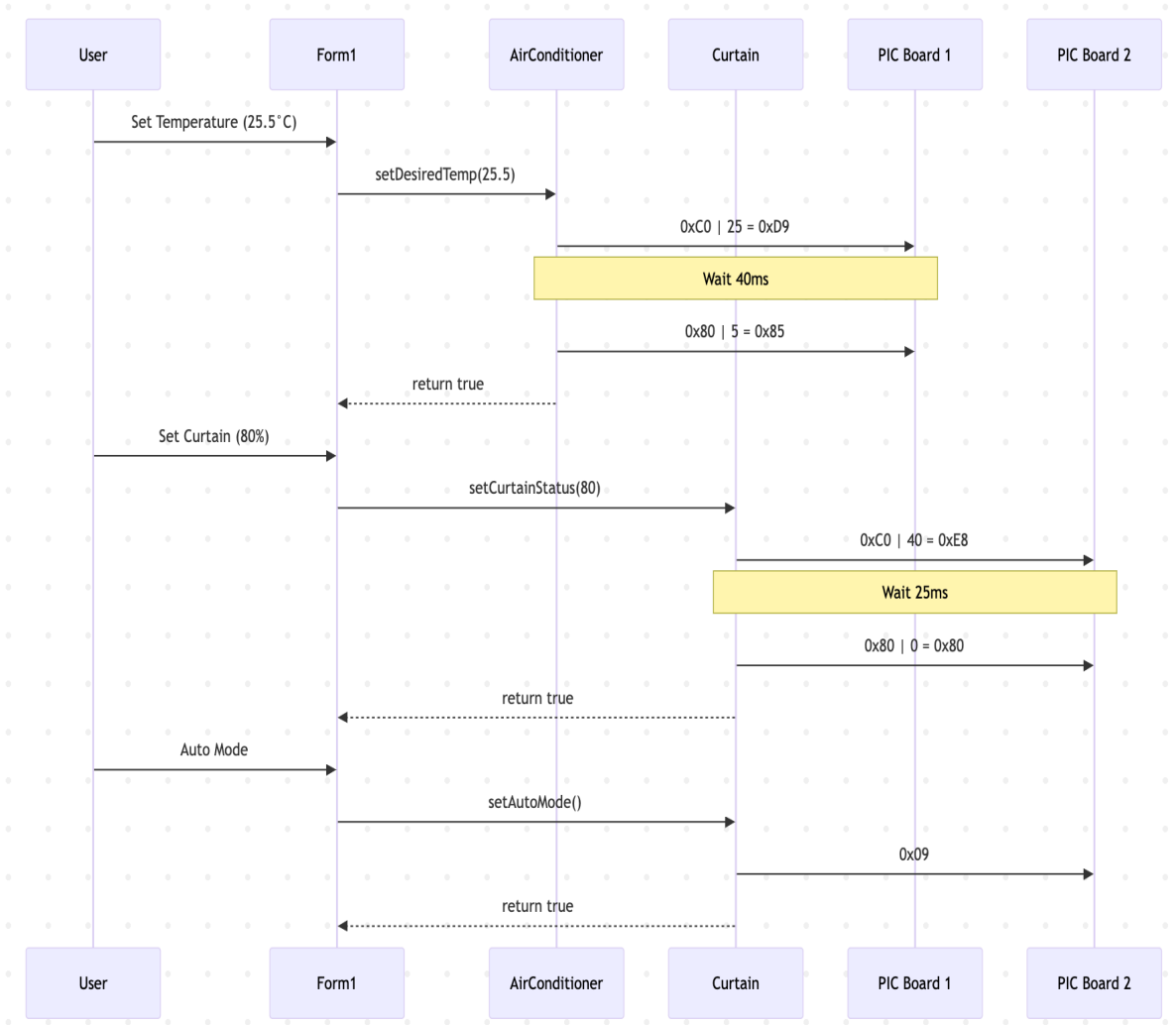

*Curtain Control API - State Diagram*




*UML Class Diagram for API*




*Sequence Diagram- Update Flow*




*Sequence Diagram- Set Commands*

## 3. Task Assignment

152120211096, Abdullah Eren COŞKUN, API Coding
152120221053, Meryem DİNÇ, Construction and coding Board #1
151220212126, Buse BAYRAMLI, Construction Board #2, coding LCD and BMP180 modules, writing report
151220222101, Yağız SÜRÜCÜ, Construction Board #2, coding UART and Potentiometer modules, writing report
151220222104, Furkan ARSLAN, Construction Board #2, coding Step Motor and LDR modules, writing report

## Conclusion

"Smart Home System" term project successfully accomplished its goal of designing a comprehensive automated environment with the use of PIC16F877A microcontroller. By simply incorporating two standalone systems, "Home Air Conditioner System" with Board #1 and "Curtain Control System" with Board #2, this project showcased the applicability of embedded systems concepts like real-time sensor reading, actuators, and serial communication. Using Assembly language in all code perfectly addressed low-level hardware control, with all time-critical functions, such as multiplexed display and stepper motor control, working with utmost efficacy. In Board #1, temperature control circuitry is implemented with considerable validation mechanism regarding acceptable temperatures ranging between 10.0 and 50.0. As far as Board #2 is concerned, it showcased advanced control functionalities with equal emphasis on both manual and automatic control systems for curtains. Implementation of stepper motor control required considerable phase and step counting algorithms for corresponding percentage opening values ranging between 0% and 100%. Moreover, "Night Mode" functionality successfully implemented LDR sensor reading for automated curtain closure below critical light threshold of 40% when ambient light intensity reduced, thus further optimizing overall energy efficiency and user satisfaction levels.The overall data transfer mechanism, realized between boards, successfully involved UART communication at the speed of 9600 bauds. As part of overall system configuration, this communication interface enabled application software in PC to work as central control station, allowing users to remotely monitor reading of sensors (temperature, pressure, and light intensity levels), as well as relay control signals. Even though communication code part for BMP180 pressure sensor is accomplished and laid down in proper coding efforts, due to restrictions in PICSimLab simulation environment, this module remained deactivated during final simulation. It, in fact, led to importance of considering simulation environment and hardware environment in proper perspective.In this manner, this assignment offered considerable experience in terms of software modularity, proper ISR management, and interfacing with peripherals. The effective cooperation of the team members in task allocation, ranging from coding of the API to creating a simulation of the hardware, made sure that all the requirements of the project were completed effectively. The proposed system marks a practical proof of concept for a smart home environment and provides a good platform for further enhancements, such as PWM fan speed control or wireless communication addition.

**Project Source Code:** https://github.com/erencoskun11/SmartHomeSystem