LABORATORY REPORT

To: Dr.Randy Hoover

From: Benjamin Lebrun, Benjamin Garcia

Subject: Lab Assignment 4: Remote Control

Date: March 26, 2019

Introduction

This project required us to create a console application that would read whether a pin was connected to a 5-volt source (would read 'high') or was not (would read 'low'). The application was also required to allow setting other pins to output high/low and to drive LEDs. The console portion was required to output an appropriate error message if invalid inputs were provided. The types of invalid inputs that could be detected included invalid commands (those other than 'set' or 'read'), invalid pins (set operations were only valid for pins 8 and 10, while read operations were only valid for pins 9 and 11), and invalid states (anything other than 'high' or 'low').

We began by creating an input interrupt handler, a buffer for input, and a stream object for output. This allowed us to read input from the interrupt handler into the buffer, and to output information back to the console using the stream object. Once basic IO was handled, we wrote code to tokenize the input on spaces and check the inputs for validity. If input was valid, the appropriate state values were set and execution continued. If input was invalid, the state flags would be set to error values and an appropriate message would be output. Once input processing was finished, low-level IO functions would be invoked based on the command and pin specified, and read or write operations with the pins were performed. After all processing and reading/writing, a message indicating the result of the command is printed to the terminal, and the program awaits further input.

Physically, we use push-buttons connected to the breadboard to handle reading high or low values from pins 9 and 11. If the button is pressed, the pins read 'high' as they are connected to the 5-volt source on the board, otherwise they read low. We have attached LEDs to the buttons to provide a visible indication of when the pins should read 'high' or 'low'.

On reset, the output pins start low, and the input pins are pulled low by the internal pull-down resistors. This means that the LEDs begin off and the read pins are 'low' before any user input is performed. When pin 8 is set high by the 'set 8 high' command, our blue LED will turn on. Similarly, when pin 10 is set high, the red LED will turn on. If either is set low by the 'set (8—10) low' command, the respective LED will turn off. The read pins (9 and 11) read low until their respective buttons are pushed. When the button for pin 9 is pressed, the yellow LED will turn on, and a read from the pin will report 'high'. Similarly, when the button for pin 11 is pressed, the green LED will turn on, and a read on pin 11 will report 'high'.

Equipment

This lab required the following equipment:

- 4 resistors (220Ω)
- 4 LEDs
- 2 push-buttons
- 8 wires (male-male)
- 1 USB-A to USB-B cable
- 1 breadboard
- 1 Arduino UNO (ATMega328P)
- AVR 8-bit GCC toolchain
- Putty

Configuration

For the circuit configuration we have a simple line from pins 8 and 10 from the Arduino unit to the LEDs behind two resistors to ground. For our voltage inputs we have a simple normally open push button switch. For each input, we have a 5 volt rail entering into one side of the push button, the other side has a hardware indication LED connected to ground with a pulldown resistor and our line to input on the arduino. When the button is pressed, it should put 5 volts of potential on the rail and appropriately light the LED.

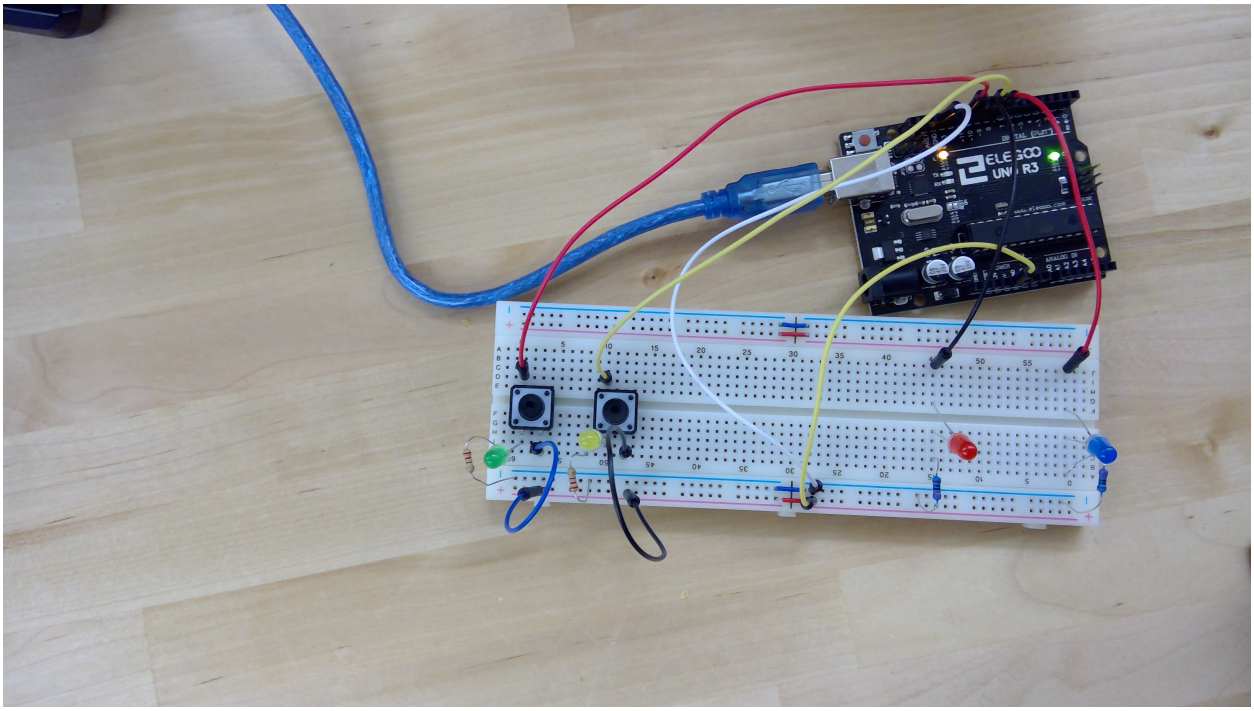

Figure 1: Picture of the wiring setup

Implementation

Our program stores global state information in a struct named *STATE*. This structure is used to conveniently provide functions with state information through a single object.

int main(void)

Performs startup logic by calling *initUART*, *initPins*, and *createDefaultState*, then loops forever using a 'while(1)' empty loop. After startup, program is entirely interrupt driven.

ISR(USART_RX_vect) (interrupt)

Every character of user input is handled by this function, which inserts the character into the *iobuff* character array. The program expects input to be terminated with a carriage-return ('\r'), at which point the *processMessage* will handle the input, and the buffer will be cleared.

void processMessage(STATE* s)

This function tokenizes the input and calls a series of state modifying functions that will each perform error checking. If the input passes validation checks, the command is executed, and the appropriate operations are performed. Invokes *setMessageType*, *setPinNumber*, and *setPinMode* to perform error checking and state setting. If input is empty the invalid command message is printed. The standard library *strtok* function is used to parse the input, and three pointers *combuff*, *pinbuff*, and *setbuff* are used to access the relevant segments.

void setMessageType(STATE state)

Checks the command segment of the input, which is stored in the state parameter, to determine if it is one of the valid commands. If it is not, the *type* state variable is set to MSG_INV and the *inv* state variable is set to INVALID_COMMAND. Otherwise the *type* variable is set to MSG_SET or MSG_READ if the command was 'write' or 'read' respectively. Input can be capitalized (HIGH) or lowercase (low) but not mixed (HiGh).

void setPinNumber(STATE* state)

Once the command has been validated, the next segment of the input is processed to determine which pin the operation applies to (assuming the pin is permissible for that operation). If the pin is invalid for the command, the *type* variable is set to MSG_INV, the *inv* variable is set to INVALID_PIN, and *pin* is set to PIN_NONE. Otherwise pin will have one of the following values:

- PIN_EIGHT
- PIN_NINE
- PIN_TEN
- PIN_ELEVEN

void setPinMode(STATE* state)

If the selected command was 'write' then this function determines if the pin is to be set 'high' or 'low'. if the input is neither 'high' nor 'low' the *pin* value is set to NONE, the *type* variable is set to MSG_INV, and *inv* is set to INVALID_STATE.

void setReadState(int pinStatus, STATE* state)

Sets the *readState* variable in the state struct to HIGH if pinStatus is 0, LOW if pinStatus is 1, and NONE if pinStatus has any other value. The value of pinStatus is expected to be the result of *ReadPinDigital*.

STATE createDefaultState()

Functions as a constructor for the STATE object as C does not support the concept of a constructor natively like C++ does. Initializes a new STATE object to expected default values and returns it.

void initUART()

Configure the UART baud rate and RX Complete interrupt.

void printMsg()

Prints a message to the terminal using the *mystdout* stream based on the state variables. Format strings are stored in strings.h and accessed using the state variables to index into arrays of pointers to the strings. Strings are combined using *fprintf*.

static int uart_putchar(char c, FILE* stream)

Stream input function for *mystdout*. Sends parameter characters one by one through the UART to the terminal.

enum MSG_TYPE

Enumeration of message types. Used to determine what message to print on output and what operation is being performed. Values are used to access the *uiMsgs* array to retrieve format strings for use in *printMsg*;

enum TGT_PIN

Enumeration of pins usable by the program. Operations on pins use these values to indicate which pin is the target of a read or write operation. Usable as an index into the *pinMsgs* array of strings representing the pin name.

enum SET_TYPE

Enumeration of values a pin can have. Can be used as an index into the *stateMsgs* array of strings containing state messages. The NONE value is used in error cases, while HIGH and LOW correspond to their respective pin states.

enum INVALID_TYPE

Used with the MSG_INV value of the MSG_TYPE enum to specify why an input was invalid. Indexes into the *errorMsgs* array to specify which string to use in the MSG_INV format string.

void initPins()

Initializes the states of the pins used in the program. All pins are written low to begin, and (labeled) pins 8 and 10 are marked for output.

int ReadPinDigital(enum TGT_PIN pin

Handles reading from the pin indicated by *pin* and returns 0 if the pin is low or 1 if the pin is high. If the pin is invalid for the read operation -1 is returned.

int WritePinDigital(enum TGT_PIN pin, enum SET_TYPE mode

Handles writing to the pin indicated by *pin*. Sets the pin to the mode indicated by *mode* and returns 1 if the pin is valid, otherwise returns 0 indicating an invalid setup.

strings.h

We use strings.h to hold several c-strings for handling output to the terminal. We access these strings by means of arrays of pointers to the strings, where the arrays can be accessed by enum values. This makes the code more general and easier to work with.

Discussion

The first of many troubles, for user input we had an issue with carriage returns versus newlines, how our code would determine the end of a buffer input. We initially checked only for `'\n'` inside of our buffer and initiated a buffer flush and process the string. The buffer was failing to flush and read properly, when we checked for a carriage return instead of a newline character our buffer was properly able to flag the end of a user command and begin processing the string

Pin set configuration and port directions and masks were the next issue. This was simply a lapse in judgement on the programmer's part in how to properly twiddle bits around in code to single out the necessary register bit.

Pulldown/Pullup internal resistor with our input ports was another fabulous discovery. We used buttons to activate our pins however we discovered that their results weren't entirely deterministic. To solve this and add a nice feature on top we used an external puldown resistor paired with an LED to indicate when power was flowing through the switch as a hardware fallback indicator.

In the real world, many components use UART communication where more advanced or complicated data must be comunicated. This also applies to communication between computers or other microcontrollers on a unified system. Because of this, we must know how to also communicate using different interfaces, such as GPIO when UART is either occupied or unavailable.

In pursuit of this, we learned how to utilize processing commands with UART communication between two machines in this lab. We also explored reading information from external sources by using the GPIO pins on our microcontroller board.

Responses

1. Checksums are often used to verify the integrity of transmitted data. What is a checksum? Walk through the calculation of the modular sum checksum for the ASCII string ”CENG447”. Now generate a longitudinal parity byte for the same string.

   A checksum is a piece of data used to test incoming data for transmission integrity. There's several ways to calculate a checksum, one example of a checksum is a modular sum checksum. This simple checksum adds the digits of the message together, continuously performing modulus with every sum until we have a final remainder.

   For our test string ”CENG447”, using a modular sum checksum, induvidually adding together each letter and modulating the result continuously by 255, our checksum comes out to 189.

2. Several different communication standards are used for inter-system serial communication, most notably RS-232 and RS-485. What are the differences between RS-232, RS-485, and the UART on our ATMega328Ps?

   For the serial standards RS-232 and RS-485, the major difference between these two standards is the voltage level of the devices that use these standards. The RS-485, uses an activation voltage of 200 mV in either the positive and negative direction, but the RS-232 uses a positive/negative voltage of 3V, both activate as digital 1 on the negative edge.

   The RS-232 has a lower minimum impedance, but has a smaller transmission range across it's cable and a smaller transmission rate. The RS-485 has a superior transmission cable length and superior transmission rate.

3. Our ATMega328Ps have built-in pullup resistors. What is a pullup/pulldown resistor, and what is a risk of not using them? How does one enable the ATMega328P's internal pullup resistors?

   A pullup or pulldown resistor assist the signal in being able to hold either a grounded or reference voltage signal. Without an internal or external pullup or pulldown resistor, the pin generally ”floats” or does not hold either signal discriminately, and could return improper digital readings when accessed. To enable the board's internal pullup resistor, while the pin is in read mode, we write a 1 to the necessary pin bit with its appropriate *PORTX* register, replacing *X* with our port.

Appendices

The following files are included as appendices:

- main.c - entry file, manages initialization logic and interrupt handling
- pinops.h - header file for the pin operations file
- pinops.c - handles pin operations

- strings.h - has static string constants for testing and printing. Also declares the *printMsg* function for outputting to the terminal.

- strings.c - defines the *printMsg* function.

- atoi.h - declaration for the atoi function.

- atoi.c - definition for the atoi function.

- state.h - contains the definition for the STATE struct and declarations for state setting functions.
- state.c - contains the definitions for the state setting functions.
- enums.h - has enumerations for program state and output

Appendix A: main.c

```
#define F_CPU 16000000
#define USART_BAUDRATE 9600
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)
#include "avr/interrupt.h"
#include "avr/io.h"
#include "pinops.h"
#include "state.h"
#include "stdio.h"
#include "strings.h"
#include "util/delay.h"

/* FUNCTION PROTOTYPES */
static int uart_putchar(char c, FILE* stream);
void initUART();

/* Program State */
static STATE globalState;

/* stdout stream */
static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL, _FDEV_SETUP_WRITE);

// program entry point, runs initialization routines.
int main(void)
{
    initUART();
    initPins();
    globalState.type = MSG_INIT;
    globalState = createDefaultState();
    // set global interrupts
    sei();
    printMsg(&mystdout, &globalState);
    while (1)
    {
    }
    return 1;
}

void processMessage(STATE* s)
{
    setBuffers(s);
    if (s->combuff == NULL)
    {
        s->type = MSG_INV;
        s->inv = INVALID_COMMAND;
    }
    else
    {
        // these functions change global state
        // based on the input buffer
        setMessageType(s);
        setPinNumber(s);
        switch (s->type)
        {
        case MSG_READ:
            setReadState(ReadPinDigital(s->pin), s);
            break;
        case MSG_SET:
            setPinMode(s);
            if (s->type != MSG_INV)
                WritePinDigital(s->pin, s->setState);
            break;
        case MSG_INV:
        case MSG_INIT:
            break;
        }
    }

    printMsg(&mystdout, s);
}

ISR(USART_RX_vect)
{
    char inByte = UDR0;

    // processing goes here
    if (inByte == '\r')
    {
        globalState.buffend = 0;
        // process string here
        processMessage(&globalState);
        // clear io buffer when done
        memset(globalState.iobuff, 0, 32 * sizeof(char));
    }
    else
    {
        // continue to write to a fifo buffer
        globalState.iobuff[globalState.buffend++] = inByte;
        inByte = 0;
        // check if the buffer is at capacity
```

```
        // (shouldn't ever happen under) normal circumstances
        if (globalState.buffend == 31)
        {
            // print buffer overflow error msg.
            fprintf(&mystdout, overflowMsg);
            // set index back to 0
            globalState.buffend = 0;
            // clear the buffer
            memset(globalState.iobuff, 0, 32 * sizeof(char));
        }
    }
}

void initUART()
{
    // init uart
    UCSR0B |= 0x98;
    UCSR0C |= 0x06;
    UBRR0L = BAUD_PRESCALE;
    UBRR0H = (BAUD_PRESCALE >> 8);
}

static int uart_putchar(char c, FILE* stream)
{
    if (c == '\n')
        uart_putchar('\r', stream);
    loop_until_bit_is_set(UCSR0A, UDRE0);
    UDR0 = c;
    return 0;
}
```

Appendix B: pinops.h

```
#ifndef _PINOPS_H_
#define _PINOPS_H_
#include "state.h"
#include <avr/io.h>
void initPins();
int ReadPinDigital(enum TGT_PIN pin);
int WritePinDigital(enum TGT_PIN pin, enum SET_TYPE mode);
#endif
```

Appendix C: pinops.c

```
#include "pinops.h"

/*
 * Sets pins into correct modes
 * 8   - out(1)
 * 9   - in (0)
 * 10  - out(1)
 * 11  - in (0)
 */
void initPins()
{
    DDRB = 0x05;
    PORTB = 0x00;
    return;
}

/*
 * Reads current status of target pin
 * returns 0 if no signal, returns -1
 * for invalid pin, otherwise non zero
 * numbers indicate pin is set
 *
 * Pins 9 and 11 are inputs
 *
 * RETURNS:
 * -1 - Invalid pin
 * 0 - Pin not set
 * Else - Pin set
 */
int ReadPinDigital(enum TGT_PIN pin)
{
    int pinstatus = PINB;
    switch (pin)
    {
    case PIN_NINE:
        return ((pinstatus & 0x02) ? 1 : 0);
        break;
    case PIN_ELEVEN:
        return ((pinstatus & 0x08) ? 1 : 0);
        break;
    default:
        return -1;
    }
    return -1;
}

/*
 * Sets target pin to target mode,
 * If invalid pin will return 1,
 * otherwise should return 0
 *
 * Pin 8 and 10 are outputs
 *
 * RETURNS:
 * 1 - Pin not set
 * 0 - Pin was set
 */
int WritePinDigital(enum TGT_PIN pin, enum SET_TYPE mode)
{
    switch (pin)
    {
    case PIN_EIGHT:
        switch (mode)
        {
        case HIGH:
            PORTB |= 0x01;
            break;
        case LOW:
            PORTB &= ~0x01;
            break;
        default:
            return 1;
        }
        break;
    case PIN_TEN:
        switch (mode)
        {
        case HIGH:
            PORTB |= 0x04;
            break;
        case LOW:
            PORTB &= ~0x04;
            break;
        default:
            return 1;
        }
        break;
    default:
        return 1;
```

```
    }
    return 0;
}
```

Appendix D: enums.h

```
#ifndef _ENUMS_H_
#define _ENUMS_H_
/* ENUM TYPES */
enum MSG_TYPE
{
    MSG_INV = 0,
    MSG_SET,
    MSG_READ,
    MSG_INIT
};

enum TGT_PIN
{
    PIN_EIGHT = 0,
    PIN_NINE,
    PIN_TEN,
    PIN_ELEVEN,
    PIN_NONE
};

enum SET_TYPE
{
    LOW = 0,
    HIGH,
    NONE
};

enum INVALID_TYPE
{
    INVALID_STATE = 0,
    INVALID_PIN,
    INVALID_COMMAND,
    INVALID_NONE
};
#endif
```

Appendix E: strings.h

```
#ifndef _STRINGS_H_
#define _STRINGS_H_
#include "state.h"
#include "stdio.h"

// ui msgs
static const char invalidMsg[] =
    "%s, Command structure: 'READ/ (WRITE) PIN_NUM (HIGH|LOW)'\r\n";
static const char setPin[] = "Pin %s set %s\r\n";
static const char readPin[] = "Pin %s reads %s\r\n";
// state msgs
static const char lowMsg[] = "LOW";
static const char highMsg[] = "HIGH";
static const char noneMsg[] = "NONE";
// pin msgs
static const char pinEight[] = "eight";
static const char pinNine[] = "nine";
static const char pinTen[] = "ten";
static const char pinEleven[] = "eleven";
static const char pinNone[] = "none";
// invalid types
static const char invalidState[] = "INVALID STATE";
static const char invalidPin[] = "INVALID PIN";
static const char invalidCommand[] = "INVALID COMMAND";
static const char invalidNone[] = "";

// initial msgs
static const char commands[] = "Available commands are: WRITE, READ\r\n"
                               "Instruction formats:\r\n"
                               "WRITE: WRITE (out_pin) (HIGH|LOW)\r\n"
                               "sets 'out_pin' high or low\r\n"
                               "READ: READ (in_pin)\r\n"
                               "reads the state of 'in_pin'\r\n";
static const char pins[] = "Valid pins:\r\n"
                           "in_pin: 9, 11\r\n"
                           "these pins may be used with the READ command\r\n"
                           "out_pin: 8, 10\r\n"
                           "these pins may be used with the WRITE command\r\n";
static const char additionalNotes[] =
    " Additional notes:\r\n"
    "- commands and set states may be all-uppercase or all-lowercase, but not "
    "mixed\r\n"
    "- commands are expected to be space separated, and input is processed on "
    "newline input\r\n";

// overflow msg
static const char overflowMsg[] = "ERROR: input over max length (31 characters "
                                  "+ carriage return). clearing buffer.";

// msg lists
static const char* uiMsgs[] = {invalidMsg, setPin, readPin};
static const char* stateMsgs[] = {lowMsg, highMsg, noneMsg};
static const char* pinMsgs[] = {pinEight, pinNine, pinTen, pinEleven, pinNone};
static const char* errorMsgs[] = {invalidState, invalidPin, invalidCommand,
                                  invalidNone};

// prototype for printMsg
void printMsg(FILE* out, STATE* state);

#endif
```

Appendix F: strings.c

```
#include "strings.h"

void printMsg(FILE* out, STATE* state)
{
    if (state->type == MSG_INV)
    {
        fprintf(out, uiMsgs[state->type], errorMsgs[state->inv]);
    }
    else if (state->type == MSG_SET)
    {
        fprintf(out, uiMsgs[state->type], pinMsgs[state->pin],
                stateMsgs[state->setState]);
    }
    else if (state->type == MSG_READ)
    {
        fprintf(out, uiMsgs[state->type], pinMsgs[state->pin],
                stateMsgs[state->readState]);
    }
    else if (state->type == MSG_INIT)
    {
        fprintf(out, commands);
        fprintf(out, pins);
        fprintf(out, additionalNotes);
```

```
    }
}
```

Appendix G: atoi.h

```
#ifndef _STRING_OPS_H_
#define _STRING_OPS_H_
int atoi(char* c);
#endif
```

Appendix H: atoi.c

```
#include "atoi.h"

// convert a null terminated string containing
// the ascii representation of a number to its
// integer representation
int atoi(char* c)
{
    int result = 0;

    for (int i = 0; c[i] != '\0'; i++)
    {
        result = result * 10 + (c[i] - 0x30);
    }

    return result;
}
```

Appendix I: state.h

```
#ifndef _STATE_H_
#define _STATE_H_
#include "atoi.h"
#include "enums.h"
#include "string.h"
/* State struct, contains the global state of the program*/
typedef struct STATE
{
    // state flags
    enum MSG_TYPE type;
    enum TGT_PIN pin;
    enum SET_TYPE setState;
    enum SET_TYPE readState;
    enum INVALID_TYPE inv;

    // input buffer
    char iobuff[32];
    char* combuff;
    char* pinbuff;
    char* setbuff;
    int buffend; // tracks end of iobuff
} STATE;

/* STATE object constructor */
STATE createDefaultState();

/* State manipulation functions*/
void setBuffers(STATE* state);
void setMessageType(STATE* state);
void setPinNumber(STATE* state);
void setPinMode(STATE* state);
void setReadState(int pinStatus, STATE* state);

#endif
```

Appendix J: state.c

```
#include "state.h"

STATE createDefaultState()
{
    STATE out;
    out.type = MSG_INIT;
    out.pin = PIN_NONE;
    out.setState = NONE;
    out.readState = NONE;
    out.inv = INVALID_NONE;
    out.buffend = 0;
```

```
    return out;
}

void setBuffers(STATE* state)
{
    state->combuff = strtok(state->iobuff, " ");
    state->pinbuff = strtok(NULL, " ");
    state->setbuff = strtok(NULL, " ");
}

void setMessageType(STATE* state)
{
    if (!memcmp("write", state->combuff, 3) ||
        !memcmp("WRITE", state->combuff, 3))
    {
        state->type = MSG_SET;
    }
    else if (!memcmp("read", state->combuff, 4) ||
             !memcmp("READ", state->combuff, 4))
    {

        state->type = MSG_READ;
    }
    else
    {
        state->inv = INVALID_COMMAND;
        state->type = MSG_INV;
    }
}

void setPinNumber(STATE* state)
{

    state->inv = INVALID_PIN;
    state->pin = PIN_NONE;
    int input = atoi(state->pinbuff);
    switch (state->type)
    {
    case MSG_SET:
        if (input == 8)
        {
            state->pin = PIN_EIGHT;
            state->inv = INVALID_NONE;
        }
        else if (input == 10)
        {
            state->pin = PIN_TEN;
            state->inv = INVALID_NONE;
        }
        break;
    case MSG_READ:
        if (input == 9)
        {
            state->pin = PIN_NINE;
            state->inv = INVALID_NONE;
        }
        else if (input == 11)
        {
            state->pin = PIN_ELEVEN;
            state->inv = INVALID_NONE;
        }
        break;
    case MSG_INV:
        break;
    }

    if (state->inv == INVALID_PIN)
    {
        state->type = MSG_INV;
    }
}

void setPinMode(STATE* state)
{
    if (!memcmp("high", state->setbuff, 4) ||
        !memcmp("HIGH", state->setbuff, 4))
    {
        state->setState = HIGH;
    }
    else if (!memcmp("low", state->setbuff, 3) ||
             !memcmp("LOW", state->setbuff, 3))
    {
        state->setState = LOW;
    }
    else
    {
        /*ERROR*/
        state->type = MSG_INV;
        state->inv = INVALID_STATE;
    }
}
```

```
void setReadState(int pinStatus, STATE* state)
{
    if (pinStatus == 0)
    {
        state->readState = LOW;
    }
    else if (pinStatus == 1)
    {
        state->readState = HIGH;
    }
    else
    {
        state->readState = NONE;
    }
}
```