Lab 5 Report

Sage Marks, Matt Krueger

3360:0001 - Embedded Systems

Professor Beichel

University of Iowa, College of Engineering

1. Introduction

We are beginning to C why there are few and far between assembly-only programmers. Lab 5 served as an introduction to the C programming language, which gives the developer more freedom than AVR assembly which was used prior. To dip our toes before the final project, Lab 5 was used as a playground. On this playground, we learned device to device communication and expanded our signal processing toolbox by programming a DAC and ADC. The product was a terminal emulator, which can be used to take measurements of voltage and set voltages at respective locations. Communication between the terminal and the microcontroller was handled by RS232 and communication between the microcontroller and the MAX518 was handled via I2C communication. The ADC is on-chip and thus controlled via registers.

Listed below is an exhaustive list of all the hardware necessary to complete this remote logging system, including a potentiometer to generate analog voltages of variable levels.

Hardware	Quantity	Description
Atmega 328P µC	1	Programmable µC
10KΩ Resistor	2	Pull-Up resistors for I2C communication
0.01µF Capacitor	1	Decoupling Capacitor
B103 Potentiometer	1	Potentiometer for Analog Voltage Control
MAX518 DAC	1	Digital to Analog Converter

Figure 1: Materials List

2. Schematic

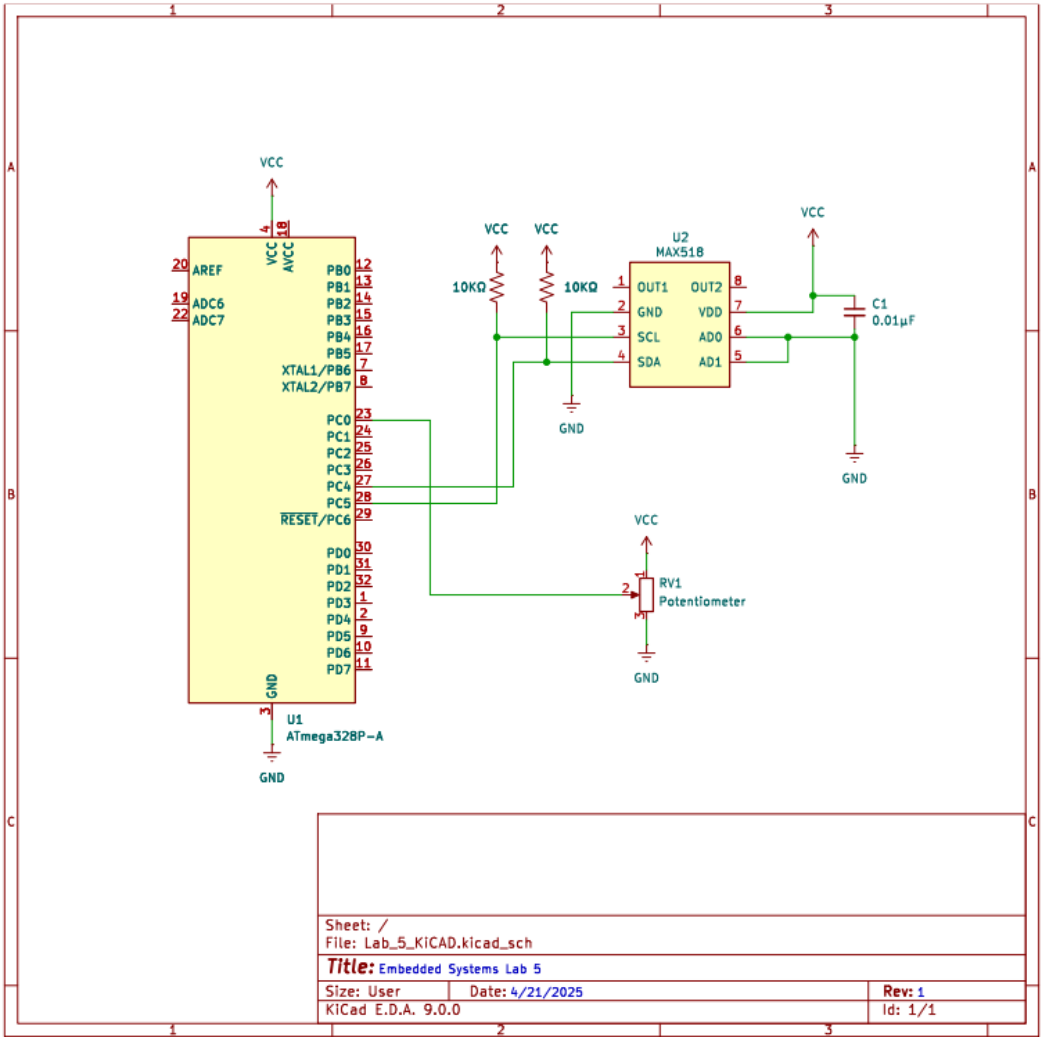
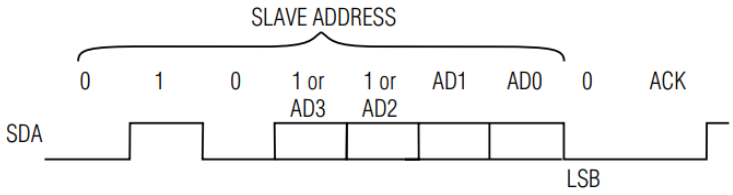


Figure 2: Electrical circuit schematic created using KiCAD

Compared to the previous lab, lab 5 did not have many hardware considerations. The main components of our circuit consisted of a B103 potentiometer and a MAX518 digital to analog converter. To read values from the potentiometer we connected the outer pins to 5V and ground and connected the middle pin (the signal wire) to A0 on the microcontroller. A0 corresponds to channel 0 of the built-in analog to digital converter included in the Arduino Uno.

To communicate with the MAX518 DAC we utilized I2C communication. This is a synchronous communication system that only requires 2 I/O pins for operation. One line is for data and the other is a clock line, when data is sent across I2C the line is pulled low, hence the need for 10KΩ pullup resistors on each line. Each I2C device has an I2C address that must be known for communication to occur. The MAX518 has a configurable address that can be changed by setting the AD0 and AD1 pins to ground or 5 volts. We opted to set the address pins to ground giving us an address of 0x2C. It is usual for I2C addresses to be only 7 bits. This

address can be found from the data sheet and is shown below in **Figure 3**. Finally, to reduce power supply noise a 0.01µF decoupling capacitor was connected to the MAX518.



Binary: 0101100 = Hex: 2C

Figure 3: MAX518 I2C Address

One frustrating issue encountered during this lab dealt with the potentiometer. The legs of the potentiometer found in the lab kit did not fit well into the bread board being used for circuit development. To counter this, a through hole PCB was utilized with soldering techniques to mount the potentiometer and ensure solid connections for more accurate voltage readings. This is shown in **Figure 4** below.

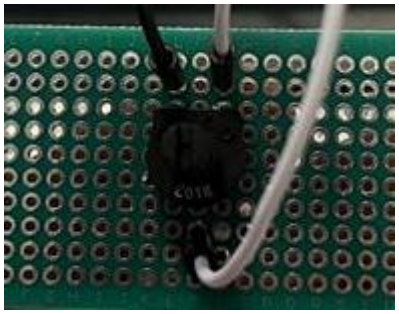


Figure 4: Soldered Potentiometer

3. Discussion

The goal of lab 5 was to develop a data logging interface that utilizes RS232 serial communication and I2C communication. The logging interface has three commands.

1. **G**: Return an analog voltage reading
2. **M**: Return **n** analog voltage readings with **dt** seconds between readings
3. **S**: Set analog voltage on channel **c** at voltage **n.nn**

The first step in implementing this functionality was to enable serial communication between the Data Visualizer in Atmel Studio and the microcontroller. To do this the Atmega328P datasheet was consulted and was found to have some very helpful C functions for USART communication. The functions are as follows:

USART0_Init()

USART0_Transmit()

USART0_Receive()

These functions are self-explanatory providing methods for configuring communication and sending and receiving bytes. Utilizing these three functions almost all required communication between the computer and microcontroller can be accomplished. Within the initialization function the desired communication Baud Rate is set to 9600. The Atmega328P operates at a oscillation frequency of 16Mhz. Then a calculation is performed to find the actual Baud Rate being used by the communication channel.

$$MYUBBR = \frac{FOSC}{BAUD - 1}$$

By passing MYUBBR to the initialization function serial communication was enabled and we were off and running. To send strings across USART the transmit function was used in a function that goes through each character in a string that is to be sent and individually sends said characters until the end of the string is reached.

To retrieve analog voltage readings, the on-board analog to digital converter of the Arduino Uno had to be utilized. Two functions were implemented to complete this task. The first of which was the initialization function. Within this function a reference voltage of 5V was set. This voltage is used for all voltage computations. Then a frequency of 125Khz was opted for as in the ATMEGA328P data sheet it is stated that the ADC operates best at a frequency in the range if 50 – 200Khz. The calculation for this can be seen below.

$$125\ Khz = \frac{16\ Mhz}{128}$$

The other analog to digital converter function was used to get voltage readings. First all channels were cleared, then channel 0 was enabled. The conversion is started by setting the ADSC bit in the ADCSRA (control and status) register. Once the ADSC bit is no longer set the conversion is complete and the value is read into a unsigned 16 bit integer. This integer is then converted into a float voltage with the following equation.

$$Float\ Voltage = \frac{Integer\ Voltage}{5.0 * 1023}$$

Peter Fleury's "i2cmaster" library was used to handle I2C communication. With the help of his library, all communication with the DAC was very straightforward requiring only a few LOC. For example, to write to DAC channel 0, the I2C start function was called, passing the I2C address and I2C_WRITE which was defined in the header file. Then the I2C write function was called to set the output channel of the DAC, then finally the voltage we want to set is sent as an integer. Then finally the I2C stop function is called.

To process and read serial commands a read line function was created that has a command timeout. The purpose of this function is to find the length of the command that is sent and then save the command to a buffer string that can be processed by separate command functions. It reads the command and looks for new line characters that would indicate a terminated string. If data was received the length of the buffer is saved. The timeout functionality helped with an issue that was encountered during the creation of this logger. Whenever the S command was run and enter pressed, nothing would occur until enter was pressed again. With the timeout no matter what after 100 milliseconds the command is processed.

In combination with this read function a command processing function was implemented. This is a simple switch and break function that only reads the first character of the buffer. If the character is equal to G you are to be sent to the G logic function and so on. If the character was not recognized an error was to be thrown.

The G command implementation was straightforward. Each time G was entered into the data visualizer a voltage is read and saved. Then with the help of *sprintf* a new buffer string is formatted to include the floating-point voltage that was calculated to 3 decimal point precision. Then the function for sending a string over USART is called. To test if we were reading correct voltage levels a multimeter was hooked up to the potentiometers signal wire and ground. The message exchange and voltage test can be seen below in **Figure 5**.

```
G
```

```
v = 2.076 V
```

Figure 5: G Command and Voltage Check

The next command that was implemented was the M command. The function created for processing the M command required a pointer to the parameter buffer string to be passed. By doing this the function could easily access the buffer command and parse out the important information. We did this by using *strchr*. This enables you to find the first instance of a specific character in a string. In this case that character was a comma. By doing this one can find where relevant data is located and determine the address in the buffer it is at. All data in this command is separated by commas. If the number of samples to be retrieved did not fall into the 2-20 range an error was thrown same as if the time between readings was not from 1-10 seconds an error was thrown.

To convert the characters to integers *atoi* was used. This converts the ASCII number associated with each character to an integer. Then similar logic was used as was for the G command when reading the voltage levels. Except a delay was included between readings by looping over a 1 second delay. This delay would loop the number of times that was specified in the command. Then the elapsed time is updated for printing. A sample message exchange can be seen below in **Figure 6**.

```
M,4,4
```

```
t=0 s, v=2.076 V
t=4 s, v=0.000 V
t=8 s, v=2.274 V
t=12 s, v=3.898 V
```

Figure 6: M Command

The final command to be implemented was the S command. This command would enable the user to set the voltage at the output of the DAC for any value in the range of 0-5 V. Once again when calling this command, a pointer to the parameter buffer string is passed. *strchr* is used again to parse the command and separate important information by finding where commas were located. Errors were thrown is the channel was not 0 or 1 and the voltage was not in the required range. Then a voltage conversion is done by converting the float number given by the user to an integer. The DAC requires an integer to be passed to it to set the voltage. Once this conversion is done, we convert the actual voltage being set by the DAC back to a float and then format that in the string that was sent. An example of this process can be seen below along with the sample message exchange, for a voltage input of 1.67. To ensure the outputs were being set correctly we tested once again with the multimeter. As you can see below our calculations were correct.

$$\frac{1.67 * 256}{5} = 85.504 \sim 86$$

Initial integer conversion (sent to DAC)

$$\frac{86 * 5.0}{256} = 1.6796875 \sim 1.68$$

Convert Again to Display Actual Voltage at DAC Output

The message exchange for the S command can be seen below in **Figure 7**.

```
S,1,1.67
```

```
DAC channel 1 set to 1.68 V (86d)
```

Figure 7: S Command

4. Conclusion

Lab 5 was like Moses: parting the Red C and saving us from the woe of AVR assembly. Although we were not using Holy C in this lab, we were given the opportunity to leave AVR assembly to use the father of all modern programming languages. Using the C programming language, communication methods such as RS232 and I2C were picked up with ease. The libraries used communication methods simplify the connection between microcontrollers and peripherals. As we wrap up the semester of labs, the knowledge gained will be on full display in our final project.

5. Appendix A: Source Code

```
#ifndef F_CPU
#define F_CPU 16000000UL                //Define the F_CLK of the micro controller
#endif

#include <avr/io.h>                             //Included for AVR specific registers (USART & ADC)
#include <stdio.h>                              //Included for string formatting (sprintf)
#include <stdlib.h>                             //Included for string to number conversion (string to
                                                integer & string to float)
#include <string.h>                             //Included to analyze strings sent over USART
#include <util/delay.h>                 //Included for delays
#include <i2cmaster.h>                          //Include the i2c library we use for communication with
                                                MAX518 DAC

#define FOSC 16000000                           //Define oscillator frequency for USART initialization
                                                (datasheet)
#define BAUD 9600                               //Define BAUD Rate (datasheet)
#define MYUBRR FOSC/16/BAUD-1           //Defined for USART initialization (datasheet)

#define MAX518_ADDR 0x2C                //Define the i2c address of the MAX518 DAC
#define COMMAND_TIMEOUT_MS 100                  //Define the length of time until a timeout occurs

//////////////////////////////////////////////////////////
// USART Functions
//////////////////////////////////////////////////////////

void USART0_Init(unsigned int ubrr)
{
        //USART initialization function from the ATMEGA328P datasheet
        UBRR0H = (unsigned char)(ubrr>>8);      //Set the baud rate
        UBRR0L = (unsigned char)ubrr;
        UCSR0B = (1<<RXEN0)|(1<<TXEN0);  //Enable transmitter and receiver
        UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);//Format for 8 data bits no parity and one stop bit
}
//USART transmission function from the ATMEGA328P datasheet
void USART0_Transmit(unsigned char data)
{
        while (!(UCSR0A & (1<<UDRE0)));//Wait for transmit buffer to be empty by reading
                                                the flag
        UDR0 = data;                          //put data into buffer and send
}

//USART receive function from the ATMEGA328P datasheet
unsigned char USART0_Receive(void)
{

        while (!(UCSR0A & (1<<RXC0)));          //Read the complete receive flag
        return UDR0;                                    //Get data from the buffer

}

//USART send string function that utilizes the transmit function
void USART0_SendString(const char* str)
{
        while (*str) //Loop through each character in the string until null-terminator is
                        reached
        {
                USART0_Transmit(*str++); //Transmit each character

        }
}

//////////////////////////////////////////////////////////
// ADC Functions
//////////////////////////////////////////////////////////

//Function that initializes the ADC
```

```
void ADC_init()
{
        ADMUX = (1<<REFS0); //Reference voltage of 5V internal
        ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);    //best ADC accuracy is
                                                                 reached with a frequency
                                                                        from 50Khz - 200kHz (stated
                                                                        in data sheet)
        //We opt for a prescalar of 128 to get a frequency of 125Khz
        ADCSRA |= (1 << ADEN);  //Enable ADC

        _delay_ms(1);   //Delay for ADC stabilization
}

//Function that reads from the ADC
float read_ADC()
{

        ADMUX &= 0xF0;
                                //Clear channel selection bits in MUX
        ADMUX |= (0 & 0x07);
                        //Enable channel 0
        ADCSRA |= (1 << ADSC);
                        //Start conversion by setting the ADSC bit
        while (ADCSRA & (1 << ADSC));
                //wait for conversion to complete
        uint16_t adc_Value = ADC;
                        //Read the 10 bit ADC value
        float voltage = adc_Value * (5.0 / 1023.0);
        //Convert the ADC value to a float with equation from the data sheet
        _delay_ms(1);
                                //Allow time for ADC to settle
        return voltage;
                                //Return the voltage value as a float
}

////////////////////////////////////////////////////////
// DAC Functions
////////////////////////////////////////////////////////

//Function that sets DAC channel 0
void set_DAC_channel0(int value)
{
        i2c_start((MAX518_ADDR << 1) | I2C_WRITE);
        //Start the i2c communication in write mode using the i2c address
        i2c_write(0x00);
                        //Write to output channel 0
        i2c_write(value);
                                //Write the value to be set at the DAC output
        i2c_stop();
                                        //Stop i2c communication
}

//Function that sets DAC channel 1
void set_DAC_channel1(int value)
{
        i2c_start((MAX518_ADDR << 1) | I2C_WRITE);
        //Start the i2c communication in write mode using the i2c address
        i2c_write(0x01);
                        //Write to output channel 1
        i2c_write(value);
                                //Write the value to be set at the DAC output
        i2c_stop();
                                        //Stop i2c communication
}

//Function that converts the DAC value to an integer from a float
int DAC_voltage_conversion(float voltage)
        //A value from 0 - 255 is needed to set the DAC
```

```
{
                                                //Uses a reference voltage of 5
        int return_voltage = voltage * 256.0 / 5.0 + 0.5;                                       //Add
0.5 for correct rounding float to int

        return return_voltage;
                        //Return integer voltage value
}

////////////////////////////////////////////////////////
// Command Functions
////////////////////////////////////////////////////////

//Reading from the ADC and G command
void G_command()
{
        float voltage = read_ADC();
                        //Read a voltage measurement through ADC channel 0

        char vstr[16];
                                //Initialize a buffer voltage string to be sent across serial
communication
        sprintf(vstr, "%.3f", voltage);
                //Format the float into the string
        USART0_SendString("v = ");
                        //Sent over USART in this format
        USART0_SendString(vstr);
                //Send voltage string
        USART0_SendString(" V\r\n");
                //End with a carriage and new line character
}

//M command that takes multiple readings from the ADC at set intervals and time amounts
void M_command(char* params)
{

                                                //Find the comma separators

                                                //strchr looks for instances of specific characters in a
string, in this case it is looking for the commas needed for the command
        char* first_comma = strchr(params, ',');
        //Find the address of the first comma
        char* second_comma;
                                //Create a pointer to the second comma character
        if (first_comma != NULL)
                //If the first comma was detected we can look for the second
        {
                second_comma = strchr(first_comma + 1, ',');
        //Search for second comma and save address in pointer
        }
        else
        {
                second_comma = NULL;
                        //No first comma found set to NULL
        }

        if (!first_comma || !second_comma)
                //If two commas are not detected the formatting is incorrect and send a message saying so
        {
                USART0_SendString("Error: Incorrect M format\r\n");
                return;
        }

        int num_readings = atoi(first_comma + 1);
        //Convert the ASCII label for the number of readings (starting address after the first comma) to
an integer
        int delay_seconds = atoi(second_comma + 1);
        //Convert the ASCII label for the delay between readings (starting address after the second comma)
to an integer
```

```
        if (num_readings < 2 || num_readings > 20)
        //Ensure number of readings is between 2 and 20
        {
                USART0_SendString("Error: Number of readings must be 2-20\r\n");
                return;
        }

        if (delay_seconds < 1 || delay_seconds > 10)
        //Ensure that the delay time is between 1 and 10 seconds
        {
                USART0_SendString("Error: Delay must be 1-10 seconds\r\n");
                return;
        }

        int elapsed_time = 0;
                        //track the time elapsed for USART communication
        char time_str[16];
                                //empty time string
        char vstr[16];
                                //empty voltage string

        for (int i = 0; i < num_readings; i++)
        //Take a measurement for each reading
        {
                float voltage = read_ADC();
                        //Take reading at current time

                sprintf(time_str, "t=%d s, v=", elapsed_time);
        //Format string with a float using sprintf to include elapsed time

                USART0_SendString(time_str);
                //Send time string over USART

                sprintf(vstr, "%.3f V", voltage);
                //Format string with a float using sprintf to include voltage reading
                USART0_SendString(vstr);
                //Send voltage string
                USART0_SendString("\r\n");
                        //Send carriage and new line character

                if (i < num_readings - 1)
                        //For all measurements except the last one
                {
                        for (int j = 0; j < delay_seconds; j++)
        //Delay between readings
                        {
                                _delay_ms(1000);
                        //Delay 1-second intervals
                        }
                        elapsed_time += delay_seconds;
                //Update time elapsed in seconds for display
                }
        }
}

//S command and DAC functionality
void S_command(char* params)
{

                                                // Find the comma separators

                                                //strchr looks for first instance of specific characters
in a string, in this case it is looking for the commas needed for the command
        char* first_comma = strchr(params, ',');
        char* second_comma;
                                //Create a pointer to the second comma character
        if (first_comma != NULL)
                //If the first comma was detected we can look for the second
```

```
        {
                second_comma = strchr(first_comma + 1, ',');
        //Search for second comma and save address in pointer
        }
        else
        {
                second_comma = NULL;
                        //No first comma found set to NULL
        }
        if (!first_comma || !second_comma)
                //If there are not two commas send an error
        {
                USART0_SendString("Error: Incorrect S format\r\n");
                return;
        }

        int channel = atoi(first_comma + 1);
        //Convert the ASCII label for the channel (address after the first comma) to an integer
        float input_voltage = atof(second_comma + 1);
        //Convert the ASCII label for the voltage input (starting address after the second comm) to a
float

        if (channel != 0 && channel != 1)
                //Ensure valid channel was input
        {
                USART0_SendString("Error: Channel must be 0 or 1\r\n");
                return;
        }

        if (input_voltage < 0.0 || input_voltage > 5.0)
        //Ensure valid voltage was input
        {
                USART0_SendString("Error: Voltage must be 0.0-5.0\r\n");
                return;
        }

        int dac_value = DAC_voltage_conversion(input_voltage);
        //Convert the input voltage into a value that the DAC can read (0-255)

        float actual_voltage = dac_value * (5.0 / 256.0);
        //Calculate the actual voltage that will be present on DAC output after float to integer rounding

        if (channel == 0)
                                //If channel 0 was specified
        {
                set_DAC_channel0(dac_value);
                //Set channel 0 output voltage
                USART0_SendString("DAC channel 0 set to ");
        //Initial display string for channel 0
        }
        else
                                        //Channel 1 was specified
        {
                set_DAC_channel1(dac_value);
                //Set channel 1 output voltage
                USART0_SendString("DAC channel 1 set to ");
        //Initial display string for channel 1
        }

        char vstr[32];
                                //Initialize buffer to put voltage information in
        sprintf(vstr, "%.2f V (%dd)\r\n", actual_voltage, dac_value);                   //Use sprintf to
format display and round the voltage, while showing the integer vlaue sent to the DAC
        USART0_SendString(vstr);
                //Send string over USART
}

//By reading the command character sends you to that command logic to perform action
//Takes the character that represents a specific command and the pointer to the command parameters
```

```
//Returns nothing
void process_command(char command, char* params)
{
        switch (command)
        {

                case 'G':
                                        //If Case G go to the G command
                G_command();
                break;
                case 'S':
                                        //If Case S go to the S command
                S_command(params);
                break;
                case 'M':
                M_command(params);
                                //If Case M go to the M command
                break;
                default:
                USART0_SendString("Unknown command\r\n");
        //If none of the commands let the user know input is unknown
                break;
        }
}


//Function that reads a line of text from the USART until newline character or timeout has been reached
//Takes a pointer to the buffer string and the max number of characters to be read
//Returns the length of the string (number of characters that have been read)
int read_line(char* buffer, int max_length)
{
        int idx = 0;
                                //Buffer index (counts the number of characters read in serial
communication)
        unsigned long start_time = 0;
                //Track time that has elapsed for a timeout

        while (idx < max_length - 1)
                //Continue reading until buffer is full (excluding null terminating character)
        {
                if (UCSR0A & (1<<RXC0))
                {
                                                //Check if character is available to be received (UCSR0A
-> USART Control and Status Register A)
                                                        //RXC0 -> specific bit position in this register
                        char c = UDR0;
                                //Read the character from the serial communication

                        if (c == '\r' || c == '\n')
                        {
                                                //If the character is the carriage character or the new
line character
                                buffer[idx] = '\0';
                                //Null terminate the string
                                return idx;
                                        //Return the number of characters
                        }
                        buffer[idx++] = c;
                                //Store characters in the buffer and increment the index
                        start_time = 0;
                                // Reset timeout counter when character is received
                }
                else
                {
                        _delay_ms(1);
                                //No character to read, delay
                        if (++start_time > COMMAND_TIMEOUT_MS)
        //If we have a timeout
                        {
```

```
                                buffer[idx] = '\0';
                                //Null-terminate the string

                                if (idx > 0)
                                {
                                        return idx;
                                        //Return character count if data was received
                                }
                                else
                                {
                                        return -1;
                                        //Return -1 for timeout with no data
                                }
                        }
                }
        }
        buffer[max_length - 1] = '\0';
                //Reaching here means buffer is full

                                                //Null terminate the buffer string
        return max_length - 1;
                        //Return the number of characters that have been read
}

////////////////////////////////////////////////////////
// Main Program
////////////////////////////////////////////////////////

void setup() {
        USART0_Init(MYUBRR);
                        //Initialize the USART RS232 Serial Communication
        ADC_init();
                                        //Initialize the analog to digital conversion
        i2c_init();
                                        //Initialize I2C communication (from library)
}

int main(void) {
        setup();
                                //Call setup function

        char command_buffer[32];
                //Initialize a string that is 32 characters in length to read commands

        while (1) {
                int bytes_read = read_line(command_buffer, 32);
        //Loop that reads from the Serial terminal continuously

                if (bytes_read > 0) {
                        //If data is read, process it as a command
                        char command = command_buffer[0];
                //First character of buffer is command type
                        char* params;
                                //Initialize a pointer to the parameters

                        if (bytes_read > 1) {
                                params = command_buffer + 1;
                //If there are parameters, point to them
                                } else {
                                params = "";
                                //Otherwise, use an empty string
                        }

                        process_command(command, params);
                //Call the command processing function
                }
        }

        return 0;}
```

6. Appendix B: References

Beichel, Reinard. *Embedded Systems: Lab 5 Considerations. The University of Iowa, 2025.* <*https://uiowa.instructure.com/courses/248357/files/30236693?module_item_id=822649*>

Beichel, Reinard. *Embedded Systems: C Programming. The University of Iowa, 2025.* <*https://uiowa.instructure.com/courses/248357/files/30138332?module_item_id=8205647* >

Beichel, Reinard. *Embedded Systems: Serial Communication. The University of Iowa, 2025.* <*https://uiowa.instructure.com/courses/248357/files/30164502?module_item_id=8213613* >

Beichel, Reinard. *Embedded Systems: Serial Interconnect Buses. The University of Iowa, 2025.*<*https://uiowa.instructure.com/courses/248357/files/30236666?module_item_id=8223421* >

Ramsay, Chris. *I2cmaster. 2014* < *https://github.com/chrisramsay/arduino-projects/blob/master/libraries/I2Cmaster/i2cmaster.h* >

MAXIM. *MAX517/MAX518/Max519 2-wire serial 8-bit dacs. 2002.* <https://www.analog.com/media/en/technical-documentation/data-sheets/max517-max519.pdf >

Pighixxx. *The Definitive Arduino Uno Pinout Diagram. May 5, 2013.* <https://uiowa.instructure.com/courses/248357/files/29320694?module_item_id=8042318>