University of British Columbia

Electrical and Computer Engineering

ELEC 291: Electrical Engineering Design Studio I 2023

Instructor's Name: Dr. Jesús Calviño-Fraga

Section: L2B

PROJECT 2 - Magnetic Field Controlled Robot

Date: April 12, 2023

Group number: B6

Student number	Student Name	% Points	Signature
48395651	Andrey Abushakhmanov	85	
46590261	Bevio Chen	85	
90300815	Pratham Goel	130	
88191713	Victor Lui	115	
63156905	Tomas Bang	85	

TABLE OF CONTENTS

1. Introduction

The purpose of this project was to design, build, program, and test a remote-controlled robot using a varying magnetic field. The robot was controlled through a microcontroller system and powered by batteries. The project was divided into two main modes of operation: track mode and command mode. In track mode, the robot was programmed to maintain a constant distance from the magnetic transmitter, while in command mode, it received signals through the magnetic field and executed corresponding actions. This report presents the design process, programming details, and testing results of the robot, as well as an evaluation of its performance and potential for further development. Overall, this project serves as an example of how magnetic field sensing can be used to create innovative autonomous systems with practical applications in a variety of fields. Below are the block diagrams for the software and hardware components of the transmitter and receiver.

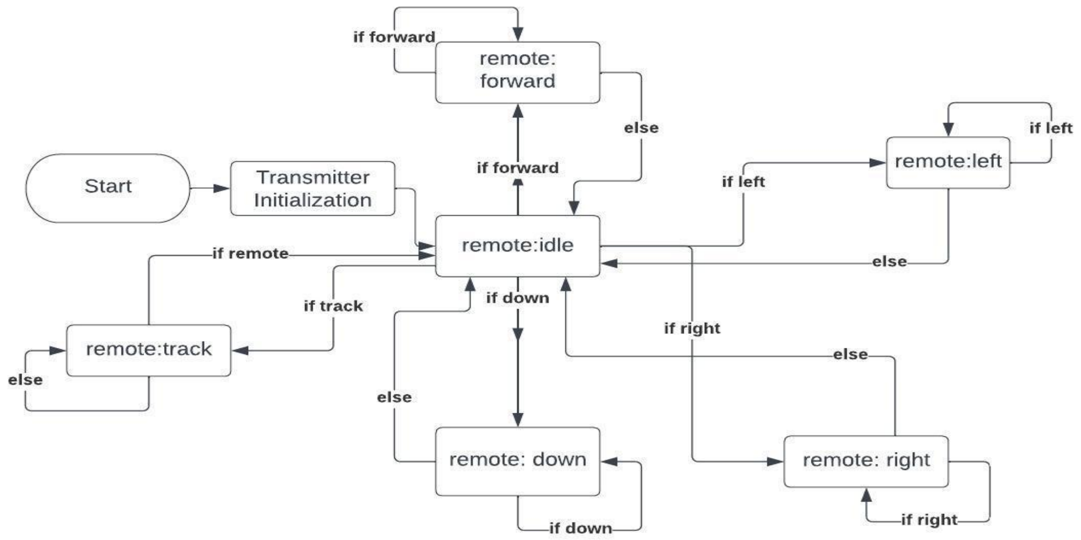

Figure 1: Software Block Diagram for Transmitter

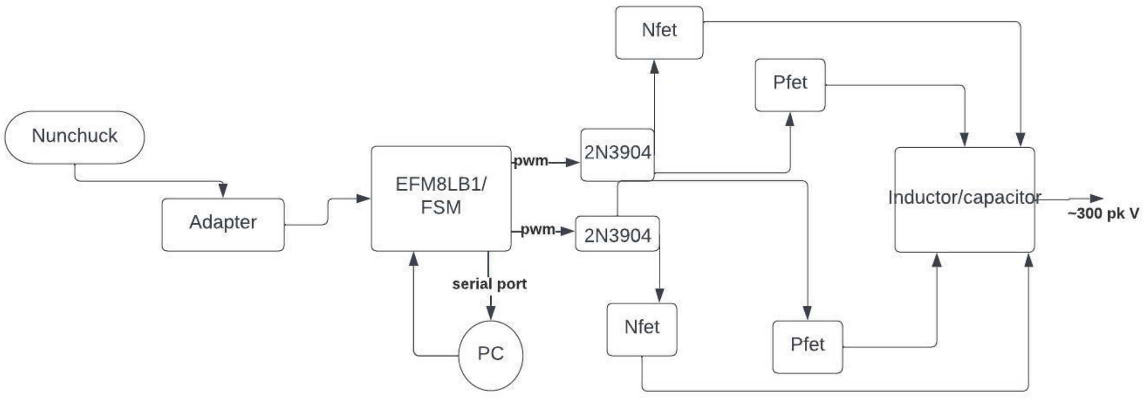

Figure 2: Hardware Block Diagram for Transmitter

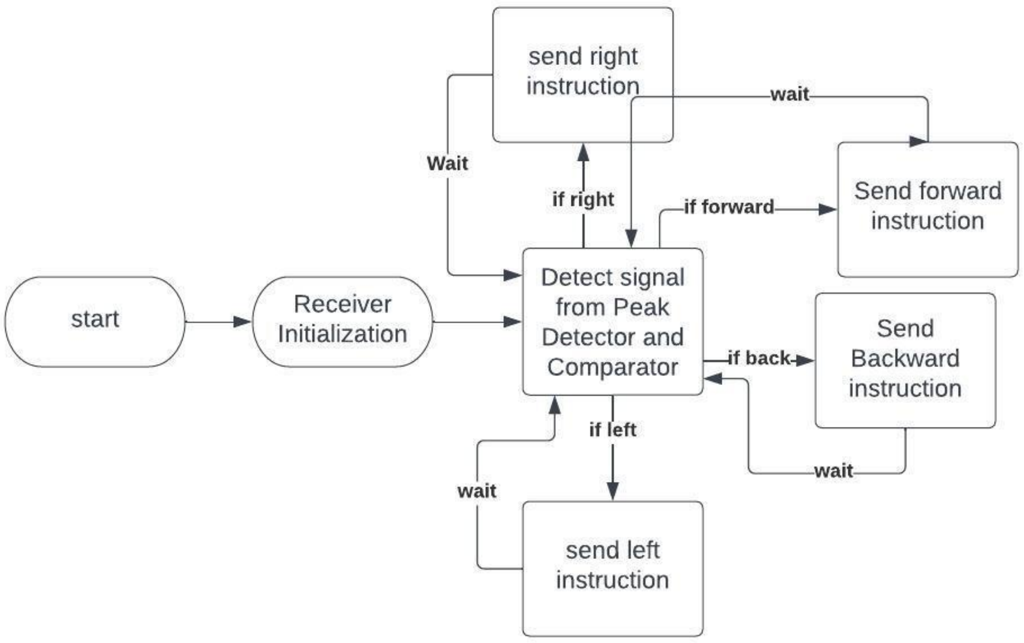

Figure 3: Software Block Diagram for Receiver

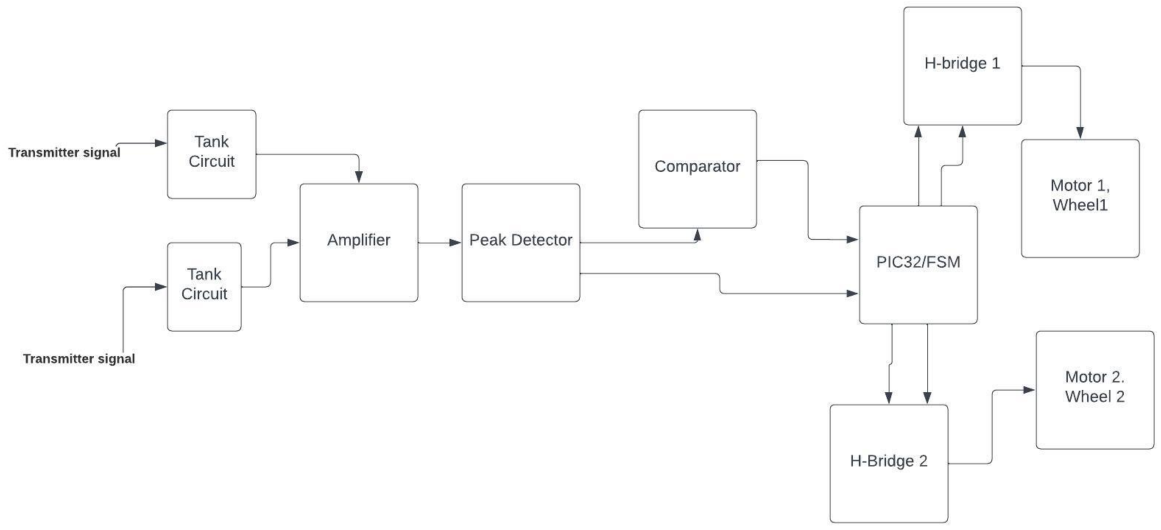

Figure 4: Hardware Block Diagram for Receiver

2.0 Investigation

To implement the design, several approaches were considered. However, before choosing a specific approach, a thorough investigation needs to be conducted to identify the relevant parameters and materials necessary to develop a better understanding of the project requirements. This section outlines the methodology used to generate and investigate ideas, design the experiment, and collect and synthesize data.

2.1 Idea Generation

Our team followed a structured approach for the project by dividing it into smaller, more manageable parts based on the list of requirements that needed to be fulfilled. This approach made it easier to generate solutions and complete the project efficiently. We brainstormed various ideas to solve the challenges, and through discussion and filtering, we selected the most suitable design for each element.

We successfully tested each part separately (transmitter and receiver) before combining them to create a complete magnetic field controlled robot. This method allowed us to achieve all the requirements while optimizing the overall performance of the system. By implementing a modular design with independent components, we were able to improve workflow and collaboration within the team, resulting in a more efficient and effective project.

2.2 Investigation Design

Prior to putting the magnetic field controlled robot into action, a thorough process of identifying the required components is carried out to ensure that all necessary parts are identified and that the appropriate part numbers are selected in order to fulfill the project's requirements. If any parts are found to be missing, a functional replacement that closely matches the original component’s capabilities will be selected, taking into consideration any potential variations in specifications or performance characteristics. Moreover, datasheets and spec sheets for all components are gathered from trustworthy sources before the circuit design is implemented on the breadboard. We used a variety of lab tools for data gathering and data testing. Including but not limited to multimeter, function generator, oscilloscope, and power supply.

2.3 Data Collection and Synthesis

To ensure the accuracy and reliability of the magnetic field controller robot, various methods were employed for data collection and synthesis. The distance between the receiver and transmitter was measured using a ruler, and the robot's movements were observed to ensure it was functioning as expected. Calibration was conducted to validate the accuracy of the magnetic sensors, and multiple measurements of the same magnetic field were taken to ensure consistency.

Other data collection methods included comparison with a known magnetic field, statistical analysis, simulation testing, field testing, peer review, software validation, and documentation review. These methods ensured that the data obtained was reliable and accurate. During the testing phase, readings from the magnetic sensors were observed, and data from different sensors was compared to check for consistency and identify discrepancies. Simulation testing was conducted to validate the magnetic field controller and predict expected results.

In addition, data was collected for a potential design using tools such as a multimeter, function generator, oscilloscope, and power supply. During the testing phase, readings from the oscilloscope and multimeter, values on PuTTy, and physical movement of the robot were observed. The tank circuits were tested by observing the DC signal displayed on an oscilloscope. The data obtained was synthesized to ensure that the magnetic field controller robot was functioning accurately and reliably. Experts in the field were consulted for peer review, and software validation and documentation review were conducted to ensure consistency with expected results and specifications.

2.4 Analysis of Results

The performance of the magnetic field controller robot was analyzed through several methods. Initially, the distance between the receiver and transmitter was measured using a ruler, and the robot's movement was observed through simple commands such as right, left, forward, and backward. The team members agreed that the robot performed as expected.

In addition to these basic validation methods, more advanced techniques were also used to ensure accuracy and consistency. Repeat measurements were taken to ensure that the results were consistent and reliable. Sensor calibration was also conducted using another inductor connected to an LED to ensure the magnetic sensors were working correctly and providing accurate readings. These additional validation techniques ensured that the data obtained was accurate and reliable.

3. Design

3.1 Use of Process

While the parameters of this project were clear, that did not mean we were constricted to one method of solution. In fact, we had the choice of 6 families of microcontrollers to base our project off of, opening the possibilities for problems unique to our design in aspect to software and hardware. As such, whenever we encountered complex and open-ended roadblocks, our go-to method would be to trace signals via oscilloscope from the start to the end of the datapath on our circuit. This would immediately identify whether the problem was due to wiring issues or code.

Reference to datasheets for the components of our circuit - op amps, comparators, diodes, BJTs, mosfets, microcontrollers - were paramount for designing the layout of the circuit and how we would distribute space on the breadboard for functions of the circuit. In particular, this project required our circuit components to be compact with each other, so having access to all datasheets made it possible to organize wiring as best as we can, which leads to an easier developmental process when we encounter bugs.

For software, we found that using the technique of setting qualitative flags in sections of the code would help us identify where our problem is. For instance, if we wanted to determine if a bug was part of a nested loop, we would plant a print function in our suspected buggy loop. If the print statement appeared on our hardware, then we could assume that loop is fine, and test the next loop. If the print statement did not appear, then that loop would be identified as the problem, to which only then would we start a line-by-line analysis. This method proved very helpful, as our code is tall, and narrowing it down with this method not only saved more time, but was more effective than, per se, simply scanning through the code multiple times and expecting to find the error.

3.2 Needs and Constraints

Our group defined the specifications of the project through class-wide consultation with Professors Jesus, Teaching Assistants, and peers from other groups. This was conducted in a variety of methods, from in-person consults, to emails, to using the online class forum Piazza.

In remote mode, the transmitter must send varying signals that respond to left, right, forward, back, and idle. The receiver needs to recognize this signal and move the robot, using wheels connected to a motor, in accordance with the received signal.

In track mode, the transmitter must send a signal of constant frequency, independent of distance, and the receiver must recognize this signal and move the robot again accordingly to keep this signal constant.

The constraint of this project required us to use two microcontrollers of different families for the transmitter and receiver, and using batteries to power. With this in mind, the needs of this project required us to take into account the feasibility of implementing extra features, as well as the time frame of the project.

3.3 Problem Specification

The transmitter must send varying signals using data modulation for the receiver to convert into logic, which the software then interprets. However, it is evident that pulsing signals from the transmitter must be great in difference for there to be a detectable difference.

As such, the method of data modulation must be one that employs a constant frequency among all instructions, while being able to instruct the receiver without interferences in the signal, which would arise from aspects such as capacitor discharge and saturated voltages. Thus, we need a clean method of changing the signals in the transmitter quickly. We found that instead of using buttons, which are slow and are limited to a single input, we could use a Wii nunchuck, acting as a joystick. Not only is this specification capable of generating a faster response, it also allows constant input from the transmitter.

In addition, our group defined another specification that allowed for user interface. Using an LCD in our circuit, we would display the current mode, as well as the current instruction . Not only does this make our final production have a cleaner demonstration with this user interface, but it also serves as a useful tool in the debugging process, using techniques listed in earlier sections.

3.4 Solution Generation

The solution process begins with designing a precise and efficient magnetic field generator, which acts as the transmitter. The generator creates a controlled magnetic field pattern that can be altered according to the desired robot movements. These patterns are then wirelessly transmitted through the environment and captured by the receiver, which is integrated into the robot's navigation system. For the sake of convenience, we choose to use the EFM8LB12 (8051 family) microcontroller for the transmitter and PIC32MX130 (MIPS family) microcontroller for the receiver due to our team's familiarity with them in lab 6.

Upon receiving the magnetic field patterns, the receiver's signal processing module decodes the information, converting it into corresponding robotic motion commands. Advanced algorithms and calibration techniques are employed to ensure that the robot accurately interprets and responds to the transmitted signals, minimizing any potential signal distortion or interference from external magnetic sources. Throughout the solution generation process, a strong emphasis is placed on optimizing the system's overall efficiency, responsiveness, and reliability, enabling the robot to seamlessly perform instructions under different operating conditions (mode).

3.5 Solution Evaluation

The evaluation of the magnetic field controlled robot solution is crucial to ensure its effectiveness, reliability, and overall performance. A comprehensive assessment is conducted through a combination of simulations studies and laboratory tests. In the simulation phase, computer simulations (putty) are employed to analyze the system's behaviour under various conditions, enabling the identification of potential design optimizations and the prediction of the robot's performance. During the laboratory tests, the robot's responsiveness, accuracy, and ability to follow the joystick commands are rigorously examined in controlled environments, allowing for fine-tuning of the algorithms and hardware components.

The solution evaluation process for the magnetic field controlled robot is also heavily focused on ensuring that the developed solution meets the project's specific requirements and objectives. These requirements are established during the initial planning phase and may encompass aspects such as system response time, precision of movements, effective communication range, and compatibility with various operating environments. Through trial and error, the final design is chosen and implemented which meets all the requirements of the project.

3.6 Detailed Design

3.6.1 Transmitter Software

Using an EF8M microcontroller, we used Two output pins to generate output signals in the form of square waves with varying frequencies for every command. This is done by initializing the EFM8's Timer 2 to a reload rate that corresponds to the frequency we want to exhibit. At its interrupt service routine, the bits of the outputs are flipped to generate a square wave. As seen in Appendix I, the pins must be configured for a push-pull output in the initializations in order for a favorable square wave to be generated. In order to implement a data modulation technique, we saw to keep the amplitude and widths of the pulses constant, but modified the time at which they are off. In the source code, this is done in the Timer 2 interrupt service routine. We used counter loops to control the frequency of on-off signals, which would flip the bits after a certain amount of times the interrupt occurs, thereby outputting different square waves. The output signal is fed to H-Bridge and then sent to the inductor to send the signal to the robot car.

3.6.1.2 Transmitter Hardware

The Hardware of the transmitter consists of the EFM8 microcontroller, the nunchuck and its adapter, an H-bridge, and an LC circuit that transmits the wave. Inputs from the nunchuck are fed into the adapter, which become analogue inputs into two pins of the microcontroller. The system then determines which modulated pulse to send, according to the direction which we assign the pulses to. This output is fed into the H-Bridge, and the resulting voltage between the inductor and capacitor is around 300 volts.

3.6.2 Robot Car

3.6.2.1 Hardware

We opted to utilize 1.0 µF capacitors in our LC (tank) circuits due to their affordability and easy accessibility. In order to guarantee that the signal obtained by the inductors matched the produced signal, we implemented a resonance frequency formula to achieve a gain of approximately 34. Furthermore, we required two op-amp amplifiers to amplify the signal and enable our comparator to register the signals transmitted by the transmitter, as the single amplifier alone did not allow the signal to reach the threshold values.

We have two breadboards to attach the components on the robot car. the signal from tanks is sent through the amplifiers to amplify the signals and then fed into the comparator. The comparator outputs square waves which are then sent to the PIC32 microcontroller to analyze. PIC32 microcontroller decodes the signal and sends it to the optocoupler which controls the motors.

3.6.2.2 Software

i. Track mode

For the track mode, it takes analog inputs at pin 4 and pin 5 of PIC32 and then analyzes the signal to keep the values of V1 (signal from left tank) and V2 (signal from right tank) constant. If the car is sent to track mode and the car is far away from the transmitter, the PIC32 controller commands the motors to go closer towards the car. PIC32 also handles the directions by trying to turn the car left or as per requirments. Additionally, the threshold values of V1 and V2 are also set to keep at about 10 cm away from the transmitter so, if the robot car is very near to the transmitter, it is commanded to go backwards.

ii. Remote Mode

The software for Remote mode is pretty straight forward, which takes input at digit input pin 14 from the comparator and reads the frequency using the GET_PERIOD() function.

After the signal is received from the comparator, the PIC32 decodes the signal by checking what is the value of frequency and sent the command with respect to this.

As seen in the code in Appendix II, Value f is defined for frequency (Hz) and after testing the transmitter code, the values of different commands were noted and the modified to receiver code.

3.7 Solution Assessment

To evaluate the complete system, a series of tests were carried out, targeting each relevant part of the design. The magnetic field generator, for instance, was tested for its consistency in creating desired magnetic field patterns and its resilience against external interference. The joystick interface was assessed for its responsiveness, ease of use, and the accuracy of command translation. The robot's navigation system and signal processing module were evaluated for their ability to accurately decode and execute received commands, even in challenging environments.

Some of the design strrengths identified included the system's overall responsiveness, the intuitive nature of the joystick interface, and the robustness of the robot's navigation system. Weaknesses, on the other hand, included potential susceptibility to strong external magnetic interference and the need for further optimization of the signal processing algorithms to enhance the precision of robot movements.

By thoroughly testing each relevant part of the design and analyzing the resulting data, our team was able to assess the system's performance against the established requirements, needs, and constraints, while also identifying areas for improvement. This rigorous evaluation process not only ensured the successful development of the magnetic field controlled robot solution but also provided a strong foundation for future iterations and enhancements.

4. Live-Long Learning

During the magnetic field controlled robot project, our team utilized various technical concepts from our co-prerequisite courses, including and not limited to ELEC 211 (Engineering Electromagnetics) and CPSC 259 (Data Structures and Algorithms for Electrical Engineers). Specifically, CPSC 259 is particularly useful in developing the code and structures in C programming language as well as ELEC 211 was incredibly useful for our team to understand how magnetic fields work.

Moreover, we learned how to document the project, including circuit diagrams, code, and mechanical design, to facilitate future maintenance and improvements. We now understand the importance of isolating the MOSFETs from the microcontroller system using opto-isolators to improve the reliability of the robot and the principles of MOSFET drivers and how to use them to control motors. Lastly, we gained practical experience in designing and building mechanical components of a robot using specific parts and tools.

In addition to technical skills, our team also learned the importance of working effectively in a team and communicating well. We found that open and frequent communication and collaboration were essential to ensure that everyone understood the project goals and tasks and to ensure that we were all working towards the same objectives. We delegated tasks based on individual strengths and availability, which allowed us to utilize each team member’s strengths and maximize productivity.

Overall, this project provided valuable experience with embedded systems, circuit analysis, hardware design, construction, and teamwork, which will not only be beneficial for our future studies as electrical engineering students and our future careers but also for our personal growth as effective communicators.

5. Conclusion

The design for the magnetic field-controlled robot project involved an aluminum chassis with two motors, batteries as the power source for wheels and circuits and inductors to receive a magnetic field from the transmitter. The receiver and transmitter are built from different families of microcontrollers: EMF8 and PIC32 respectively. Robot has two modes of operation that can be seen on the LCD mounted on the transmitter circuit. At first, the track mode robot keeps the desired distance between the transmitter and itself through small and accurate changes in position. In the second command, the mode robot changes its position by giving instructions from the user using the Wii Nunchuck controller mounted on the transmitter circuit as well. The robot is able to comply with given instructions within approximately 80 cm away from the transmitter. After that it loses the magnetic field that the transmitter creates. Overall, the robot performs all instructions relatively fast and accurately which means that desired and efficient circuits and code were created.

During the project, the main challenges were designing and building the mechanical components of the robot because of the lack of space and breadboard and robot chassis itself. Moreover, developing the code to control the robot and read the electromagnetic signal from the transmitter was very challenging as well as electromagnetic fields from robots of other teams and machines in class were disturbing our measurements and calculations. The project required careful planning and testing, as small errors in the code or mechanical components caused significant issues. The project took approximately 100 hours of work as a team to complete, including planning, design, construction, and testing.

References

1. Abbott, J. J., Diller, E., & Petruska, A. J. (2020). Magnetic methods in robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, *3*, 57-90.
2. Nam, J., Lee, W., Jung, E., & Jang, G. (2017). Magnetic navigation system utilizing a closed magnetic circuit to maximize magnetic field and a mapping method to precisely control magnetic field in real time. *IEEE Transactions on Industrial Electronics*, *65*(7), 5673-5681.

Bibliography

1. J.Calvino-Fraga. ELEC 291. Project, Topic: “ELEC291_Project_2_Magnetic_Field_Controlled_Robot.” Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC
2. J.Calvino-Fraga. ELEC 291. Class Lecture, Topic: “Lecture slides for March 17th. (ELEC291_Project_2_2023.pdf).” Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC
3. J.Calvino-Fraga. ELEC 291. Project, Topic: “Robot_assembly.pdf.” Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC
4. J.Calvino-Fraga. ELEC 291. Microcontroller, Topic: “The PIC32 Microcontroller System.pdf.” Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC
5. J.Calvino-Fraga. ELEC 291. Microcontroller, Topic: “The EFM8 Microcontroller System.pdf..” Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC

Appendix I: Transmitter Source Code

```
//initializations
#include <EFM8LB1.h>
#include "globals.h"
#include <stdio.h>

char _c51_external_startup (void)
{
	// Disable Watchdog with key sequence
	SFRPAGE = 0x00;
	WDTCN = 0xDE; //First key
	WDTCN = 0xAD; //Second key

	VDM0CN |= 0x80;
	RSTSRC = 0x02;

	#if (SYSCLK == 48000000L)
		SFRPAGE = 0x10;
		PFE0CN  = 0x10; // SYSCLK < 50 MHz.
		SFRPAGE = 0x00;
	#elif (SYSCLK == 72000000L)
		SFRPAGE = 0x10;
		PFE0CN  = 0x20; // SYSCLK < 75 MHz.
		SFRPAGE = 0x00;
	#endif

	#if (SYSCLK == 12250000L)
		CLKSEL = 0x10;
		CLKSEL = 0x10;
		while ((CLKSEL & 0x80) == 0);
	#elif (SYSCLK == 24500000L)
		CLKSEL = 0x00;
		CLKSEL = 0x00;
		while ((CLKSEL & 0x80) == 0);
	#elif (SYSCLK == 48000000L)
		// Before setting clock to 48 MHz, must transition to 24.5 MHz first
		CLKSEL = 0x00;
		CLKSEL = 0x00;
		while ((CLKSEL & 0x80) == 0);
		CLKSEL = 0x07;
		CLKSEL = 0x07;
		while ((CLKSEL & 0x80) == 0);
	#elif (SYSCLK == 72000000L)
		// Before setting clock to 72 MHz, must transition to 24.5 MHz first
		CLKSEL = 0x00;
		CLKSEL = 0x00;
		while ((CLKSEL & 0x80) == 0);
		CLKSEL = 0x03;
		CLKSEL = 0x03;
		while ((CLKSEL & 0x80) == 0);
	#else
		#error SYSCLK must be either 12250000L, 24500000L, 48000000L, or 72000000L
	#endif

	// Configure the pins used for square output
	P2MDOUT|=0b_0000_0011;
	P0MDOUT |= 0x10; // Enable UART0 TX as push-pull output
	P1MDOUT |= 0b_1000_0000;
	//CHANGED XBR0 (0x01 ?) AND XBR1 (0x10 ?)================
	XBR0     = 0b_0000_0101; // Enable UART0 on P0.4(TX) and P0.5(RX)
	XBR1     = 0X00; // Enable T0 on P0.0
	XBR2     = 0x40; // Enable crossbar and weak pull-ups

	//CHANGED FROM 0xFFL to 0x100=============
	#if (((SYSCLK/BAUDRATE)/(2L*12L))>0x100)
		#error Timer 0 reload value is incorrect because (SYSCLK/BAUDRATE)/(2L*12L)
> 0xFF
	#endif
	// Configure Uart 0
```

```
        SCON0 = 0x10;
        CKCON0 |= 0b_0000_0000 ; // Timer 1 uses the system clock divided by 12.
        TH1 = 0x100-((SYSCLK/BAUDRATE)/(2L*12L));
        TL1 = TH1;      // Init Timer1
        TMOD &= ~0xf0;  // TMOD: timer 1 in 8-bit auto-reload
        TMOD |=  0x20;
        TR1 = 1; // START Timer1
        TI = 1;  // Indicate TX0 ready


        // Configure and enable SMBus
        SMB0CF = 0b_0101_1100; //INH | EXTHOLD | SMBTOE | SMBFTE ;
        SMB0CF |= 0b_1000_0000;  // Enable SMBus
        // Configure Timer 0 as the I2C clock source

        CKCON0 |= 0b_0000_0100; // Timer0 clock source = SYSCLK
        TMOD &= 0xf0;  // Mask out timer 1 bits
        TMOD |= 0x02;  // Timer0 in 8-bit auto-reload mode
        // Timer 0 configured to overflow at 1/3 the rate defined by SMB_FREQUENCY
        TL0 = TH0 = 256-(SYSCLK/SMB_FREQUENCY/3);
        TR0 = 1; // Enable timer 0






        // Initialize timer 2 for periodic interrupts
        TMR2CN0=0x00;   // Stop Timer2; Clear TF2;
        CKCON0|=0b_0001_0000;
        TMR2RL=(-(SYSCLK/(2*DEFAULT_F))); // Initialize reload value
        TMR2=0xffff;   // Set to reload immediately
        ET2=1;         // Enable Timer2 interrupts
        TR2=1;         // Start Timer2



        /*
        // Initialize timer 3 for periodic interrupts
        TMR3CN0=0x00;   // Stop Timer3; Clear TF3;
        CKCON0|=0b_0100_0000; // Timer 3 uses the system clock
        TMR3RL=(-(SYSCLK/(2*DEFAULT_F))); // Initialize reload value
        TMR3=0xffff;   // Set to reload immediately
        EIE1|=0b_1000_0000;     // Enable Timer3 interrupts
        TMR3CN0|=0b_0000_0100;  // Start Timer3 (TMR3CN0 is not bit addressable)

        // Initialize timer 5 for periodic interrupts
        SFRPAGE=0x10;
        TMR5CN0=0x00;   // Stop Timer5; Clear TF5; WARNING: lives in SFR page 0x10
        CKCON1|=0b_0000_0100; // Timer 5 uses the system clock
        TMR5RL=(-(SYSCLK/(2*DEFAULT_F))); // Initialize reload value
        TMR5=0xffff;   // Set to reload immediately
        EIE2|=0b_0000_1000; // Enable Timer5 interrupts
             // Start Timer5 (TMR5CN0 is bit addressable)

        */
        EA=1; // Global interrupt enable

        return 0;
}

//  LCD in 4-bit interface mode
#include <EFM8LB1.h>
#include <stdio.h>
#include "lcd.h"
#include "globals.h"

// Uses Timer4 to delay <ms> mili-seconds.

void Timer4ms(unsigned char ms)
{
        unsigned char i;// usec counter
        unsigned char k;

        k=SFRPAGE;
        SFRPAGE=0x10;
        // The input for Timer 4 is selected as SYSCLK by setting bit 0 of CKCON1:
        CKCON1|=0b_0000_0001;

        TMR4RL = 65536-(SYSCLK/1000L); // Set Timer4 to overflow in 1 ms.
        TMR4 = TMR4RL;                 // Initialize Timer4 for first overflow

        TF4H=0; // Clear overflow flag
        TR4=1;  // Start Timer4
        for (i = 0; i < ms; i++)       // Count <ms> overflows
        {
                while (!TF4H);  // Wait for overflow
                TF4H=0;         // Clear overflow indicator
        }
        TR4=0; // Stop Timer4
        SFRPAGE=k;
}


void I2C_write (unsigned char output_data)
{
        SMB0DAT = output_data; // Put data into buffer
        SI = 0;
        while (!SI); // Wait until done with send
}

unsigned char I2C_read (void)
{
        unsigned char input_data;

        SI = 0;
        while (!SI); // Wait until we have data to read
        input_data = SMB0DAT; // Read the data

        return input_data;
}

void I2C_start (void)
{
        ACK = 1;
        STA = 1;     // Send I2C start
        STO = 0;
        SI = 0;
        while (!SI); // Wait until start sent
        STA = 0;     // Reset I2C start
}

void I2C_stop(void)
{
        STO = 1;      // Perform I2C stop
        SI = 0;// Clear SI
        //while (!SI);    // Wait until stop complete (Doesn't work???)
}

void nunchuck_init(bit print_extension_type)
{
        unsigned char i, buf[6];

        // Newer initialization format that works for all nunchucks
        I2C_start();
        I2C_write(0xA4);
        I2C_write(0xF0);
        I2C_write(0x55);
        I2C_stop();
        Timer4ms(1);
        I2C_start();
        I2C_write(0xA4);
        I2C_write(0xFB);
        I2C_write(0x00);
        I2C_stop();
        Timer4ms(1);


        // Read the extension type from the register block.  For the original Nunchuk it
should be
        // 00 00 a4 20 00 00.
        I2C_start();
        I2C_write(0xA4);
        I2C_write(0xFA); // extension type register
        I2C_stop();
        Timer4ms(3); // 3 ms required to complete acquisition

        I2C_start();
        I2C_write(0xA5);

        // Receive values
        for(i=0; i<6; i++)
        {
                buf[i]=I2C_read();
        }
        ACK=0;
        I2C_stop();
        Timer4ms(3);


        if(print_extension_type)
        {
                printf("Extension type: %02x  %02x  %02x  %02x  %02x  %02x\n",
                        buf[0],  buf[1], buf[2], buf[3], buf[4], buf[5]);
        }

        // Send the crypto key (zeros), in 3 blocks of 6, 6 & 4.

        I2C_start();
        I2C_write(0xA4);
        I2C_write(0xF0);
        I2C_write(0xAA);
        I2C_stop();
        Timer4ms(1);


        I2C_start();
        I2C_write(0xA4);
        I2C_write(0x40);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_stop();
        Timer4ms(1);


        I2C_start();
        I2C_write(0xA4);
        I2C_write(0x40);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_stop();
        Timer4ms(1);
        I2C_start();
        I2C_write(0xA4);
        I2C_write(0x40);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_write(0x00);
        I2C_stop();
        Timer4ms(1);

}

void nunchuck_getdata(unsigned char * s)
{
        unsigned char i;

        // Start measurement
        I2C_start();
        I2C_write(0xA4);
        I2C_write(0x00);
        I2C_stop();
        Timer4ms(3);   // 3 ms required to complete acquisition


        // Request values
        I2C_start();
        I2C_write(0xA5);

        // Receive values
        for(i=0; i<6; i++)
        {
               s[i]=(I2C_read()^0x17)+0x17; // Read and decrypt
        }
        ACK=0;
        I2C_stop();
}



void LCD_pulse (void)
{
        LCD_E=1;
        Timer4ms(1);
        LCD_E=0;
}

void LCD_byte (unsigned char x)
{
        // The accumulator in the C8051Fxxx is bit addressable!
        ACC=x; //Send high nible
        LCD_D7=ACC_7;
        LCD_D6=ACC_6;
        LCD_D5=ACC_5;
        LCD_D4=ACC_4;
        LCD_pulse();
        Timer4ms(1);
        ACC=x; //Send low nible
        LCD_D7=ACC_3;
        LCD_D6=ACC_2;
        LCD_D5=ACC_1;
        LCD_D4=ACC_0;
        LCD_pulse();
}

void WriteData (unsigned char x)
{
        LCD_RS=1;
        LCD_byte(x);
        Timer4ms(2);
```

```
}

void WriteCommand (unsigned char x)
{
	LCD_RS=0;
	LCD_byte(x);
	Timer4ms(5);
}

void LCD_4BIT (void)
{
	LCD_E=0; // Resting state of LCD's enable is zero
	//LCD_RW=0; // We are only writing to the LCD in this program
	Timer4ms(20);
	// First make sure the LCD is in 8-bit mode and then change to 4-bit mode
	WriteCommand(0x33);
	WriteCommand(0x33);
	WriteCommand(0x32); // Change to 4-bit mode

	// Configure the LCD
	WriteCommand(0x28);
	WriteCommand(0x0c);
	WriteCommand(0x01); // Clear screen command (takes some time)
	Timer4ms(20); // Wait for clear screen command to finsih.
}

void LCDprint(char * string, unsigned char line, bit clear)
{
	int j;

	WriteCommand(line==2?0xc0:0x80);
	Timer4ms(5);
	for(j=0; string[j]!=0; j++)  WriteData(string[j]);// Write the message
	if(clear)  for(;  j<CHARS_PER_LINE;  j++)  WriteData(' ');  //  Clear  the  rest  of  the
line
}


//main function for transmitter
//ELEC 291 LB6
#include <EFM8LB1.h>
#include <stdlib.h>
#include <stdio.h>
#include "globals.h"
#include "lcd.h"

volatile int counter = 0;
volatile int track = 0;
volatile int forward = 0;
volatile int backward = 0;
volatile int right = 0;
volatile int left = 0;


void Timer2_ISR (void) interrupt INTERRUPT_TIMER2
{
	if(forward){

		if(counter<200){
			TF2H=0;
			OUT0=1;
			OUT1=!OUT0;
			counter++;
		}
		else{
			TF2H=0;
			OUT0=0;
			OUT1=!OUT0;
			counter=0;

		}
```

```
}


else if(backward){


       if(counter<250){
              TF2H=0;
              OUT0=1;
              OUT1=!OUT0;
              counter++;
       }
       else{
              TF2H=0;
              OUT0=0;
              OUT1=!OUT0;
              counter=0;
       }



}

else if(left){

       if(counter<150){
              TF2H=0;
              OUT0=1;
              OUT1=!OUT0;
              counter++;
       }
       else{
              TF2H=0;
              OUT0=0;
              OUT1=!OUT0;
              counter=0;
       }

}
else if(right){

       if(counter<300){
              TF2H=0;
              OUT0=1;
              OUT1=!OUT0;
              counter++;
       }
       else{
              TF2H=0;
              OUT0=0;
              OUT1=!OUT0;
              counter=0;
       }

}
else if(track){

       if(counter<125){
              TF2H=0;
              OUT0=1;
              OUT1=!OUT0;
              counter++;
       }

       else{
              TF2H=0;
              OUT0=0;
              OUT1=!OUT0;
              counter=0;
       }
```

```
        }


        else{ //IDLE MODE FOR REMOTE

                if(counter <400){
                        TF2H = 0; // Clear Timer2 interrupt flag
                        OUT0=1;
                        OUT1=!OUT0;
                        counter++;
                }

                else{
                        TF2H=0;
                        OUT0=0;
                        OUT1=!OUT0;
                        counter=0;
                }

        }

}
void main (void)
{
        unsigned char rbuf[6];
        unsigned long int x, f;

        int joy_x, joy_y, off_x, off_y, acc_x, acc_y, acc_z;
        bit but1, but2;
        //int width_new;
        //int reload_new;
        //int tmr2_origin = TMR2RL;



        //char buff [17];
        //Configure LCd
        LCD_4BIT();
        LCDprint("thisworking", 1, 1);
        Timer4ms(500);


        printf("\x1b[2J"); // Clear screen using ANSI escape sequence.

        //printf("New frequency=");
        //scanf("%lu", &f);
        f=17550;
        x=(SYSCLK/(2L*f));


        TR2=0; // Stop timer 2
        TMR2RL=0x10000L-x; // Change reload value for new frequency
        TMR3RL = TMR2RL*2;
        TR2=1; // Start timer 2
        //tmr2_origin = TMR2RL;  //store TMR2RL
        f=SYSCLK/(2L*(0x10000L-TMR2RL));
        printf("\nActual frequency: %lu\n", f);




        printf("\n\nEFM8LB1 WII Nunchuck I2C Reader\n");
        printf("Variable frequency generator for the EFM8LB1.\r\n"
               "Check pins P2.0 and P1.7 with the oscilloscope.\r\n");


        Timer4ms(200);
        nunchuck_init(1);
        Timer4ms(100);


        nunchuck_getdata(rbuf);

        off_x=(int)rbuf[0]-128;
        off_y=(int)rbuf[1]-128;
        printf("Offset_X:%4d Offset_Y:%4d\n\n", off_x, off_y);

        while(1)
        {
         /* PWM CONFIGURATION tentative
          50% duty cycle = control mode
          40% duty cycle = track mode
          60% duty cycle = move forward
          70% duty cycle = move backward
          80% duty cycle = move right
          90% duty cycle = move left
          */

          //NESTED WHILE LOOPS. outer while loop is remote control state
          // and inner loop is tracking state
               nunchuck_getdata(rbuf);

               joy_x=(int)rbuf[0]-128-off_x;
               joy_y=(int)rbuf[1]-128-off_y;
               acc_x=rbuf[2]*4;
               acc_y=rbuf[3]*4;
               acc_z=rbuf[4]*4;

               but1=(rbuf[5] & 0x01)?1:0;
               but2=(rbuf[5] & 0x02)?1:0;
               if (rbuf[5] & 0x04) acc_x+=2;
               if (rbuf[5] & 0x08) acc_x+=1;
               if (rbuf[5] & 0x10) acc_y+=2;
               if (rbuf[5] & 0x20) acc_y+=1;
               if (rbuf[5] & 0x40) acc_z+=2;
               if (rbuf[5] & 0x80) acc_z+=1;

               //DEFAULT REMOTE CONTROL STATE

               LCDprint("mode:remote", 1, 1);
               LCDprint("idle", 2, 1);
               if(joy_x >= 80){
                      LCDprint("right", 2, 1);
                      right=1;
                      while(joy_x>=80){
                             nunchuck_getdata(rbuf);

                             joy_x=(int)rbuf[0]-128-off_x;
                             joy_y=(int)rbuf[1]-128-off_y;
                             acc_x=rbuf[2]*4;
                             acc_y=rbuf[3]*4;
                             acc_z=rbuf[4]*4;

                             but1=(rbuf[5] & 0x01)?1:0;
                             but2=(rbuf[5] & 0x02)?1:0;
                             if (rbuf[5] & 0x04) acc_x+=2;
                             if (rbuf[5] & 0x08) acc_x+=1;
                             if (rbuf[5] & 0x10) acc_y+=2;
                             if (rbuf[5] & 0x20) acc_y+=1;
                             if (rbuf[5] & 0x40) acc_z+=2;
                             if (rbuf[5] & 0x80) acc_z+=1;

                             printf("Buttons(Z:%c,      C:%c)      Joystick(%4d,      %4d)
Accelerometer(%3d, %3d, %3d)\x1b[0J\r",
                                    but1?'1':'0',  but2?'1':'0',  joy_x,  joy_y,  acc_x,
acc_y, acc_z);

                             Timer4ms(100);
```

```
			}
		}
		right=0;
		counter = 0;

		if(joy_x <= -80){
			LCDprint("left", 2, 1);
			//90% duty cycle
			left=1;
			while(joy_x <=-80){
				nunchuck_getdata(rbuf);

				joy_x=(int)rbuf[0]-128-off_x;
				joy_y=(int)rbuf[1]-128-off_y;
				acc_x=rbuf[2]*4;
				acc_y=rbuf[3]*4;
				acc_z=rbuf[4]*4;

				but1=(rbuf[5] & 0x01)?1:0;
				but2=(rbuf[5] & 0x02)?1:0;
				if (rbuf[5] & 0x04) acc_x+=2;
				if (rbuf[5] & 0x08) acc_x+=1;
				if (rbuf[5] & 0x10) acc_y+=2;
				if (rbuf[5] & 0x20) acc_y+=1;
				if (rbuf[5] & 0x40) acc_z+=2;
				if (rbuf[5] & 0x80) acc_z+=1;

				printf("Buttons(Z:%c,      C:%c)      Joystick(%4d,      %4d)
Accelerometer(%3d, %3d, %3d)\x1b[0J\r",
					but1?'1':'0',  but2?'1':'0',  joy_x,  joy_y,  acc_x,
acc_y, acc_z);

				Timer4ms(100);
			}
		}
		left=0;
		counter=0;


		if(joy_y >= 80){
			LCDprint("forward", 2, 1);
			//60% duty cycle
			forward=1;
			while(joy_y>80){
				nunchuck_getdata(rbuf);

				joy_x=(int)rbuf[0]-128-off_x;
				joy_y=(int)rbuf[1]-128-off_y;
				acc_x=rbuf[2]*4;
				acc_y=rbuf[3]*4;
				acc_z=rbuf[4]*4;

				but1=(rbuf[5] & 0x01)?1:0;
				but2=(rbuf[5] & 0x02)?1:0;
				if (rbuf[5] & 0x04) acc_x+=2;
				if (rbuf[5] & 0x08) acc_x+=1;
				if (rbuf[5] & 0x10) acc_y+=2;
				if (rbuf[5] & 0x20) acc_y+=1;
				if (rbuf[5] & 0x40) acc_z+=2;
				if (rbuf[5] & 0x80) acc_z+=1;

				printf("Buttons(Z:%c,      C:%c)      Joystick(%4d,      %4d)
Accelerometer(%3d, %3d, %3d)\x1b[0J\r",
					but1?'1':'0',   but2?'1':'0',   joy_x,   joy_y,   acc_x,
acc_y, acc_z);

				Timer4ms(100);

			}
		}
		forward=0;
```

```
            counter = 0;

            if(joy_y <= -80){
                  LCDprint("backward", 2, 1);
                  //70% duty cycle
                  backward=1;
                  while(joy_y <= -80){
                        nunchuck_getdata(rbuf);

                        joy_x=(int)rbuf[0]-128-off_x;
                        joy_y=(int)rbuf[1]-128-off_y;
                        acc_x=rbuf[2]*4;
                        acc_y=rbuf[3]*4;
                        acc_z=rbuf[4]*4;

                        but1=(rbuf[5] & 0x01)?1:0;
                        but2=(rbuf[5] & 0x02)?1:0;
                        if (rbuf[5] & 0x04) acc_x+=2;
                        if (rbuf[5] & 0x08) acc_x+=1;
                        if (rbuf[5] & 0x10) acc_y+=2;
                        if (rbuf[5] & 0x20) acc_y+=1;
                        if (rbuf[5] & 0x40) acc_z+=2;
                        if (rbuf[5] & 0x80) acc_z+=1;

                        printf("Buttons(Z:%c,      C:%c)      Joystick(%4d,      %4d)
Accelerometer(%3d, %3d, %3d)\x1b[0J\r",
                              but1?'1':'0',   but2?'1':'0',   joy_x,   joy_y,   acc_x,
acc_y, acc_z);

                        Timer4ms(100);

                  }
            }
            backward=0;
            counter = 0;

            printf("Buttons(Z:%c,  C:%c)  Joystick(%4d,  %4d)  Accelerometer(%3d,  %3d,
%3d)\x1b[0J\r",
                     but1?'1':'0', but2?'1':'0', joy_x, joy_y, acc_x, acc_y, acc_z);

            Timer4ms(100);

            if(but1==0){
                  LCDprint("mode:track", 1, 1);
                  LCDprint("track", 2, 1);
                  Timer4ms(100);
                  track = 1;

                  while(but2!=0){
                        nunchuck_getdata(rbuf);

                        joy_x=(int)rbuf[0]-128-off_x;
                        joy_y=(int)rbuf[1]-128-off_y;
                        acc_x=rbuf[2]*4;
                        acc_y=rbuf[3]*4;
                        acc_z=rbuf[4]*4;

                        but1=(rbuf[5] & 0x01)?1:0;
                        but2=(rbuf[5] & 0x02)?1:0;
                        if (rbuf[5] & 0x04) acc_x+=2;
                        if (rbuf[5] & 0x08) acc_x+=1;
                        if (rbuf[5] & 0x10) acc_y+=2;
                        if (rbuf[5] & 0x20) acc_y+=1;
                        if (rbuf[5] & 0x40) acc_z+=2;
                        if (rbuf[5] & 0x80) acc_z+=1;

                        printf("Buttons(Z:%c,      C:%c)      Joystick(%4d,      %4d)
Accelerometer(%3d, %3d, %3d)\x1b[0J\r",
                              but1?'1':'0',   but2?'1':'0',   joy_x,   joy_y,   acc_x,
acc_y, acc_z);
Timer4ms(100);
                        }
                        track = 0;
                        counter = 0;
                }


        }
}
```

Appendix II: Receiver Source Code

```
#include <XC.h>
#include <sys/attribs.h>
#include <stdio.h>
#include <stdlib.h>

/* Pinout for DIP28 PIC32MX130:
                                          --------
                                   MCLR -|1     28|- AVDD
  VREF+/CVREF+/AN0/C3INC/RPA0/CTED1/RA0 -|2     27|- AVSS
        VREF-/CVREF-/AN1/RPA1/CTED2/RA1 -|3     26|- AN9/C3INA/RPB15/SCK2/CTED6/PMCS1/RB15
                PGED1/AN2/C1IND/C2INB/C3IND/RPB0/RB0     -|4                          25|-
CVREFOUT/AN10/C3INB/RPB14/SCK1/CTED5/PMWR/RB14
  PGEC1/AN3/C1INC/C2INA/RPB1/CTED12/RB1 -|5     24|- AN11/RPB13/CTPLS/PMRD/RB13
   AN4/C1INB/C2IND/RPB2/SDA2/CTED13/RB2 -|6     23|- AN12/PMD0/RB12
     AN5/C1INA/C2INC/RTCC/RPB3/SCL2/RB3 -|7     22|- PGEC2/TMS/RPB11/PMD1/RB11
                                    VSS -|8     21|- PGED2/RPB10/CTED11/PMD2/RB10
                     OSC1/CLKI/RPA2/RA2 -|9     20|- VCAP
                OSC2/CLKO/RPA3/PMA0/RA3 -|10    19|- VSS
                         SOSCI/RPB4/RB4 -|11    18|- TDO/RPB9/SDA1/CTED4/PMD3/RB9
         SOSCO/RPA4/T1CK/CTED9/PMA1/RA4 -|12    17|- TCK/RPB8/SCL1/CTED10/PMD4/RB8
                                    VDD -|13    16|- TDI/RPB7/CTED3/PMD5/INT0/RB7
                    PGED3/RPB5/PMD7/RB5 -|14    15|- PGEC3/RPB6/PMD6/RB6
                                          --------
*/


// Configuration Bits (somehow XC32 takes care of this)
#pragma config FNOSC = FRCPLL       // Internal Fast RC oscillator (8 MHz) w/ PLL
#pragma config FPLLIDIV = DIV_2     // Divide FRC before PLL (now 4 MHz)
#pragma config FPLLMUL = MUL_20     // PLL Multiply (now 80 MHz)
#pragma config FPLLODIV = DIV_2     // Divide After PLL (now 40 MHz)
#pragma config FWDTEN = OFF         // Watchdog Timer Disabled
#pragma config FPBDIV = DIV_1       // PBCLK = SYCLK
#pragma config FSOSCEN = OFF        // Turn off secondary oscillator on A4 and B4

// Defines
#define SYSCLK 40000000L
#define FREQ 100000L // We need the ISR for timer 1 every 10 us
#define Baud2BRG(desired_baud)( (SYSCLK / (16*desired_baud))-1)

volatile int ISR_pwm1=150, ISR_pwm2=150, ISR_cnt=0, ISR_frc, ISR_cnt2=0;
long int time_ISR = 0;
long int Prev_V_ISR, Peak_V_ISR = 0;
volatile int movement_instruction_ISR=0;
volatile int bitone, bittwo, bitthree;
volatile int entered_if_statement = 0;
long int timer_count = 0;
long int adc_four;
long int v1_thres ;
long int v2_thres ;

int ADCRead(char analogPIN)
{
    AD1CHS = analogPIN << 16;    // AD1CHS<16:19> controls which analog pin goes to the ADC

    AD1CON1bits.SAMP = 1;        // Begin sampling
    while(AD1CON1bits.SAMP);     // wait until acquisition is done
    while(!AD1CON1bits.DONE);    // wait until conversion done

    return ADC1BUF0;             // result stored in ADC1BUF0
}

// The Interrupt Service Routine for timer 1 is used to generate one or more standard
// hobby servo signals.  The servo signal has a fixed period of 20ms and a pulse width
// between 0.6ms and 2.4ms.
void __ISR(_TIMER_1_VECTOR, IPL5SOFT) Timer1_Handler(void)
```

```
{
	IFS0CLR=_IFS0_T1IF_MASK; // Clear timer 1 interrupt flag, bit 4 of IFS0

	ISR_cnt++;
	if(ISR_cnt==ISR_pwm1)
	{
		LATAbits.LATA3 = 0;
	}
	if(ISR_cnt==ISR_pwm2)
	{
		LATBbits.LATB4 = 0;
	}
	if(ISR_cnt == 100){
	}


	if(ISR_cnt>=2000)
	{
		/*20msCount++;*/
		ISR_cnt=0; // 2000 * 10us=20ms
		LATAbits.LATA3 = 1;
		LATBbits.LATB4 = 1;
		ISR_frc++;
	}



}

void delay_ms (int msecs)
{
	int ticks;
	ISR_frc=0;
	ticks=msecs/20;
	while(ISR_frc<ticks);
}




void SetupTimer1 (void)
{
	// Explanation here: https://www.youtube.com/watch?v=bu6TTZHnMPY
	__builtin_disable_interrupts();
	PR1 =(SYSCLK/FREQ)-1; // since SYSCLK/FREQ = PS*(PR1+1)
	TMR1 = 0;
	T1CONbits.TCKPS  =  0;  //  3=1:256  prescale  value,  2=1:64  prescale  value,  1=1:8
prescale value, 0=1:1 prescale value
	T1CONbits.TCS = 0; // Clock source
	T1CONbits.ON = 1;
	IPC1bits.T1IP = 5;
	IPC1bits.T1IS = 0;

	PR2 =(SYSCLK/FREQ)-1; // since SYSCLK/FREQ = PS*(PR1+1)
	TMR2 = 0;
	T2CONbits.TCKPS  =  0;  //  3=1:256  prescale  value,  2=1:64  prescale  value,  1=1:8
prescale value, 0=1:1 prescale value
	T2CONbits.TCS = 0; // Clock source
	T2CONbits.ON = 1;
	IPC2bits.T2IP = 5;
	IPC2bits.T2IS = 0;

	IFS0bits.T1IF = 0;
	IEC0bits.T1IE = 1;

	IFS0bits.T2IF = 0;
	IEC0bits.T2IE = 1;

	INTCONbits.MVEC = 1; //Int multi-vector
	__builtin_enable_interrupts();
}
```

```
// Use the core timer to wait for 1 ms.
void wait_1ms(void)
{
    unsigned int ui;
    _CP0_SET_COUNT(0); // resets the core timer count

    // get the core timer count
    while ( _CP0_GET_COUNT() < (SYSCLK/(2*1000)) );
}

void waitms(int len)
{
       while(len--) wait_1ms();
}

#define PIN_PERIOD (PORTB&(1<<5))

// GetPeriod() seems to work fine for frequencies between 200Hz and 700kHz.
long int GetPeriod (int n)
{
       int i;
       unsigned int saved_TCNT1a, saved_TCNT1b;

    _CP0_SET_COUNT(0); // resets the core timer count
       while (PIN_PERIOD!=0) // Wait for square wave to be 0
       {
              if(_CP0_GET_COUNT() > (SYSCLK)) return 0;
       }

    _CP0_SET_COUNT(0); // resets the core timer count
       while (PIN_PERIOD==0) // Wait for square wave to be 1
       {
              if(_CP0_GET_COUNT() > (SYSCLK)) return 0;
       }

    _CP0_SET_COUNT(0); // resets the core timer count
       for(i=0; i<n; i++) // Measure the time of 'n' periods
       {
              while (PIN_PERIOD!=0) // Wait for square wave to be 0
              {
                     if(_CP0_GET_COUNT() > (SYSCLK)) return 0;
              }
              while (PIN_PERIOD==0) // Wait for square wave to be 1
              {
                     if(_CP0_GET_COUNT() > (SYSCLK)) return 0;
              }
       }

       return  _CP0_GET_COUNT();
}


void UART2Configure(int baud_rate)
{
    // Peripheral Pin Select
    U2RXRbits.U2RXR = 4;    //SET RX to RB8
    RPB9Rbits.RPB9R = 2;    //SET RB9 to TX

    U2MODE = 0;         // disable autobaud, TX and RX enabled only, 8N1, idle=HIGH
    U2STA = 0x1400;     // enable TX and RX
    U2BRG = Baud2BRG(baud_rate); // U2BRG = (FPb / (16*baud)) - 1

    U2MODESET = 0x8000;     // enable UART2
}

void uart_puts(char * s)
{
       while(*s)
       {
              putchar(*s);
              s++;
       }
}

char HexDigit[]="0123456789ABCDEF";
void PrintNumber(long int val, int Base, int digits)
{
       int j;
       #define NBITS 32
       char buff[NBITS+1];
       buff[NBITS]=0;

       j=NBITS-1;
       while ( (val>0) | (digits>0) )
       {
              buff[j--]=HexDigit[val%Base];
              val/=Base;
              if(digits!=0) digits--;
       }
       uart_puts(&buff[j+1]);
}

// Good information about ADC in PIC32 found here:
// http://umassamherstm5.org/tech-tutorials/pic32-tutorials/pic32mx220-tutorials/adc
void ADCConf(void)
{
    AD1CON1CLR = 0x8000;    // disable ADC before configuration
      AD1CON1  =  0x00E0;           //  internal  counter  ends  sampling  and  starts  conversion
(auto-convert), manual sample
    AD1CON2 = 0;            // AD1CON2<15:13> set voltage reference to pins AVSS/AVDD
    AD1CON3 = 0x0f01;       // TAD = 4*TPB, acquisition time = 15*TAD
    AD1CON1SET=0x8000;      // Enable ADC
}


void ConfigurePins(void)
{
    // Configure pins as analog inputs
    ANSELBbits.ANSB2 = 1;   // set RB2 (AN4, pin 6 of DIP28) as analog pin
    TRISBbits.TRISB2 = 1;   // set RB2 as an input
    ANSELBbits.ANSB3 = 1;   // set RB3 (AN5, pin 7 of DIP28) as analog pin
    TRISBbits.TRISB3 = 1;   // set RB3 as an input

       // Configure digital input pin to measure signal period
       ANSELB &= ~(1<<5); // Set RB5 as a digital I/O (pin 14 of DIP28)
    TRISB |= (1<<5);   // configure pin RB5 as input
    CNPUB |= (1<<5);   // Enable pull-up resistor for RB5

    // We can do the three lines above using this instead:
    // ANSELBbits.ANSELB5=0;  Not needed because RB5 can not be analog input?
    // TRISBbits.TRISB5=1;
    // CNPUBbits.CNPUB5=1;

    // Configure output pins
       TRISAbits.TRISA0 = 0; // pin  2 of DIP28
       TRISAbits.TRISA1 = 0; // pin  3 of DIP28
       TRISBbits.TRISB0 = 0; // pin  4 of DIP28
       TRISBbits.TRISB1 = 0; // pin  5 of DIP28
       //TRISAbits.TRISA2 = 0; // pin  9 of DIP28
       //TRISAbits.TRISA3 = 0; // pin 10 of DIP28
       //TRISBbits.TRISB4 = 0; // pin 11 of DIP28
       INTCONbits.MVEC = 1;

       //Configure output pins for motor
       TRISBbits.TRISB0 = 0; // pin4
       TRISBbits.TRISB1 = 0; // pin5
       TRISAbits.TRISA2 = 0; // pin9
       TRISAbits.TRISA4 = 0; // pin12
}

void PrintFixedPoint (unsigned long number, int decimals)
```

```
{
	int divider=1, j;

	j=decimals;
	while(j--) divider*=10;

	PrintNumber(number/divider, 10, 1);
	uart_puts(".");
	PrintNumber(number%divider, 10, decimals);
}

// movements
void go_forward(){
	LATBbits.LATB0 = 0; //pin 4
	LATBbits.LATB1 = 1; //pin 5
	LATAbits.LATA2 = 0; //pin 9
	LATAbits.LATA4 = 1; //pin 12
}

void go_backward(){
	LATBbits.LATB0 = 1; //pin 4
	LATBbits.LATB1 = 0; //pin 5
	LATAbits.LATA2 = 1; //pin 9
	LATAbits.LATA4 = 0; //pin 12
}

void turn_right(){
	LATBbits.LATB0 = 0; //pin 4
	LATBbits.LATB1 = 1; //pin 5
	LATAbits.LATA2 = 1; //pin 9
	LATAbits.LATA4 = 0; //pin 12
}

void turn_left(){
	LATBbits.LATB0 = 1; //pin 4
	LATBbits.LATB1 = 0; //pin 5
	LATAbits.LATA2 = 0; //pin 9
	LATAbits.LATA4 = 1; //pin 12
}

void stop_motors(){
	//printf("\n\r0ol");
	//all motor pins off
	LATBbits.LATB0 = 0; //pin 4
	LATBbits.LATB1 = 0; //pin 5
	LATAbits.LATA2 = 0; //pin 9
	LATAbits.LATA4 = 0; //pin 12
}



int mode_handler(int instruction,int mode){

if((50<instruction)&&(instruction<=250)){
	mode = 1;
	return mode; // remote mode //

	}
	else {

	return 0; // track mode //

	}


}

long int real_time_average_V1(){
	int count1 = 0;
	long int sum_V1 = 0;
	while(count1 < 10){
```

```
              sum_V1 += ADCRead(4);
              count1++;
              delay_ms(10);
       }
       //printf("\r\v1: %d", sum_V1*3290L/1203L);
       return sum_V1 * 3290L / 1023L/20L;
}

long int real_time_average_V2(){
       int count2 = 0;
       long int sum_V2 = 0;
       while(count2 < 10){

              sum_V2 += ADCRead(5);
              count2++;
              delay_ms(10);
       }
//     printf("\r\v2: %d", sum_V2*3290L/1203L);
       return sum_V2 * 3290L / 1023L/20L;
}

// In order to keep this as nimble as possible, avoid
// using floating point or printf() on any of its forms!
void main(void)
{
       volatile unsigned long t=0;
    long int adcval1, adcval2;
    long int v1,v2;
       unsigned long int count, f;
       int movement_instruction = 9;
       int mode;
       float left_right_difference;


       CFGCON = 0;

    UART2Configure(115200);  // Configure UART2 for a baud rate of 115200
    ConfigurePins();
       ADCConf(); // Configure ADC
    SetupTimer1();

    waitms(50); // Give PuTTY time to start
       uart_puts("\x1b[2J\x1b[1;1H"); // Clear screen using ANSI escape sequence.
       uart_puts("\r\nPIC32 multi I/O example.\r\n");
       uart_puts("Measures the voltage at channels 4 and 5 (pins 6 and 7 of DIP28
package)\r\n");
       uart_puts("Measures period on RB5 (pin 14 of DIP28 package)\r\n");
       uart_puts("Toggles RA0, RA1, RB0, RB1, RA2 (pins 2, 3, 4, 5, 9, of DIP28
package)\r\n");
       uart_puts("Generates Servo PWM signals at RA3, RB4 (pins 10, 11 of DIP28
package)\r\n\r\n");

       //Prev_V_ISR = ADCRead(4) * 3290.0 / 1023.0;

       //set motors off initially
       LATAbits.LATA2 = 0;
       LATAbits.LATA3 = 0;
       LATBbits.LATB4 = 0;
       LATAbits.LATA4 = 0;
       mode = 1;




       while(1)
       {

              v1 = real_time_average_V1();

              v1_thres = 200;
    v2_thres = 200;

    if(v1> v1_thres && v1 < 500){

    v1_thres = v1; // to update the max value of recived signal

    }




    v2 = real_time_average_V2();

if(v2> v2_thres && v2 < 500){

    v2_thres = v2;

    }

 count=GetPeriod(100);
    if(count>0)
    {
            f=((SYSCLK/2L)*100L)/count;
            //uart_puts("f=");
            //PrintNumber(f, 10, 7);
            //uart_puts("Hz\r\n");
    //      PrintNumber(count, 10, 6);
    //      uart_puts("        \r\n");
    }
    else
    {
            uart_puts("NO SIGNAL                     \r");
    }

    mode = mode_handler(f,mode);



    //if track mode (mode = 0)

    if(mode == 0/* && v1 != 0*/){



            if(v1 <= v1_thres && v2<=v2_thres ){
                    go_forward();
                    delay_ms(500);
                    stop_motors();
                    if((v1 == v1_thres)){

            turn_left();
            delay_ms(500);
            stop_motors();
                    printf("Turning left\r\n");
                    }

                    if((v2 == v2_thres)){

            turn_right();
            delay_ms(500);
            stop_motors();
                    printf("Turning right\r\n");



             }
            }

            else if((v1 >= v1_thres) && (v2 >= v2_thres)){
                    go_backward();
```

```
                delay_ms(700);
                stop_motors();
                printf("Turning forward\r\n");
        }


}
//if remote mode (mode = 1)
if(mode == 1){

/*1. 0 means go forward
  2. 1 means go backward
  3. 2 means turn left
  4. 3 means turn right*/

        printf("\n\r %d", movement_instruction_ISR);
        if((170<=f)&&(f<=195)){
        movement_instruction_ISR=1;


        }
        else if((135<=f)&&(f<=150)){
        movement_instruction_ISR=2;


        }
        else if((210<=f)&&(f<=250)){
        movement_instruction_ISR=3;

        }
        else if((110<=f)&&(f<=125)){
        movement_instruction_ISR=4;

        }
        else{
                movement_instruction_ISR = 0;
                }


        if(movement_instruction_ISR  == 1)
        {
                go_forward();
                delay_ms(500);
                stop_motors();
        }
        else if(movement_instruction_ISR == 2)
        {
                go_backward();
                delay_ms(500);
                stop_motors();

        }


        else if(movement_instruction_ISR == 3)
        {
                turn_left();
                delay_ms(500);
                stop_motors();

        }
        else if(movement_instruction_ISR == 4)
        {
                turn_right();
                delay_ms(500);
                stop_motors();

        }                       else{
                stop_motors();
                delay_ms(700);
                movement_instruction_ISR = 0;
```

```
                        }



                }

        }
        delay_ms(10);
}
```