BONE-KL25Z Tank

ECE331 Final Project: User Friendly Control Interface for an Electric Motor Tank

Humberto Aboud Torres Lobo and David Ametsitsi
CM 1492/2956

February 25th, 2014

Table of Contents

Introduction

Overview

The idea of this project is to take a DC motor based tank and create two frames of interface for a user to control it. The first objective consists of a wireless motion based control that consists of a device controlled by the user, the tank will move based on the movements of the controller. The second objective is a real time stream video captured by a camera on the tank that can be viewed on any electronic device with access to the internet, such as computers, smartphones or tablets.

Objectives

I. To efficiently and effectively implement key skills learned and utilized during our quarter working with Embedded Systems in Dr. Jian Jian Song's Class.
II. To challenge ourselves and to gain real-world experience in Embedded Systems and coding methodology.
III. To gain profound knowledge and research further independent projects.

Equipment

Lenovo W530

Texas Instruments Beagle Bone Black

GCC compiler

Freescale Freedom FRDM-KL25Z

FreeScale Code Warrior IDE and Debugger

Digi International XBee S1

MCC7805CT Voltage Regulator

Tank Frame

User Manual:

Notes

This device complies with part 15 of the FCC rules. Operation is subject to the following two conditions: (1) this device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

Requirements

BONE-KL25Z Tank controller

BONE-KL25Z Tank unit

9.6V battery pack INCLUDED

9V battery NOT INCLUDED

Use

Allow yourself to get familiar with the operations and use of the BONE-KL25Z Tank and Tank Controller.

Hold the controller right-side up with the antenna pointing up. Navigate to the right hand side of the controller and flip the switch toward the antenna.

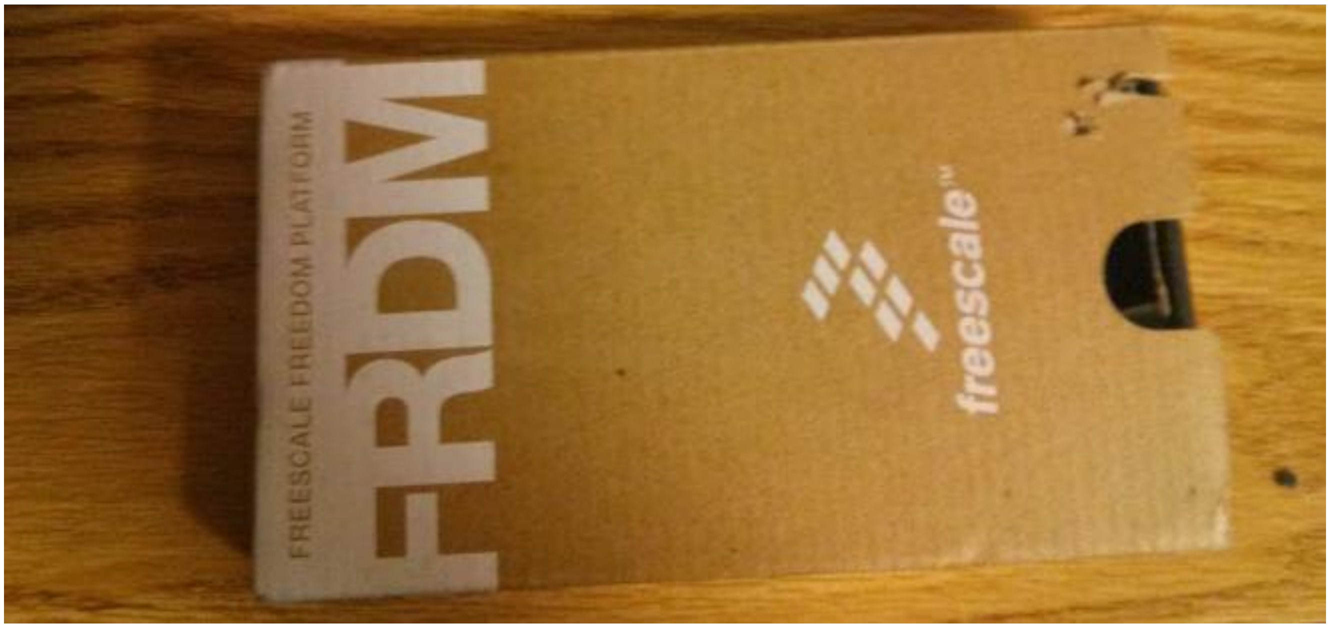

Figure 1. Controller unit

Next, navigate to the back of the tank and flip the toggle switch to the "I" position.

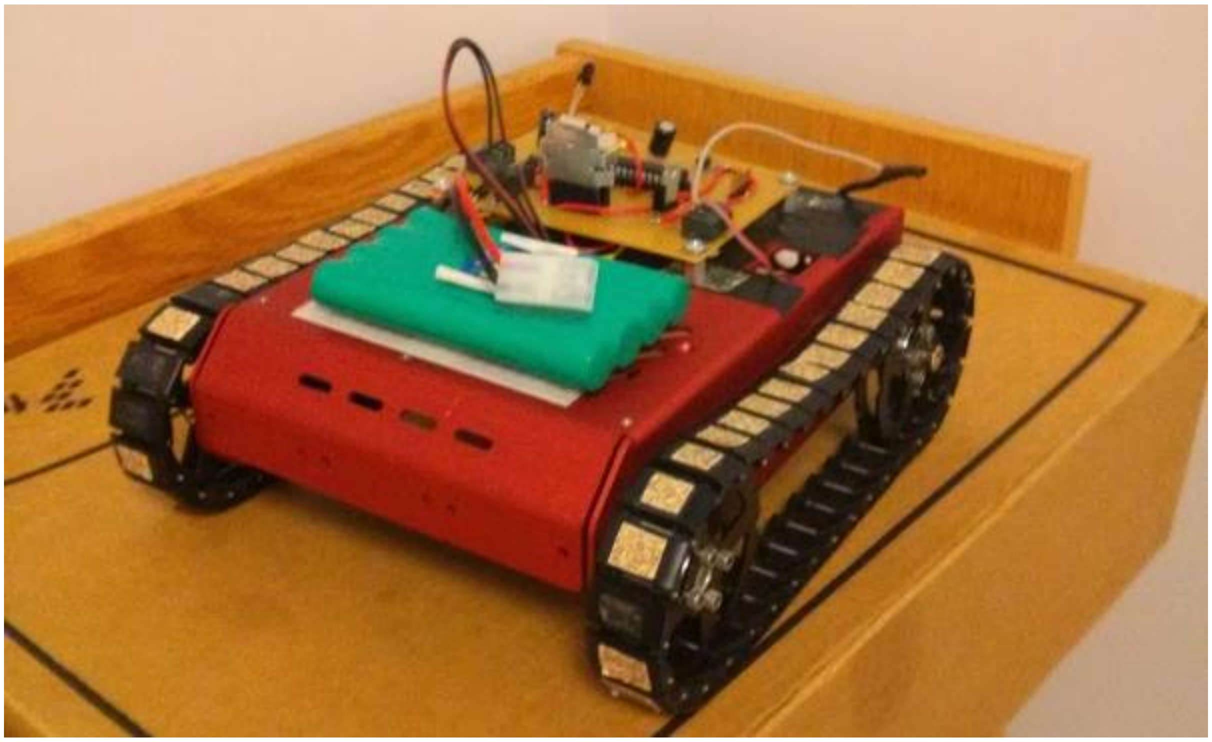

Figure 2. photo of tank device

Wait for the BLUE LED next to the XBee module to light up and you are ready to begin controlling the tank.

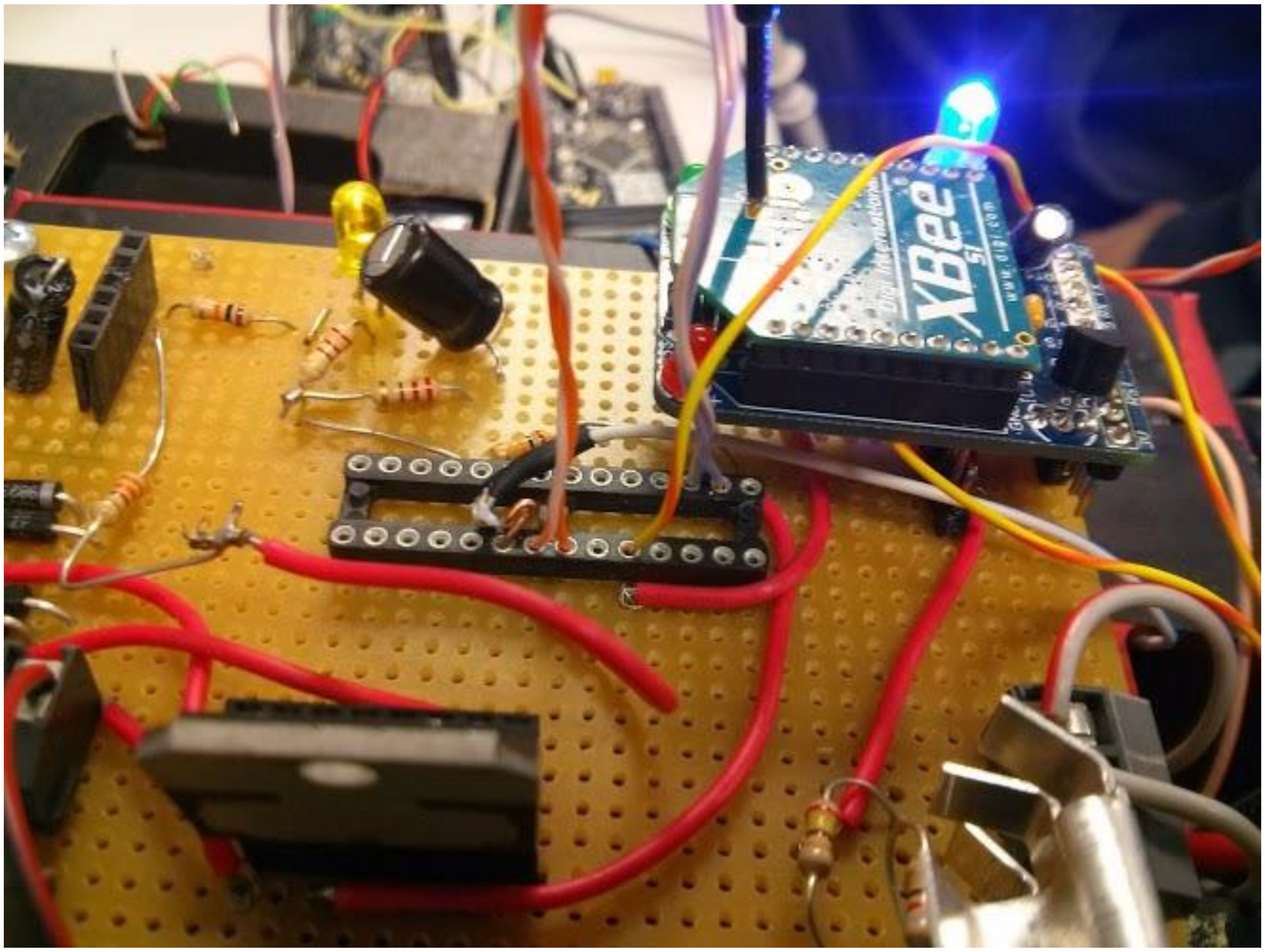

Figure 3. example of blue LED illuminated when the tank is ready

You are able to use the pitch and roll of the controller to control the movement of the tank. Familiarize yourself with the movement and be sure that you find an open area to ensure safe use of this Device.

System Overview

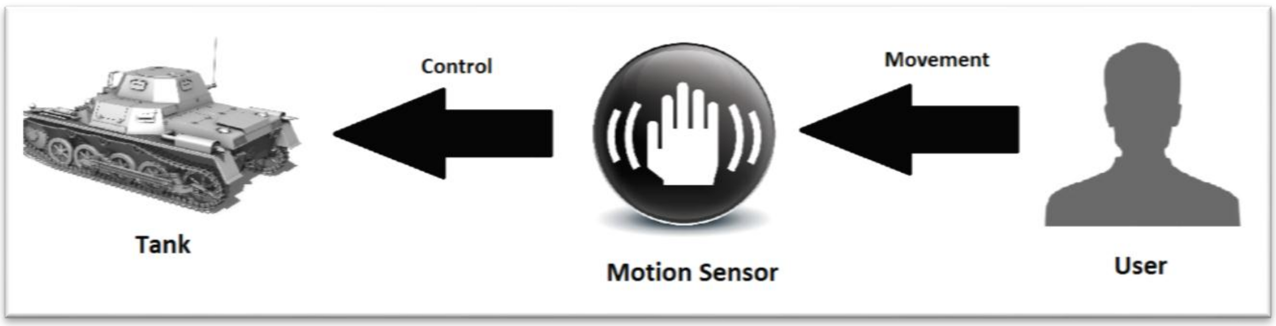

Figure 4. Objective I overview

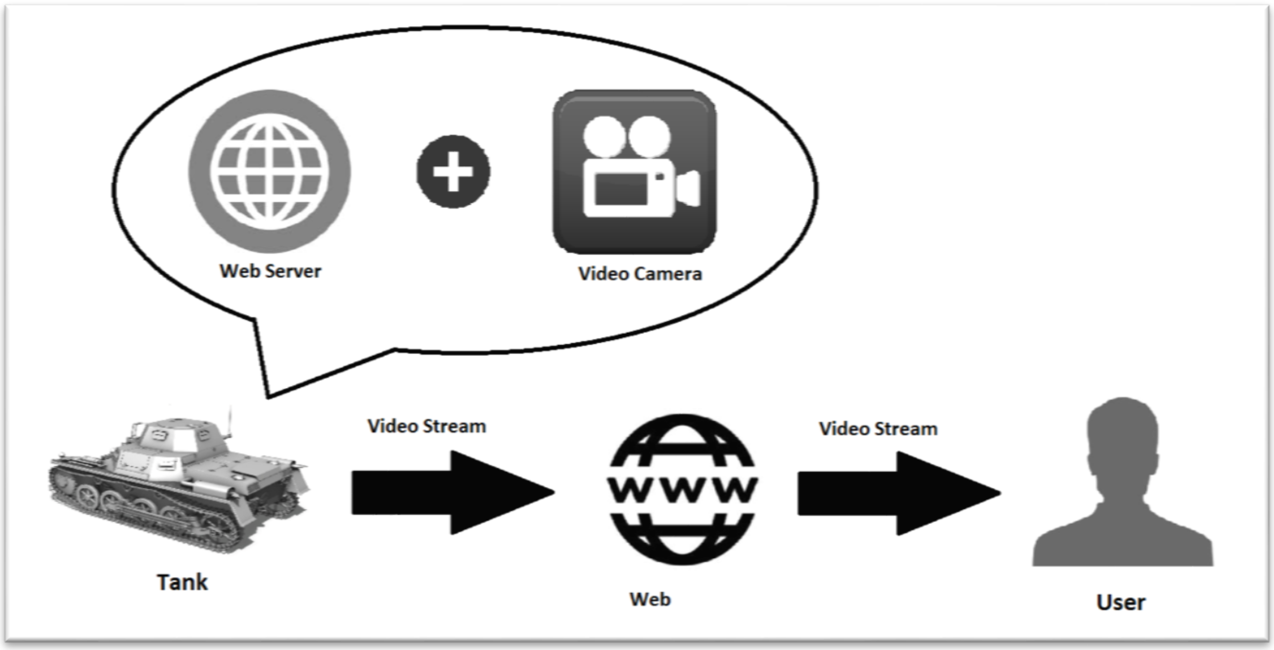

Figure 5. Objective II Overview

Internal Operation

BONE-KL25Z Tank

The tank unit had an embedded Beagle Bone Black microcontroller on the underside of its chassis. This provided us with the streamlined look and easy wiring. Connected to the BBB we have the WiFi module for communication, the XBee unit for serial data from the FRDM, motor controller wires to generate the PWM to the motor controller, and the +5V and ground lines to the motorcontroller.

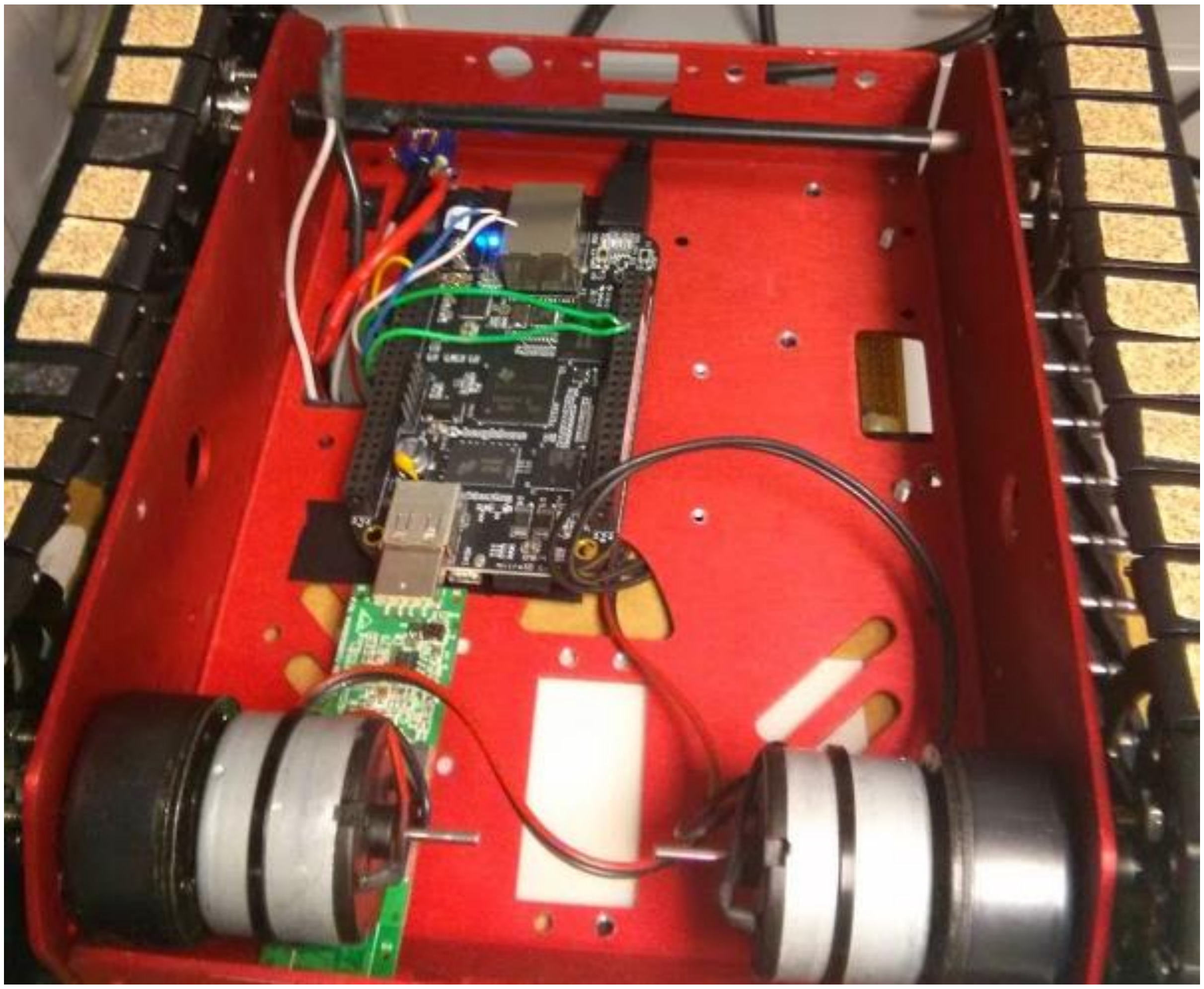

Figure 6. Internals of the Tank Device

BONE-KL25Z Tank Controller

We have the tank controller embedded into an FRDM box with easy assembly and battery change. We designated areas for the battery, XBee and the FRDM fit conveniently in the center. We also created an area for the switch to be easy accessible without opening the case.

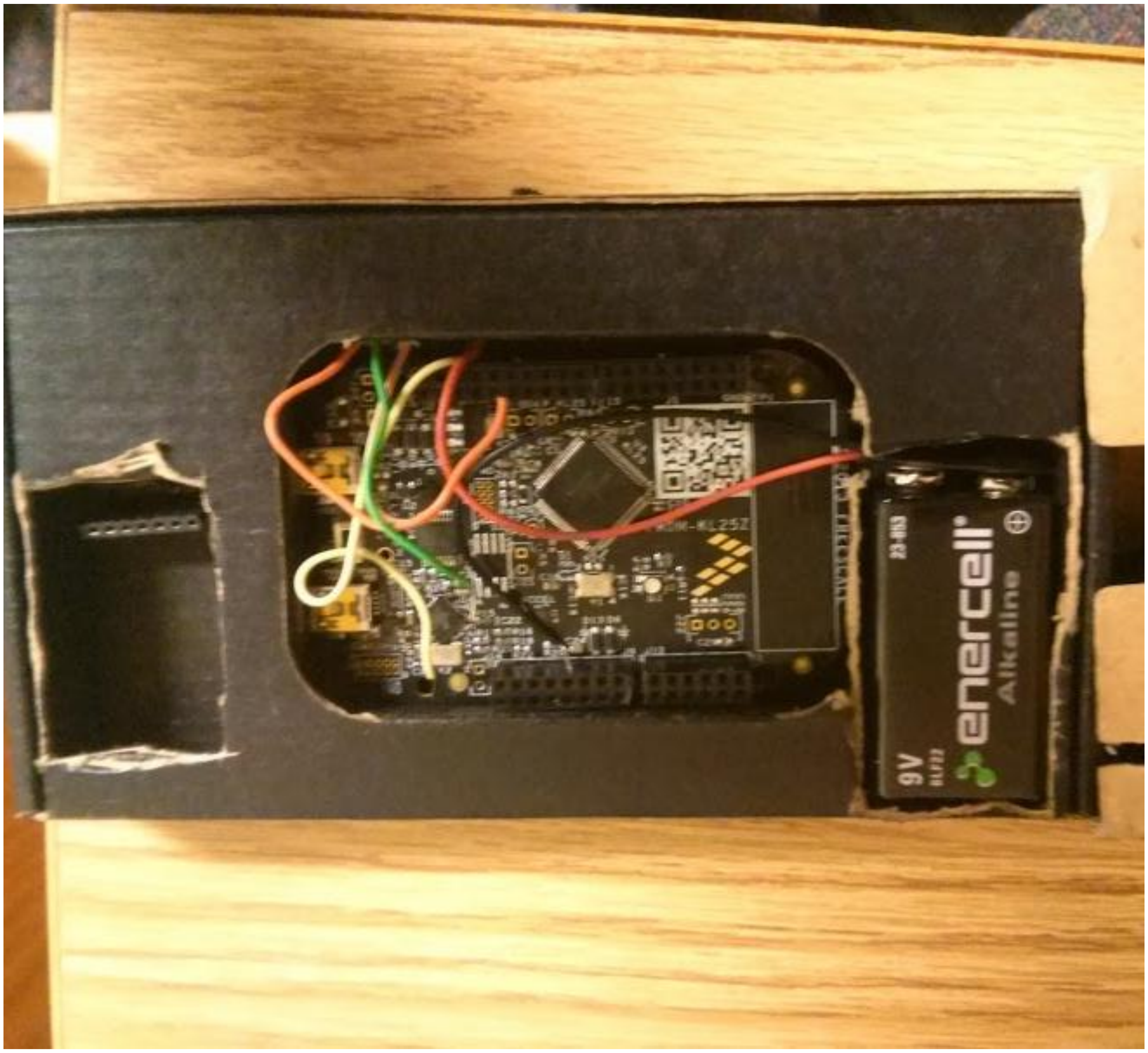

Figure 7. Internals of the FRDM controller

Testing Procedure

PWM – Our first objective was to program a successful PWM on both microcontrollers in case we had any unexpected surprises. We tested this by designating GPIOs to function as PWM ports and tested the outputs on the lab oscilloscope in order to verify the produced duty cycle. This test was an integral part for the motor controller in order to generate the correct voltage magnitudes to the motors.

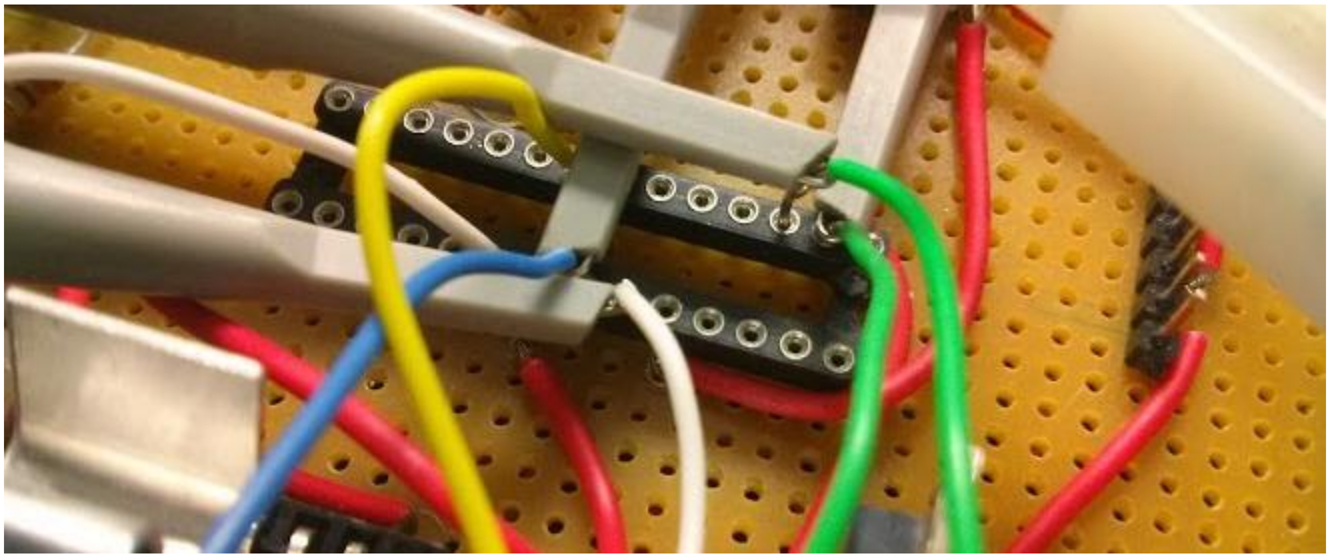

Figure 8. Oscilloscope probes testing the PWM outputs

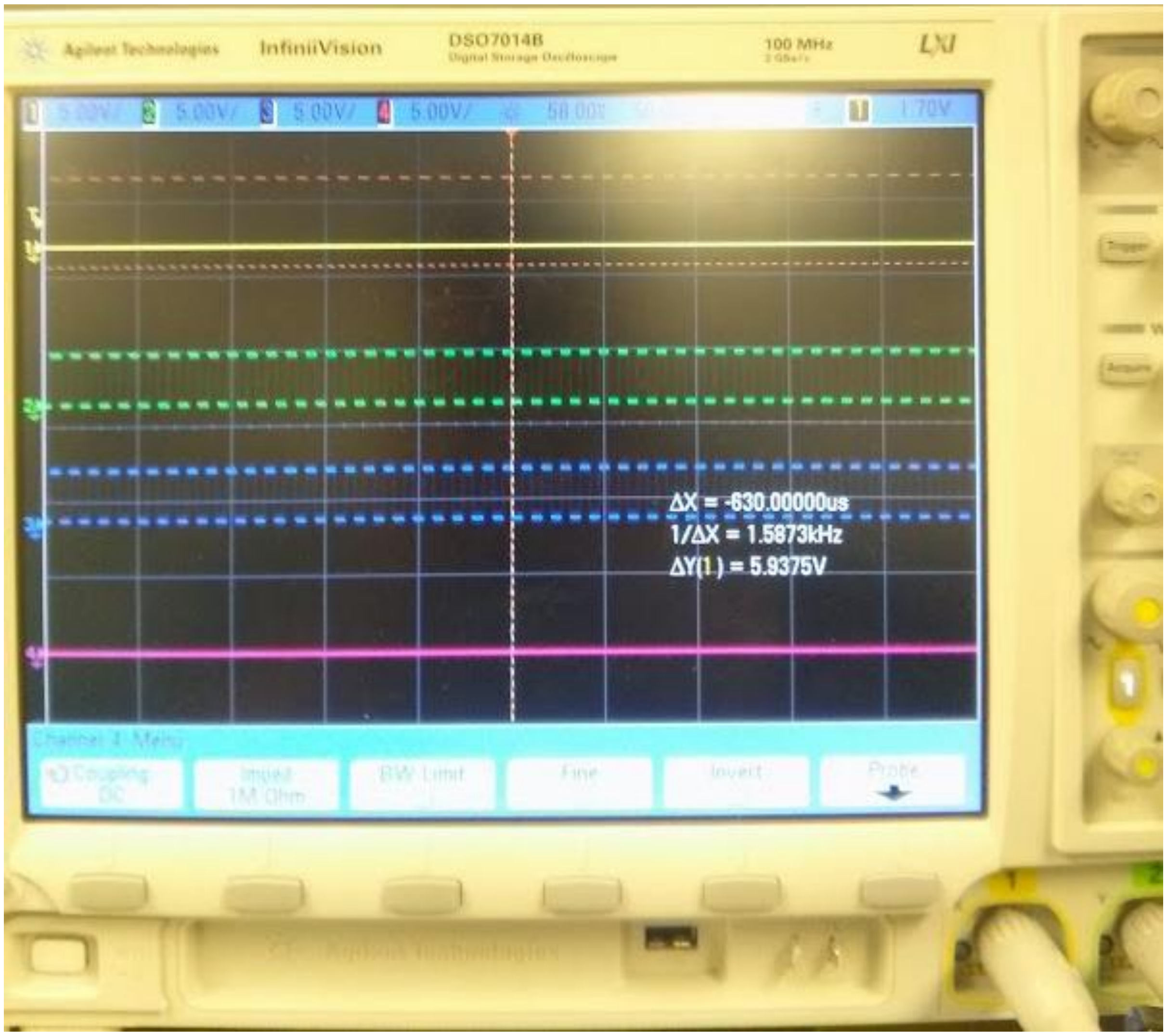

Figure 9. Oscilloscope sample showing successfully generated PWMs for the left motor(top) and right motor(bottom)

XBee/UART – Our second objective was to generate and receive UART Serial communication to talk with the XBee and transmit the motor controlling data in the form of a string. We tested this by setting up both XBees, sending data to one of them and receiving the data on the other XBee using the same microcontroller. This ensured that all data would be transmitted and received correctly. Proximity and direction of the antennas was also tested, we determined that the max distance the device could be controlled was about a meter and a half.

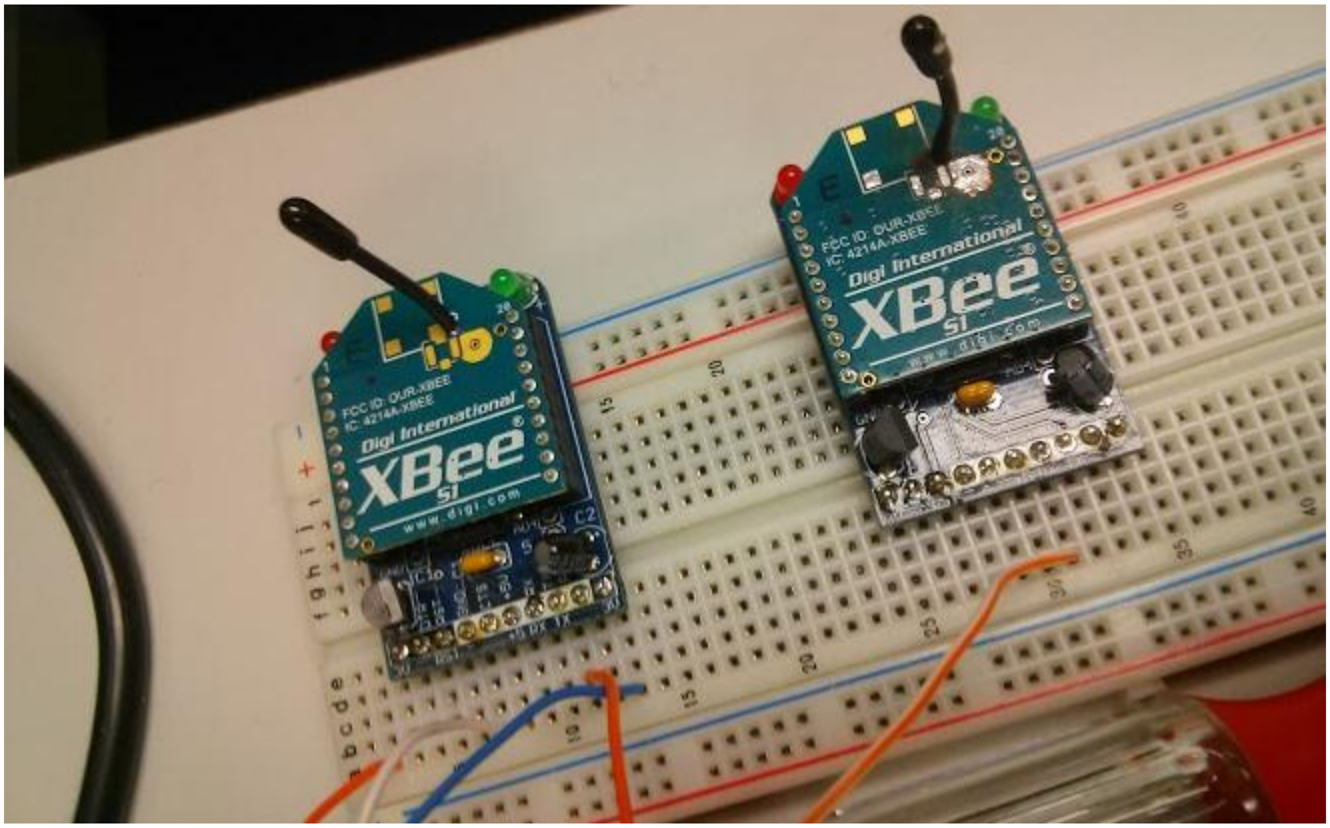

Figure 10. Preliminary testing for adjacent UART communication

Communication – Thirdly, communication was tested by setting the FRDM-KL25Z to send a constant duty cycle to the Beagle Bone Black. An oscilloscope probe was attached to the XBee to determine a successfully transmitted signal. During this test, we were able to get an accurate estimate for the operating conditions that best worked.

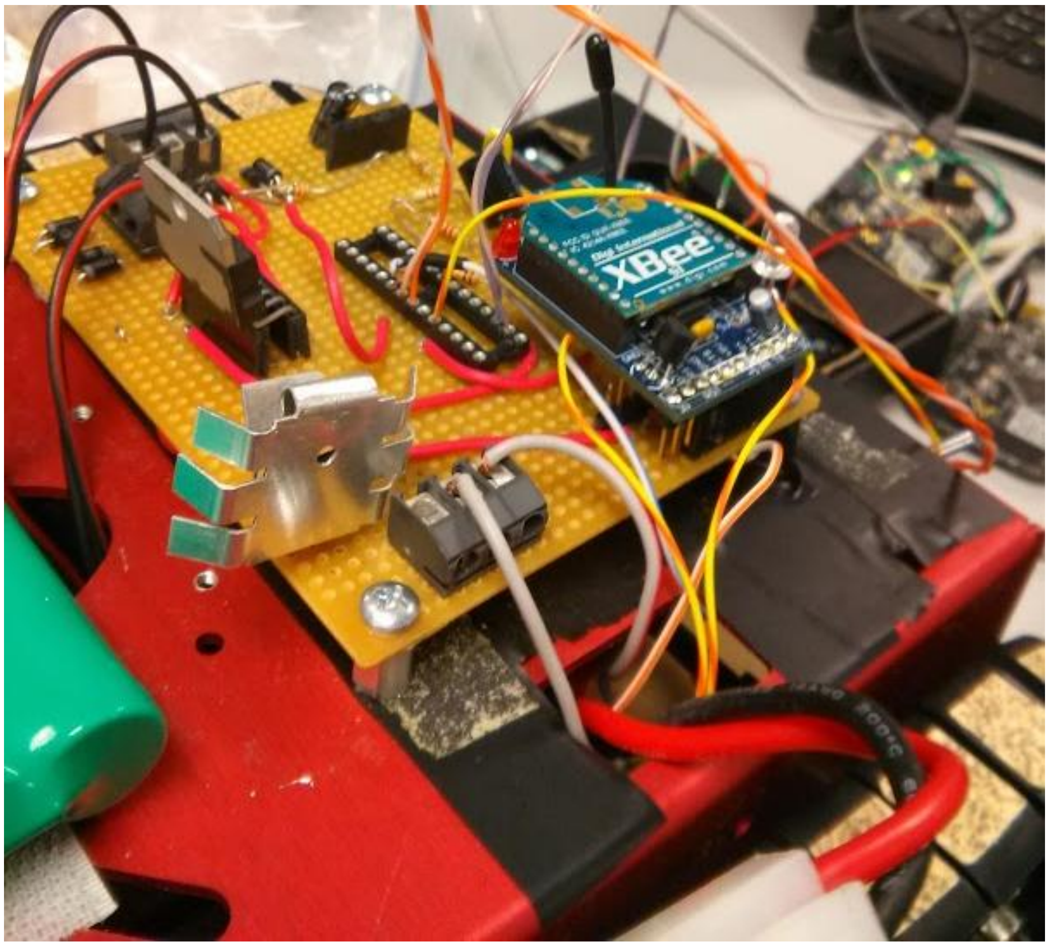

Figure 11. Final communication testing to determine best conditions

Accelerometer – Finally, accelerometer data was read from the FRDM and used to determine the duty cycles of the two motors. This was accomplished by using I2C to interface with the embedded accelerometer on the FRDM microcontroller. This was tested by hooking the tank up to the FRDM directly and testing the pitch and roll of the FRDM controller.

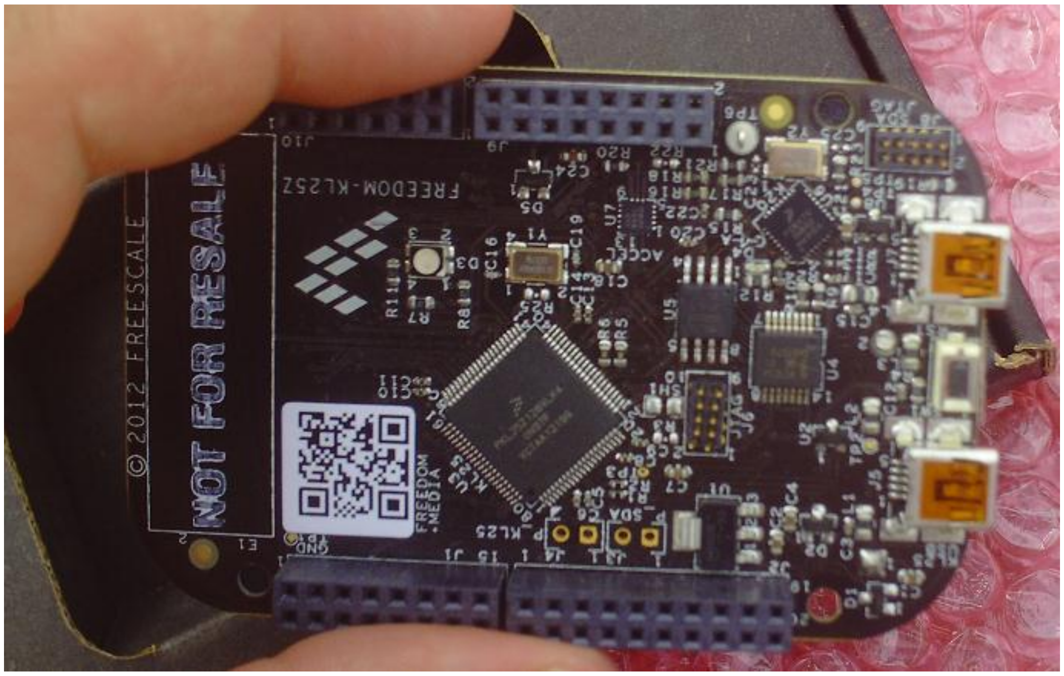

Figure 12. FRDM microcontroller with embedded accelerometer

Testing Results:

PWM Generation

We were able to generate and test a successful PWM on both microcontrollers, this was not only limited to a single PWM. We were able to generate up to 4 which included all the necessary PWMs for the project.

XBee/UART

We were able to generate and test successful UART serial communication over pins. Receiving and Transmitting on both microcontrollers.

Communication

Although initial communication was successful, the signal began to suffer at a range of about a few feet. After Speaking with Dr. Yoder, a sponsoring professor, we were able to determine that the main reason for the discrepancies and range loss was due to an antenna that suffered a fracture earlier this year.

Accelerometer

PWM and accelerometer functioned well together and we were able to get two wheels responding to the accelerometer data.

Schedule:

■
Projected Schedule

x
Time Spent(hours)

x
Time Spent(unanticipated hours)

	T	W	R	F	S	S	M	T	W	R	F	S	S	M	T
Task	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
• Objective I															
o Motion sensing of the FRDM	3	3	3	3	3										
o ZigBee's protocol	3	3	4	5	3										
o XBee's protocol					4	5									
o Communication					■	3	3	5							
o Embed the BBB on the tank							■							3	
o Embed the FRDM into unit							■							3	
o BBB's control of the tank							■	■	2	2	2	2	5	10	10

Bill of Materials:

Qty	Part	Description	Cost
1	Tank frame	The tank hardware	Borrowed
1	mcc7805ct	Voltage regulator used to convert the tanks onboard 9.6V to 5V for BBB.	Borrowed
1	FRDM-KL25Z	The FRDM microcontroller used to the motion control	Acquired
1	Beagle Bone	The Beagle Bone microcontroller used to control the tank and its features	Borrowed
1	Case for BBB	Case to house the Beagle Bone Black underneath the tank	Acquired
1	XBee	The XBee to make the communication between the microcontrollers	Borrowed
1	Camera	Viewport for determining tank's direction of travel. Used I2C and FIFO storage.	---
		Total:	$0

Final Results

Due to the complexities we discovered from the ZigBee we took the chance to switch to an XBee module with greater support for the Beagle Bone, FRDM, and UART. We then had further challenges from the XBee due to its damaged antenna and we were forced to resort to a "wired" approach. The tank continued to function correctly but the wireless aspect was lost.

Acknowledgements

Dr. Song – FRDM lend, debugging & parts

Dr. Yoder – BBB lend

Freescale Forums – FRDM support

Alex Alvarez – FRDM support

Mark Crosby – BBB 3D printed case

Chris Hopwood – BBB support

Code Source:

I2C – Accelerometer

```
#include <MKL25Z4.h>
#include "I2C_MMA8451_API.h"

void MMA8451_init(){
		uint8_t temp;

    temp = I2C_read(I2C_ADDR_MMA8451, MMA8451_CTRL_REG1);
    I2C_write(I2C_ADDR_MMA8451, MMA8451_CTRL_REG1, temp|0x01);
}

void I2C_init(){

		/*Clock configuration*/
		SIM_SCGC4 |= SIM_SCGC4_I2C0_MASK;
		//Enable clock for I2C0
		SIM_SCGC5 |= SIM_SCGC5_PORTE_MASK;
			//Enable clock for PORTE


   PORTE_PCR24 = (PORTE_PCR24 & ~PORT_PCR_MUX_MASK) | PORT_PCR_MUX(5);	//PORTE24 is
I2C0 SCL
   PORTE_PCR25 = (PORTE_PCR25 & ~PORT_PCR_MUX_MASK) | PORT_PCR_MUX(5);	//PORTE25 is
I2C0 SDA

    I2C0_F = I2C_F_ICR(0x14) | I2C_F_MULT(0x00);						//Sets
the frequency divider
		I2C0_C1 |= I2C_C1_IICEN_MASK;;// | I2C_C1_IICIE_MASK;
		//Enables I2C0

}
```

```
uint8_t I2C_read(uint8_t device_addr, uint8_t reg_addr){
        static int i;
        uint8_t data;

        I2C0_C1 |= (I2C_C1_MST_MASK | I2C_C1_TX_MASK);          //Set Master (start signal) and
Transmitter mode

        I2C0_D = (device_addr<<1) | I2c_write_bit;               //Send the address of the device
followed by write bit
        I2C_wait_flag();                                                  //Wait the
transmission to conclude

        I2C0_D = I2C_D_DATA(reg_addr);                                    //Select the
address in the register
        I2C_wait_flag();                                                  //Wait the
transmission to conclude

        I2C0_C1 |= I2C_C1_RSTA_MASK;                                      //Repeat start
        I2C0_D = (device_addr<<1) | 0x01;                           //Send the address of the
desired device followed by read bit
        I2C_wait_flag();                                                  //Wait the
transmission to conclude

        I2C0_C1 &= ~I2C_C1_TX_MASK;                                             //Set
Receiver mode

        I2C0_C1 |= I2C_C1_TXAK_MASK;                                      //Set nack
        data = I2C0_D;
        //Dummy read
        I2C_wait_flag();                                                  //Wait the
transmission to conclude

        I2C0_C1 &= ~I2C_C1_MST_MASK;                   //Stop signal

        data = I2C0_D;


        for(i=0;i<40;i++)
                asm("nop");

        return data;
}

void I2C_write(uint8_t device_addr, uint8_t reg_addr, uint8_t data){
        static int i;

        I2C0_C1 |= I2C_C1_TX_MASK | I2C_C1_MST_MASK;            //Set Master (start signal) and
Transmitter mode

        I2C0_D = (device_addr<<1) | I2c_write_bit;               //Send the address of the device
followed by write bit
        I2C_wait_flag();                                                  //Wait the
transmission to conclude

        I2C0_D = I2C_D_DATA(reg_addr);                                    //Select the
register in the device
        I2C_wait_flag();                                                  //Wait the
transmission to conclude
```

```
	I2C0_D = I2C_D_DATA(data);					//Send the data
	I2C_wait_flag();						//Wait the
transmission to conclude

	I2C0_C1 &= ~I2C_C1_MST_MASK;					//Stop signal

	for(i=0;i<40;i++)
		asm("nop");						//Wait
loop
}

void I2C_wait_flag(){
	while(!(I2C0_S & I2C_S_IICIF_MASK));

	I2C0_S |= I2C_S_IICIF_MASK;
}
```

PWM – Motorcontroller

```
#include <MKL25Z4.h>
#include "PWM_API.h"

void TPM0_CH_SetDutyCycle(uint8_t channel, uint16_t pulse_width){
	/*Set the TPM0_CnV register to set the duty cycle
	 * of the PWM wave generated in the specified channel
	 * */

	/*TPM Counter*/
	/*Clear the counter so CnV can be set (I don't know why it must be done)*/
	TPM0_CNT = TPM_CNT_COUNT(0x00);

	/*PWM Pulse Width*/
	/*TPM0_C0V: C0V=0x0000, initial duty cycke = 0% */
	TPM_CnV_REG(TPM0_BASE_PTR,channel) = TPM_CnV_VAL(pulse_width);
}

void PWM_init(){
/* Initialize the TPM0 to use channels 1, 2, 3 and 4 in the Edge-Aligned
 * PWM mode to generate two PWM waves. Set and use PORTA and PORTE as output.
 * PORTA4 = TMP0_CH1
 * PORTA5 = TPM0_CH2
 * PORTE30 = TPM0_CH30
 * PORTE31 = TPM0_CH4
 * */

	/*Clock configuration*/
	/*Enable clock for PORTA and PORTE*/
	SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTE_MASK;
	/*Enable clock source for TPM module*/
	SIM_SOPT2 |= SIM_SOPT2_TPMSRC(0x01);
	/*Enable clock for TPM0*/
	SIM_SCGC6 |= SIM_SCGC6_TPM0_MASK;

	/*TPM0 configuration*/
	/*Status and Control Register*/
	/* TPM0_SC: ??=0,DMA=0,TOF=0,TOIE=0,CPWMS=0,CMOD=1,PS=2 */
	TPM0_SC = (TPM_SC_CMOD(0x01) | TPM_SC_PS(0x02));
	/* Reset counter register */
	/* TPM0_CNT: COUNT=0x0000 */
	TPM0_CNT = TPM_CNT_COUNT(0x00);
	/*PWM period*/
	/* TPM0_MOD: MOD=0xffff */
	TPM0_MOD = TPM_MOD_MOD(0x0fff);

	/*TPM0 Channels configuration*/
	/*Edge-Aligned PWM mode*/
	/*TPM0_C0SC: ??=0,CHF=0,CHIE=0,MSB=1,MSA=0,ELSB=1,ELSA=0,??=0,DMA=0 */
	TPM0_C1SC = (TPM_CnSC_CHIE_MASK | TPM_CnSC_MSB_MASK | TPM_CnSC_ELSB_MASK);
	TPM0_C2SC = (TPM_CnSC_CHIE_MASK | TPM_CnSC_MSB_MASK | TPM_CnSC_ELSB_MASK);
	TPM0_C3SC = (TPM_CnSC_CHIE_MASK | TPM_CnSC_MSB_MASK | TPM_CnSC_ELSB_MASK);
	TPM0_C4SC = (TPM_CnSC_CHIE_MASK | TPM_CnSC_MSB_MASK | TPM_CnSC_ELSB_MASK);
	/*PWM Pulse Width*/
	/*TPM0_C0V: C0V=0x0000, initial duty cycke = 0% */
	TPM0_C1V = TPM_CnV_VAL(0x4fff);
	TPM0_C2V = TPM_CnV_VAL(0);
	TPM0_C3V = TPM_CnV_VAL(0x4fff);
	TPM0_C4V = TPM_CnV_VAL(0);

	/*Pin Control Register*/
	/*ISF=0 (no interrupt); MUX=3 (TPM pins are Alt 3)*/
	PORTA_PCR4 = ((PORTA_PCR4 & ~(PORT_PCR_ISF_MASK | PORT_PCR_MUX(0x04))) |
(PORT_PCR_MUX(0x03)));
	PORTA_PCR5 = ((PORTA_PCR5 & ~(PORT_PCR_ISF_MASK | PORT_PCR_MUX(0x04))) |
(PORT_PCR_MUX(0x03)));
	PORTE_PCR30 = ((PORTE_PCR30 & ~(PORT_PCR_ISF_MASK | PORT_PCR_MUX(0x04))) |
(PORT_PCR_MUX(0x03)));
	PORTE_PCR31 = ((PORTE_PCR31 & ~(PORT_PCR_ISF_MASK | PORT_PCR_MUX(0x04))) |
(PORT_PCR_MUX(0x03)));
}
```

Serial Communication – XBee

```
#include "MKL25Z4.h"

void init_tx(){
   /*Enable clock for peripherals*/
	SIM_SOPT2 |= SIM_SOPT2_UART0SRC(0x01);
	SIM_SOPT2 &= ~SIM_SOPT2_PLLFLLSEL_MASK;

	SIM_SCGC4 |= SIM_SCGC4_UART0_MASK;
	SIM_SCGC5 |= SIM_SCGC5_PORTE_MASK;
	SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK;

	/*PORTE20 ISF=0 (no interrupt); MUX=4 (UART0 pins are Alt 4)*/
	PORTE_PCR20 = ((PORTE_PCR20 & ~(PORT_PCR_MUX_MASK)) |
(PORT_PCR_MUX(0x04)))|PORT_PCR_ISF_MASK;
	//PORTE_PCR21 = ((PORTE_PCR21 & ~(PORT_PCR_MUX_MASK)) |
(PORT_PCR_MUX(0x04)))|PORT_PCR_ISF_MASK;

   /* PORTA_PCR1: ISF=0,MUX=2 */
	PORTA_PCR1 = (uint32_t)((PORTA_PCR1 & (uint32_t)~0x01000500UL) | (uint32_t)0x0200UL);
    /*PORTA_PCR2  ISF=0,MUX=2 */
	PORTA_PCR2 = (uint32_t)((PORTA_PCR2 & (uint32_t)~0x01000500UL) | (uint32_t)0x0200UL);

	UART0_C2 &= ~(UART0_C2_TE_MASK | UART0_C2_RE_MASK);
	  UART0_BDH = 0x01;
	  UART0_BDL = 0x38;
	  UART0_C4 = 0x06;
	  UART0_C1 = 0x00;
	  UART0_C3 = 0x00;
```

```
                UART0_MA1 = 0x00;
                UART0_MA1 = 0x00;
                UART0_S1 |= 0x1F;
                UART0_S2 |= 0xC0;
               UART0_C2 |= UART0_C2_TE_MASK | UART0_C2_RE_MASK;

}

void init_rx(){
    /*Enable clock for peripherals*/
    SIM_SCGC4 |= SIM_SCGC4_UART0_MASK;
//          SIM_SCGC5 |= SIM_SCGC5_PORTE_MASK;
    SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK;

               /*PORTE21 ISF=0 (no interrupt); MUX=4 (UART0 pins are Alt 4)*/
               PORTE_PCR21 = ((PORTE_PCR21 & ~(PORT_PCR_ISF_MASK | PORT_PCR_MUX_MASK)) |
(PORT_PCR_MUX(0x04)));

    /* PORTA_PCR1: ISF=0,MUX=2 */
//    PORTA_PCR1 = (uint32_t)((PORTA_PCR1 & (uint32_t)~0x01000500UL) | (uint32_t)0x0200UL);
    /* PORTA_PCR2: ISF=0,MUX=2 */
//    PORTA_PCR2 = (uint32_t)((PORTA_PCR2 & (uint32_t)~0x01000500UL) | (uint32_t)0x0200UL);

               UART0_C2 |= UART0_C2_RE_MASK;
}

void send_character(char c){
    while(!(UART0_S1&UART_S1_TDRE_MASK) && !(UART0_S1&UART_S1_TC_MASK));
    UART0_D  = c;
}

char receive_character(){
               char c;

               while(!(UART0_S1&UART_S1_RDRF_MASK));
               c = UART0_D;
}
```