University of British Columbia

Department of Electrical and Computer Engineering

ELEC 291/292 L2A 2023 W2: Electrical Engineering Design Studio I

Dr. Jesus Calvino-Fraga P.Eng

Project 2 – Metal Detector Robot

Group Number: A2

Name	Student Number	Score %	Signature
Jeffrey He	32613788	120	JH
Yuqian Song	90293630	110	YS
Sidharth Sudhir	11255635	100	SS
Ao Sun	25909979	90	AS
Harry Zhang	13013727	90	HZ
Sam Zhao	30678635	90	SZ

Date of Submission: April 12th, 2024

Table of Content

1. Introduction

The primary objective of this project is to develop a robot that can effectively detect metal objects using a joystick controller. The robot, along with its remote controller, are both powered by batteries and use distinct microcontroller systems from different families to ensure robust control and functionality. The transmitter utilizes an EFM8 microcontroller, while the robot is controlled by STM32F1 coded in C language.

Some requirements of the robot include being able to move smoothly in various patterns such as '8', 'square', and 'I', and responding to changes in metal proximity with appropriate audio-visual signals. The robot's movements are facilitated by two-gear motors and controlled via a joystick. We integrated a sonar sensor into the car's safety system to prevent collisions by detecting obstacles, enhancing its autonomous capabilities. The car features multiple operational modes, including a self-driving option, which can be seamlessly toggled using a joystick, allowing for versatile and adaptive handling under various conditions. This project not only challenges our technical skills but also enables practical learning through the assembly and programming of a fully functional robotic system. This report will cover the comprehensive steps taken from conceptualization to the final demonstration, reflecting both the theoretical and practical aspects of building a sophisticated embedded system.

2. Investigation

2.1. Idea Generation

In the idea generation phase, our group met together and brainstormed collaboratively to determine our strategies for tackling this project. Through the experience and knowledge gained from completing lab 5 and lab 6, we learned some basics of using C to code different microcontrollers and effectively use makefiles to streamline our code compilations.

Our brainstorming sessions led to multiple potential solutions for each component of the project, from motor control to metal detection and radio communication. Each idea was evaluated based on feasibility, resource availability, and integration capability with other components, leading to the selection of the most promising solutions for development. For example, our team hypothesized that the inconsistencies in radio transmission might be due to interference caused by multiple radios operating simultaneously in the vicinity. This interference could potentially disrupt the signal integrity between the remote and the robot. To mitigate potential interference, we implemented a private channel for the exclusive use of our project's JDY-40 radios by setting unique frequencies. We worked towards the goal that the robot moved smoothly and responded quickly to control inputs and metal detection, which were critical to the project's success.

2.2. Investigation Design

Unlike the labs we have done, the metal-detecting robot required comprehensive research, thorough analysis, and a series of practical tests. Our team made sure that our design decisions were supported by solid data. The guidance from our instructor's notes was particularly crucial, clarifying the basic concepts of the H-bridge and the specific needs of the controller, which shaped the early phases of our design. Furthermore, a detailed examination of the datasheets for the components allowed us to understand their capabilities, constraints, and how they could be best integrated. This careful selection and analysis of components guaranteed their compatibility and maximized the overall efficiency of the system.

When addressing hardware problems, especially those related to the accuracy of temperature measurements, we consulted datasheets extensively for direction. Additionally, online platforms like technical forums and community discussions were crucial in troubleshooting and providing solutions to specific issues. These resources offered professional insights into programming in C, circuit design, and debugging methods, enhancing our approach to problem-solving. They also presented optimization techniques that went beyond the guidance in our instructor's materials or the datasheets, greatly enriching our development process.

2.3. Data Collection

For data collection, we relied on essential tools including a multimeter, oscilloscope, and function generator. During the testing and debugging phases, these instruments, combined with lab facilities, enabled comprehensive data gathering. Initially, we closely monitored motor power to validate its performance. Ensuring data accuracy involved conducting tests, documenting results, and comparing readings for consistency. More insights into our data synthesis procedures are elaborated in the Data Synthesis section below.

2.4. Data Synthesis

- Our team employed a methodical approach in synthesizing data during the testing phase of our metal detector robot, using it to enhance system functionality and reliability. This involved monitoring waveform changes, verifying electrical parameters, and debugging software outputs.
- We utilized Oscilloscopes and Multimeters to monitor the Colpitts oscillator for waveform changes and to ensure components stayed within safe operational limits.

- During our design process, Putty Served as a terminal emulator for real-time monitoring and debugging, crucial for testing software functionalities and communication protocols.
- Components were methodically integrated, from basic motor functions to complex systems like the metal detector and communication modules. This step-by-step approach helped isolate and address issues effectively.

2.5. Analysis of Results

We carefully examined our results to minimize errors within the expected hardware tolerances. An important part of our approach was validating our findings through actual robot operations in real-world conditions. We assessed the functionality of our robot commands—including moving forwards, backward, and turning—to ensure our design performed reliably across a range of test scenarios.

Our group's assessment of the validity of the conclusions drawn from the results involves a multifaceted approach that focuses on the extent of error and the limitations of our theoretical understanding and measurement capabilities.

3. Design

3.1. Use of Process

To ensure our robot not only met all basic functionalities but also included extra features, our group diligently adhered to the engineering design process. We meticulously followed each step of the process to maximize the robot's effectiveness. The first step involved identifying the problems the robot was intended to solve and detailing the proposed solutions. These challenges ranged from constructing essential components like the JDY-40 and the joystick to mounting the circuit on the robot and establishing communication between the robot and the remote, among others.

To tackle these issues effectively, we organized ourselves into three pairs: one focused on building the robot and its circuitry, another on controlling the robot and its communication with the remote, and the third on the remote control system and its display. Each subgroup specialized in solving their respective issues and then collaborated to integrate their solutions, particularly where system interdependencies like communication protocols were crucial.
We employed evaluation metrics to objectively assess each proposed solution, scoring them to determine the most effective. After selecting the best solutions, we proceeded to develop and test them. This phase, like many others, required iterative revisions due to unforeseen challenges, involving extensive debugging of our code, reassembling the circuit, and troubleshooting components like the JDY-40.

Having refined our design through brainstorming, solution generation, evaluation, testing, and revision, we confirmed the robot's functionality and moved on to add extra features. The implementation of these additional features followed a similar, albeit slightly less rigorous, process as that of the main robot development.

3.2. Need and Constraint Identification

Our team started by thoroughly reviewing the lab manual and consulting with the professor to clarify any ambiguities, ensuring a comprehensive understanding of the project's scope. The fundamental requirements for the metal detector robot included:

Dual Microcontroller Systems: Utilizing two different microcontroller families to independently control the robot and the remote.

Battery Operation: Both the robot and the remote needed to be battery-powered, adhering to the specifications provided.

Metal Detection: Designing a metal detector capable of detecting all types of Canadian coins, with detection sensitivity indicated through variable beep frequencies from the remote's speaker.

Remote Interface: Incorporating an LCD to display the metal detection strength and a speaker to indicate detection events.

Communication: Employing JDY-40 radio modules for reliable communication between the robot and the remote.

Movement Patterns: Enabling the robot to navigate smoothly in complex patterns ('8', 'I', and square) with adjustable speeds and minimal drift.

Addressing Constraints and Scheduling

Time Management:

- **Course Load and Scheduling Conflicts:** With two team members enrolled in third-year courses, coordinating work schedules was challenging as we shared different class schedules.
- **Midterm Examinations:** A compact midterm schedule influenced our project timeline. We planned intensive work sessions post-midterms to make optimal use of the remaining time.
- **Holiday and Personal Commitments:** Absences from personal issues and holidays such as during the Easter holiday, required us to accelerate work in advance to keep the project on track.

Technical Constraints:

- **Radio Module Reliability:** We faced issues with one of the JDY-40 modules' reliability, which necessitated adjustments in our code to maintain effective communication.

Strategic Planning

To mitigate these constraints, our planning included:

Advanced Preparation: Completing as much work as possible in the available lab hours and scheduling extra sessions when needed.]

Code Robustness: Enhancing the communication code to accommodate the less reliable JDY-40, ensuring no loss in functionality.

By carefully defining the robot and remote's requirements and proactively addressing potential constraints, our team established a solid foundation for the project. This strategic approach not only facilitated smooth development phases but also optimized our efforts amidst scheduling challenges and technical issues, setting a clear path toward achieving our project goals.

3.3. Problem Specification

From the specifications outlined in the project manual, we clearly defined how to tackle each requirement. Given the need for two different microcontrollers, we chose the EFM8 for its familiarity and the STM32 for its proven reliability from previous labs. In constructing the robot, we adhered to a detailed tutorial for building the components and decided to include four additional LEDs to indicate turning and reversing actions—two as headlights and two as taillights.

Moreover, to compensate for the motor's uneven power distribution to the wheels, we introduced a "PWM" parameter in our code. This adjustment allowed us to control the power supplied to each wheel, ensuring smooth operation with minimal drifting. For metal detection, we implemented a variable named “diff” to measure the metal's signal strength, enabling the speaker to emit beeps at varying frequencies based on this strength. We established four strength levels: Level 1 indicated no metal (no beep), Level 2 a small metal object like a dime (400ms beep), Level 3 a larger object like a loonie or toonie (200ms beep), and Level 4 significant metal presence such as a stack of coins (100ms beep).

For the remote control and movement of the robot, we utilized a master-slave setup, where the robot communicated the detected metal's strength only upon request from the remote. The remote’s LCD displayed this strength and the robot’s speed (adjustable through three-speed settings), controlled via both joystick and buttons.

3.4. Solution Generation

Selection of Microcontrollers and Component Integration

We chose the EFM8 and STM32 microcontrollers for their familiarity, which expedited the initial development phases. To address the incompatibility of JDY-40 radio modules and the PS2 joystick with our breadboards, we explored two connection methods: using jumper wires and soldering metal wires directly onto the components' pins. We opted for soldering, which provided a more reliable and secure connection.

Communication Protocol and Data Transmission

For the robot and remote communication, we initially tested continuous signal exchange but encountered issues with loops becoming stuck. Following our professor's recommendation, we implemented a master-slave relationship, which proved to be more efficient. In transmitting data from the robot's metal detector, we initially considered sending raw frequency data but ultimately chose to send a calculated 'strength' measurement, which was more intuitive and streamlined the code.

Motor Control and Movement Optimization

Initially, we hardcoded specific power values to control the motors, which was ineffective due to variability in motor performance and battery depletion. We switched to a variable power setting system, allowing dynamic adjustments to accommodate motor inconsistencies and maintain smooth operation, especially necessary for the robot’s complex navigation patterns.

3.5. Solution Evaluation

Given that most of our proposed solutions met the project's needs, our team developed a set of evaluation criteria to ascertain the most effective solutions. These parameters were functionality, accuracy, efficiency, and clarity, arranged in order of importance. Each solution was rated on a scale from 1 to 5, with 1 indicating poor performance and 5 representing excellence.

In the construction of the robot and remote, using jumper wires was compared to soldering. While jumper wires rated high in terms of efficiency and clarity for their ease of use, they fell short in accuracy and functionality. Although quick to set up, jumper connections were less reliable and effective over time compared to soldered connections, which provided long-term stability and performance.

Regarding robot communication, the master-slave setup was superior to the waiting-in-loops approach across all criteria. It facilitated shorter, clearer code, consistently bug-free operations, and quicker responses due to minimized code delays.

For transmitting information from the robot to the remote, sending signal strength was slightly less accurate than transmitting exact frequencies due to potential calculation or measurement errors. However, it scored better in functionality, efficiency, and clarity, making it the preferred method for balancing performance with ease of use.

Lastly, in terms of robot movement, using a fixed power setting did not meet the basic project requirements and thus was not further considered. Our consensus was that a variable, adjustable power setting was necessary and most effective, aligning with our highest standards for functionality, accuracy, efficiency, and clarity.

The final design was chosen to integrate these top-scoring solutions, ensuring that the robot operated as effectively as possible. For detailed scores and evaluations, please see Figure 1 in the appendix.

3.6. Safety/Professionalism

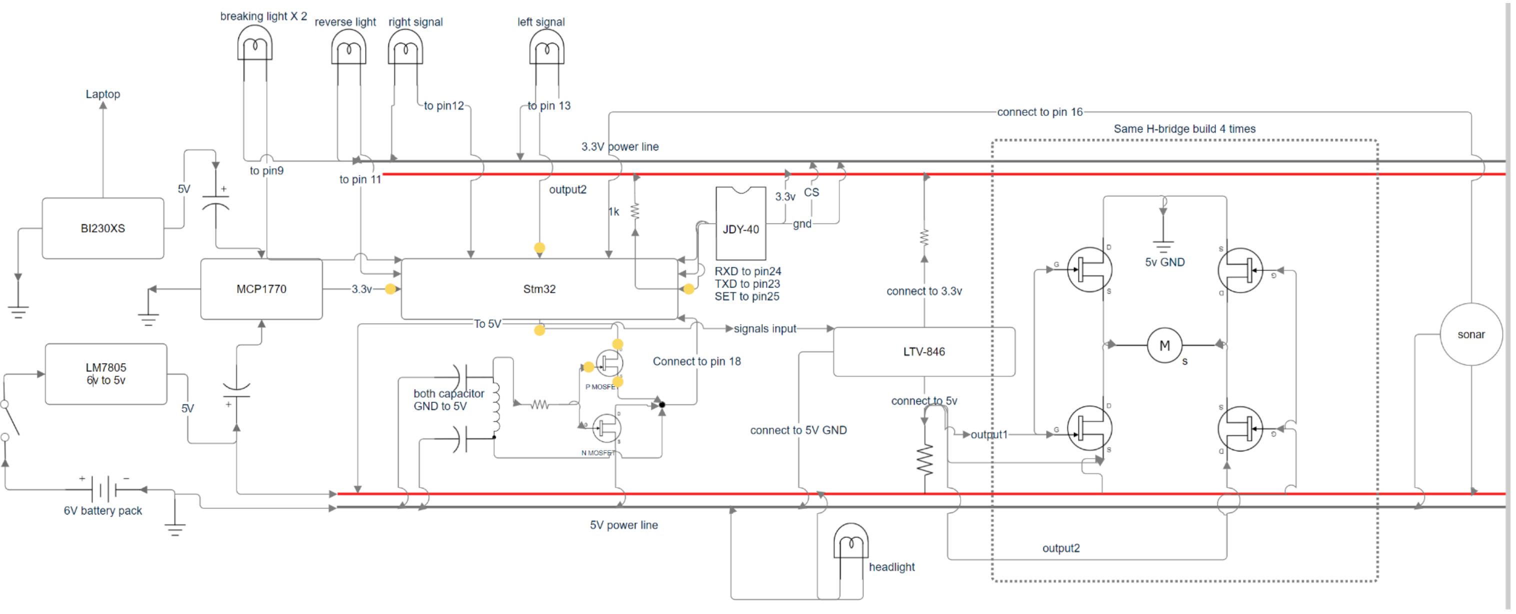
While constructing the robot, ensuring the safety of the team and users was the top priority for our group. Protective measures like wearing safety goggles when soldering or cutting wires were strictly adhered to, and we always worked in teams of two for better attention. Additionally, maintaining a tidy workspace was critical to avoid potential hazards not just for our team members, but for anyone in the surrounding. This involved promptly disposing of any unwanted parts to eliminate tripping hazards, such as a robot left on the ground. In terms of the robot's design, we incorporated safety components such as resistors linked to LEDs and other outputs to minimize the risk of overheating or damage.

Furthermore, we installed a switch on the battery to cut power when not in use, preventing accidental engagements with other elements.

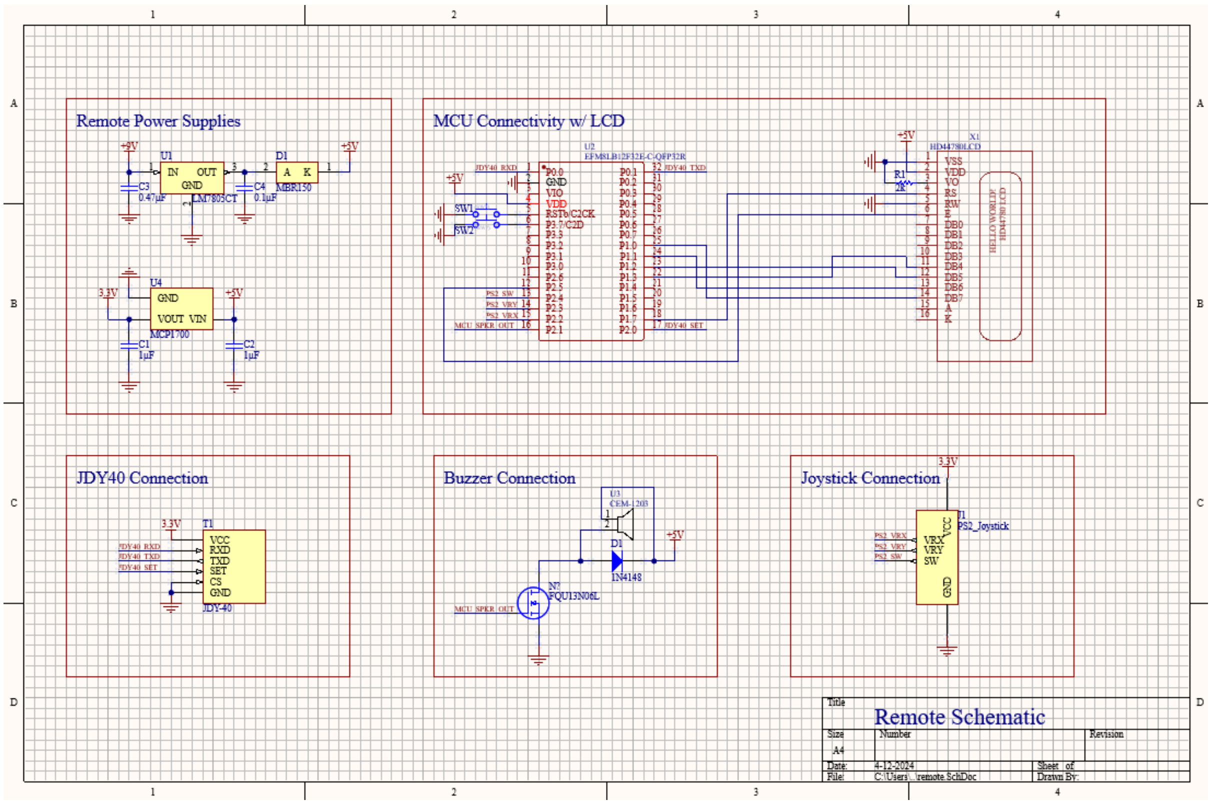
4. Detailed Design

4.1. Hardware Block Diagram

4.1.1. Robot Car Circuit Diagram



4.1.2. Remote Circuit Diagram



4.2. Circuit explained

The robot circuit can be separated into four segments: power supply and connection to laptop, microcontroller, Communications, and Robot power system.

4.2.1. Robot Car Circuit explained

4.2.1.1. Power supply

The power supply consists of a 6-volt battery pack, which is connected to a switch, and a 6v to 5v voltage regulator LM7805. The 5v from the LM7805 is then sent to MCP1770, which transforms 5v into 3.3v and connects to the upper power line on the breadboard. The bottom power line is connected with the 6v battery. The serial port BI230XS can also supply 5v of voltage, so we need to connect it with the MCP1770 to transform it to 3.3 volts. There are also two diodes before the voltage enters MCP1770 to avoid too much voltage entering the regulator.

4.2.1.2. Microcontroller

We choose STM32 to be the microcontroller for our robot car. The microtroller took 3.3 volts and is also connected to the 3.3-volt ground.

4.2.1.3. Communication

We used jdy40 for our communication, which requires a 3.3V power supply, and the connected pin is connected as shown in the diagram.

4.2.1.4. Robot power system

The power system is the most complicated part of our circuit. First, we need to use LTV-846 to filter out the noise from the rest of the circuit. Next, we need to build an H-bridge for each circuit, each H-bridge is consistent with a P and an N MOSFET. Using the property of MOSFET, we can change our input and make the current flow in both directions, allowing the motor to rotate forward it backward.

4.2.1.5. Light system

We have a light system to simulate everyday driving, including braking light, headlight, reversed light, and the Turing signal. We solder wire the resistor and the light bulbs together to hide the light's wiring under the breadboard.

4.2.1.6. Sonar

The sonar is connected to 5v straight from LM7805, then connected to the 6v ground, the output will be a pulse wave, which we can use to calculate the distance.

4.2.2. Remote Circuit explained

4.2.2.1 Power Supply

The power supply consists of a 9-volt battery pack, which is connected to a 9v to 5v voltage regulator LM7805. The 5v from the LM7805 is then sent to MCP1770, which transforms 5v into 3.3v and connects to JDY-40, 5V also powers our EFM8 chip directly.

4.2.2.2 LCD Display

The EFM8 is attached to an LCD that serves as the visual feedback element for the user. This display indicates the presence of metal detected by the robot and conveys the intensity of the PWM signals that determine the robot's speed and direction. Additionally, the LCD outlines the robot's current operating mode, whether it be manual mode or automatic mode.

4.2.2.3 Buzzer

A buzzer is interfaced with the microcontroller and provides auditory cues to assist in determining the metal's proximity. Varied tones from the buzzer indicate how close the robot is to the target—higher tones mean closer distance and lower tones suggest it's farther away.

4.2.2.4 PS2 Joystick

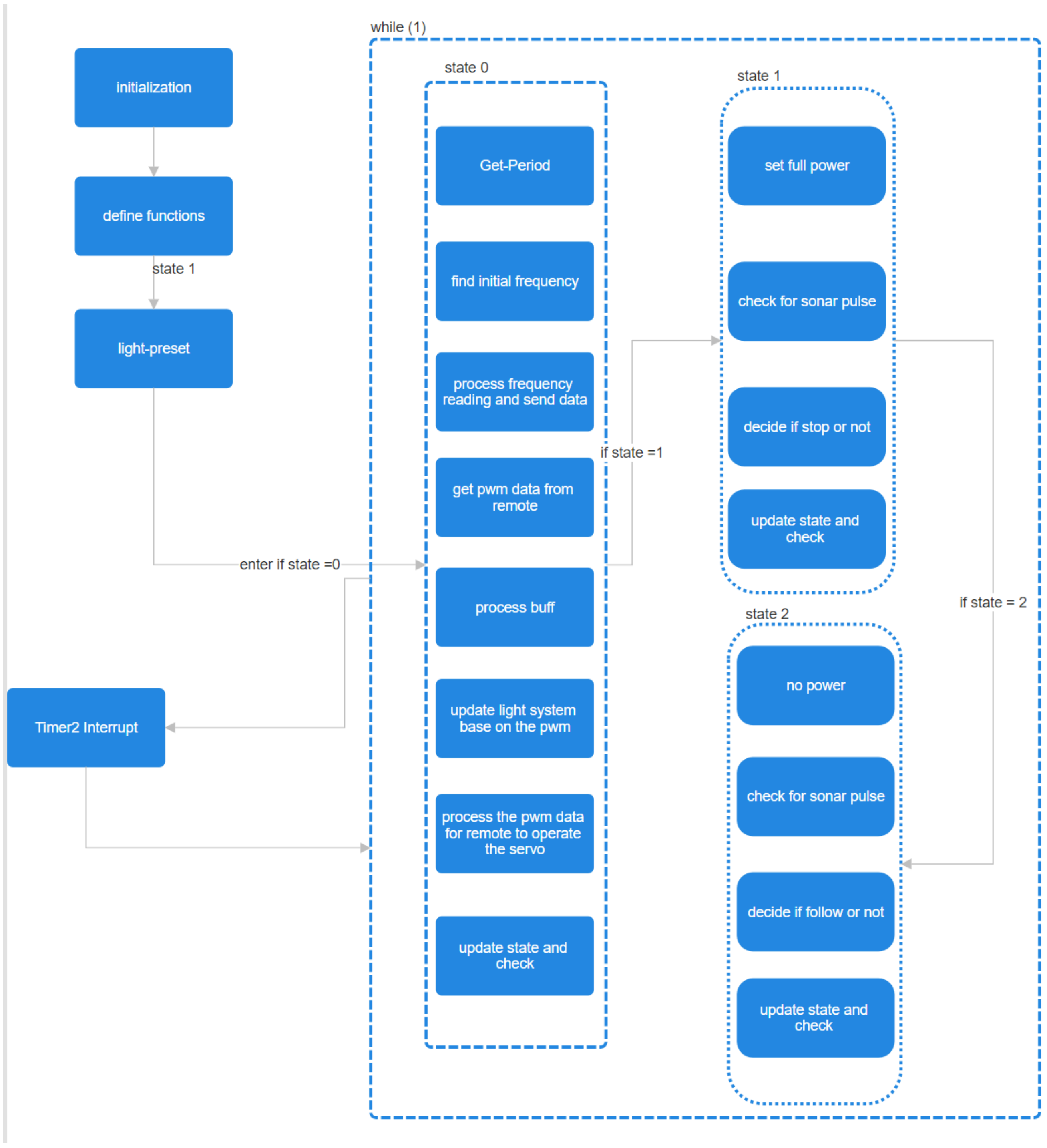
User input for manual control comes through a PS2 joystick, which relays PWM commands to the robot through its VRx and VRy pins. The joystick's SW pin has a dual role, acting as a button for toggling between the robot's operational modes.

4.2.2.1 JDY40

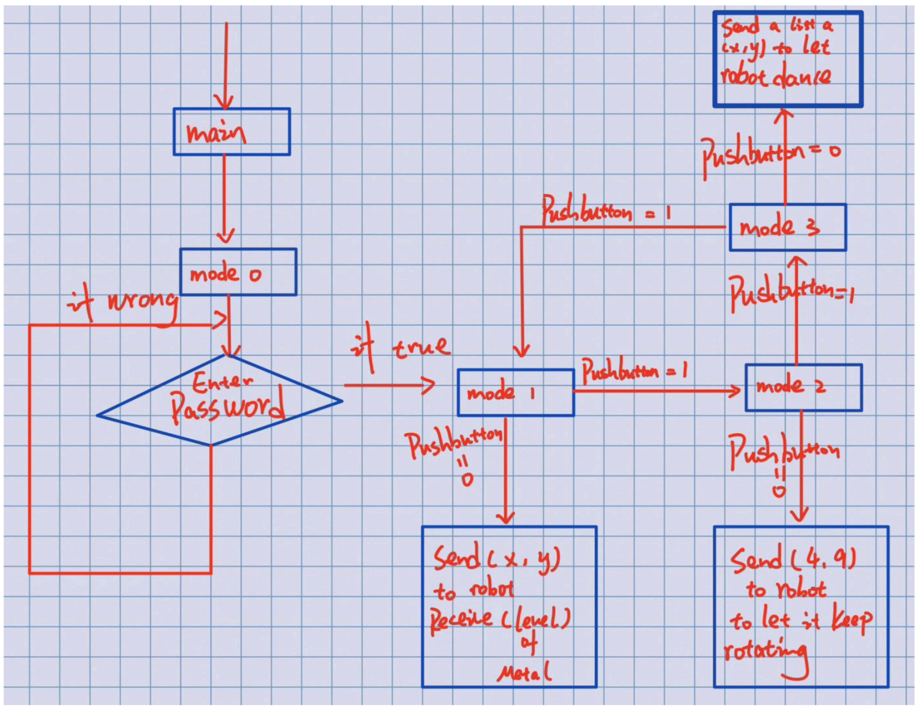
Lastly, the JDY40 module is a key component for wireless communication. It ensures that the PWM signals and mode changes initiated by the operator are received by the robot. In turn, the module sends back information about the robot's metal proximity, keeping the operator fully informed. This integration ensures that the remote controller and robot maintain a steady and reliable connection, crucial for smooth operation.

4.3. Software block diagram

4.3.1. Robot Software



4.3.2. Remote Software



4.4. Software explained

4.4.1. Robot software

4.4.1.1. Initialization

The code initializes an STM32 microcontroller, configuring GPIO pins, timers, and UART communication, as well as includes standard libraries for the microcontroller series. define variable and delay functions and variable. PWM hardware initialization configures output pins and Timer 2 for PWM generation.

4.4.1.2. Frequency and data sending

The program will first calculate the base frequency of the inductor. then if will keep reading the frequency and send the level of metal detected through jdy40.

4.4.1.3. Data Receiving

Receive data form JDY40 and decoding it(from Prof. source code)

4.4.1.4. process_buff

This function extracts integers for 'x' and 'y' coordinates from a buff, it checks for the format to ensure a single comma separates them. If successful, it updates update, if wrong, it keeps the old value.

4.4.1.5. PWM calculation

Process the signal integer data from the JDY40 and process it into a value that can be used in the PWM program in the timer2 interrupt. After the data is processed, the pins will also be initialized so that the H-bridge can work

4.4.1.6. Timer2 interrupt

Interrupt will activate every period. This part of the code is crucial for the robot control, it will send a square wave based on pwm1 and pwm2, which will determine how much power the motor will receive, allowing us to determine the direction and speed of the motor.

4.4.1.7. State1&2

This program will use the function get_pulse we defined to get data from the sonar and decide whether the car is moving forward or data.

4.4.2. Remote Software

4.4.2.1. Initialization

We use an EFM8LB1 microcontroller to manage various hardware operations. It sets up the microcontroller's clock, disables the watchdog timer, and configures UART0 and UART1 for communication, along with timers for periodic interrupts.

4.4.2.2. Program Definition

Key functionalities include initializing ADC pins for analog input, configuring UART1 for serial communication, and implementing timing routines using Timer3 for precise microsecond and millisecond delays, as well as set up LCD.

4.4.2.3. Mode 0

The remote's software is structured around a finite state machine (FSM) that manages various modes of operation. The initial state, mode0, prompts the user to enter a password to unlock the remote. The system is preset with the password '1321', and only upon entering the correct password does the remote unlock, otherwise, it remains locked.

4.4.2.4. Mode 1

The mode1 function displays "Mode 1" on the LCD and continuously displays the coordinates sent via JDY40. It waits for JDY input to calculate the signal digital integer and adjusts the sound output based on the calculated value and updates the LCD to indicate signal strength or metal detection. If the third voltage input indicates a specific condition, the loop is interrupted and exits.

4.4.2.5. Mode 2 &3

Pressing the PS2 joystick switches the remote to Mode 2, where the robot sends programmed PWM values. A subsequent press of the joystick switches operation to Mode 3, enabling the robot to perform dance moves. Pressing the joystick again returns the remote to Mode 1, restoring manual control. FSM facilitates this pattern cycle.

4. 5. Solution Assesment

We tested the robot's motion through a straight (5 cm off center) path, an eight shape path, and a square path. We also tested the metal detection level in the induction by calibrating the frequency, with an accuracy of 89%. Additionally, we tested the sonar accuracy and got a fairly accurate pulse reading range of 4cm. We also tested that the maxim communication distance between the remote control and the car is about 20 meters. Using this information, we can modify our robots to make them more agile, accurate, and intelligent.

5. Life Long Learning

Similar to the reflow oven controller, the prerequisite knowledge we obtained from ELEC 201 and CPEN 211 was heavily applied here. From this project, we delved deeper into embedded programming with C and learned about how it could potentially be applied in the real world. We also learned the importance of organization as we encountered issues with wiring which led to delays in producing a complete product. In particular, we gained experience in debugging and resolving issues in a physical circuit, which proved crucial given the remote-robot communication. Furthermore, signal consistency was also thoroughly debugged to ensure that the robot could receive clear, unambiguous input signals.

6. Conclusions

Our team successfully engineered a robotic system that not only meets but also exceeds the stipulated project requirements by integrating advanced extra features. The system features an STM32 microcontroller for the robot and an EFM8LB12 microcontroller for the transmitter, facilitating sophisticated operations. A highlight of our design is two automated modes, one is circling mode which perform circular sweep continuously, and one will perform a series of dance move for celebration purposes. We also integrated a MB1043 sonar detecter onto the robot to detect distance ahead. It is useful as it can avoiding collison that could cause damage to the robot and the object it hit.

Challenges Encountered

Throughout the development process, we encountered various challenges that sharpened our problem-solving skills and deepened our technical understanding. One major hurdle was signal interference from other groups' projects, which initially caused confusion and operational disruptions in our system. After conducting tests outside the lab environment where the system operated flawlessly, we pinpointed the interference issue and resolved it by modifying our device's DVID and RF channel settings. This adjustment secured a stable and reliable communication link between our robot and the transmitter.

Project Completion Timeline

The project was labor-intensive, demanding around 50 hours of concerted effort from each member of our team. This duration encompassed assembly parts, programming functions, extensive troubleshooting, and final testing phases. These stages were critical not only to fine-tune our design but also to validate its operational reliability and functionality. Despite the challenges faced, the project served as an invaluable learning opportunity, enhancing our proficiency in software development and equipping us with the skills to tackle and resolve unforeseen technical problems.

7. References

[1] J. Calvino-Fraga, "Project 2 –Magnetic Field Controlled Robot", Department of Electrical and Computer Engineering, University of British Columbia, 2007 (Revised 2024).
[2] J. Calvino-Fraga, "Robot Assembly", Department of Electrical and Computer Engineering, University of British Columbia, 2007 (Revised 2023).
[3] J. Calvino-Fraga, "The STM32 Microcontroller System", Department of Electrical and Computer Engineering, University of British Columbia, 2007 (Revised 2024).
[4] J. Calvino-Fraga, "The EFM8 Microcontroller System", Department of Electrical and Computer Engineering, University of British Columbia, 2007 (Revised 2024).

8. Bibliography

SparkFun. (n.d.). LM7805 Datasheet. Retrieved from:

https://www.sparkfun.com/datasheets/Components/LM7805.pdf

Digi-Key. (n.d.). LTV-816, 826, 846 Datasheet. Retrieved from:

https://mm.digikey.com/Volume0/opasdata/d220001/medias/docus/967/LTV-816_826_846.pdf

Microchip. (n.d.). MCP1700 Data Sheet. Retrieved from:

https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/MCP1700-Data-Sheet-20001826F.pdf

ElectroDragon. (n.d.). EY-40 English Manual. Retrieved from:

https://w.electrodragon.com/w/images/0/05/EY-40_English_manual.pdf

MaxBotix. (n.d.). HRLV-MaxSonar-EZ Datasheet. Retrieved from:

https://maxbotix.com/pages/hrlv-maxsonar-ez-datasheet

9. Appendices

9.1. Robot Source Code

```
#include "../Common/Include/stm32l051xx.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../Common/Include/serial.h"
#include "UART2.h"

#define SYSCLK 32000000L
#define DEF_F 15000L

volatile int PWM_Counter = 0;
volatile unsigned char pwm1=0, pwm2=0;
volatile int pwmx;
volatile int pwmy;
#define PA8 (GPIOA->IDR & BIT8)
#define PIN_PERIOD (GPIOA->IDR&BIT8)
#define PIN_SONAR (GPIOA->IDR&BIT6)

// LQFP32 pinout
//             ----------
//       VDD -|1       32|- VSS
//      PC14 -|2       31|- BOOT0
//      PC15 -|3       30|- PB7
//      NRST -|4       29|- PB6
//      VDDA -|5       28|- PB5
//       PA0 -|6       27|- PB4
//       PA1 -|7       26|- PB3
//       PA2 -|8       25|- PA15 (Used for RXD of UART2, connects to TXD of JDY40)
//       PA3 -|9       24|- PA14 (Used for TXD of UART2, connects to RXD of JDY40)
//       PA4 -|10      23|- PA13 (Used for SET of JDY40)
//       PA5 -|11      22|- PA12
//       PA6 -|12      21|- PA11
//       PA7 -|13      20|- PA10 (Reserved for RXD of UART1)
//       PB0 -|14      19|- PA9  (Reserved for TXD of UART1)
//       PB1 -|15      18|- PA8  (pushbutton)
//       VSS -|16      17|- VDD
//              ----------

#define F_CPU 32000000L
// Uses SysTick to delay <us> micro-seconds.
void Delay_us(unsigned char us)
{
	// For SysTick info check the STM32L0xxx Cortex-M0 programming manual page 85.
	SysTick->LOAD = (F_CPU/(1000000L/us)) - 1;  // set reload register, counter rolls
over from zero, hence -1
	SysTick->VAL = 0; // load the SysTick counter
	SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk; // Enable
SysTick IRQ and SysTick Timer */
	while((SysTick->CTRL & BIT16)==0); // Bit 16 is the COUNTFLAG.  True when counter
rolls over from zero.
	SysTick->CTRL = 0x00; // Disable Systick counter
}

void waitms (unsigned int ms)
{
	unsigned int j;
	unsigned char k;
	for(j=0; j<ms; j++)
		for (k=0; k<4; k++) Delay_us(250);
}

void Hardware_Init(void)
```

```
{
       GPIOA->OSPEEDR=0xffffffff; // All pins of port A configured for very high speed! Page 201 of RM0451


       RCC->IOPENR |= (BIT0 | BIT1); // Peripheral clock enable for port A and b
       //RCC->IOPENR |= BIT0; // peripheral clock enable for port A

    GPIOA->MODER = (GPIOA->MODER & ~(BIT27|BIT26)) | BIT26; // Make pin PA13 output (page 200 of RM0451, two bits used to configure: bit0=1, bit1=0))
       GPIOA->ODR |= BIT13; // 'set' pin to 1 is normal operation mode.

       GPIOA->MODER = (GPIOA->MODER & ~(BIT6|BIT7)) | BIT6;    // PA3
       GPIOA->OTYPER &= ~BIT3; // Push-pull
    GPIOA->MODER = (GPIOA->MODER & ~(BIT8|BIT9)) | BIT8;    // PA4
       GPIOA->OTYPER &= ~BIT4; // Push-pull
    GPIOA->MODER = (GPIOA->MODER & ~(BIT10|BIT11)) | BIT10; // PA5
       GPIOA->OTYPER &= ~BIT5; // Push-pull
       GPIOA->MODER = (GPIOA->MODER & ~(BIT14|BIT15)) | BIT14; // PA7
       GPIOA->OTYPER &= ~BIT7; // Push-pull



       GPIOA->MODER &= ~(BIT16 | BIT17); // Make pin PA8 input
       // Activate pull up for pin PA8:
       GPIOA->PUPDR |= BIT16;
       GPIOA->PUPDR &= ~(BIT17);

       GPIOA->MODER &= ~(BIT12 | BIT13); // Make pin PA6 input
       // Activate pull up for pin PA6:
       GPIOA->PUPDR |= BIT12;
       GPIOA->PUPDR &= ~(BIT13);

       GPIOA->MODER &= ~(BIT2 | BIT3); // Make pin PA1 input PIN7
       // Activate pull up for pin PA1:
       GPIOA->PUPDR |= BIT2;
       GPIOA->PUPDR &= ~(BIT3);
}

void pwm_harware(void)
```

```
{
       // Set up output pins
    GPIOB->MODER = (GPIOB->MODER & ~(BIT15|BIT14)) | BIT14; // Make pin Pb7 output
    //GPIOB->OTYPER &= ~BIT7; //push_pull-for
    GPIOB->MODER = (GPIOB->MODER & ~(BIT13|BIT12)) | BIT12; // Make pin Pb6 output
    //PIOB->OTYPER &= ~BIT6; //push_pull - back

    GPIOB->ODR &= ~BIT6; // Set Pb8 low


    GPIOB->MODER = (GPIOB->MODER & ~(BIT11|BIT10)) | BIT10; // Make pin Pb5 output
    //GPIOB->OTYPER &= ~BIT5; //push_pull - for
    GPIOB->MODER = (GPIOB->MODER & ~(BIT9|BIT8)) | BIT8; // Make pin Pb4 output
    //GPIOA->OTYPER &= ~BIT4; //push_pull - back

    GPIOB->ODR &= ~BIT4; // Set Pb8 low

       // Set up timer
       RCC->APB1ENR |= BIT0;  // turn on clock for timer2 (UM: page 177)
       TIM2->ARR = F_CPU/DEF_F-1;
       NVIC->ISER[0] |= BIT15; // enable timer 2 interrupts in the NVIC
       TIM2->CR1 |= BIT4;      // Downcounting
       TIM2->CR1 |= BIT7;      // ARPE enable
       TIM2->DIER |= BIT0;     // enable update event (reload event) interrupt
       TIM2->CR1 |= BIT0;      // enable counting

       __enable_irq();
}

void SendATCommand (char * s)
{
       char buff[40];
       printf("Command: %s", s);
       GPIOA->ODR &= ~(BIT13); // 'set' pin to 0 is 'AT' mode.
       waitms(10);
       eputs2(s);
       egets2(buff, sizeof(buff)-1);
       GPIOA->ODR |= BIT13; // 'set' pin to 1 is normal operation mode.
       waitms(10);
       printf("Response: %s", buff);
```

```
}

void forward_ini(void)
{
        GPIOB->OTYPER &= ~BIT7; // set to push and pull for square wave
        GPIOB->OTYPER &= ~BIT5;

        GPIOB->ODR &= ~BIT6; //set to 0
        GPIOB->ODR &= ~BIT4;
}

void backward_ini(void)
{
        GPIOB->OTYPER &= ~BIT6;
        GPIOB->OTYPER &= ~BIT4;

        GPIOB->ODR &= ~BIT7;
        GPIOB->ODR &= ~BIT5;
}


void TIM2_Handler(void)
{
        TIM2->SR &= ~BIT0; // clear update interrupt flag
        PWM_Counter++;

    if(pwmy >= 0){
        if(pwm1>PWM_Counter){
                GPIOB->ODR |= BIT7;
            }
        else{
                GPIOB->ODR &= ~BIT7;
            }

        if(pwm2>PWM_Counter){
                GPIOB->ODR |= BIT5;
            }
        else{
                GPIOB->ODR &= ~BIT5;
            }

        if (PWM_Counter > 255)
         {
                PWM_Counter=0;
                GPIOB->ODR |= (BIT7|BIT5);
            }
        }

    if(pwm1 ==0  && pwm2 ==0 )
    {
                GPIOB->ODR &= ~BIT6;
                GPIOB->ODR &= ~BIT4;
                GPIOB->ODR &= ~BIT7;
                GPIOB->ODR &= ~BIT5;
        }



    if(pwmy <0){
        if(pwm1>PWM_Counter){
                GPIOB->ODR |= BIT6;
            }
        else{
                GPIOB->ODR &= ~BIT6;
            }

        if(pwm2>PWM_Counter){
                GPIOB->ODR |= BIT4;
            }
        else{
                GPIOB->ODR &= ~BIT4;
            }

        if (PWM_Counter > 255){
                PWM_Counter=0;
                GPIOB->ODR |= (BIT6|BIT4);
         }
        }
}
```

```
void process_buff(const char *buff, char *xbuff, char *ybuff) {
    static char last_x[1] = "1"; // Static storage for last good 'x' value
    static char last_y[1] = "1"; // Static storage for last good 'y' value
    int x, y;
    char postCheck;

    // Validate format: exactly one ',' and x, y are single digits (0-9)
    if (sscanf(buff, "%1d,%1d", &x, &y, &postCheck) == 2) {
        // Successfully parsed x and y as single digits without extra characters following
        sprintf(xbuff, "%d", x); // Convert integer x to string and store in xbuff
        sprintf(ybuff, "%d", y); // Convert integer y to string and store in ybuff

        // Update last good values
        strcpy(last_x, xbuff);
        strcpy(last_y, ybuff);
    } else {
        // Parsing failed or validation not passed, fallback to last good values
        strcpy(xbuff, last_x);
        strcpy(ybuff, last_y);
    }
}

long int GetPeriod (int n)
{
       __disable_irq();
       int i;
       unsigned int saved_TCNT1a, saved_TCNT1b;

       SysTick->LOAD = 0xffffff;  // 24-bit counter set to check for signal present
       SysTick->VAL = 0xffffff; // load the SysTick counter
       SysTick->CTRL   =  SysTick_CTRL_CLKSOURCE_Msk  |  SysTick_CTRL_ENABLE_Msk;  //  Enable
SysTick IRQ and SysTick Timer */
       while (PIN_PERIOD!=0) // Wait for square wave to be 0
       {
              if(SysTick->CTRL & BIT16) return 0;
       }
       SysTick->CTRL = 0x00; // Disable Systick counter
```

```
	SysTick->LOAD = 0xffffff;  // 24-bit counter set to check for signal present
	SysTick->VAL = 0xffffff; // load the SysTick counter
	SysTick->CTRL   =  SysTick_CTRL_CLKSOURCE_Msk  |  SysTick_CTRL_ENABLE_Msk;  //  Enable
SysTick IRQ and SysTick Timer */
	while (PIN_PERIOD==0) // Wait for square wave to be 1
	{
		if(SysTick->CTRL & BIT16) return 0;
	}
	SysTick->CTRL = 0x00; // Disable Systick counter

	SysTick->LOAD = 0xffffff;  // 24-bit counter reset
	SysTick->VAL = 0xffffff; // load the SysTick counter to initial value
	SysTick->CTRL   =  SysTick_CTRL_CLKSOURCE_Msk  |  SysTick_CTRL_ENABLE_Msk;  //  Enable
SysTick IRQ and SysTick Timer */
	for(i=0; i<n; i++) // Measure the time of 'n' periods
	{
		while (PIN_PERIOD!=0) // Wait for square wave to be 0
		{
			if(SysTick->CTRL & BIT16) return 0;
		}
		while (PIN_PERIOD==0) // Wait for square wave to be 1
		{
			if(SysTick->CTRL & BIT16) return 0;
		}
	}
	SysTick->CTRL = 0x00; // Disable Systick counter

	__enable_irq();
	return 0xffffff-SysTick->VAL;
}


long int GetPulse (void)
{
	__disable_irq();
	unsigned int saved_TCNT1a, saved_TCNT1b;

	SysTick->LOAD = 0xffffff;  // 24-bit counter set to check for signal present
	SysTick->VAL = 0xffffff; // load the SysTick counter
```

```
        SysTick->CTRL    =  SysTick_CTRL_CLKSOURCE_Msk|  SysTick_CTRL_ENABLE_Msk;  //  Enable
SysTick IRQ and SysTick Timer */
        while (PIN_SONAR!=0) // Wait signal to be 0
        {
                if(SysTick->CTRL & BIT16) return 0;
        }
        SysTick->CTRL = 0x00; // Disable Systick counter

        SysTick->LOAD = 0xffffff;  // 24-bit counter set to check for signal present
        SysTick->VAL = 0xffffff; // load the SysTick counter
        SysTick->CTRL   =  SysTick_CTRL_CLKSOURCE_Msk  |  SysTick_CTRL_ENABLE_Msk;  //  Enable
SysTick IRQ and SysTick Timer */
        while (PIN_SONAR==0) // Wait for signal to be 1
        {
                if(SysTick->CTRL & BIT16) return 0;
        }
        SysTick->CTRL = 0x00; // Disable Systick counter

        SysTick->LOAD = 0xffffff;  // 24-bit counter reset
        SysTick->VAL = 0xffffff; // load the SysTick counter to initial value
        SysTick->CTRL   =  SysTick_CTRL_CLKSOURCE_Msk  |  SysTick_CTRL_ENABLE_Msk;  //  Enable
SysTick IRQ and SysTick Timer */

        while (PIN_SONAR!=0) // Wait for square wave to be 0
        {
                if(SysTick->CTRL & BIT16) return 0;
        }

        SysTick->CTRL = 0x00; // Disable Systick counter

        __enable_irq();
        return 0xffffff-SysTick->VAL;
}



#define PA4_0 (GPIOA->ODR &= ~BIT4)
#define PA4_1 (GPIOA->ODR |=  BIT4)
#define PA5_0 (GPIOA->ODR &= ~BIT5)
#define PA5_1 (GPIOA->ODR |=  BIT5)
```

```
#define PA3_0 (GPIOA->ODR &= ~BIT3)

#define PA3_1 (GPIOA->ODR |=  BIT3)

#define PA7_0 (GPIOA->ODR &= ~BIT7)

#define PA7_1 (GPIOA->ODR |=  BIT7)





int main(void){

       char buff[10];

       char buffs[30];

    int cnt=0;


    //pwm

    int npwm;

    int power;

    char xbuff[1];

    char ybuff[1];

    int tempx;

    int tempy;


    //metal dector

    int level;

    long int ini_f = 1000000;

    long int ini_t = 0;

    int i;

    int flag = 1;

    long int count = 0;

       long int f;

    int pre_y = 4;


    //sonar

    long int pulse1;

    int state = 0;


       Hardware_Init();

       pwm_harware();

       initUART2(9600);
```

```
        waitms(1000); // Give putty some time to start.
        printf("\r\nJDY-40 test\r\n");

        // We should select an unique device ID.  The device ID can be a hex

        SendATCommand("AT+DVID1314\r\n");

        // To check configuration
        SendATCommand("AT+VER\r\n");
        SendATCommand("AT+BAUD\r\n");
        SendATCommand("AT+RFID1314\r\n");
        SendATCommand("AT+DVID1314\r\n");
        SendATCommand("AT+RFC007\r\n");
        SendATCommand("AT+POWE\r\n");
        SendATCommand("AT+CLSS\r\n");



        printf("\r\nPress   and   hold   a   push-button   attached   to   PA8   (pin   18)   to
transmit.\r\n");
        //__disable_irq();
                PA3_0;
                PA4_0;
                PA5_0;
                PA7_0;
                //PB3_1;

        cnt=0;
        //state =0;
        //printf("%d",state);

        while(1){

                count=GetPeriod(30);
                //pwm_harware();
                //__disable_irq();
                //__enable_irq();
                //count = 0;
```

```
   //pwm_harware();
   if(count>0)
   {
          f=(F_CPU*30)/count;

 if (flag) {
        for (int i = 0; i < 30; i++) {
         if(ini_f > f){
             ini_f = f;
         }
   flag = 0;
 }

          //printf("inif: %d, f: %d\n\r",ini_f,f);
   }
          //printf("inif: %d, f: %d\n\r",ini_f,f);
          //eputs("f=");
          //PrintNumber(ini_f, 10, 7);
          //eputs("Hz, count=");
          //PrintNumber(f, 10, 7);
          //eputs("          \r\n");
   }




          //sprintf(buffs, "  12\r\n");
          //eputs2(buffs);
          //waitms(20);
   if (f<ini_f){
   eputs("0\n\r");
   sprintf(buffs, "000\r\n");
   eputs2(buffs);
waitms(20);
   }
   if(f>ini_f&&f<ini_f+90){
   eputs("1\n\r");
   sprintf(buffs, "111\r\n");
   eputs2(buffs);
```

```
waitms(20);
    }
    if(f>=ini_f+90&&f<ini_f+180){
    sprintf(buffs, "222\r\n");
    eputs2(buffs);
waitms(20);
    //eputc2('2\n\r');
    }
    if(f>=ini_f+180){
    eputs("3\n\r");
    sprintf(buffs, "333\r\n");
    eputs2(buffs);
waitms(20);
    }

    if (ReceivedBytes2()>0){
            egets2(buff, sizeof(buff)-1);
            //printf("%s\n\r", buff);

 process_buff(buff, xbuff, ybuff);

 tempx = atoi(xbuff);
    tempy = atoi(ybuff);

 printf("x:%d y:%d\n\r", tempx, tempy);

 if (tempy > 4){
        double resulty = (tempy-4.0)/5.0*100.00;
                  pwmy = (int)resulty;
    }
    if (tempy < 4){
            double resulty = (tempy-4.0)/4.0*100.00;
                  pwmy = (int)resulty;
                  }
 if(tempy == 4){
     pwmy = 0;
 }

 double resultx = (tempx-4.0)/4.0*100.00;
```

```
pwmx = (int)resultx;

if(pwmx>100){
pwmx=100;
}

//printf("x:%d\n\r",pwmx);
//printf("y:%d\n\r",pwmy);


if(pwmy > 0 && pwmy < pre_y){
 PA7_1;
 waitms(300);
}
if(pwmy == 0 && pwmx ==0){
       PA7_1;
}
else{
PA7_0;
}

if(pwmy < 0){
       PA3_1;
}
if(pwmy>=0){
       PA3_0;
}

if(pwmx == 0){
       PA4_0;
       PA5_0;
}
if(pwmx<0){
       PA5_0;
       PA4_1;
       }
if(pwmx>0){
       PA4_0;
       PA5_1;
```

```
                    }

                pre_y = pwmy;

        if(pwmy > 0){

                forward_ini();

                double resultp = 255.0*abs(pwmy)/100.0;
                power = (int)resultp;
                //printf("%d\n",power);
                if (pwmx > 0){
                    //eputs("right \n");
                    double result1 = power - (power * abs(pwmx) / 100.0);
                    pwm1 = (int)result1;
                    //printf("1:%d\n\r",pwm1);
                    pwm2 = power;
                    //printf("2:%d\n\r",pwm2);
                    }
                else if(pwmx < 0){
                    pwm1 = power;
                    //printf("1:%d\n\r",pwm1);
                    double result = power - (power * abs(pwmx) / 100.0);
                    pwm2 = (int)result;
                    //printf("2:%d\n\r",pwm2);
                    }
                if(pwmx == 0){
                    //eputs("right\n\n");
                    pwm1 = power;
                    //eputs("%d",pwm1\n);
                    pwm2 = power-10;
                    //eputs("%d",pwm2\n);
                }
           }

    if(pwmy <0){
                backward_ini();
                double resultp = 255.0*abs(pwmy)/100.0;
```

```
              power = (int)resultp;
              //printf("%d\n",power);
              if (pwmx > 0){
              //eputs("right \n");
              double result1 = power - (power * abs(pwmx) / 100.0);
              pwm1 = (int)result1-8;
              //printf("1:%d\n\r",pwm1);
              pwm2 = power;
              //printf("2:%d\n\r",pwm2);
              }
              else if(pwmx < 0){
              pwm1 = power-8;
              //printf("1:%d\n\r",pwm1);
              double result = power - (power * abs(pwmx) / 100.0);
              pwm2 = (int)result;
              //printf("2:%d\n\r",pwm2);
              }
              if(pwmx==0){
              //eputs("right\n\n");
              pwm1 = power;
              //eputs("%d",pwm1\n);
              pwm2 = power-10;
              //eputs("%d",pwm2\n);
              }
       }

if(pwmy ==0){

              forward_ini();
              double result = 255.0*abs(pwmx)/100.0;
              power = (int)result;
              //printf("power:%d\n\r",power);
              if(pwmx > 0){
                     pwm1 = 0;
                     pwm2 = power;
                     //eputs("right\n");
              }
              else if (pwmx < 0){
                     pwm1 = power;
```

```
                pwm2 = 0;
        }
        if(pwmx ==0){
                pwm1 = 0;
                //printf("right %d\n",pwm1);
                pwm2 = 0;
                //printf("%d\n",pwm1);
                }
        }
        if(pwm1<0){
        pwm1 = 0;
        }
}
printf("%d\r\n",state);



if((GPIOA->IDR&BIT1)==0){
        waitms(200);
        if((GPIOA->IDR&BIT1)==0){
                state++;
                }
}

//else{
//      printf("Nothing transmitted.\n\r");
//      waitms(500);
//}

while(state == 1){
//printf("%d\r\n",state);
        backward_ini();

        if(PIN_SONAR!=0){
        pulse1=GetPulse();
        }

        printf("T=%d       \r", pulse1);

        pwm1 = 100;
```

```
pwm2 = 100;
        if(pulse1 <= 9600){
                backward_ini();
                printf("in if");
        pwm1 = 0;
        pwm2 = 0;

//waitms(200);
}
if((GPIOA->IDR&BIT1)==0){
        waitms(200);
        if((GPIOA->IDR&BIT1)==0){
                state++;
                break;
                }
        }

}
//pin7
while(state == 2){
printf("s3");
        backward_ini();

        if(PIN_SONAR!=0){
        pulse1=GetPulse();
        }

        printf("3T=%d       \r", pulse1);
        pwm1 = 0;
        pwm2 = 0;

        if(pulse1 >= 10000){
                backward_ini();
                printf("in if");
        pwm1 = 100;
        pwm2 = 100;

//waitms(200);
}}}}
```

9.2. Remote Source Code

```
#include <EFM8LB1.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define SYSCLK 72000000
#define BAUDRATE 115200L
#define SARCLK 18000000L
#define TIMER_5_FREQ 2048 //should be set 2048L later
#define TIMER_2_FREQ 1000L//For a 1ms tick

#define LCD_RS P1_7
// #define LCD_RW Px_x // Not used in this code.  Connect to GND
#define LCD_E  P2_5
#define LCD_D4 P1_3
#define LCD_D5 P1_2
#define LCD_D6 P1_1
#define LCD_D7 P1_0
#define CHARS_PER_LINE 16

#define BUTTON1 (!P3_7)
#define BUTTON2 (!P3_2)
#define BUTTON3 (!P3_0)

idata char buff[30];
volatile unsigned int TickCount = 0;
volatile unsigned int Sound_out = 1000;
volatile unsigned int state = 0;

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

        VDM0CN=0x80;       // enable VDD monitor
        RSTSRC=0x02|0x04;  // Enable reset on missing clock detector and VDD
```

```
#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
```

```
	P0MDOUT |= 0x11; // Enable UART0 TX (P0.4) and UART1 TX (P0.0) as push-pull outputs
	P2MDOUT |= 0x01; // P2.0 in push-pull mode
	P2MDOUT|=0b_0000_0010;
	XBR0     = 0x01; // Enable UART0 on P0.4(TX) and P0.5(RX)
	XBR1     = 0X00;
	XBR2     = 0x41; // Enable crossbar and uart 1

	// Configure Uart 0
	#if (((SYSCLK/BAUDRATE)/(2L*12L))>0xFFL)
		#error Timer 0 reload value is incorrect because (SYSCLK/BAUDRATE)/(2L*12L)
> 0xFF
	#endif
	SCON0 = 0x10;
	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

	//Initialize timer 2 for periodic interrupts
	TMR2CN0=0x00;   // Stop Timer2; Clear TF2;
	CKCON0|=0b_0001_0000; // Timer 2 uses the system clock
	TMR2RL=(0x10000L-(SYSCLK/(2*TIMER_2_FREQ))); // Initialize reload value
	TMR2=0xffff;   // Set to reload immediately
	ET2=1;         // Enable Timer2 interrupts
	TR2=1;         // Start Timer2 (TMR2CN is 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=(0x10000L-(SYSCLK/(2*TIMER_5_FREQ))); // Initialize reload value
	TMR5=0xffff;   // Set to reload immediately
	EIE2|=0b_0000_1000; // Enable Timer5 interrupts
	TR5=1;         // Start Timer5 (TMR5CN0 is bit addressable)
	SFRPAGE=0x00;

	EA=1;  // Enable global interrupts
	return 0;
}

void Timer2_ISR (void) interrupt INTERRUPT_TIMER2
{
	SFRPAGE=0x0;
	TF2H = 0; // Clear Timer2 interrupt flag
	TickCount++;
	if(TickCount>=Sound_out){
	SFRPAGE = 0x10;
	TR5 ^= 1;
	TickCount=0;
	SFRPAGE = 0x00;
	}
}

void Timer5_ISR (void) interrupt INTERRUPT_TIMER5
{
	SFRPAGE=0x10;
	TF5H = 0; // Clear Timer5 interrupt flag
	P2_1=!P2_1;
	SFRPAGE=0x00;
}


void InitADC (void)
{
	SFRPAGE = 0x00;
	ADEN=0; // Disable ADC

	ADC0CN1=
		(0x2 << 6) | // 0x0: 10-bit, 0x1: 12-bit, 0x2: 14-bit
	   (0x0 << 3) | // 0x0: No shift. 0x1: Shift right 1 bit. 0x2: Shift right 2 bits.
0x3: Shift right 3 bits.
		(0x0 << 0) ; // Accumulate n conversions: 0x0: 1, 0x1:4, 0x2:8, 0x3:16,
0x4:32

	ADC0CF0=
            ((SYSCLK/SARCLK) << 3) | // SAR Clock Divider. Max is 18MHz. Fsarclk =
(Fadcclk) / (ADSC + 1)
            (0x0 << 2); // 0:SYSCLK ADCCLK = SYSCLK. 1:HFOSC0 ADCCLK = HFOSC0.

    ADC0CF1=
            (0 << 7)   | // 0: Disable low power mode. 1: Enable low power mode.
            (0x1E << 0); // Conversion Tracking Time. Tadtk = ADTK / (Fsarclk)

    ADC0CN0 =
            (0x0 << 7) | // ADEN. 0: Disable ADC0. 1: Enable ADC0.
            (0x0 << 6) | // IPOEN. 0: Keep ADC powered on when ADEN is 1. 1: Power down
when ADC is idle.
            (0x0 << 5) | // ADINT. Set by hardware upon completion of a data conversion.
Must be cleared by firmware.
            (0x0 << 4) | // ADBUSY. Writing 1 to this bit initiates an ADC conversion
when ADCM = 000. This bit should not be polled to indicate when a conversion is complete.
Instead, the ADINT bit should be used when polling for conversion completion.
            (0x0 << 3) | // ADWINT. Set by hardware when the contents of ADC0H:ADC0L
fall within the window specified by ADC0GTH:ADC0GTL and ADC0LTH:ADC0LTL. Can trigger an
interrupt. Must be cleared by firmware.
            (0x0 << 2) | // ADGN (Gain Control). 0x0: PGA gain=1. 0x1: PGA gain=0.75.
0x2: PGA gain=0.5. 0x3: PGA gain=0.25.
            (0x0 << 0) ; // TEMPE. 0: Disable the Temperature Sensor. 1: Enable the
Temperature Sensor.

    ADC0CF2=
            (0x0 << 7) | // GNDSL. 0: reference is the GND pin. 1: reference is the AGND
pin.
            (0x1 << 5) | // REFSL. 0x0: VREF pin (external or on-chip). 0x1: VDD pin.
0x2: 1.8V. 0x3: internal voltage reference.
            (0x1F << 0); // ADPWR. Power Up Delay Time. Tpwrtime = ((4 * (ADPWR + 1)) +
2) / (Fadcclk)

    ADC0CN2 =
            (0x0 << 7) | // PACEN. 0x0: The ADC accumulator is over-written.  0x1: The
ADC accumulator adds to results.
            (0x0 << 0) ; // ADCM. 0x0: ADBUSY, 0x1: TIMER0, 0x2: TIMER2, 0x3: TIMER3,
0x4: CNVSTR, 0x5: CEX5, 0x6: TIMER4, 0x7: TIMER5, 0x8: CLU0, 0x9: CLU1, 0xA: CLU2, 0xB:
CLU3
```

```
        ADEN=1; // Enable ADC
}
void InitPinADC (unsigned char portno, unsigned char pin_num)
{
        unsigned char mask;

        mask=1<<pin_num;

        SFRPAGE = 0x20;
        switch (portno)
        {
                case 0:
                        P0MDIN &= (~mask); // Set pin as analog input
                        P0SKIP |= mask; // Skip Crossbar decoding for this pin
                break;
                case 1:
                        P1MDIN &= (~mask); // Set pin as analog input
                        P1SKIP |= mask; // Skip Crossbar decoding for this pin
                break;
                case 2:
                        P2MDIN &= (~mask); // Set pin as analog input
                        P2SKIP |= mask; // Skip Crossbar decoding for this pin
                break;
                default:
                break;
        }
        SFRPAGE = 0x00;
}

unsigned int ADC_at_Pin(unsigned char pin)
{
        ADC0MX = pin;   // Select input from pin
        ADINT = 0;
        ADBUSY = 1;     // Convert voltage at the pin
        while (!ADINT); // Wait for conversion to complete
        return (ADC0);
}
#define VDD 3.3 // The measured value of VDD in volts
```

```
float Volts_at_Pin(unsigned char pin)
{
         return ((ADC_at_Pin(pin)*VDD)/16383.0);
}



void UART1_Init (unsigned long baudrate)
{
    SFRPAGE = 0x20;
        SMOD1 = 0x0C; // no parity, 8 data bits, 1 stop bit
        SCON1 = 0x10;
        SBCON1 =0x00;   // disable baud rate generator
        SBRL1 = 0x10000L-((SYSCLK/baudrate)/(12L*2L));
        TI1 = 1; // indicate ready for TX
        SBCON1 |= 0x40;   // enable baud rate generator
        SFRPAGE = 0x00;
}

// Uses Timer3 to delay <us> micro-seconds.
void Timer3us(unsigned char us)
{
        unsigned char i;               // usec counter

        // The input for Timer 3 is selected as SYSCLK by setting T3ML (bit 6) of CKCON0:
        CKCON0|=0b_0100_0000;

        TMR3RL = (-(SYSCLK)/1000000L); // Set Timer3 to overflow in 1us.
        TMR3 = TMR3RL;                 // Initialize Timer3 for first overflow

        TMR3CN0 = 0x04;                 // Sart Timer3 and clear overflow flag
        for (i = 0; i < us; i++)       // Count <us> overflows
        {
                while (!(TMR3CN0 & 0x80));  // Wait for overflow
                TMR3CN0 &= ~(0x80);         // Clear overflow indicator
        }
        TMR3CN0 = 0 ;                   // Stop Timer3 and clear overflow flag
}

void waitms (unsigned int ms)
```

```
{
       unsigned int j;
       unsigned char k;
       for(j=0; j<ms; j++)
              for (k=0; k<4; k++) Timer3us(250);
}

//LCD
void LCD_pulse (void)
{
       LCD_E=1;
       Timer3us(40);
       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();
       Timer3us(40);
       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);
       waitms(2);
}
```

```
void WriteCommand (unsigned char x)

{

	LCD_RS=0;

	LCD_byte(x);

	waitms(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

	waitms(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)

	waitms(20); // Wait for clear screen command to finsih.

}


void LCDprint(char * string, unsigned char line, bit clear)

{

    int j;


    WriteCommand(line==2?0xc0:0x80);

    waitms(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

}



void putchar1 (char c)

{

    SFRPAGE = 0x20;
```

```
        while (!TI1);
        TI1=0;
        SBUF1 = c;
        SFRPAGE = 0x00;
}

void sendstr1 (char * s)
{
        while(*s)
        {
                putchar1(*s);
                s++;
        }
}

char getchar1 (void)
{
        char c;
    SFRPAGE = 0x20;
        while (!RI1);
        RI1=0;
        // Clear Overrun and Parity error flags
        SCON1&=0b_0011_1111;
        c = SBUF1;
        SFRPAGE = 0x00;
        return (c);
}

char getchar1_with_timeout (void)
{
        char c;
        unsigned int timeout;
    SFRPAGE = 0x20;
    timeout=0;
        while (!RI1)
        {
                SFRPAGE = 0x00;
                Timer3us(20);
                SFRPAGE = 0x20;
				timeout++;
				if(timeout==25000)
				{
					SFRPAGE = 0x00;
					return ('\n'); // Timeout after half second
				}
		}
		RI1=0;
		// Clear Overrun and Parity error flags
		SCON1&=0b_0011_1111;
		c = SBUF1;
		SFRPAGE = 0x00;
		return (c);
}

void getstr1 (char * s)
{
		char c;

		while(1)
		{
				c=getchar1_with_timeout();
				if(c=='\n')
				{
					*s=0;
					return;
				}
				*s=c;
				s++;
		}
}

// RXU1 returns '1' if there is a byte available in the receive buffer of UART1
bit RXU1 (void)
{
		bit mybit;
    SFRPAGE = 0x20;
		mybit=RI1;
		SFRPAGE = 0x00;
```

```
        return mybit;
}

void waitms_or_RI1 (unsigned int ms)
{
        unsigned int j;
        unsigned char k;
        for(j=0; j<ms; j++)
        {
                for (k=0; k<4; k++)
                {
                        if(RXU1()) return;
                        Timer3us(250);
                }
        }
}

void SendATCommand (char * s)
{
        printf("Command: %s", s);
        P2_0=0; // 'set' pin to 0 is 'AT' mode.
        waitms(5);
        sendstr1(s);
        getstr1(buff);
        waitms(10);
        P2_0=1; // 'set' pin to 1 is normal operation mode.
        printf("Response: %s\r\n", buff);
}



int pwm_txd(int y, int x, unsigned int time_sec)
{
    unsigned char i;
    float voltage;
    int v;

    for(i = 0; i < time_sec*10; i++){

        sprintf(buff, "%d,%d\n\r", x,y);
        sendstr1(buff);
        printf(buff);

        waitms_or_RI1(100);

        voltage = Volts_at_Pin(QFP32_MUX_P2_4);
        v = (voltage == 0.00) ? 0 : 1;
        if(v == 0)
        {
            return 1;
        }

    }

    return 0; // Completed the loop without button press
}
unsigned char unlockSequence[]= {1, 3, 2, 1};
unsigned char currentStep;

void mode0(void)
{
       int one_count = 0;
       currentStep = 0;
       waitms(500);
       waitms(500);
       waitms(500);
       LCDprint("Enter password", 1,1);
       //LCD_4BIT();
          while(currentStep < 4) {
        if (BUTTON1 && unlockSequence[currentStep] == 1) {
        //LCDprint("1", 1,1, 1);
            while(BUTTON1); // Wait for button release
            currentStep++;
            one_count++;
            if(one_count == 1){
               WriteCommand(0x01);
               LCDprint("1",1,1);
            }
            else if(one_count == 2)
            {
               WriteCommand(0x01);
               LCDprint("1321", 1, 1);
            }
        } else if (BUTTON3 && unlockSequence[currentStep] == 3) {
        WriteCommand(0x01);
        LCDprint("13", 1, 1);
            while(BUTTON3); // Wait for button release
            currentStep++;
        } else if (BUTTON2 && unlockSequence[currentStep] == 2) {
        WriteCommand(0x01);
        LCDprint("132", 1, 1);
            while(BUTTON2); // Wait for button release
            currentStep++;
        }
        else if (currentStep > 0 && (BUTTON1 || BUTTON2 || BUTTON3)) {
            // If any button is pressed out of sequence, reset the step counter
            LCDprint("     Error!    ", 1, 1);
            waitms(2500);
            mode0();
            //currentStep = 0;
        }
        waitms(100);


    }


       waitms(2000);
       LCDprint("Starting...",1,1);
       waitms(2000);
       LCDprint("           ",1,1);
}

void mode1(void)
{

       char a='0';
       int sum=0;
       float freq_index=0.0;
float v[3];
float x,y;
int xx,yy,dig_v;

LCDprint("Mode 1", 1, 1);

while(1)
{       int i;
        sum=0;
        freq_index=0.0;
        v[0] = Volts_at_Pin(QFP32_MUX_P2_2);
        v[1] = Volts_at_Pin(QFP32_MUX_P2_3);

        y=9/3.3*v[0];
        x=9/3.3*v[1];

        yy = (int)y;
        xx = (int)x;

        sprintf(buff, "%d,%d\n\r", xx,yy);
        printf("x:%d,y:%d\n",xx,yy);
        sendstr1(buff);
        //LCDprint(buff, 2, 5, 1);
        waitms_or_RI1(100);

        if(RXU1())
 {

     a = getchar1_with_timeout();
     for(i=0;i<20;i++){
     sum+=a-'0';
     }
     freq_index= sum/20;
     printf("%f\r\n",freq_index);

     if(freq_index>=0&&freq_index<1)
     {
        Sound_out = 2000;
        LCDprint("WEAK", 2, 1);
```

```
            }
            else if(freq_index>=1&&freq_index<=2){

              Sound_out = 1000;
              LCDprint("Medium", 2, 1);
            }
            else if(freq_index>2&&freq_index<3){

            Sound_out = 400;
            LCDprint("Metal Detected", 2, 1);

            }
            else {

              Sound_out = 200;
              LCDprint("Metal Detected", 2, 1);
            }

            waitms(10);
        }

        v[2] = Volts_at_Pin(QFP32_MUX_P2_4);
              dig_v = (v[2] == 0.00) ? 0 : 1;
        if(dig_v == 0)
        {
              break;
        }

    }
}

void mode2(void)
{
    int i;
    LCDprint("Mode 2", 1, 1);
    while(1)
    {
       i = pwm_txd(4, 9, 6);
```

```
        if(i)
        {
                break;
        }
        /*
        i = pwm_txd(4, 1, 2);
        if(i)
        {
                break;
        }
        i = pwm_txd(9, 4, 6);
        if(i)
        {
                break;
        }
        i = pwm_txd(4, 1, 3);
        if(i)
        {
                break;
        }
        i = pwm_txd(9, 4, 6);
        if(i)
        {
                break;
        }
        i = pwm_txd(4, 1, 3);
        if(i)
        {
                break;
        }
        i = pwm_txd(9, 4, 6);
        if(i)
        {
                break;
        }*/


    }
}
```

```
void mode3(void)
{
        int i;
        LCDprint("Mode 3", 1, 1);
        while(1)
        {
                i = pwm_txd(4, 0 , 6);
        if(i)
        {
                break;
        }
        /*i = pwm_txd(9, 4, 6);
        if(i)
        {
                break;
        }
        i = pwm_txd(4, 7, 2);
        if(i)
        {
                break;
        }
        i = pwm_txd(9, 4, 6);
        if(i)
        {
                break;
        }
        i = pwm_txd(4, 7, 2);
        if(i)
        {
                break;
        }
        i = pwm_txd(9, 4, 6);
        if(i)
        {
                break;
        }*/
        }
}



void main (void)
{
    char a='0';
       int sum=0;
       float freq_index=0.0;
       //float v[3];
       //float x,y;
       //int xx,yy,dig_v;
       float voltage;
       int v1;
       waitms(500);
       printf("\r\nJDY-40 test\r\n");
       UART1_Init(9600);

       InitPinADC(2, 2); // Configure P2.2 as analog input
       InitPinADC(2, 3); // Configure P2.3 as analog input
       InitPinADC(2, 4); // Configure P2.4 as analog input
    InitADC();
    LCD_4BIT();

       // To configure the device (shown here using default values).
       // For some changes to take effect, the JDY-40 needs to be power cycled.
       // Communication can only happen between devices with the
       // same RFID and DVID in the same channel.

       //SendATCommand("AT+BAUD4\r\n");
       //SendATCommand("AT+RFID8899\r\n");
       //SendATCommand("AT+DVID1122\r\n"); // Default device ID.
       //SendATCommand("AT+RFC001\r\n");
       //SendATCommand("AT+POWE9\r\n");
       //SendATCommand("AT+CLSSA0\r\n");

       // We should select an unique device ID.  The device ID can be a hex
       // number from 0x0000 to 0xFFFF.  In this case is set to 0xABBA
       SendATCommand("AT+DVID1314\r\n");
```

```
// To check configuration
SendATCommand("AT+VER\r\n");
SendATCommand("AT+BAUD\r\n");
SendATCommand("AT+RFID1314\r\n");
SendATCommand("AT+DVID1314\r\n");
SendATCommand("AT+RFC007\r\n");
SendATCommand("AT+POWE\r\n");
SendATCommand("AT+CLSS\r\n");

//printf("\r\Press and hold the BOOT button to transmit.\r\n");
LCDprint("Mode 1", 1, 1);



mode0();

 while (1) {
 voltage = Volts_at_Pin(QFP32_MUX_P2_4);
 v1 = (voltage == 0.00) ? 0 : 1;

 if (v1 == 0) {
        waitms(300) ;
        if (v1 == 0) {
     state = (state + 1) % 3;
     }
 }


 switch (state) {
     case 0:
         mode1();
         break;
     case 1:
         mode2();
         break;
     case 2:
         mode3();
         break;
```

```
                default:

                    continue;

          }


         }

}
```