Final Report

Automatic and Remote Operated Coin Picking Robot

SUBMITTED TO: DR. JÉSUS CALVIÑO-FRAGA
ELEC 291 L2A GROUP A07

Prepared by	Group A07
Reference ID	PROJ-FRPT-1.0-FINAL
Version	1.0
Status	Final
Date of Issue	2025-04-08

Student	Student ID	Points (%)	Signature
Chathil Rajamanthree	32523201	105	
Colin Yeung	70611371	105	
Eric Feng	70120548	75	
Rishi Upath	18259374	105	
Santo Neyyan	55096309	105	
Warrick Lo	47938733	105	

Contents

1 Introduction

The objective of this project is to design and build a coin-picking robot capable of detecting and retrieving coins. The robot operates in two distinct modes. In manual mode, it is remotely controlled by an external operator who can send commands to collect coins. In automatic mode, the robot autonomously navigates within a boundary defined by an AC-carrying wire and identifies and collects all types of Canadian currency coins. The project was successfully completed according to the following specifications:

- The controller and robot systems must use microcontrollers from different families and programmed through the C programming language.
- The controller and robot systems must be entirely battery operated.
- Coins must be detected using a metal detector. Additionally, the robot must detect all Canadian coins currently in circulation.
- The robot motors must be controlled using MOSFETs and may be isolated from the micrcontroller system using opto-isolators.
- In automatic mode, the robot must be able to detect the AC current carrying wire and remain within a minimum defined perimeter of $0.5\,\mathrm{m}^2$. The AC source can be generated using a function generator or oscillator chip.
- In automatic mode, the robot should autonomously pickup 20 coins (four of each: $0.05, $0.10, $0.25, $1.00, $2.00) without leaving the defined perimeter. After 20 coins have been picked up, the robot waits for next commands from the operator.
- In manual mode, the robot is controlled using a remote and must carry out the commands: forward, backward, left, right, and coin pickup. The remote should include a LCD to display the metal detection strength returned by the robot, as well as a speaker that beeps when metal is detected. The frequency of the beep should increase with increasing metal detection strength.

1.1 Hardware Design Overview

The hardware system consists of the robot and controller modules. The robot is powered by a $9\,\mathrm{V}$ battery, which is regulated to $5\,\mathrm{V}$ using the LM7805 and further to $3.3\,\mathrm{V}$ using the MCP1700 to power the EFM8LB1 microcontroller and other low-voltage components like the ultrasonic sensor and radio transmitter. A separate $6\,\mathrm{V}$ battery (made from four $1.5\,\mathrm{V}$ batteries in series) powers the motors, servos and electromagnet. These high-power components are electrically isolated from the microcontroller circuitry using the LTV-847 optocoupler to prevent noise interference. Additionally, the motors are controlled using standard H-bridge circuits composed of N-MOS and P-MOS transistors. Similarly, the servos and electromagnet are also optoisolated using the LTV-847.

The controller hardware is built around the STM32L051 microcontroller system and is powered through a 9 V regulated to 5 V using the LM7805 and further to 3.3 V by the MCP1700. For user input, the system consists of a joystick for robot control and ADC push buttons. For feedback to the user, the controller has an LCD display, speaker, and LED indicators. The controller uses the JDY-40 to transmit and receive information with the robot. See Figure A.1 for the high-level hardware block diagram. Also see Figure A.3 and Figure A.4 for detailed hardware circuit schematics of the robot and controller.

1.2 Software Design Overview

The software system consists of two modules: the controller (the "master") and the robot (the "slave"). Upon startup, the system enters manual mode, where the user can control the robot via the remote controller. In this mode, the controller continuously sends movement instructions to the robot based on joystick input. The robot listens for these instructions and responds accordingly. Additionally, the robot transmits electromagnet strength data correlated with its proximity to coins back to the controller. This data is mapped to variable frequency beeps from the speaker and displayed on the LCD.

When the user presses the mode change push button, the system switches to automatic mode. In this mode, the robot continuously moves forward until it detects either the perimeter wire, an obstacle, or a coin. If it detects a perimeter or obstacle, it first moves backward for a specified amount of time. Then, it performs a turn at a randomized angle. If a coin is detected, it stops and moves backwards slightly to position the coin directly in front of it. Then, the coin pickup will be executed with servos moving in a sweeping motion to pick the coin up. This process repeats until the robot has collected 20 coins, after which the system automatically returns to manual mode. See Figure A.2 for the high-level software block diagram.

2 Investigation

2.1 Idea Generation

Before starting any physical prototyping and testing, our group took time to thoroughly review the project requirements and the resources provided. Building on what worked well during Project 1 of ELEC 291, we continued to use a dedicated Discord channel to facilitate communication and brainstorming throughout the project. This served as a central hub for sharing ideas, proposing functionalities, and collaboratively refining the project specifications.

To generate ideas, we divided the project into sub-categories and discussed potential approaches for each section. We also searched online for inspiration and explored how features from similar projects could be adapted to our own design.

As we explored potential features and design choices, we evaluated each idea based on technical complexity, feasibility, required resources, and overall contribution to the project goals. This structured approach ensured that everyone was aligned on the project direction and aware of the implementation priorities.

2.2 Investigation Design

We began by designing and testing each project module individually on a separate code file and test board. This approach allowed us to debug efficiently and work on different components in parallel without affecting the overall system functionality. We applied this method to various subsystems for the robot module, including:

- Metal detector circuit utilizing the Colpitts oscillator.
- Two perimeter detector circuits.
- Motor control using H-Bridge circuit configurations.
- Electromagnet controlled using a signal from the microcontroller.
- Both servos of the arm, carrying out the "sweeping" motion.
- Ultrasonic sensor capable of object detection.
- LED headlights.

We also applied this method to various subsystems for the controller module, including:

- LCD display.
- Speaker capable of beeping faster and slower.

- LED mode of operation indicators.
- ADC joystick control.
- ADC push button operation.
- JDY-40 radio transmittal and recieval (for both robot and controller).

After each of these modules worked individually, we then integrated them into modules "closely" related to one another. For example, after the arm servos and the electromagnet were individually working we then integrated them together to make a single "coin-pickup" function. We repeated this process until all of the robot and controller code was integrated into a single C file. Although a considerable time was spent debugging, this method of integration helped tremendously in narrowing down errors.

2.3 Data Collection

The robot base collects data from 4 different inputs: input commands from the JDY-40 Radio, voltage from the perimeter detectors, frequency readings from the metal detector and sonar data from the ultrasonic sensor. The JDY-40 Radio sends characters to the robot in manual mode. To ensure that these commands are received, we utilized the serial port with PuTTY to collect data on this communication. For the perimeter detector, we displayed collected voltage readings and tested the functionality using signals generated by the frequency generator. Similarly, with our metal detection circuit, we displayed the frequency readings using PuTTY. For both the metal detection and perimeter detector circuits, we utilized an oscilloscope to visualize collected data. We also used a multimeter during testing to ensure that the wiring was accurate and that all components were powered with the correct voltage. The microcontroller also receives distance data from the ultrasonic sensor detecting objects within a $5\,\mathrm{cm}$range. The remote controller similarly collects data from the robot base using the JDY-40 radio receiving metal strength readings.

2.4 Data Synthesis

In terms of data synthesis, the data from the metal detector, perimeter detector and ultrasonic sensor needed to be synthesized.

The frequency readings are synthesized by the software program as the microcontroller measures the period of the square signal at the input pin. Then, this period data is converted into frequency and displayed with PuTTY. Using multiple different coins, we confirmed that the metal detection circuit worked properly, seeing spikes in frequency.

With perimeter detection, the input pin to the microcontroller is first configured as an ADC input as it receives analog data. This data is then converted into a voltage using numerical conversions in the code and printed in PuTTY for analysis. Observing these voltage readings, we confirmed that the robot could detect the square signals in the perimeter wire.

For the sonar, the ultrasonic sensor transmits a pulse via the trigger pin hitting any objects in front. Simultaneously, the microcontroller starts a timer. When the pulse hits the object, it is reflected and received by the echo pin, the timer is also stopped. The microcontroller then calculates the time difference, which is used to determine the distance to the object by correlating pulse duration with travel distance.

2.5 Analysis of Results

To validate our measurement accuracy and conclusions, we conducted extensive tests to evaluate perimeter detection, coin pickup and sonar object sensing.

Testing the perimeter detection through PuTTY required iterative adjustments to achieve a 100% detection rate. Due to the non-ideal behaviour of components, the startup calibration which established a baseline voltage reading proved crucial for accurate detection by accounting for variations in ambient electromagnetic noise. Similarly, sonar detection was fairly straightforward due to only detecting objects in front of the robot. This only meant that a suitable distance value had to be found to perform a turn.

Coin detection and the electromagnet dealt with frequency readings from the tank circuit. We observed that older, rusty coins were fine to read, but difficult to pick up, hence a limitation for our robot.

Arm servo control and pickup range required PWM signals. Thus, the smoothness of operation was tested such that each coin pick-up would not fling the coin due to the arm movement being too fast. With about a 5cm pickup radius, this proved satisfactory for our application as we achieved a 100% pick-up rate.

3 Design

3.1 Use of Process

Our group first gathered and read both the lecture slides and the project document to determine the key requirements of this project. We divided the project into smaller parts to be completed by each team member, and the various smaller sections of the project are integrated into our main code at the end. In developing various sections of the project, individually or in small groups, we focused on communicating with each other to ensure that individual sections could be integrated into the overall system. We also assisted fellow team members in solving software bugs or wiring problems in hardware. Using this model, we can ensure that our group was able to finish more parts of the project in the limited time given while finishing each section faster with the aid of other teammates. In the process of integrating different software and hardware components, our group tried to optimize performance while also ensuring efficiency and organization in our design. Individual pieces of the system are also repeatedly tested, both individually and combined with other components, to ensure successful operation and to determine potential problems.

3.2 Need and Constraint Identification

Our group reviewed the project document extensively to determine both the requirements and extra features for this project. In project 2, our group is required to design a coin-picking robot powered by a battery, operating in automatic and manual mode. The coin-picking mechanism picks up coins using an electromagnet and deposits the coin into a storage compartment. As clearly stated by the project document, the coin-picking system must reliably detect coins and successfully pick up the coins each time, and hence it is especially important to consider accuracy and reliability when designing the system. The system also needs to be easy to operate and clearly accessible for the user, so all relevant information should be clearly displayed on the remote and all design considerations should focus on making the robot more convenient and accessible for the user. Designs and potential additional features should also focus on addressing the possible challenges that the user might encounter or functions which may be practical and plausible for a potential user.

The EFM8 and STM32 microcontrollers used in this project are limited in the number of input pins and processing power. Therefore, functional requirements are implemented more carefully on the hardware when taking into account of the limited input pins. Extra features are also limited by the number of input pins as well, and many of our original ideas for extra features could not be implemented. The coin-picking robot was also required to be designed, built, and tested in a limited time setting, which limited the number of additional features that we could add.

3.3 Problem Specification

Due to the limited number of pins on the EFM8 and STM32 microcontrollers that we used, our team was required to strategically plan the usage of the GPIO ports. To address the potential challenge of other obstacles on the track, a sonar sensor was added to detect objects directly in the path of the coin-picking robot and turn in a different direction in automatic mode, similar to its behaviour when it detects the perimeter. After 20 coins are picked up in automatic mode, the system will perform a victory dance to indicate its completion before switching to manual mode, improving accessibility and convenience for the user.

3.4 Solution Generation

To meet the functional specifications outlined in the project document, we explored multiple design functions as well as additional features for our coin picker to ensure versatility, accuracy, and convenience. Below is a brief summary:

1. Sonar Sensor:
    - A sonar sensor was added to the automatic mode to detect any objects directly in front of the robot; the robot would move back and turn in a random direction upon detecting an object in its path, similar to its behaviour when it reaches the perimeter wire.
2. ADC Push Buttons:
    - Analog-to-Digital Converter push buttons were used on the remote to control different features for the system while minimizing the number of input pins.
3. Victory Dance:
    - After the coin-picking robot finishes picking up 20 coins in automatic mode, it will do a victory dance to indicate that the automatic mode cycle is finished before moving to manual mode.
4. Headlights:
    - Headlights are added to the front of the robot mostly for aesthetic purposes, and also considering the possibility that the user should want to manually operate the robot in a dark environment.

3.5 Solution Evaluation

The coin-picking process using the electromagnet was designed with the necessary design specifications in mind. During the testing process, a challenge our team faced was the coin falling off the electromagnet when it swings up abruptly to move the coin to the storage unit. This problem was identified and resolved by first moving the electromagnet arm to a halfway point and then moving the arm up to avoid abrupt movement which could cause the coin to fall. We also recognized the possibility both in examining the functions of the inductor and in actual testing that the coin might vary slightly in its position from the electromagnet. Therefore, the electromagnet was programmed to sweep the area detectable by the inductor to account for all possibilities for the location of the coin. Since the project criteria and the instructor has stated explicitly that coins must be reliably picked up by the system, we made it a priority to ensure the consistency of the coin-picking mechanism for each iteration. In addition, a speaker on the remote will beep at higher frequencies when a coin is near the coin picker in manual mode. This was a requirement stated in the project document, and was therefore prioritized in its implementation.

In terms of additional features, we included a victory dance after 20 coins have been picked up to indicate to the user that automatic mode is finished. This decision is made in consideration of convenience for the user, as the user could more easily determine when the automatic process is finished. A sonar sensor is also added in automatic mode to detect the presence of objects directly in front of the coin-picking robot, in which the robot would move back and turn in a random direction if objects are detected. This accounts for potential obstacles along the track when the user operates the robot, and hence improves user experience.

3.6 Detailed Design

This section will outline the methodology and engineering principles employed throughout the development of this projects key components. We will provide an in-depth explanation of each block and the approaches that were taken to design each part, supported by relevant diagrams and source code. Refer to diagrams A.3 and A.4 for comprehensive circuit diagrams for the robot and controller respectively.

3.6.1 Startup Calibration

On startup, the robot pauses for approximately 10 seconds to perform perimeter and coin detection solely for the purpose of establishing baseline values. Specifically, the frequency is measured five times and the voltage ten times, with the highest values from each used as the base frequency and base voltage, respectively. Determining these values during startup ensures consistency in baseline readings for each session. This calibration is important, as frequency and voltage readings often varies between sessions. For both perimeter and coin detection, the robot checks whether the current readings exceed the respective baseline values to confirm the presence of a coin or the perimeter wire. It is also essential that the robot is positioned away from both the perimeter wire and any coins during startup to find the correct base values.

3.6.2 Wireless Communications

Communications between the remote controller and robot subsystems were done wirelessly. For this, we utilised the JDY-40 radio controller. We employ a master-slave communication architecture, with the remote controller being the master. The rationale behind this is to avoid collisions when communicating. This way, both microcontrollers can recognise when it is their time to send or receive data.

The communication starts when the master (remote controller) sends the start transmission byte (`STX 0x02`). The payload is then sent, with the encoding as described in Table 3.1. The communication between two subsystems are relatively simple and minimal, therefore we chose to omit sending an acknowledge (`ACK 0x06`) or end transmission (`ETX 0x03`) after the

7	6	5	4	3	2	1	0
`pushbutton`				`JS_X`		`JS_Y`	

(a) Subcaption.

Bit	3	2	1	0
Button 7	0	0	0	1
Undefined	X	X	X	0

(b) Encoding for `pushbutton`.

Bit	1	0
Undefined	0	0
Left	0	1
Right	1	0
Center	1	1

(c) Encoding for `JS_X`.

Bit	1	0
Undefined	0	0
Down	0	1
Up	1	0
Center	1	1

(d) Encoding for `JS_Y`.

Table 3.1. Payload byte.

payload. When requesting for data from the slave (robot), the master sends the enquiry byte (`ENQ 0x05`) to the slave. The slave then sends back an ASCII character array with the data. The array is terminated by a null character (`NUL 0x00`). On the receiving end (master), the characters are decoded to integers one at a time with the C standard library function `atoi()`. Initially, `sscanf()` was utilised. However, the microcontroller would consistently be stuck in a loop when executing the function, hence our decision to use `atoi()`.

3.6.3 Perimeter Detection

To restrict the robot's movement in automatic mode, we implemented a perimeter detector on the robot. As a prerequisite, the perimeter must be lined with a wire that carries an alternating current. We found that a frequency of $5000\,\mathrm{Hz}$ works best in our specific test cases. This signal can either be generated from an arbitrary waveform generator (AWG), or a 555 timer oscillator circuit [1] with resistor and capacitor values determined by the equation:

$$f = \frac{1}{\log(2)\,(R_A + 2R_B)\,C}.$$

The alternating current produces a time-varying magnetic field. An inductive sensor is utilised to detect the field, operating on the principle of **Faraday's law of induction**, where the induced voltage is given by

$$\varepsilon = -N\frac{\mathrm{d}\Phi}{\mathrm{d}t}. \tag{3.1}$$

The energy from the eddy current is then stored in the LC tank circuit, which will oscillate at an angular frequency $\omega = (LC)^{-1/2}$. The signals from the oscillations are then passed through a high-gain non-inverting op-amp and into a peak detector, which consists of a diode and a capacitor which follows and stores the peak voltage from the input. The peak detector output is finally passed to an analog input on the EFM8.

After the microcontroller performs the analog-to-digital conversion, the computed voltage is compared to a threshold voltage. Through testing, we found that $0.2\,\mathrm{V}$ above the baseline voltage found during initialisation works well as the threshold. When the microcontroller detects a voltage higher than the threshold, it starts a sequence of instructions to reroute the robot. This sequence begins by stopping the motors and reversing for approximately $400\,\mathrm{ms}$, which was added to reduce the chance of the robot escaping the boundaries when approaching the perimeter at an extremely acute angle. The microcontroller then generates a pseudorandom number in the range $[500, 1500)$ using the `rand()` C standard library function. The robot then turns left for the amount of time (in milliseconds) that was generated. The random turn amount is to reduce the probability of the robot being "stuck" in a loop turning between two points on the perimeter.

Two inductive sensors, arranged orthogonal to each other, are used to ensure reliable detection of the perimeter. This is due to the fact that the magnetic field generated by the perimeter wire is entirely in the azimuthal direction, as given by **Ampère's law** through a straight wire:

$$\mathbf{B} = \frac{\mu_0 I}{2\pi r}\hat{\mathbf{e}}_\varphi.$$

If the coil of inductive sensor is nearly parallel to the perimeter wire, the magnetic field through the plane of the coil will be extremely small. Thus, the magnetic flux will not induce a voltage large enough for the EFM8 to recognise the perimeter.

3.6.4 Object Detection

In automatic mode, the robot is able to detect and avoid foreign objects and obstacles. This is achieved using a sonar sensor. A $10\,\mu\text{s}$ pulse is first sent to the sonar. The return time, in microseconds, is then recorded. Using the equation

$$s = vt,$$

where s is the displacement, $v \approx 343\,\mathrm{ms}^{-1}$ is the speed of sound in air [2], and t is the time, we can calculate the distance from the robot to the object. We calibrated the robot to respond to objects within approximately $6\,\mathrm{cm}$. When the robot detects an object, it will choose a random direction and turn for a random amount of time, similar to the perimeter detector in section 3.6.3.

3.6.5 Coin Detection

The coin/metal detection circuit is essentially a Colpitts oscillator with a discrete CMOS inverter that generates a high-frequency AC signal. The frequency of the oscillator is derived from the equation

$$f = \frac{1}{2\pi\sqrt{LC}}, \quad \text{where } C = \frac{C_1 C_2}{C_1 + C_2}.$$

When a conductive object (such as a coin) enters the inductors electromagnetic field, eddy currents form within the metal and change the magnetic field. This results in a slight change in inductance, which is reflected as a change in the oscillator's frequency. The microcontroller then detects this frequency change, thus confirming the presence of metal and executes the coin pickup task.

The circuit consists of a PMOS-NMOS inverter stage that sustains oscillations in the inductor and capacitor tank circuit. As the robot gets closer to the coin, the inductance of the inductor reduces, increasing the frequency readings received by the robot. The output is then processed by the microcontroller, where frequency changes are measured and used to trigger the coin pickup mechanism. This allows the robot to detect coins efficiently while differentiating between metal and non-metal objects. See Table B.1 for details on frequency readings for specific coins.

3.6.6 Motor Control

Optocouplers were used to isolate the motor subsystem from the rest of the robot. This is done to isolate unwanted noise from inductive loads such as motors and the electromagnet. These work by utilising light to send signals, creating a physical disconnect between the two circuits. The output signals of the optocouplers are fed into the an H-bridge constructed using MOSFETs. By setting the signal high on MOSFETs diagonal to each other, we are able to change the direction of the motor spin.

3.6.7 Electromagnetic Arm

The electromagnetic arm is controlled by two servomotors, allowing for two degrees of freedom. The shoulder servo controls the rotation around the shoulder joint, connected to the robot base at the bottom. This allows movement in the horizontal plane. The elbow servo further allows movement in the vertical direction. Servo motors are typically controlled with a pulse-width modulated (PWM) signal. To change the position of the servomotors, the duty cycle of the PWM signal is altered by the microcontroller. A $7\,\Omega$ electromagnet is attached to the end of the arm. When activated, the electromagnet draws from a source of $6\,\text{V}$, resulting in a current of $0.857\,\text{A}$. From various tests conducted, we determined that the electromagnet has sufficient strength to pick up the heaviest denomination of Canadian coins.

3.6.8 Controller Joystick

The joystick is utilised to control the direction of the robot in manual mode. The joystick gave three outputs: two analog and one digital. The x- and y-direction (analog) outputs are given to the ADC to perform the calculation. We then send the result through the JDY-40 radio. The joystick switch is connected to a digital pin and likewise sent through the radio. Through many tests, we found that our joystick does not handle small inputs well. We found that in most cases, the we retrieve only three discrete values from the joystick per direction.

3.6.9 Analog Push-Button Array

To allow for adding multiple pushbuttons to the remote controller, an array of pushbuttons were configured as input to the analog-to-digital converter (ADC). The array of buttons acts as a voltage divider. The ADC then calculates the voltage of the input to determine the button pressed.

3.6.10 Automatic Mode

Using the perimeter and coin detection subsystems, the robot is able to pick up a specified number of coins automatically. As covered in sections 3.6.3 and 3.6.4, the robot will perform a random walk upon detecting the perimeter or obstacles. The robot will also automatically detect coins and activate the electromagnetic arm sequence. Eventually, this would result in all of the coins being picked up automatically. Once the robot has picked up the specified number of coins, the robot will stop, inform the user, and go back into manual mode. The number of coins to pick up is by default 20, but this may be changed at compile time with the preprocessor macro `NUM_COINS`—*exempli gratia*, `c51 -DNUM_COINS=25 -`

`o main.hex main.c.`

3.7 Solution Assessment

Movement (motor and joystick) control

First, movement in automatic mode was tested by setting the control signals high/low to get movement forward, backward, and rotation clockwise and anticlockwise. Furthermore, we tested the responsiveness of the joystick in manual mode to ensure the robot is not lagging.

Perimeter detection

Initial perimeter detection tests were evaluated by printing on PuTTY whenever a piece of wire carrying AC current was near the relevant inductors. However, to test in a more practical setting, we set up the mat with an AC current-carrying wire. Further tweaking and testing resulted in a 100% detection rate of the wire.

Coin detection and electromagnet

Similarly, coin detection was first tested by printing on PuTTY. Through further testing with the electromagnet, we found that older, rusty coins were hard to pickup. This is expected as the rusting process of the metal in the coin leads to impurities that interfere with our electromagnet. To improve this limitation, we would have to increase the strength of the electromagnet by increasing the number of turns, or the current. For practical purposes, 100% efficiency was achieved as all coins within our perimeter were picked up. See Table B.1 for data on frequency values.

Arm servo control

Following the coin detection tests, we integrated the arm servo control to it and tested using pwm. This allowed us to test the range of motion that can be achieved and so, created a function to test this. It was measured that the robot can pick up coins within a 5cm radius of the coin detector.

JDY-40 radio transmission and reception

To test the JDY-40 we tested multiple data transmissions to ensure that communication commands from master to slave were working properly. PuTTY prints were used to debug transmission and reception commands as well as data feedback of metal detection strength.

4 Life-Long Learning

Throughout this project, we identified learning gaps in applying physics concepts related to electromagnetism. While we had a foundational understanding of the physics, we had to independently learn how they apply to metal detection and perimeter sensing. Additionally, working with the JDY-40 wireless module and wireless communication was a fresh experience for many of us. It required us to get a good grasp of its communication protocol and learn to study the datasheet closely. Another key area of learning was coding for different microcontrollers, as each has its own syntax, register configurations, and nuances in timers. Beyond technical knowledge, we also developed better organizational skills, ensuring that circuit components were properly tracked and connected correctly to prevent short circuits. Moreover, this experience pushed us mentally to develop reliable debugging strategies, especially as multiple modules often broke during rewiring.

Many of our previous courses in first semester helped us during the design and execution of this project. For example, ELEC 211 provided the necessary background on electromagnetic principles, helping us understand how the Colpitts oscillator detects metal objects and how the perimeter detector picks up induced signals from the boundary wire. As well, APSC 101 from first year provided a valuable experience in building and controlling a robotic claw. This gave us confidence in assembling the base of the robot, the coin picking mechanism and other mechanical parts. Overall, this project significantly expanded our knowledge of robotics, wireless communication and developing a complex system.

5 Conclusions

The coin picking robot was designed for detecting and collecting coins in two modes of operation: manual and automatic. In automatic mode, the robot effectively detects both coins and the perimeter wire. It moves continuously and turns randomly when it detects the boundary or an object in front of it. In manual mode, our remote sends commands to the robot with very little delay. The remote controls the motors adequately for omnidirectional movement. Additionally, the robot routinely transmits metal detection data. The remote then activates the buzzer with the frequency directly correlating to how close the coin is to the robot. The LCD will also display the metal strength, similar to a real metal detector. Notable features in our design include sonar sensor, ADC push buttons, victory dance functionality, and aesthetic headlights. The ultrasonic sonar allowed the robot to avoid objects in its path during coin collection and requires both hardware and software integration with the sensor. The ADC buttons were added to introduce more functionality to the remote controller in sending specific commands. The victory dance, occurring after picking up 20 coins, is a specific robot action where the robot spins around. Headlights were added to the robot for both aesthetics and practicality to provide lighting in darker environments.

Most problems encountered were due to errors in wiring, faulty components and transmission problems. In particular, we faced many problems during integration due to wiring errors when connecting different modules. However, these were thoroughly debugged and helped ensure our understanding of every component. Throughout the project, we had two servos become faulty due to mechanical wear and needed to have them replaced. Furthermore, our electromagnet had issues with drawing too much current because of its low resistance. This was fixed by consulting with the professor and finding a replacement with higher resistance. With the radio, there were issues with sending commands and the robot freezing after the certain commands. This was because delays in our program cause issues with the robot receiving commands from the remote controller. These problems were fixed after sifting through datasheets and clearing the receiving buffer after each delay.

The project required approximately 75 hours of work:

- Hardware Assembly: 15 hours
- Software Development: 10 hours
- Module integration: 35 hours
- Bonus Features: 7 hours
- Final testing and calibration: 8 hours

6 References

[1] "555 & 556 Timers," Signetics, 1973. [Online]. Available: https://www.rfcafe.com/miscellany/cool-products/Signetics-555-556-Timer-1973-Databook.pdf. [Accessed: Apr 5, 2025].

[2] Wikipedia contributors, "Speed of Sound," Wikipedia, The Free Encyclopedia, Apr 5, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Speed_of_sound. [Accessed: Apr 7, 2025].

7 Bibliography

C. K. Alexander and M. N. O. Sadiku, Fundamentals of Electric Circuits, 5th ed. New York, NY, USA: McGraw-Hill, 2013. Eindhoven University of Technology, 8051 Instruction Set, Eindhoven University of Technology, [Online]. Available: https://aeb.win.tue.nl/comp/8051/set8051.html. [Accessed: Apr 8, 2025].

Microchip Technology Inc., "MCP1700 Low Quiescent Current LDO Data Sheet," Rev. E, 2018. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/MCP1700-Low-Quiescent-Current-LDO-20001826E.pdf. [Accessed: Apr 8, 2025].

RCS Components, "JDY-40 Datasheet," [Online]. Available: https://www.rcscomponents.kiev.ua/datasheets/JDY-40-datasheet.pdf. [Accessed: Apr 8, 2025].

Silicon Laboratories, "EFM8 Sleepy Bee Family EFM8SB1 Data Sheet," Rev. 1.5, Mar. 2024. [Online]. Available: https://www.silabs.com/documents/public/data-sheets/efm8sb1-datasheet.pdf. [Accessed: Apr 8, 2025].

STMicroelectronics, "STM32F205xx STM32F207xx Datasheet," Rev. 18, Jul. 2020. [Online]. Available: https://www.st.com/resource/en/datasheet/cd00237391.pdf. [Accessed: Apr 8, 2025].

Tower Pro, "MG90S Micro Servo Datasheet," [Online]. Available: https://www.electronicoscaldas.com/datasheet/MG90S_Tower-Pro.pdf. [Accessed: Apr 8, 2025].

Appendix A. Diagrams

A.1 Hardware System Block Diagram

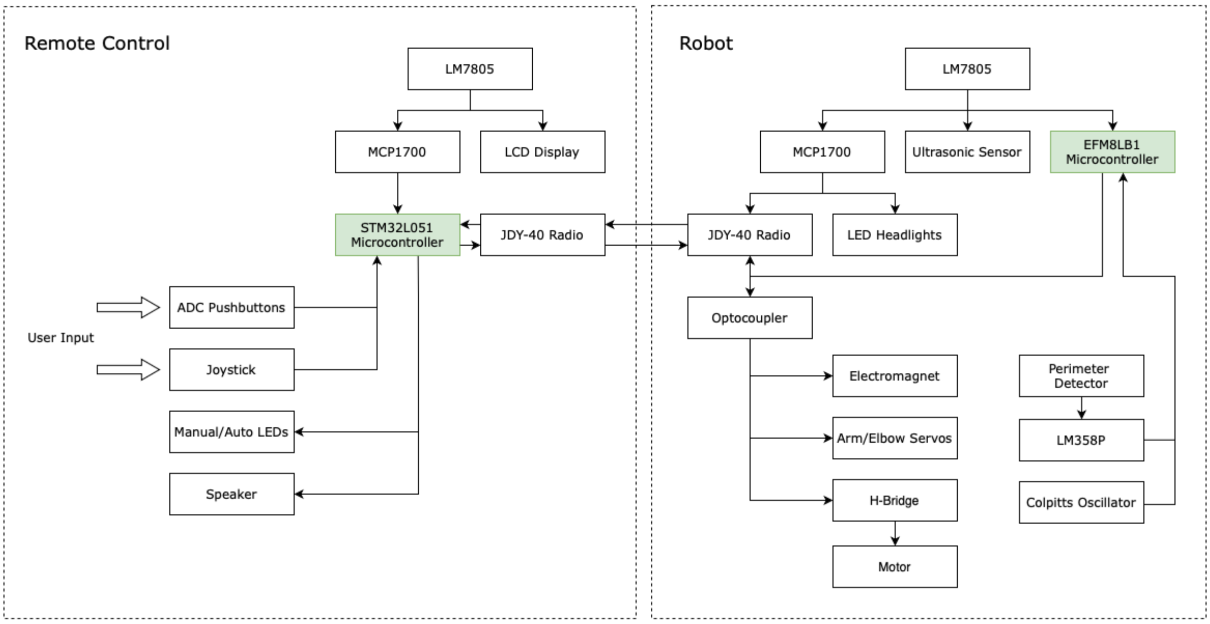

Figure A.1. Hardware block diagram.

A.2 Software System Block Diagram

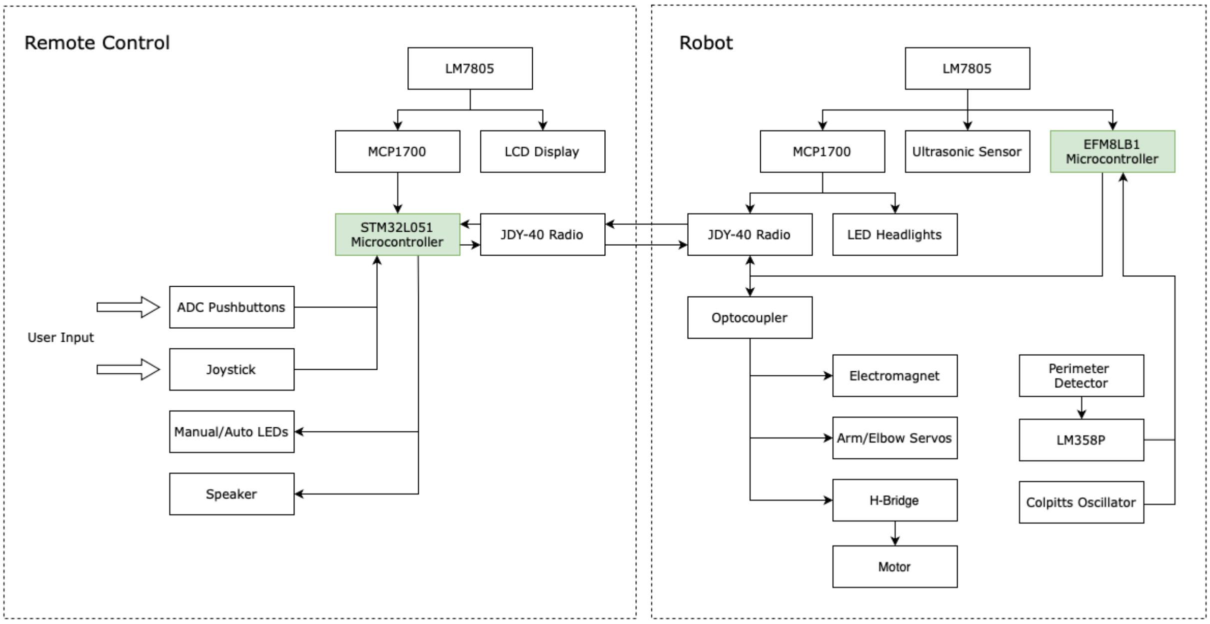

Figure A.2. Software block diagram.

A.3 Detailed Robot Circuit Diagram

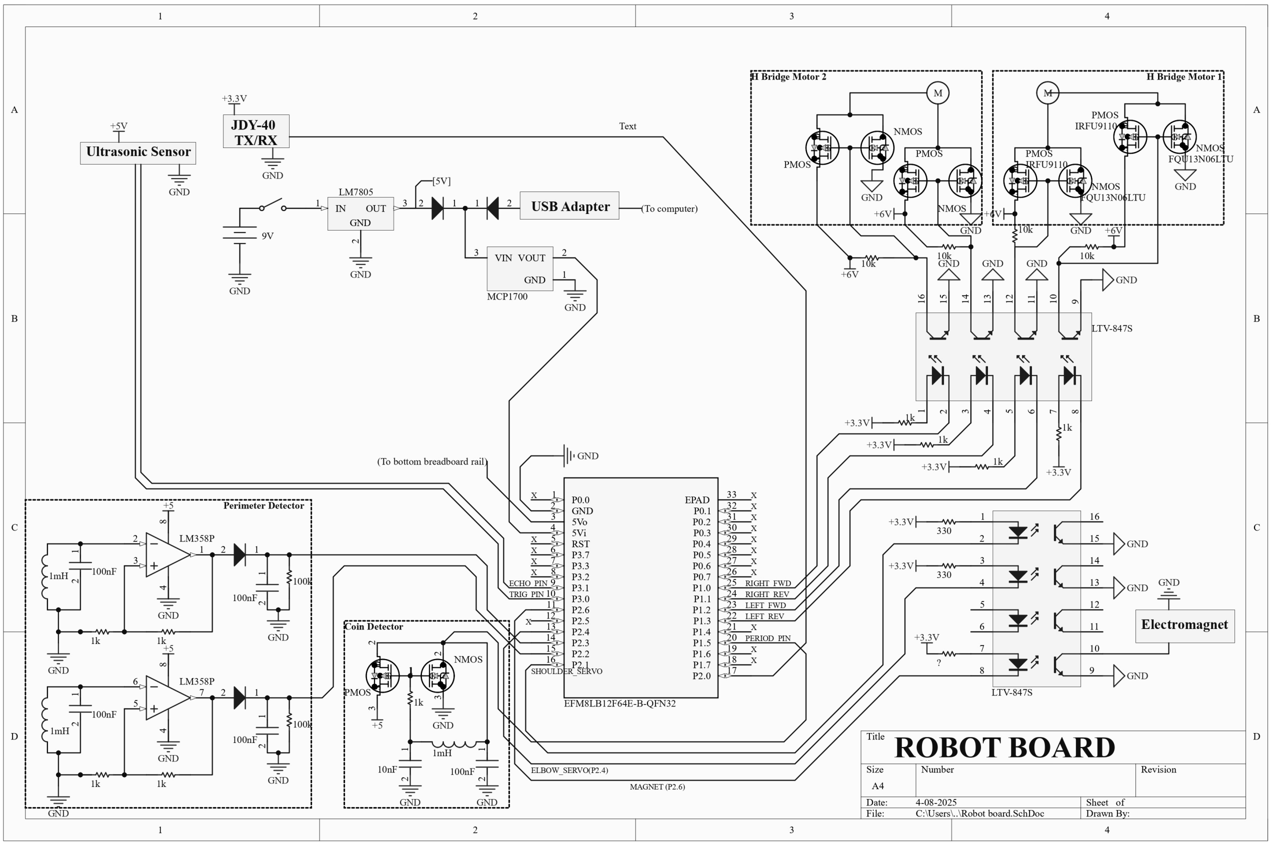

Figure A.3. Detailed robot circuit schematic.

A.4 Detailed Controller Circuit Diagram

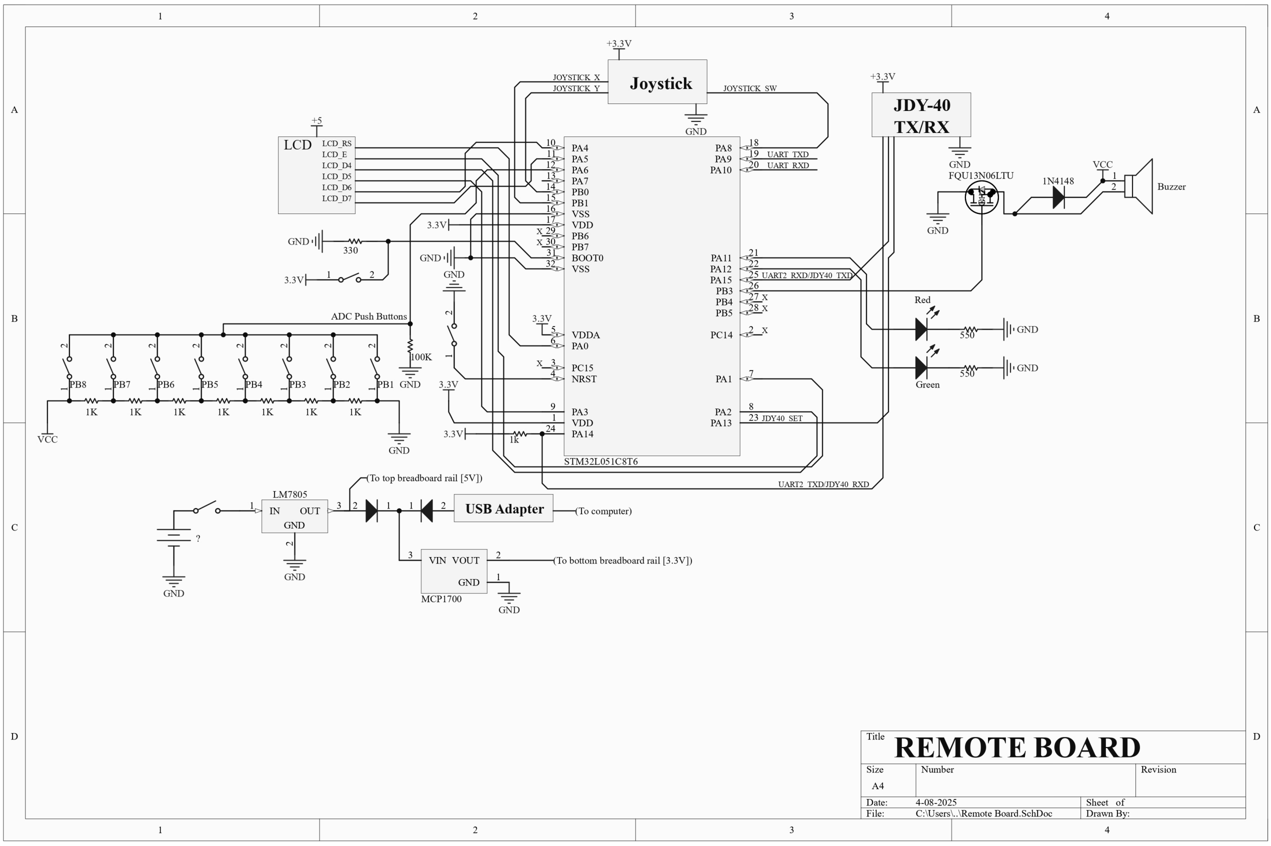

Figure A.4. Detailed controller circuit schematic.

Appendix B. Data

	Frequency, f (Hz)		Change, Δf (Hz)		
	Trial 1	Trial 2	Trial 1	Trial 2	Average change
Base	57 221	57 242	—	—	—
Nickel	57 468	57 480	247	238	242.5
Dime	57 428	57 452	207	210	208.5
Quarter	57 479	57 489	258	247	252.5
Loonie	57 497	57 516	276	274	275.0
Toonie	57 536	57 720	315	478	396.5

Table B.1. Frequency data collected for each type of coin. Two trial runs were conducted. The same coin was used for both trials for each denomination.

Appendix C. Program Source Code

Remote Controller Source

Listing C.1. `util.h`.

```c
/*
 * Coin Picking Robot (Remote)
 * util.h
 */

#ifndef UTIL_H
#define UTIL_H

#define SYSCLK 32000000L

#define LCD_RS_0 (GPIOA->ODR &= ~BIT0)
#define LCD_RS_1 (GPIOA->ODR |= BIT0)
#define LCD_E_0 (GPIOA->ODR &= ~BIT1)
#define LCD_E_1 (GPIOA->ODR |= BIT1)
#define LCD_D4_0 (GPIOA->ODR &= ~BIT2)
#define LCD_D4_1 (GPIOA->ODR |= BIT2)
#define LCD_D5_0 (GPIOA->ODR &= ~BIT3)
#define LCD_D5_1 (GPIOA->ODR |= BIT3)
#define LCD_D6_0 (GPIOA->ODR &= ~BIT4)
#define LCD_D6_1 (GPIOA->ODR |= BIT4)
#define LCD_D7_0 (GPIOA->ODR &= ~BIT5)
#define LCD_D7_1 (GPIOA->ODR |= BIT5)
#define CHARS_PER_LINE 16
#define MAXBUFFER 64

typedef struct ComBuffer {
        unsigned char buffer[MAXBUFFER];
        unsigned head, tail;
        unsigned count;
} ComBuffer;

void sleep(unsigned int ms);
void usleep(unsigned char us);

void lcd_init(void);
void lcd_print(char *s, unsigned char line, unsigned char clear);
void lcd_write_command(unsigned char x);
void lcd_write_data(unsigned char x);
void lcd_byte(unsigned char x);
void lcd_pulse(void);

void adc_init(void);
int adc_read(unsigned int channel);

void uart2_init(int baud);
int uart2_received(void);
int com2_read(int max, unsigned char *buf);
int com2_write(int count, unsigned char *buf);
int com2_egets(char *s, int size);
int com2_eputs(char *s);
int com2_echos(char *s, int size);
```

```
char com2_egetc(void);
void com2_eputc(char c);
char com2_echoc(void);

#endif /* UTIL_H */
```

Listing C.2. `util.c`.

```
/*
 * Coin Picking Robot (Remote)
 * util.c
 */

#include "include/stm32l051xx.h"
#include "util.h"

unsigned com2_open, com2_error, com2_busy;
ComBuffer com_tx_buf, com_rx_buf;

unsigned char com_getbuf(ComBuffer *buf);
int com_putbuf(ComBuffer *buf, unsigned char data);
unsigned com_length(ComBuffer *buf);
void usart2_tx(void);
void usart2_rx(void);

void USART2_Handler(void) {
        if (USART2->ISR & BIT7)
                usart2_tx();
        if (USART2->ISR & BIT5)
                usart2_rx();
}

void sleep(unsigned int ms) {
        unsigned int i;
        for (i = 0; i < 4*ms; ++i)
                usleep(250);
        return;
}

void usleep(unsigned char us) {
        SysTick->LOAD = (SYSCLK / 1000000L * us) - 1;
        SysTick->VAL = 0;
        SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk;
        while ((SysTick->CTRL & BIT16) == 0);
        SysTick->CTRL = 0x00;
        return;
}

void lcd_init(void) {
        LCD_E_0;
        sleep(20);

        lcd_write_command(0x33);
        lcd_write_command(0x33);
        lcd_write_command(0x32);

        lcd_write_command(0x28);
        lcd_write_command(0x0C);
        lcd_write_command(0x01);
        sleep(20);

        return;
}

void lcd_print(char *s, unsigned char line, unsigned char clear) {
        int i;

```

```c
	lcd_write_command(line == 1 ? 0x80 : 0xC0);
	sleep(5);
	for (i = 0; s[i] != 0; ++i)
		lcd_write_data(s[i]);
	if (clear)
		for (; i < CHARS_PER_LINE; ++i)
			lcd_write_data(' ');
	return;
}

void lcd_write_command(unsigned char x) {
	LCD_RS_0;
	lcd_byte(x);
	sleep(5);
	return;
}

void lcd_write_data(unsigned char x) {
	LCD_RS_1;
	lcd_byte(x);
	sleep(2);
	return;
}

void lcd_byte(unsigned char x) {
	if (x & 0x80) LCD_D7_1; else LCD_D7_0;
	if (x & 0x40) LCD_D6_1; else LCD_D6_0;
	if (x & 0x20) LCD_D5_1; else LCD_D5_0;
	if (x & 0x10) LCD_D4_1; else LCD_D4_0;
	lcd_pulse();

	usleep(40);

	if (x & 0x08) LCD_D7_1; else LCD_D7_0;
	if (x & 0x04) LCD_D6_1; else LCD_D6_0;
	if (x & 0x02) LCD_D5_1; else LCD_D5_0;
	if (x & 0x01) LCD_D4_1; else LCD_D4_0;
	lcd_pulse();

	return;
}

void lcd_pulse(void) {
	LCD_E_1;
	usleep(40);
	LCD_E_0;
	return;
}

void adc_init(void) {
	RCC->APB2ENR |= BIT9;

	/* ADC clock selection procedure (page 746 of RM0451). */
	ADC1->CFGR2 |= ADC_CFGR2_CKMODE;

	/* ADC enable sequence procedure (page 745 of RM0451). */
	ADC1->ISR |= ADC_ISR_ADRDY;
	ADC1->CR |= ADC_CR_ADEN;
	if ((ADC1->CFGR1 & ADC_CFGR1_AUTOFF) == 0) {
		while ((ADC1->ISR & ADC_ISR_ADRDY) == 0);
	}

	/* Calibration code procedure (page 745 of RM0451). */
	if ((ADC1->CR & ADC_CR_ADEN) != 0) {
		ADC1->CR |= ADC_CR_ADDIS;
	}
```

```
        ADC1->CR |= ADC_CR_ADCAL;
        while ((ADC1->ISR & ADC_ISR_EOCAL) == 0);
        ADC1->ISR |= ADC_ISR_EOCAL;
}

int adc_read(unsigned int channel) {
        /* Single conversion sequence code example - software trigger
         * (page 746 of RM0451). */
        ADC1->CFGR1 |= ADC_CFGR1_AUTOFF;
        ADC1->CHSELR = channel;
        ADC1->SMPR |= ADC_SMPR_SMP_0 | ADC_SMPR_SMP_1 | ADC_SMPR_SMP_2;
        if (channel == ADC_CHSELR_CHSEL17) {
                ADC->CCR |= ADC_CCR_VREFEN;
        }

        /* Perform the AD conversion. */
        ADC1->CR |= ADC_CR_ADSTART;
        while ((ADC1->ISR & ADC_ISR_EOC) == 0);

        return ADC1->DR;
}

void uart2_init(int baud) {
        int baud_rate_divisor;

        __disable_irq();

        com_rx_buf.head = com_rx_buf.tail = com_rx_buf.count = 0;
        com_tx_buf.head = com_tx_buf.tail = com_tx_buf.count = 0;
        com2_open = 1;
        com2_error = 0;

        RCC->IOPENR |= BIT0;

        baud_rate_divisor = SYSCLK;
        baud_rate_divisor = baud_rate_divisor / (long)baud;

        GPIOA->OSPEEDR |= BIT28;
        GPIOA->OTYPER &= ~BIT14;
        GPIOA->MODER = (GPIOA->MODER & ~(BIT28)) | BIT29;
        GPIOA->AFR[1] |= BIT26;

        GPIOA->MODER = (GPIOA->MODER & ~(BIT30)) | BIT31;
        GPIOA->AFR[1] |= BIT30;

        RCC->APB1ENR |= BIT17;

        USART2->CR1 |= (BIT2|BIT3|BIT5|BIT6);
        USART2->CR2 = 0x00000000;
        USART2->CR3 = 0x00000000;
        USART2->BRR = baud_rate_divisor;
        USART2->CR1 |= BIT0;

        NVIC->ISER[0] |= BIT28;

        __enable_irq();
}

int uart2_received(void) {
        return com_length(&com_rx_buf);
}

int com2_read(int max, unsigned char *buf) {
        unsigned i;

        if (!com2_open)
```

```
                return -1;

        i = 0;
        while ((i < max-1) && (com_length(&com_rx_buf)))
                buf[i++] = com_getbuf(&com_rx_buf);

        if (i > 0) {
                buf[i]=0;
                return i;
        } else {
                return 0;
        }
}

int com2_write(int count, unsigned char *buf) {
        unsigned i;

        if (!com2_open)
                return -1;

        if (count < MAXBUFFER)
                while ((MAXBUFFER - com_length(&com_tx_buf)) < count);
        else
                return -2;

        for (i = 0; i < count; ++i)
                com_putbuf(&com_tx_buf, buf[i]);

        if ((USART2->CR1 & BIT3) == 0) {
                USART2->CR1 |= BIT3;
                USART2->TDR = com_getbuf(&com_tx_buf);
        }

        return 0;
}

int com2_egets(char *s, int max) {
        int len;
        char c;

        if (!com2_open)
                return -1;

        len = 0;
        c = 0;
        while ((len < max-1) && (c != '\n')) {
                while (!com_length(&com_rx_buf));
                c = com_getbuf(&com_rx_buf);
                s[len++] = c;
        }

        if (len > 0) {
                s[len] = 0;
        }

        return len;
}

int com2_eputs(char *s) {
        if (!com2_open)
                return -1;

        while (*s)
                com2_write(1, s++);

        return 0;
```

```
}

int com2_echos(char *s, int max) {
        int len;
        char c;

        if (!com2_open)
                return -1;

        len = 0;
        c = 0;

        while ((len < max-1) && (c != '\r')) {
                while (!com_length(&com_rx_buf));
                c = com_getbuf(&com_rx_buf);
                com2_eputc(c);
                s[len++] = c;
        }

        if (len > 0) {
                s[len] = 0;
        }

        return len;
}

char com2_egetc(void) {
        return com_getbuf(&com_rx_buf);
}

void com2_eputc(char c) {
        com2_write(1, &c);
}

char com2_echoc(void) {
        char c;
        c = com2_egetc();
        com2_eputc(c);
        return c;
}

unsigned char com_getbuf(ComBuffer *buf) {
        unsigned char data;

        if (buf->count==0)
                return 0;

        __disable_irq();

        data = buf->buffer[buf->tail++];
        if (buf->tail == MAXBUFFER) buf->tail = 0;
        buf->count--;

        __enable_irq();

        return data;
}

int com_putbuf(ComBuffer *buf, unsigned char data) {
        if ((buf->head == buf->tail) && (buf->count != 0))
                return 1;

        __disable_irq();

        buf->buffer[buf->head++] = data;
        buf->count++;
```

```
		if (buf->head == MAXBUFFER)
			buf->head = 0;

		__enable_irq();

		return 0;
}

unsigned int com_length(ComBuffer *buf) {
		return buf->count;
}

void usart2_tx(void) {
		if (com_length(&com_tx_buf)) {
			USART2->TDR = com_getbuf(&com_tx_buf);
		} else {
			USART2->CR1 &= ~BIT3;
			if (USART2->ISR & BIT6) USART2->ICR |= BIT6;
			if (USART2->ISR & BIT7) USART2->RQR |= BIT4;
		}
}

void usart2_rx(void) {
		if (com_putbuf(&com_rx_buf, USART2->RDR))
			com2_error = 1;
}
```

Listing C.3. `main.c`.

```
/*
* Coin Picking Robot (Remote)
* main.c
*/

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

#include "include/stm32l051xx.h"
#include "include/serial.h"
#include "util.h"

#define VER_MAJOR 1
#define VER_MINOR 0

#define SYSCLK 32000000L
#define TICK_FREQ 1000L

#ifndef NDEBUG
		#define DEBUG_PRINT(fmt, ...)                                          \
			fprintf(stderr, "DEBUG: %s:%d: %s(): " fmt "\r\n",             \
				__FILE__, __LINE__, __func__, __VA_ARGS__)
#else
		#define DEBUG_PRINT(fmt, ...) do {} while (0)
#endif

/*
 *                     ----------
 *             VDD -|1        32|- VSS
 *            PC14 -|2        31|- BOOT0
 *            PC15 -|3        30|- PB7
 *            NRST -|4        29|- PB6
 *            VDDA -|5        28|- PB5
 *    LCD_RS   PA0 -|6        27|- PB4
 *    LCD_E    PA1 -|7        26|- PB3   BUZZER
 *    LCD_D4   PA2 -|8        25|- PA15  UART2_RXD/JDY40_TXD
```

```
 *      LCD_D5   PA3 -|9       24|- PA14  UART2_TXD/JDY40_RXD
 *      LCD_D6   PA4 -|10      23|- PA13  JDY40_SET
 *      LCD_D7   PA5 -|11      22|- PA12  MANUAL (GREEN) LED
 *      ADC PB   PA6 -|12      21|- PA11  AUTOMATIC (RED) LED
 *               PA7 -|13      20|- PA10  UART1_RXD
 * JOYSTICK_Y   PB0 -|14      19|- PA9   UART1_TXD
 * JOYSTICK_X   PB1 -|15      18|- PA8   JOYSTICK_SW
 *               VSS -|16      17|- VDD
 *                     ----------
 */

volatile int count = 0;
volatile int count_threshold = 1000;

void init(void);
void timer2_init(void);
void send_command(char *s);
void reception_off(void);
int pb_read(void);

int main(void) {
        int x, y, timeout;
        char *buf;
        int frequency = 1000;
        int duty_cycle = 50;
        int count = 0;
        char buff[80];
        int metal_strength;
        char lcd_buff[80];

        init();
        timer2_init();
        uart2_init(9600);
        adc_init();
        lcd_init();

        DEBUG_PRINT("Coin picking robot, version %d.%d (%s %s)", \
                VER_MAJOR, VER_MINOR, __DATE__, __TIME__);

        sleep(1000);

        reception_off();

        /* Retrieve current configuration. */
        send_command("AT+VER\r\n");
        send_command("AT+BAUD\r\n");
        send_command("AT+RFID\r\n");
        send_command("AT+DVID\r\n");
        send_command("AT+RFC\r\n");
        send_command("AT+POWE\r\n");
        send_command("AT+CLSS\r\n");

        /* Set device ID to 0xC0A8 and switch to channel 108. */
        send_command("AT+DVIDC0A8\r\n");
        send_command("AT+RFC108\r\n");

        buf = malloc(80);
        strcpy(buf, 0);

        sleep(500);

        GPIOA->ODR ^= BIT12; // Set manual LED

        while (1) {
                x = adc_read(ADC_CHSELR_CHSEL8);
                DEBUG_PRINT("x: %d", x);
```

```
		y = adc_read(ADC_CHSELR_CHSEL9);
		DEBUG_PRINT("y: %d", y);

		buf[0] = 0;

		/* Read joystick inputs. */

		if (x <= 1800) {
			/* Left. */
			buf[0] |= 0b01 << 2;
		} else if (x >= 2300) {
			/* Right. */
			buf[0] |= 0b10 << 2;
		} else {
			/* Centre. */
			buf[0] |= 0b11 << 2;
		}

		if (y <= 1800) {
			/* Down. */
			buf[0] |= 0b01;
		} else if (y >= 2300) {
			/* Up. */
			buf[0] |= 0b10;
		} else {
			/* Centre. */
			buf[0] |= 0b11;
		}

		/* Joystick button. */
		if (!(GPIOA->IDR & BIT8)) {
			/* Software debounce. This uses a blocking delay.
			 * We may wish to change this later. */
			sleep(150);
			if (!(GPIOA->IDR & BIT8))
				buf[0] |= 0b1 << 4;
		}

		/* Read analogue push button array. */

		if (pb_read() == 8) {
			sleep(150);
			if (pb_read() == 8) {
				com2_eputc('#');
				GPIOA->ODR ^= BIT11;
				GPIOA->ODR ^= BIT12;
			}
		} else {
			com2_eputc(0x02);
		}

		DEBUG_PRINT("buf: %d", buf[0]);

		sleep(5);
		com2_eputc(buf[0]);
		sleep(5);
		com2_eputc('\n');

		sleep(5);
		com2_eputc(0x05);

		timeout = 0;

		/* Wait for response. */

		while(1) {
```

```
			if (uart2_received() > 0) break;
			if (++timeout > 250) break;
			usleep(100);
		}

		if (uart2_received() > 0) {
			com2_egets(buff, sizeof(buff));
			DEBUG_PRINT("Response: %s\r\n", buff);

			metal_strength = atoi(buff);

			DEBUG_PRINT("Metal Strength: %d\r\n", metal_strength);
			if (metal_strength < 0) {
				metal_strength *= -1;
			} else {
				sprintf(lcd_buff, "Strength: %d", metal_strength/16);
				lcd_print(lcd_buff, 1, 1);
			}

			if (metal_strength <= 100)
				count_threshold = 1000;
			else if (metal_strength > 100 && metal_strength <= 750)
				count_threshold = 700;
			else if (metal_strength > 750 && metal_strength < 1500)
				count_threshold = 400;
			else
				count_threshold = 200;
		}
		else {
			printf("No Response\r\n", buff);
		}
		sleep(50);
	}
}

void init(void) {
	/* Configure port A for very high speed (page 201). */
	GPIOA->OSPEEDR=0xFFFFFFFF;

	RCC->IOPENR |= BIT0;
	RCC->IOPENR |= BIT1;

	/* LCD display output. */
	GPIOA->MODER = (GPIOA->MODER & ~(BIT0|BIT1)) | BIT0;
	GPIOA->OTYPER &= ~BIT0;
	GPIOA->MODER = (GPIOA->MODER & ~(BIT2|BIT3)) | BIT2;
	GPIOA->OTYPER &= ~BIT1;
	GPIOA->MODER = (GPIOA->MODER & ~(BIT4|BIT5)) | BIT4;
	GPIOA->OTYPER &= ~BIT2;
	GPIOA->MODER = (GPIOA->MODER & ~(BIT6|BIT7)) | BIT6;
	GPIOA->OTYPER &= ~BIT3;
	GPIOA->MODER = (GPIOA->MODER & ~(BIT8|BIT9)) | BIT8;
	GPIOA->OTYPER &= ~BIT4;
	GPIOA->MODER = (GPIOA->MODER & ~(BIT10|BIT11)) | BIT10;
	GPIOA->OTYPER &= ~BIT5;

	/* Timer2 PWM */
	// Configure PB3 for alternate function (TIM2_CH2, pin 26 in LQFP32 package)
	GPIOB->OSPEEDR  |= BIT6; // MEDIUM SPEED
	GPIOB->OTYPER   &= ~BIT3; // Push-pull
	GPIOB->MODER    = (GPIOB->MODER & ~(BIT6)) | BIT7; // AF-Mode
	GPIOB->AFR[0]   |= BIT13; // AF2 selected (check table 16 in page 43 of "en.DM00108219.pdf")

	/* Radio output. */
	GPIOA->MODER = (GPIOA->MODER & ~(BIT27|BIT26)) | BIT26;
	GPIOA->ODR |= BIT13;
```

```

        /* Joystick button input. */
        GPIOA->MODER &= ~(BIT16|BIT17);
        GPIOA->PUPDR |= BIT16;
        GPIOA->PUPDR &= ~BIT17;

        /* Push-button array ADC input. */
        GPIOA->MODER &= ~(BIT12|BIT13);
        GPIOA->PUPDR |= BIT12;
        GPIOA->PUPDR &= ~BIT13;

        /* Manual/Automatic LEDs as output */
        GPIOA->MODER = (GPIOA->MODER & ~(BIT23)) | BIT22;
        GPIOA->MODER = (GPIOA->MODER & ~(BIT25)) | BIT24;
}

void timer2_init(void) {

        // Set up timer
        RCC->APB1ENR |= BIT0;  // turn on clock for timer2 (UM: page 177)
        //TIM2->ARR = SYSCLK/TICK_FREQ;
        TIM2->PSC = 31; // Set prescaler to 31999 to get 1ms tick (32MHz/32000=1kHz)
        TIM2->ARR=200; // Set auto-reload value to 999 for 1kHz (1ms tick)
        //TIM2->ARR = 255;
        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 PWM in channel 2 of Timer 2
        TIM2->CCMR1|=BIT14|BIT13; // PWM mode 1 ([6/5/4]=110)
        TIM2->CCMR1|=BIT11; // OC1PE=1
        TIM2->CCER|=BIT4; // Bit 4 CC1E: Capture/Compare 2 output enable.

        // Set PWM to 50%
        //TIM2->CCR2=SYSCLK/(TICK_FREQ*2);
        TIM2->CCR2=512;
        TIM2->EGR |= BIT0; // UG=1

        __enable_irq();
}

void send_command(char *s) {
        char buff[40];
        printf("Command: %s", s);
        GPIOA->ODR &= ~BIT13;
        sleep(10);
        com2_eputs(s);
        com2_egets(buff, sizeof(buff) - 1);
        GPIOA->ODR |= BIT13;
        sleep(10);
        printf("Response: %s", buff);
}

void reception_off(void) {
        GPIOA->ODR &= ~(BIT13);
        sleep(10);
        com2_eputs("AT+DVID0000\r\n");
        sleep(10);
        GPIOA->ODR |= BIT13;
        while (uart2_received() > 0)
                com2_egetc();
}

int pb_read(void) {
```

```c
	int pb_adc;
	pb_adc = adc_read(ADC_CHSELR_CHSEL6);
	DEBUG_PRINT("pb_adc: %d", pb_adc);

	/* A 1000 ohm resistor is connected to the analog push button
	 * array input pin. The following values were read for their
	 * respective buttons:
	 * - button 1: 320;
	 * - button 2: 460;
	 * - button 3: 580;
	 * - button 4: 730;
	 * - button 5: 930;
	 * - button 6: 1260;
	 * - button 7: 1920;
	 * - button 8: 4090. */
	if (pb_adc > 0xF00)
		return 8;
	if (pb_adc > 0x700)
		return 7;
	if (pb_adc > 0x480)
		return 6;
	if (pb_adc > 0x380)
		return 5;
	if (pb_adc > 0x280)
		return 4;
	if (pb_adc > 0x200)
		return 3;
	if (pb_adc > 0x180)
		return 2;
	if (pb_adc > 0x100)
		return 1;
	return -1;
}

void TIM2_Handler(void)
{
	TIM2->SR &= ~BIT0; // clear update interrupt flag
	count++;
	if (count > count_threshold)
	{
		count = 0;

		if(TIM2->CCR2 == 128)
			TIM2->CCR2 = 192; // 75%
		else
			TIM2->CCR2 = 128; // 50%
	}

}
```

Robot Source

Listing C.4. main.c.

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

#ifndef NUM_COINS
#define NUM_COINS 20
#endif

idata char buff[20];
#define PERIOD_PIN P1_5

```

```c
//Motor Pins
#define OUTPIN1    P1_0
#define OUTPIN2    P1_1
#define OUTPIN3    P1_2
#define OUTPIN4    P1_3

#define SYSCLK 72000000L // SYSCLK frequency in Hz
#define BAUDRATE 115200L

#define SARCLK 18000000L
#define RELOAD_10us (0x10000L-(SYSCLK/(12L*100000L))) // 10us rate

#define VDD 3.3035 // The measured value of VDD in volts

//Servo Vars
volatile unsigned int pwm_reload;
volatile unsigned char pwm_state=0;
volatile unsigned char count20ms;
volatile unsigned int servo_switch = 0; //0 for elbow, 1 for shoulder

#define ELBOW_SERVO P2_4
#define SHOULDER_SERVO P2_1
#define MAGNET    P2_6
#define ELBOW_MODE 0
#define SHOULDER_MODE 1
#define RELOAD_10MS (0x10000L-(SYSCLK/(12L*100L)))
#define ARM_DELAY 500

#define TRIG_PIN P3_0
#define ECHO_PIN P3_1

#define BACKLED_PIN P0_3

#ifndef NDEBUG
        #define DEBUG_PRINT(fmt, ...) printf("DEBUG: %s:%d: " fmt "\r\n", __FILE__, __LINE__, __VA_A
#else
        #define DEBUG_PRINT(fmt, ...) do {} while (0)
#endif

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;
```

```c
		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 as outputs
	P0MDOUT |= 0b_0001_1001; // Configure UART0 TX (P0.4) and UART1 TX (P0.0) as push-pull outpu
	P1MDOUT |= 0b_1010_1111; // OUPTUT1 to OUTPUT4, coin detector
	P2MDOUT|=0b_0101_0011; //Shoulder and elbow servo, electromagnet

	P3MDOUT |= 0x01;      // P3.0 (TRIG) push-pull
	P3MDOUT &= ~0x02;     // P3.1 (ECHO) open-drain
	XBR0     = 0x01; // Enable UART0 on P0.4(TX) and P0.5(RX)
	XBR1     = 0X00; //
	XBR2     = 0x41; // Enable crossbar and uart 1

	#if (((SYSCLK/BAUDRATE)/(2L*12L))>0xFFL)
		#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

	// Initialize timer 5 for periodic interrupts
	SFRPAGE=0x10;
	TMR5CN0=0x00;
	TMR5=0xffff;   // Set to reload immediately
	EIE2|=0b_0000_1000; // Enable Timer5 interrupts
	TR5=1;         // Start Timer5 (TMR5CN0 is bit addressable)

	EA=1;

	SFRPAGE=0x00;

	return 0;
}


//-----------------------------------------------------------------------------------//
void Timer5_ISR (void) interrupt INTERRUPT_TIMER5
{
	SFRPAGE=0x10;
	TF5H = 0; // Clear Timer5 interrupt flag
	// Since the maximum time we can achieve with this timer in the
```

```
        // configuration above is about 10ms, implement a simple state
        // machine to produce the required 20ms period.

        if (servo_switch == ELBOW_MODE){
                switch (pwm_state)
                {
                                case 0:
                                        ELBOW_SERVO=1;
                                        TMR5RL=RELOAD_10MS;
                                        pwm_state=1;
                                        count20ms++;
                                break;
                                case 1:
                                        ELBOW_SERVO=0;
                                        TMR5RL=RELOAD_10MS-pwm_reload;
                                        pwm_state=2;
                                break;
                                default:
                                        ELBOW_SERVO=0;
                                        TMR5RL=pwm_reload;
                                        pwm_state=0;
                                break;
                }
        } else {
                switch (pwm_state)
                {
                                case 0:
                                        SHOULDER_SERVO=1;
                                        TMR5RL=RELOAD_10MS;
                                        pwm_state=1;
                                        count20ms++;
                                break;
                                case 1:
                                        SHOULDER_SERVO=0;
                                        TMR5RL=RELOAD_10MS-pwm_reload;
                                        pwm_state=2;
                                break;
                                default:
                                        SHOULDER_SERVO=0;
                                        TMR5RL=pwm_reload;
                                        pwm_state=0;
                                break;
                }
        }
}


// 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;
        for(j=ms; j!=0; j--)
        {
                Timer3us(249);
                Timer3us(249);
                Timer3us(249);
                Timer3us(250);
        }
}

void elbow_control(float pulse){
        servo_switch = ELBOW_MODE;
        pwm_reload=0x10000L-(SYSCLK*pulse*1.0e-3)/12.0;
}

void shoulder_control(float pulse){
        servo_switch = SHOULDER_MODE;
        pwm_reload=0x10000L-(SYSCLK*pulse*1.0e-3)/12.0;
}

void coin_pickup(void){
        MAGNET = 1;
        shoulder_control(1.5);
        waitms(ARM_DELAY);
        elbow_control(2.4);
        waitms(ARM_DELAY);
        shoulder_control(2.4);
        waitms(ARM_DELAY);
        shoulder_control(1.5);
        waitms(ARM_DELAY);
        elbow_control(1.8);
        waitms(ARM_DELAY);
        elbow_control(1.0);
        waitms(ARM_DELAY);
        shoulder_control(1.0);
        waitms(ARM_DELAY);
        MAGNET = 0;
        waitms(ARM_DELAY);
        elbow_control(1.0);
        waitms(ARM_DELAY);
        shoulder_control(1.2);
}

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:
                (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) / (
                (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
		(0x0 << 5) | // ADINT. Set by hardware upon completion of a data conversion. Must be
		(0x0 << 4) | // ADBUSY. Writing 1 to this bit initiates an ADC conversion when ADCM
		(0x0 << 3) | // ADWINT. Set by hardware when the contents of ADC0H:ADC0L fall within
		(0x0 << 2) | // ADGN (Gain Control). 0x0: PGA gain=1. 0x1: PGA gain=0.75. 0x2: PGA g
		(0x0 << 0) ; // TEMPE. 0: Disable the Temperature Sensor. 1: Enable the Temperature

	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.
		(0x1F << 0); // ADPWR. Power Up Delay Time. Tpwrtime = ((4 * (ADPWR + 1)) + 2) / (Fa

	ADC0CN2 =
		(0x0 << 7) | // PACEN. 0x0: The ADC accumulator is over-written.  0x1: The ADC accum
		(0x0 << 0) ; // ADCM. 0x0: ADBUSY, 0x1: TIMER0, 0x2: TIMER2, 0x3: TIMER3, 0x4: CNVST

	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);
}

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;
}
```

```
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, unsigned char n)
{
	char c;
	unsigned char cnt;

	cnt=0;
	while(1)
	{
```

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

                if (cnt<n)
                {
                        cnt++;
                        *s=c;
                        s++;
                }
                else
                {
                        *s=0;
                        return;
                }
        }
}

// 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, sizeof(buff)-1);
        waitms(10);
        P2_0=1; // 'set' pin to 1 is normal operation mode.
        printf("Response: %s\r\n", buff);
}

void ReceptionOff (void)
{
        P2_0=0; // 'set' pin to 0 is 'AT' mode.
        waitms(10);
        sendstr1("AT+DVID0000\r\n"); // Some unused id, so that we get nothing in RXD1.
        waitms(10);
        // Clear Overrun and Parity error flags
        SCON1&=0b_0011_1111;
        P2_0=1; // 'set' pin to 1 is normal operation mode.
```

```
}

float Volts_at_Pin(unsigned char pin)
{
	 return ((ADC_at_Pin(pin)*VDD)/0b_0011_1111_1111_1111);
}

// Measure the period of a square signal at PERIOD_PIN
unsigned long GetPeriod (int n)
{
	unsigned int overflow_count;
	unsigned char i;

	TR0=0; // Stop Timer/Counter 0
	TMOD&=0b_1111_0000; // Set the bits of Timer/Counter 0 to zero
	TMOD|=0b_0000_0001; // Timer/Counter 0 used as a 16-bit timer

	// Reset the counter
	TR0=0;
	TL0=0; TH0=0; TF0=0; overflow_count=0;
	TR0=1;
	while(PERIOD_PIN!=0) // Wait for the signal to be zero
	{
		if(TF0==1) // Did the 16-bit timer overflow?
		{
			TF0=0;
			overflow_count++;
			if(overflow_count==10) // If it overflows too many times assume no signal is
			{
				TR0=0;
				return 0; // No signal
			}
		}
	}

	// Reset the counter
	TR0=0;
	TL0=0; TH0=0; TF0=0; overflow_count=0;
	TR0=1;
	while(PERIOD_PIN!=1) // Wait for the signal to be one
	{
		if(TF0==1) // Did the 16-bit timer overflow?
		{
			TF0=0;
			overflow_count++;
			if(overflow_count==10) // If it overflows too many times assume no signal is
			{
				TR0=0;
				return 0; // No signal
			}
		}
	}

	// Reset the counter
	TR0=0;
	TL0=0; TH0=0; TF0=0; overflow_count=0;
	TR0=1; // Start the timer
	for(i=0; i<n; i++) // Measure the time of 'n' periods
	{
		while(PERIOD_PIN!=0) // Wait for the signal to be zero
		{
			if(TF0==1) // Did the 16-bit timer overflow?
			{
				TF0=0;
				overflow_count++;
			}
```

```
			}
			while(PERIOD_PIN!=1) // Wait for the signal to be one
			{
				if(TF0==1) // Did the 16-bit timer overflow?
				{
					TF0=0;
					overflow_count++;
				}
			}
		}
		TR0=0; // Stop timer 0, the 24-bit number [overflow_count-TH0-TL0] has the period in clock c

		return (overflow_count*65536+TH0*256+TL0);
}

void eputs(char *String)
{
		while(*String)
		{
			putchar(*String);
			String++;
		}
}

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

		if(val<0)
		{
			putchar('-');
			val*=-1;
		}

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


void motor_stop (void) {
		OUTPIN1=0;
		OUTPIN2=0;
		OUTPIN3=0;
		OUTPIN4=0;
}


void motor_forward (void) {
		BACKLED_PIN = 1;
		OUTPIN1=1;
		OUTPIN2=0;
		OUTPIN3=1;
		OUTPIN4=0;
}

void motor_backward (void) {
```

```
        BACKLED_PIN = 1;
        OUTPIN1=0;
        OUTPIN2=1;
        OUTPIN3=0;
        OUTPIN4=1;
}


void motor_left (void) {
        OUTPIN1=1;
        OUTPIN2=0;
        OUTPIN3=0;
        OUTPIN4=1;
}


void motor_right (void) {
        OUTPIN1=0;
        OUTPIN2=1;
        OUTPIN3=1;
        OUTPIN4=0;
}



void random_turn(void) {
        unsigned int turn_time = (rand() % 1000) + 500; // Random turn duration between 500ms and 15

        motor_left();

        waitms(turn_time);
        motor_stop();
        waitms(500); // Pause before moving forward again
}

void sonar_turn(void) {
        unsigned int turn_direction = rand() % 2; // Randomly choose left (0) or right (1) // not ne
        unsigned int turn_time_sonar = (rand() % 500) + 250; // Random turn duration between 500ms a


        if (turn_direction == 1) {
                motor_right();
        } else {
                motor_left();
        }

        waitms(turn_time_sonar);
        motor_stop();
        waitms(500); // Pause before moving forward again
}

unsigned long base_freq (void) {

        long int count, f = 0;
        long int base_f = 0;
        int n = 0;

        while (n < 5) {
                count=GetPeriod(30);
                f=(SYSCLK*30.0)/(count*12);
                if (base_f < f) {
                        base_f = f;
                }
                n++;

                waitms(100);
```

```c
        }

        DEBUG_PRINT("Base frequency: %ld", base_f);
        return base_f;
}


float base_volt(unsigned char pin) {
        float volt = 0;
        float base_v = 0;
        int n = 0;

        while (n < 10) {
                volt = Volts_at_Pin(pin);
                if (base_v < volt){
                        base_v = volt;
                }
                n++;
                waitms(100);
        }

        DEBUG_PRINT("Base voltage: %f", base_v);
        return base_v;
}

void victory_dance(void) {
        motor_left();
        waitms(3000);
        motor_stop();
        shoulder_control(1.5);
        waitms(500);
        elbow_control(2.4);
        waitms(500);
        elbow_control(1.0);
        waitms(500);
        elbow_control(2.4);
        waitms(500);
        elbow_control(1.0);
        waitms(500);
        shoulder_control(1.2);
}

void send_trigger_pulse() {
        TRIG_PIN = 1;
        Timer3us(10);  // 10 ţs pulse
        TRIG_PIN = 0;
}

unsigned int measure_echo_pulse() {
        unsigned int duration = 0;

        while (!ECHO_PIN);          // Wait for echo to go HIGH
        while (ECHO_PIN) {          // Measure time echo stays HIGH
                Timer3us(1);
                duration++;
        }

        return duration;
}

void main (void)
{
        int is_auto_mode;
        long int j, v;
        float perimeter_1, perimeter_2, base_volt_1, base_volt_2;
        long int count, f, threshold_freq;
```

```

        int coin = 0;
        char c;

        unsigned int echo_time;

        float pulse_width;
        count20ms=0;
        is_auto_mode = 0;

        waitms(500);
        printf("\r\nEFM8LB12 JDY-40 Slave Test.\r\n");

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

        TRIG_PIN = 0;  // Ensure TRIG is LOW

        UART1_Init(9600);

        ReceptionOff();

        // To check configuration
        SendATCommand("AT+VER\r\n");
        SendATCommand("AT+BAUD\r\n");
        SendATCommand("AT+RFID\r\n");
        SendATCommand("AT+DVID\r\n");
        SendATCommand("AT+RFC\r\n");
        SendATCommand("AT+POWE\r\n");
        SendATCommand("AT+CLSS\r\n");

        SendATCommand("AT+DVIDC0A8\r\n");
        SendATCommand("AT+RFC108\r\n");

        waitms(1000); // Wait a second to give PuTTy a chance to start

        motor_stop();

        f = 0;
        threshold_freq = base_freq() + 100;
        base_volt_1 = base_volt(QFP32_MUX_P2_2);
        base_volt_2 = base_volt(QFP32_MUX_P2_3);

        while (1)
        {
                count=GetPeriod(30);
                if (count>0)
                {
                        f=(SYSCLK*30.0)/(count*12);
                        eputs("f=");
                        PrintNumber(f, 10, 7);
                }
                else
                {
                        eputs("NO SIGNAL                     \r");
                }

                c = 0;
                if (RXU1()) {
                        c = getchar1();
                        DEBUG_PRINT("%c", c);
                }

                if (c == '#')
                        is_auto_mode = !is_auto_mode;
```

```

restart:
		if (is_auto_mode) {
			DEBUG_PRINT("Automatic", 0);
			goto automatic;
		} else {
			DEBUG_PRINT("Manual", 0);
			goto manual;
		}

manual:

		if(c==0x02) // Master is sending message
		{
			getstr1(buff, sizeof(buff)-1);
			DEBUG_PRINT("%c", buff[0]);

			if (buff[0] & (0b1 << 4)) {
				DEBUG_PRINT("Pickup", 0);
				motor_stop();
				// waitms(1000);
				coin_pickup();
				// continue;
				SFRPAGE = 0x20;
				c = SBUF1;
				SFRPAGE = 0x00;
			} else {
				switch (buff[0] & 0b1111) {
				case 0b0101:
				case 0b0110:
				case 0b0111:
					DEBUG_PRINT("Left", 0);
					motor_left();
					break;
				case 0b1001:
				case 0b1010:
				case 0b1011:
					DEBUG_PRINT("Right", 0);
					motor_right();
					break;
				case 0b1101:
					DEBUG_PRINT("Back", 0);
					motor_backward();
					break;
				case 0b1110:
					DEBUG_PRINT("Forward", 0);
					motor_forward();
					break;
				default:
					DEBUG_PRINT("Stop", 0);
					motor_stop();
				}
			}
		}

		goto telemetry;

automatic:
		j=ADC_at_Pin(QFP32_MUX_P2_2);
		v=(j*33000)/0x3fff;
		eputs("ADC[P2.2]=0x");
		PrintNumber(j, 16, 4);
		eputs(", ");
		PrintNumber(v/10000, 10, 1);

		putchar('.');
```

```
                PrintNumber(v%10000, 10, 4);
                eputs("V ");

                perimeter_1 = Volts_at_Pin(QFP32_MUX_P2_2);

                j=ADC_at_Pin(QFP32_MUX_P2_3);
                v=(j*33000)/0x3fff;
                eputs("ADC[P2.3]=0x");
                PrintNumber(j, 16, 4);
                eputs(", ");
                PrintNumber(v/10000, 10, 1);


                perimeter_2 = Volts_at_Pin(QFP32_MUX_P2_3);
                putchar('.');
                PrintNumber(v%10000, 10, 4);
                eputs("V ");

                // Not very good for high frequencies because of all the interrupts in the backgroun
                // but decent for low frequencies around 10kHz.

                send_trigger_pulse();
                echo_time = measure_echo_pulse();
                SFRPAGE = 0x20;
                c = SBUF1;
                SFRPAGE = 0x00;

                if (f > threshold_freq) {
                        motor_stop();
                        motor_backward();
                        waitms(200);
                        eputs("coin det");
                        motor_stop();
                        coin_pickup();
                        waitms(300);
                        SFRPAGE = 0x20;
                        c = SBUF1;
                        SFRPAGE = 0x00;

                        coin++;

                        eputs("Coin detected");
                } else if (perimeter_1 > base_volt_1 + 0.2 || perimeter_2 > base_volt_2 + 0.2) {
                        eputs("Perimeter detected");
                        motor_stop();
                        motor_backward();
                        waitms(400);
                        random_turn();
                        SFRPAGE = 0x20;
                        c = SBUF1;
                        SFRPAGE = 0x00;

                        motor_stop();
                }

                else if ((float) echo_time/58 < 3.0){
                        motor_stop();
                        motor_backward();
                        waitms(500);
                        sonar_turn();
                        SFRPAGE = 0x20;
                        c = SBUF1;
                        SFRPAGE = 0x00;

                        motor_stop();
                }
```

```
		if (coin == NUM_COINS) {
			motor_stop();
			is_auto_mode = 0;
			coin = 0;
			waitms(10);
			SFRPAGE = 0x20;
			c = SBUF1;
			SFRPAGE = 0x00;
			waitms(10);
			continue;
		}

		motor_forward();

telemetry:
		if (c == 0x05) // Master wants slave data
		{
			sprintf(buff, "%03ld\n", f - threshold_freq + 200);
			waitms(5); // The radio seems to need this delay...
			sendstr1(buff);
		}
	}

}
```