
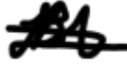






University of British Columbia
Electrical and Computer Engineering
ELEC291/ELEC292 Winter 2025
Instructor: Dr. Jesus Calvino-Fraga
Section 201

Project 2 – Coin Picking Robot

Group A13

<i>Student #</i>	<i>Student Name</i>	<i>% Points</i>	<i>Signature</i>
31668908	Yassin Abulnaga	100	
19097781	Faris Alshouani	100	
72193626	Ali Danesh	100	
44837953	Ronald Feng	100	
77438398	Drédyn Fontana	100	
26570291	Nick Unruh	100	

Date of Submission: April 8, 2025

TABLE OF CONTENTS

1. Introduction

2. Investigation

2.1. Idea Generation

2.2. Investigation Design

2.3. Data Collection

2.4. Data Synthesis

2.5. Analysis of Results

3. Design

3.1. Use of Process

3.2. Need and Constraint Identification

3.3. Problem Specification

3.4. Solution Generation

3.5. Solution Evaluation

3.6. Safety/Professionalism

3.7. Detailed Design

3.8. Solution Assessment

4. Life-Long Learning

5. Conclusions

6. References

7. Bibliography

8. Appendices

1. Introduction

Objective

Our objective for this project was to design, program, test and implement a functional coin picking robot. The robot is required to detect and collect coins whilst remaining within a designated area, using perimeter detection to ensure it does not exceed the bounds. It should be able to navigate an area without human control; however, we must also design a remote that is capable of controlling the robot's movements using a joystick. The signal intensity received by the metal detector will be monitored using an LCD located on the remote controller.

Specifications

Our project had to meet the following specifications:

- Programmed in C
- Two microcontrollers in different families
- Battery powered Robot and Remote
- MOSFET drivers for the motors
- Remote must have display, speaker, joystick or equivalent
- Radio communication with JDY-40

A detailed parts list of the robot and the remote can be found in Appendix I. Figures 1 to 4 below present the block diagrams of the hardware and software of the robot and the remote.

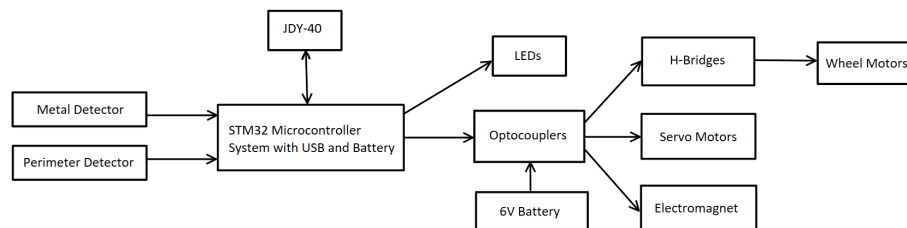


Figure 1: Block diagram of the hardware components of the robot.

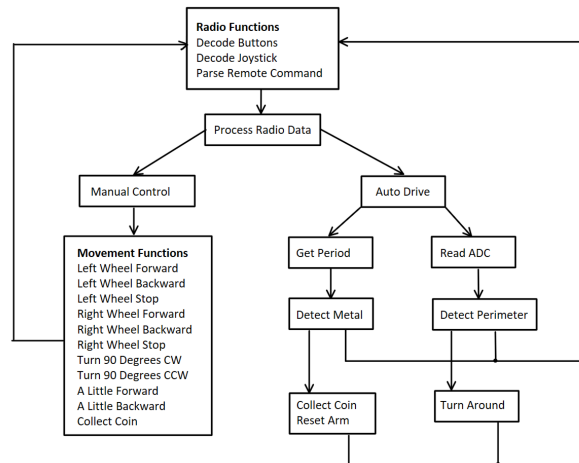


Figure 2: Block diagram of the software components of the robot.

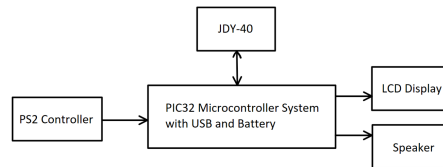


Figure 3: Block diagram of the hardware components of the remote.

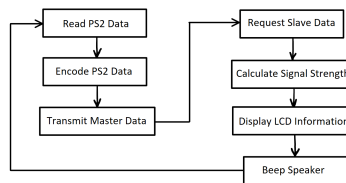


Figure 4: Block diagram of the software components of the remote.

2. Investigation

2.1 Idea Generation

Our group outlined the project's requirements and developed a plan of action accordingly. We created a Gantt chart to systematically divide tasks up amongst group members to optimize our time. After establishing the process for implementing the base project, we drafted a document

proposing potential bonus features. In the end we were left with a clear outline guiding us towards the final product.

2.2 Investigation Design

- **Research & Information Gathering:** We reviewed datasheets, previous course materials, online resources, and lab manuals to understand key components (e.g., JDY-40, STM32, PIC32, op-amps) and sensor-actuator interfacing [1], [2].
- **Prototype Circuit Development:** We developed prototype circuits for a metal detector with a Colpitts oscillator and a perimeter sensor. Additionally, we designed and tested op-amp circuits to amplify sensor signals for precise ADC measurements.
- **Experimental Measurements:** We used the Fluke 45 Digital Multimeter, TTI EX354T Power Supply, and an oscilloscope to measure voltages, verify resistor values, validate PWM signals, and confirm the accuracy of the STM32's GetPeriod function.
- **Data Collection & Analysis:** We displayed frequency and voltage information to the PuTTY terminal and compared them with experimental measures to verify accuracy.
- **Integration Testing:** We performed iterative tests with the PIC32 remote and STM32 robot to assess communication, sensor reliability, and overall system responsiveness, ensuring both automatic and manual modes met design specifications.

2.3 Data Collection

We used various tools to gather data during testing. The Fluke 45 multimeter measured voltages in the metal and perimeter circuits, while the TTI EX354T power supply provided stable voltage for robot testing. The oscilloscope verified the Colpitts oscillator frequency, PWM timing for servos, and perimeter sensor voltage near boundary wires. See Appendix II for the oscilloscope readings of the PWM. These signals aided in tuning components and detection thresholds.

Debugging LEDs on the STM32 visually confirmed metal or perimeter detection during tests. The STM32's ADC sampled perimeter detector voltages to trigger perimeter avoidance, and its GetPeriod function measured oscillator frequency for metal detection. These tools helped calibrate sensors, validate circuits, and debug software.

2.4 Data Synthesis

Real-time outputs from both the GetPeriod function and the ADC readings were printed continuously to the terminal. This allowed us to visualize how the oscillator frequency and perimeter sensor voltages changed as coins were moved near the detector or as the robot approached the boundary wire. The live serial output provided instant feedback, helping us fine-tune detection thresholds. These terminal readings confirmed that the STM32's internal peripherals were operating consistently and could be trusted for control decisions.

2.5 Analysis of Results

Our group assessed the validity of our conclusions by repeatedly performing tests of the metal and perimeter detectors, and using an oscilloscope to confirm the microcontroller's measurements. By analyzing the oscilloscope data, we could confirm any discrepancies in the microcontroller's readings and adjust our approach accordingly. These tests gave us confidence that our circuitry and our code were working effectively, allowing us to calibrate accurate thresholds for our metal and perimeter detection code.

3. Design

3.1 Use of Process

- **Problem Definition & Requirements:** We identified the need for a coin picking robot that automatically detects and collects coins within a defined perimeter while allowing manual

override via a remote. We established constraints including limited project time, component availability, and the requirement to use different microcontroller families (PIC32 for the remote and STM32 for the robot).

- **Subsystem Decomposition:** We divided the project into hardware and software components. On the hardware side, we implemented sensor circuits (metal detection using a Colpitts oscillator and perimeter detection). We discovered that the ADC voltage from the inductor was too low for reliable detection, and instead integrated an op-amp circuit for signal amplification.
- **Incremental Software Development:** We integrated sensor readings, wireless communication, and actuator control gradually to simplify debugging and future modifications.
- **Testing and Verification:** We conducted thorough testing using oscilloscopes, multimeters, and debugging LEDs to validate PWM signals, sensor outputs, and wireless communication. We verified that the op-amp amplification provided accurate perimeter detection.
- **Integration & Iterative Refinement:** We combined all subsystems and performed full-system tests in both automatic and manual modes. We iteratively refined both hardware and software until the system met all design specifications and performance criteria.

3.2 Need and Constraint Identification

Our team identified customer, user, and enterprise needs by adopting the perspective of the end user and considering key factors such as customizability, user-friendliness, and accuracy.

Constraints of achieving these needs include our limited project time, resources, and people working on the project. We needed to address these needs before adding any additional features.

The customer's need was for a robot that could reliably locate and collect coins with minimal

supervision and emphasizing efficiency. The user's need was centred around implementing an intuitive remote interface, including real-time data collection, responsive control over the robot, and simple mode switching between automatic and manual operation. The enterprise's needs were addressed by ensuring a robust, cost-effective solution to ensure affordability, maintainability, and scalability. To address these needs, we identified several constraints such as: limited development time, restricted component selection, battery-powered operation, amount of people on each task, and the requirement to use two distinct microcontrollers.

3.3 Problem Specification

From our needs analysis of the project, we specified the following design requirements:

- **Coin Detection:** The metal detector must sense all current Canadian coins when within a few centimeters, with a detection threshold set at approximately 10% above baseline frequency.
- **Coin Pickup:** The electromagnet and servo arm must reliably pick up a coin in a single attempt at least 90% of the time.
- **Boundary Enforcement:** The robot must not leave the designated perimeter; the system must react within 0.5 seconds when a perimeter sensor is triggered.
- **Remote Control:** Commands from the remote should be executed with a latency of less than 100 ms, and the feedback system (LCD and buzzer) must update frequently enough to be useful.
- **Safety and Power:** The design needed to be efficient as it operated on battery power.

Furthermore, we needed to adopt a resilient design to ensure the robot's longevity.

3.4 Solution Generation

We explored a variety design solutions to meet the functional specifications of the coin picking robot, which included successful implementations and discarded approaches:

- **User Input & Remote Interface:** A PS2 controller for the PIC32 remote was chosen over a joystick and push buttons because it created a more familiar and intuitive user input interface.
- **Wireless Communication:** We used the JDY-40 for its low power consumption, and ease of configuration in the lab environment.
- **Motor Control & Drive System:** Instead of direct microcontroller control, discrete MOSFET drivers were adopted to minimize electrical noise and to ensure robust motor control during coin collection maneuvers.
- **Coin Pickup Mechanism:** Several designs were prototyped, such as mechanical grippers and electromagnet-based systems. The final design utilizes an electromagnet combined with micro servo motors to provide fast, precise coin pickup and deposit.
- **Sensor Integration:** We used a Colpitts oscillator for metal detection as it provides reliable detection of all coin types. For perimeter detection, we designed a circuit that accurately senses the AC field from the perimeter wire.
- **Software & System Control:** The remote firmware on the PIC32 was developed to decode PS2 joystick inputs and transmit commands wirelessly, ensuring seamless integration between the two systems.

3.5 Solution Evaluation

We evaluated each solution based on its difficulty of implementation and impact on the final design to identify the most suitable options for the project.

- We had a variety of microcontrollers to choose from, such as the ATMEGA32, LPC824, and the EFM8. We chose to use the STM32 and the PIC32 microcontrollers because of their raw power and because we had previous experiences with these microcontrollers.

- We considered using a joystick and push buttons to control the robot, but we found online documentation on how to implement the PS2 controller in our design. Using a PS2 controller allowed for more input options and more functionality in our project.
- The use of two breadboards in our final design gave us a generous amount of space for circuitry, allowing for cleaner circuits and more extra features.

3.6 Safety / Professionalism

We performed safe and professional practices during the design of our project. This includes following the established laboratory rules such as using safety glasses while soldering components, keeping the laboratory food-free, and turning off the equipment and cleaning the workspace after we finished using it. We disconnected the remote and the robot from power while we were working on their circuits. The robot also has a switch to quickly turn off the power in case of an accident. Furthermore, we included buttons on our PS2 controller that could pause the robot and take it out of its autonomous mode which provided another layer of safety.

3.7 Detailed Design

Microcontroller System with USB and Battery Circuit

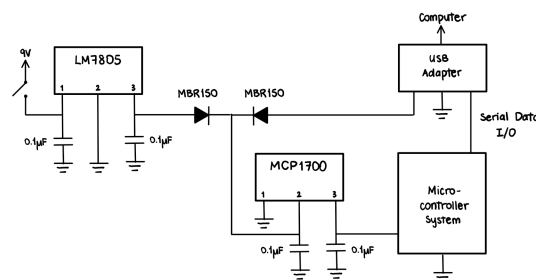


Figure 5: Microcontroller System with USB and Battery Circuit.

The STM23 and the PIC32 require 3.3V to function, so we used an MCP1700 3.3V regulator to convert the 5V from the USB adapter to 3.3V. However, we also needed to use a 9V battery to power the microcontrollers. We achieved this by regulating the 9V battery 5V using the LM7805

5V regulator. We attached this circuit to the 5V wire of the USB adapter, and included diodes to prevent unwanted current flow.

Radio Circuit

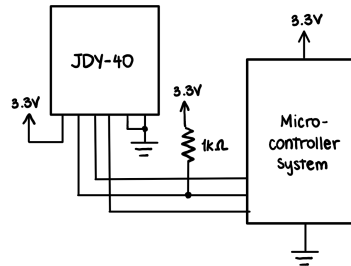


Figure 6: Radio Circuit

We referred to existing documentation to build the radio circuit for both the STM32 for the robot and the PIC32 for the remote [3].

Perimeter Detector Circuit

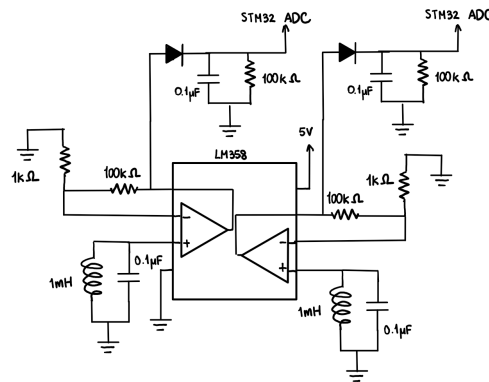


Figure 7: Perimeter Detector Circuit.

An inductor in parallel with a capacitor will create a small induced AC voltage when an AC current is next to the inductor. This voltage is amplified with a gain of 100 using an LM358 op-amp. To measure the peak voltage of the induced AC voltage, a peak detector made out of a diode, a capacitor, and a resistor is attached to the output of the op-amp. This voltage is read by

the STM32's ADC. Another perimeter detector circuit is placed perpendicular to the first one to ensure the perimeter is detected regardless of the angle of approach.

Metal Detector Circuit

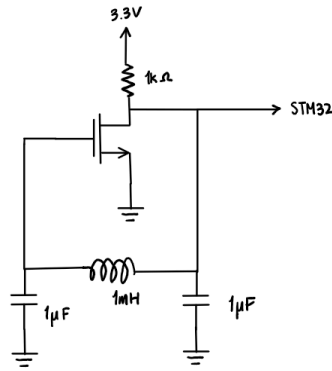


Figure 8: Metal Detector Circuit.

The metal detector is a sinusoidal oscillator made of an inductor, two capacitors, an N-MOSFET, and a resistor. When metal is placed next to the inductor, its inductance changes, and will change the frequency of the oscillator. This change in frequency is measured by the STM32, allowing it to detect metal.

Electromagnet, Servo, and Motor Circuit

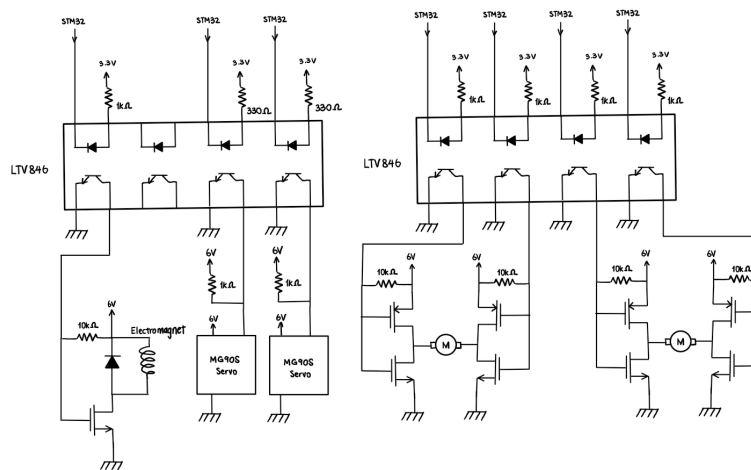


Figure 9: Electromagnet, Servo, and Motor Circuit

Two LTV846 optocouplers were used to isolate the electromagnet, the servos, and the motors from the rest of the robot circuit. The motors that control the wheels are in two H-bridge circuits each made out of two P-MOSFETs, two N-MOSFETs, and two resistors, allowing for both forward and reverse rotation in the motors [4].

Perimeter Circuit

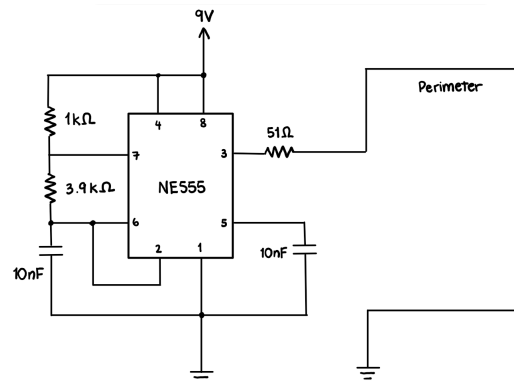


Figure 10: Perimeter Circuit

A 555 timer in astable mode produces an oscillating current through a perimeter made of wire. The resistor and capacitor values of the 555 timer were chosen so that it produces a frequency near the 16 kHz resonance frequency of the perimeter detector. A low resistor value is attached to the output of the 555 timer to maximize the AC current.

PIC32 to PS2 Controller Circuit

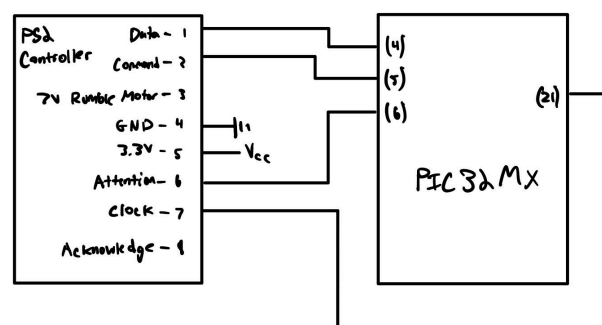


Figure 11: PIC32 to PS2 Controller Circuit

The PS2 controller has 8 pins and operates at 3.3V, which is compatible with the PIC32 microcontroller system. Four connections are used between the controller and the microcontroller: data, command, attention, and clock. The 7V rumble motor and acknowledge pins are not required. The controller sends data to the microcontroller through pin 4. The microcontroller sends commands to the controller through pin 2. Pin 6 is used for the attention signal, which acts as a chip select; when the microcontroller pulls it low, communication begins. Pin 7 is used for the clock signal to synchronize data transfer between the controller and the microcontroller.

PIC32 to LCD Circuit

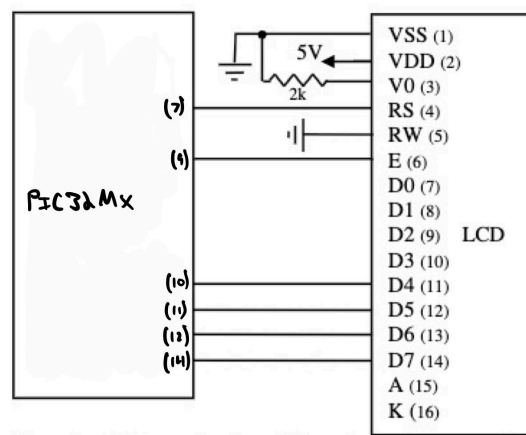


Figure 12: PIC32 to LCD Circuit

We referred to existing documentation to build the LCD circuit for the PIC32 in the remote [5].

Robot Code

The full robot code can be found in the Appendix III. This section explains how the robot decodes the radio data and how the code performs the automatic and manual modes.

- **Decoding Radio Data:** The remote sends a command in the form of a 6-character string. For each character of the string, it is decoded by either using a switch statement or if-else

statements. Each character is mapped to a corresponding predefined constant that either refers to a button on the PS2 controller or a joystick position. Afterwards, these constants are stored in a RemoteCommand struct pointer that can be accessed by other functions.

- **Automatic Mode:** The robot is in automatic mode when the variable `*mode_flag` is set to 1. In this mode, the robot moves forward continuously while periodically scanning for metal and the perimeter. To detect metal, the robot measures the oscillating voltage from the metal detector and calculates the period and frequency. Upon identifying metal, it uses an electromagnet and servo motors to execute a predefined procedure for picking up a coin. Simultaneously, the robot checks for the perimeter using the ADC value from the perimeter detector circuit. If this value exceeds a threshold, the robot responds by performing a programmed routine to back up and turns away to avoid crossing the perimeter.
- **Manual Mode:** The robot is in its manual mode when the variable `*mode_flag` is set to 0. In manual mode, the robot continuously reads and decodes the commands sent through the radio and executes actions based on those commands. Additionally, it sends the frequency detected by the metal detector through the radio to the remote, where it can use that value to calculate the signal strength of the metal detector, display the value on the LCD, and beep the speaker.

Remote Code

The full remote code can be found in the Appendix IV. This section explains how the remote encodes the PS2 data, calculates the signal strength of the metal detector, and causes the speaker to beep.

- **Encoding Data:** The remote takes an array of raw data from the PS2 controller, and encodes them into a character array to be sent over the radio. To encode the buttons, the raw data is compared with several if-else statements to assign the correct character for the button that has

been pressed. To encode the joysticks, the raw data goes through several if-else statements to see if the raw data falls into a range of values. A character is assigned for each button and range, such as 'U' for the UP button on the D'PAD or '1' to '9' for the joysticks. Finally, a terminating character '\n' is attached to the very end.

- **Calculating Signal Strength:** With the frequency sent by the robot, the remote first subtracts the frequency by a constant of 10,000. Then it clamps this value to a minimum of 0 and a maximum of 100. Afterwards, this value is multiplied by 100 and then divided by 30, adjusting the value to a normalized range for further purposes.
- **Speaker Beeping:** The beeping of the speaker relies on Timer 1's ISR in the PIC32. After calculating the signal strength, the code calculates a beepFreq value based on the signal strength, and clamps beepFreq to a maximum and minimum value. Next, the timer uses a state machine controlled by Second_Flag to create the beeping pattern. When Second_Flag = 0, the speaker is off for a period of time which depends on beepFreq. When Second_Flag = 1, the speaker constantly toggles, producing a square wave for a fixed duration of time.

3.8 Solution Assessment

Many parts of our final design could be assessed by performing tests for those components and observing if they performed as expected. For example, we could assess the functionality of the PS2 controller by displaying the output of the controller to the LCD in the remote. For the radio, we displayed the input and output messages on PuTTY for both the remote and the robot to confirm that the information was being sent reliably. We could confirm the motors and the electromagnet were working by observing if they were in their expected states during execution. As shown above in Investigation, we tested our metal and perimeter detectors by comparing the measurements displayed on PuTTY with the oscilloscope. We put metal underneath the metal

detector and a wire with an AC current next to the perimeter detector and confirmed that the microcontroller's measurements still align with the oscilloscope's readings. Figure 13 below shows some data collected during the testing of the perimeter detector.

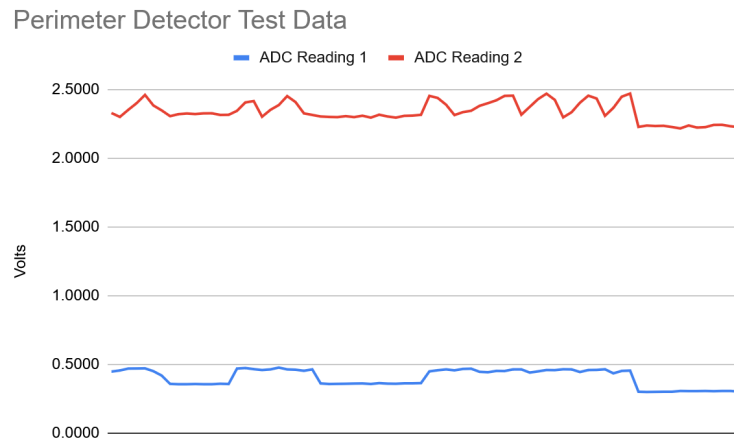


Figure 13: Plot of the voltage measured by the STM32's ADC, during times where the perimeter is near and not near the perimeter detector

Some of the strengths of our design include the usage of the PS2 controller, allowing for many possible commands and actions for the robot, as well as the use of two breadboards for lots of space for circuitry. However, improvements to our design could be made, such as the reliability of the metal detector circuit and code.

4. Life-Long Learning

This project required us to apply numerous different concepts and skills we learned from previous courses. Our understanding of MOSFET operation and H-bridge circuits from ELEC 201 helped us design the motor driver circuit to implement bidirectional control of the robot's wheels. Concepts from ELEC 211, such as inductance and induced voltage, were directly applicable in building the perimeter detection system using sensor coils. In tuning the Colpitts oscillator for metal detection, we relied on our knowledge of resonant frequency and LC tank

behavior from ELEC 202. These principles guided our component selection and analysis of the oscillator's frequency response to nearby metal objects. Additionally, our programming skills developed in APSC 160 and CPSC 259 were crucial in writing embedded C code for both the STM32 and PIC32 microcontrollers. We used timers, ADCs, PWM control, and UART communication, all requiring careful attention to efficient program structure. By integrating theoretical knowledge with hands-on problem solving, we deepened our understanding of both analog and digital systems, reinforcing the importance of continuous learning across disciplines. Furthermore, we were able to solidify our understanding of previous courses through hands-on experience.

5. Conclusions

Our project is a coin-picking robot that can be controlled manually with a remote and can automatically collect coins by itself. The remote uses a PS2 controller and sends commands through a JDY-40 radio. The robot moves using two motors and wheels and can pick up coins with two servo motors and an electromagnet. The robot has a metal detector and a perimeter detector to automatically detect coins and stay inside a perimeter. The strength of the signal from the metal detector in the robot is displayed on the LCD on the remote and causes a speaker on the remote to beep more frequently. Our design also has extra functionality, such as having complete control of the electromagnet arms of the robot, being able to pause and resume the robot's automatic mode, and displaying the coin count on the LCD. We encountered many problems such as delays debugging the radio communication and difficulties with detecting metal. However, we were able to demonstrate a completed project in the end. We estimate that we spent 60 hours on the project overall.

6. References

- [1] “RM0451 Reference manual”, STMicroelectronics, Accessed Mar. 14, 2025
https://www.st.com/resource/en/reference_manual/dm00443854-ultra-low-power-stm32l0x0-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf
- [2] “PIC32MX Family Reference Manual”, Microchip Technology Inc., Accessed Mar. 14, 2025
<http://ww1.microchip.com/downloads/en/DeviceDoc/61132B.pdf>
- [3] J. Calviño-Fraga, “Project 2: Coin Picking Robot”, University of British Columbia, Electrical and Computer Engineering, ELEC291/ELEC292, Mar. 14, 2025
- [4] “H-Bridges – The Basics | Modular Circuits.”
<https://www.modularcircuits.com/blog/articles/h-bridge-secrets/h-bridges-the-basics/>
- [5] J. Calviño-Fraga, “PIC32.zip”, University of British Columbia, Electrical and Computer Engineering, ELEC291/ELEC292, Accessed Mar. 14, 2025

7. Bibliography

- B. Emmett, “Simple wireless serial communication - notes to self”, Jan. 05, 2025.
<https://benjemmett.com/archives/790>
- LITE-ON Technology Corp. Optoelectronics, “Photocoupler Product Data Sheet”, Jan. 26, 2016.
https://mm.digikey.com/Volume0/opasdata/d220001/medias/docus/967/LTV-816_826_846.pdf
- Texas Instruments, “Industry-Standard Dual Operational Amplifiers”, Oct. 2024.
https://www.ti.com/lit/ds/symlink/lm358.pdf?ts=1743882577746&ref_url=https%253A%252F%252Fno.mouser.com%252F

8. Appendices

APPENDIX I: PARTS LIST

ROBOT HARDWARE:

- **Robot:**
 - 2x Solarbotics GM4, 2x Servo Wheel, Tamiya 70144, 4x AA Battery Holder, 1x 9V cable, 2x MG90S Servo, Coin picker assembly kit, Electromagnet, Chassis Holder, Six Pin Switch
- **Microcontroller Circuit:**
 - STM32L051 Microcontroller, BO230XS USB adapter, 2x MBR150 Diodes, LM7805, MCP1700, 6x 0.1uF Capacitor, 2x Push Buttons, 4x 330 Ohm Resistor, 1x Green LED, 2x Red LED
- **Radio Circuit:**
 - JDY40 Radio, 1k Ohm Resistor
- **Metal Detector Circuit:**
 - 1mH Inductor, 2x 1uF Capacitor, 1k Ohm Resistor, N-MOSFET
- **Perimeter Detector Circuit:**
 - LM358 Op-Amp, 2x 1mH Inductor, 4x 0.1uF Capacitor, 2x MBR150, 2x 1k Ohm Resistor, 4x 100k Ohm Resistor
- **Electromagnet, Servo, and Motors Circuit:**
 - 2x LTV846, 2x 330 Ohm Resistor, 7x 1k Ohm Resistor, 5x 10k Ohm Resistor, 4x P-MOSFET, 5x N-MOSFET, 1N4148 Diode

ROBOT SOFTWARE:

- **Remote Decoding Functions**

- `int decodeButton1(char ch);`
- `int decodeButton2(char ch);`
- `int decodeJoystick(char letter, int isYAxis);`
- `int parseRemoteCommand(char *str, RemoteCommand *cmd);`
- `int mapValue(int x, int in_min, int in_max, int out_min, int out_max);`

- **Robot Base Functions**

- `void wait_lms(void);`
- `void waitms(int len);`
- `void TIM2_Handler(void);`
- `void Hardware_Init(void);`
- `long int GetPeriod(int n);`
- `void PrintNumber(long int val, int Base, int digits);`

- **Radio Functions**

- `void SendATCommand(char *s);`
- `void ReceptionOff(void);`
- `void processRadioData(char *buff, RemoteCommand *currentCmd);`
- `int automode_toggle_check (char button);`

- **Movement Functions**

- `void leftWheelForward(void);`
- `void leftWheelBackward(void);`
- `void leftWheelStop(void);`
- `void rightWheelForward(void);`
- `void rightWheelBackward(void);`
- `void rightWheelStop(void);`
- `void turnAround(void);`
- `void turn90degreesCW(void);`

- `void turn90DegreesCCW(void);`
- `void aLittleForward(void);`
- `void aLittleBackward(void);`

- **Mode Functions**

- `void autodriven(RemoteCommand *currentCommand);`
- `void gameMode(void);`
- `void ManualControl(RemoteCommand *command);`

- **Electromagnetic Functions**

- `int detectMetal(void);`
- `int detectPerimeter(void);`
- `void collectCoin(void);`
- `void resetArm(void);`

REMOTE HARDWARE:

- **Microcontroller Circuit:**

- PIC32MX130 Microcontroller, BO230XS USB adapter, 2x MBR150 Diodes, LM7805, MCP1700, 6x 0.1uF Capacitor, 100uF Capacitor, Push Button, 330 Ohm Resistor, 1x Green LED

- **Speaker Circuit:**

- N-MOSFET, 1x 1N4148 Diode, CEM-1302 Speaker

- **Radio Circuit:**

- JDY40 Radio, 1k Ohm Resistor

- **LCD Circuit:**

- LCM-S01602DTR/M LCD Display, 2k Ohm Resistor

REMOTE SOFTWARE:

- **Remote Base Functions**

- `void Timer4us(unsigned char t);`
- `void waitms(unsigned int ms);`
- `void waitus(unsigned int us);`
- `void UART2Configure(int baud_rate);`
- `int _mon_getc(int canblock);`

- **LCD Functions**

- `void LCD_pulse(void);`
- `void LCD_byte(unsigned char x);`
- `void WriteData(unsigned char x);`
- `void WriteCommand(unsigned char x);`
- `void LCD_4BIT(void);`
- `void LCDprint(char * string, unsigned char line, unsigned char clear);`

- **Radio Functions**

- `void ClearFIFO (void);`
- `void SendATCommand(char *s);`
- `void ReceptionOff(void);`
- `void requestSlaveData(void);`
- `int calcSignalStrength(int frequency);`

- **PS2 Controller Functions**

- `void PS2_Init(void);`
- `unsigned char PS2_TransferByte(unsigned char outByte);`
- `void PS2_ReadData(unsigned char *d);`
- `void Encode_Data(unsigned char * data, char * dest)`

APPENDIX II: PWM TESTING WITH THE OSCILLOSCOPE

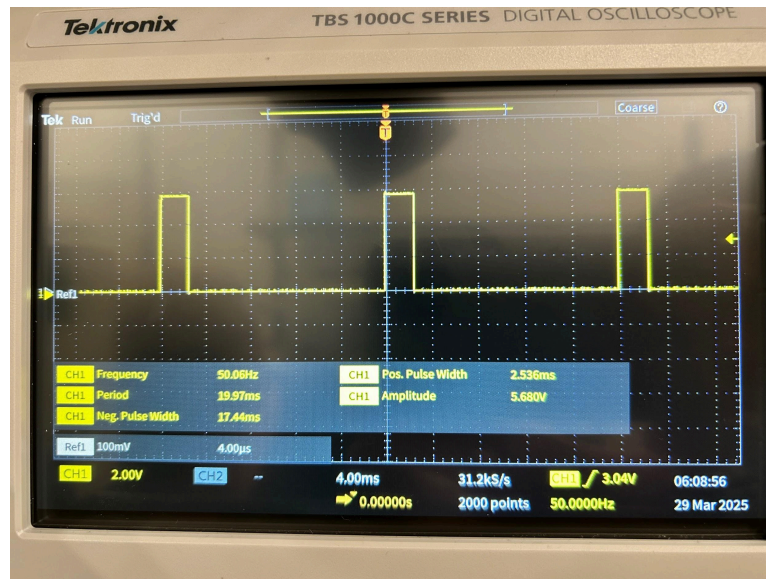


Figure 14: Oscilloscope display of PWM output of 2.55ms per 20ms

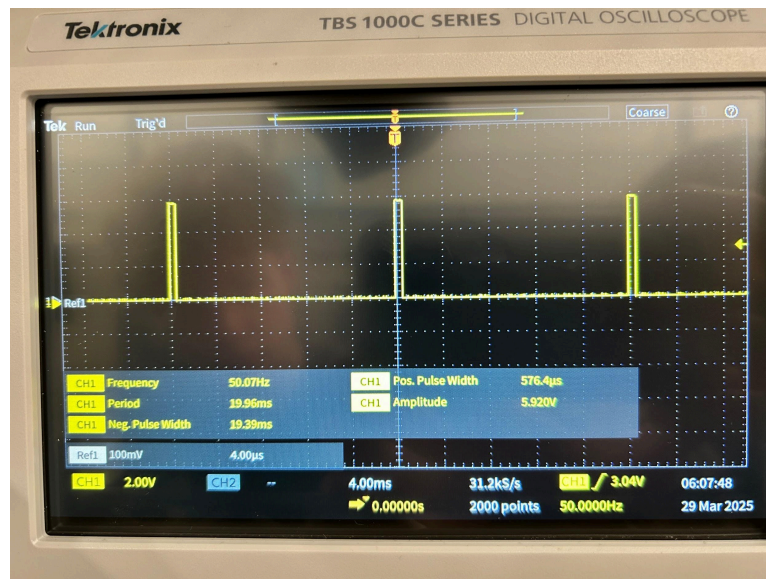


Figure 15: Oscilloscope display of PWM output of 0.6ms per 20ms

APPENDIX III: FULL ROBOT CODE

```
#include "../Common/Include/stm321051xx.h"  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include "../Common/Include/serial.h"
```



```

#include "adc.h"
#include "UART2.h"

#define F_CPU 32000000L
#define DEF_F 100000L // 10us tick

// Some 'defines' to turn pins on/off easily (pins must be configured as outputs)
#define PB3_0 (GPIOB->ODR &= ~BIT3) // Left Wheel Black
#define PB3_1 (GPIOB->ODR |= BIT3)
// Left Wheel Red
#define PB4_0 (GPIOB->ODR &= ~BIT4) // Off
#define PB4_1 (GPIOB->ODR |= BIT4) // On
// Right Wheel Red
#define PB5_0 (GPIOB->ODR &= ~BIT5) // Off
#define PB5_1 (GPIOB->ODR |= BIT5) // On
// Right Wheel Black
#define PB6_0 (GPIOB->ODR &= ~BIT6) // Off
#define PB6_1 (GPIOB->ODR |= BIT6) // On
// Electromagnet
#define PB7_0 (GPIOB->ODR &= ~BIT7) // Off
#define PB7_1 (GPIOB->ODR |= BIT7) // On

#define PA13_0 (GPIOA->ODR &= ~BIT13) // Radio
#define PA13_1 (GPIOA->ODR |= BIT13)

#define PA6_0 (GPIOA->ODR &= ~BIT6) // LED Debugging
#define PA6_1 (GPIOA->ODR |= BIT6)
#define PA7_0 (GPIOA->ODR &= ~BIT7)
#define PA7_1 (GPIOA->ODR |= BIT7)

// // A define to easily read PA14 (PA14 must be configured as input first)
// #define PA14 (GPIOA->IDR & BIT14)

// A define to easily read PA1 (PA1 must be configured as input first)
#define PA1 (GPIOA->IDR & BIT1)

// A define to easily read PA8 (PA8 must be configured as input first)
#define PA8 (GPIOA->IDR & BIT8)

// LQFP32 pinout
// -----
//          VDD -|1          32|- VSS
//          PC14 -|2         31|- BOOT0
//          PC15 -|3         30|- PB7 (OUT 5) Electromagnet
//          NRST -|4         29|- PB6 (OUT 4) Right Wheel Black (Back)
//          VDDA -|5         28|- PB5 (OUT 3) Right Wheel Red (Forward)
//          PA0 -|6          27|- PB4 (OUT 2) Left Wheel Red (Back)
// (button) PA1 -|7          26|- PB3 (OUT 1) Left Wheel Black (Forward)
//          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 (pwm2)

```



```

#define MAX_SPEED      100    // Maximum speed value.
#define MAX_TURN       50     // Maximum turning value.
#define SPEED_THRESHOLD 10    // Minimum speed to command a movement.

// ----- Define servo movement limits and step values -----
#define MIN_ISR_PWM1 60      // Vertical servo upper limit
#define MAX_ISR_PWM1 255     // Vertical servo lower limit
#define MIN_ISR_PWM2 60      // Horizontal servo left limit
#define MAX_ISR_PWM2 255     // Horizontal servo right limit
// Use larger steps and shorter delays for faster movement
#define VERTICAL_STEP 20
#define HORIZONTAL_STEP 20
#define VERTICAL_DELAY_MS 5
#define HORIZONTAL_DELAY_MS 5

/*****
*****
GLOBAL VARIABLE DECLARATIONS
*****
*****/

volatile int PWM_Counter = 0;
volatile unsigned char ISR_pwm1 = 60, ISR_pwm2 = 255;

// Could measure references at startup or hard-code values here
volatile long int reference_frequency;
volatile long int frequency;
volatile long int reference_count;
volatile long int count;
volatile int reference_voltage[2];
volatile int voltage[2];

//RemoteCommand * Controller_Data; //used to store data from PS2 controller
int * mode_flag; //used to check if robot should be in automatic pickup mode

// Global coin counter added for the game:
volatile int coin_count = 0;

// Structure to hold decoded remote command values
typedef struct {
    int b1;    // buttons1
    int b2;    // buttons2
    int rx;    // Right Joystick X
    int ry;    // Right Joystick Y
    int lx;    // Left Joystick X
    int ly;    // Left Joystick Y
} RemoteCommand;

/*****
*****
FUNCTION PROTOTYPES

```

```

*****
*****/

//Remote Decoding Functions
int decodeButton1(char ch);
int decodeButton2(char ch);
int decodeJoystick(char letter, int isYAxis);
int parseRemoteCommand(char *str, RemoteCommand *cmd);
int mapValue(int x, int in_min, int in_max, int out_min, int out_max);

//Robot Base Functions
void wait_lms(void);
void waitms(int len);
void TIM2_Handler(void);
void Hardware_Init(void);
long int GetPeriod(int n);
void PrintNumber(long int val, int Base, int digits);

//Radio Functions
void SendATCommand(char *s);
void ReceptionOff(void);
void processRadioData(char *buff, RemoteCommand *currentCmd);
//int parseRemoteCommand(char *str, RemoteCommand *cmd);
int automode_toggle_check(char button);

//Movement Functions
void leftWheelForward(void);
void leftWheelBackward(void);
void leftWheelStop(void);
void rightWheelForward(void);
void rightWheelBackward(void);
void rightWheelStop(void);
void turnAround(void);
void turn90degreesCW(void);
void turn90DegreesCCW(void);
void aLittleForward(void);
void aLittleBackward(void);

//Mode Functions
void autodrive(RemoteCommand *currentCommand);
void gameMode(void);
void ManualControl(RemoteCommand *command);

//Electromagnetic Functions
int detectMetal(void);
int detectPerimeter(void);
void collectCoin(void);
void resetArm(void);

/*****
*****
REMOTE COMMAND DECODING FUNCTIONS

```

Based on the Controller Data Encoding Table

:contentReference[oaicite:2]{index=2}<:contentReference[oaicite:3]{index=3}:

Data format: [b1Char][b2Char][rxChar][ryChar][lxChar][lyChar]

For example: "NN1515" means no buttons pressed, Right Joystick left & centre, Left Joystick left & centre.

*****/

```
int decodeButton1(char ch) {
    switch(ch) {
        case 'N': return BUTTON_NONE;    // No button pressed
        case 's': return SELECT;          // SELECT
        case 'S': return START;           // START
        case 'Z': return START_SELECT;    // START_SELECT
        case 'U': return DPAD_UP;         // DPAD UP
        case 'D': return DPAD_DOWN;       // DPAD DOWN
        case 'L': return DPAD_LEFT;       // DPAD LEFT
        case 'R': return DPAD_RIGHT;      // DPAD RIGHT
        case 'l': return L3;              // L3 (joystick press)
        case 'r': return R3;              // R3 (joystick press)
        case 'E': return BUTTON_ERROR;    // Error
        default: return 0;                // Unknown value
    }
}

int decodeButton2(char ch) {
    switch(ch) {
        case 'N': return BUTTON_NONE;    // No button pressed
        case 'X': return X_BUTTON;       // X
        case 'S': return SQUARE;         // Square
        case 'O': return O_BUTTON;       // O
        case 'T': return TRIANGLE;       // Triangle
        case 'r': return R1;              // R1
        case 'R': return R2;              // R2
        case 'l': return L1;              // L1
        case 'L': return L2;              // L2
        case 'E': return BUTTON_ERROR;    // Error
        default: return 0;
    }
}

int decodeJoystick(char letter, int isYAxis) {
    if(letter < '1' || letter > '9') return -1; // Invalid letter
    int num = letter - '0';
    // For non-inverted joystick, compute midpoint using LEVEL definitions.
    static int mappingX[9] = {
        (LEVEL0 + LEVEL1) / 2, // '1': (0+28)/2 = 14
        (LEVEL1 + LEVEL2) / 2, // '2': (28+55)/2 = 41
        (LEVEL2 + LEVEL3) / 2, // '3': (55+82)/2 = 68
        (LEVEL3 + LEVEL4) / 2, // '4': (82+110)/2 = 96
        (LEVEL4 + LEVEL5) / 2, // '5': (110+146)/2 = 128
    }
}
```

```

        (LEVEL5 + LEVEL6) / 2, // '6': (146+174)/2 = 160
        (LEVEL6 + LEVEL7) / 2, // '7': (174+201)/2 = 187
        (LEVEL7 + LEVEL8) / 2, // '8': (201+228)/2 = 214
        (LEVEL8 + LEVEL9) / 2 // '9': (228+255)/2 = 241 (integer division)
    };
    // For the y-axis, use an inverted mapping (reverse order).
    static int mappingY[9] = {
        (LEVEL8 + LEVEL9) / 2, // '1' â†’ highest: 241
        (LEVEL7 + LEVEL8) / 2, // '2': 214
        (LEVEL6 + LEVEL7) / 2, // '3': 187
        (LEVEL5 + LEVEL6) / 2, // '4': 160
        (LEVEL4 + LEVEL5) / 2, // '5': 128
        (LEVEL3 + LEVEL4) / 2, // '6': 96
        (LEVEL2 + LEVEL3) / 2, // '7': 68
        (LEVEL1 + LEVEL2) / 2, // '8': 41
        (LEVEL0 + LEVEL1) / 2 // '9' â†’ lowest: 14
    };
    if(!isYAxis) {
        return mappingX[num - 1];
    } else {
        return mappingY[num - 1];
    }
}

int parseRemoteCommand(char *str, RemoteCommand *cmd) {
    if(strlen(str) != 6) return 0; // Expect exactly 6 characters
    cmd->b1 = decodeButton1(str[0]);
    cmd->b2 = decodeButton2(str[1]);
    cmd->rx = decodeJoystick(str[2], 0); // Right Joystick X
    cmd->ry = decodeJoystick(str[3], 1); // Right Joystick Y
    cmd->lx = decodeJoystick(str[4], 0); // Left Joystick X
    cmd->ly = decodeJoystick(str[5], 1); // Left Joystick Y
    return 1;
}

// Mapping function: maps an input value x from [in_min, in_max] to [out_min, out_max].
int mapValue(int x, int in_min, int in_max, int out_min, int out_max) {
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

/*****
*****
ROBOT BASE FUNCTIONS
*****
*****/

void wait_1ms(void)
{
    // For SysTick info check the STM3210xxx Cortex-M0 programming manual.
    SysTick->LOAD = (F_CPU/1000L) - 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(int len)
{
    while(len--) wait_1ms();
}

// Interrupt service routines are the same as normal
// subroutines (or C funtions) in Cortex-M microcontrollers.
// The following should happen at a rate of 1kHz.
// The following function is associated with the TIM2 interrupt
// via the interrupt vector table defined in startup.c
void TIM2_Handler(void)
{
    TIM2->SR &= ~BIT0; // clear update interrupt flag
    PWM_Counter++;

    if(ISR_pwm1>PWM_Counter)
    {
        GPIOA->ODR |= BIT11;
    }
    else
    {
        GPIOA->ODR &= ~BIT11;
    }

    if(ISR_pwm2>PWM_Counter)
    {
        GPIOA->ODR |= BIT12;
    }
    else
    {
        GPIOA->ODR &= ~BIT12;
    }

    if (PWM_Counter > 2000) // The period is 20ms
    {
        PWM_Counter=0;
        GPIOA->ODR |= (BIT11|BIT12);
    }
}

void Hardware_Init(void)
{

```

```

    GPIOA->OSPEEDR=0xffffffff; // All pins of port A configured for very high speed! Page 201
of RM0451
    //GPIOA->OSPEEDR |= 0xfc000000; // Pins PA15, PA14, PA13 configured for very high speed!
Page 201 of RM0451

    RCC->IOPENR  |= (BIT1|BIT0); // peripheral clock enable for ports A and B

    // Configure the pin used for analog input: PB0 and PB1 (pins 14 and 15)
    GPIOB->MODER |= (BIT0|BIT1); // Select analog mode for PB0 (pin 14 of LQFP32 package)
    GPIOB->MODER |= (BIT2|BIT3); // Select analog mode for PB1 (pin 15 of LQFP32 package)

    initADC();

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

    // // Configure the pin connected to the pushbutton as input

    // GPIOA->MODER &= ~(BIT28 | BIT29); // Make pin PA14 input
    // // Activate pull up for pin PA8:
    // GPIOA->PUPDR |= BIT28;
    // GPIOA->PUPDR &= ~(BIT29);

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

    // Configure some pins as outputs:
    // Make pins PB3 to PB7 outputs (page 200 of RM0451, two bits used to configure: bit0=1,
bit1=0)
    GPIOB->MODER = (GPIOB->MODER & ~(BIT6|BIT7)) | BIT6; // PB3
    GPIOB->OTYPER &= ~BIT3; // Push-pull
    GPIOB->MODER = (GPIOB->MODER & ~(BIT8|BIT9)) | BIT8; // PB4
    GPIOB->OTYPER &= ~BIT4; // Push-pull
    GPIOB->MODER = (GPIOB->MODER & ~(BIT10|BIT11)) | BIT10; // PB5
    GPIOB->OTYPER &= ~BIT5; // Push-pull
    GPIOB->MODER = (GPIOB->MODER & ~(BIT12|BIT13)) | BIT12; // PB6
    GPIOB->OTYPER &= ~BIT6; // Push-pull
    GPIOB->MODER = (GPIOB->MODER & ~(BIT14|BIT15)) | BIT14; // PB7
    GPIOB->OTYPER &= ~BIT7; // Push-pull

    // Set up servo PWM output pins
    GPIOA->MODER = (GPIOA->MODER & ~(BIT22|BIT23)) | BIT22; // Make pin PA11 output (page 200
of RM0451, two bits used to configure: bit0=1, bit1=0)
    GPIOA->OTYPER |= BIT11; // Open-drain
    GPIOA->MODER = (GPIOA->MODER & ~(BIT24|BIT25)) | BIT24; // Make pin PA12 output (page 200
of RM0451, two bits used to configure: bit0=1, bit1=0)
    GPIOA->OTYPER |= BIT12; // Open-drain

```



```

    // Set up JDY40 output pins
    GPIOA->MODER = (GPIOA->MODER & ~(BIT26|BIT27)) | 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.

    // Set up LED Debugging output pins
    GPIOA->MODER = (GPIOA->MODER & ~(BIT12|BIT13)) | BIT12; // Make pin PA6 output
    GPIOA->OTYPER &= ~BIT6; // Push-pull
    GPIOA->MODER = (GPIOA->MODER & ~(BIT14|BIT15)) | BIT14; // Make pin PA7 output
    GPIOA->OTYPER &= ~BIT7; // Push-pull

    // Set up timers
    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();
}

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

    SysTick->LOAD = 0xffffffff; // 24-bit counter set to check for signal present
    SysTick->VAL = 0xffffffff; // load the SysTick counter
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk; // Enable SysTick
IRQ and SysTick Timer */
    while (PA8!=0) // Wait for square wave to be 0
    {
        //eputs("PA8!=0 While Loop in GetPeriod\r\n");
        if(SysTick->CTRL & BIT16) return 0;
    }
    SysTick->CTRL = 0x00; // Disable Systick counter

    SysTick->LOAD = 0xffffffff; // 24-bit counter set to check for signal present
    SysTick->VAL = 0xffffffff; // load the SysTick counter
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk; // Enable SysTick
IRQ and SysTick Timer */
    while (PA8==0) // Wait for square wave to be 1
    {
        //eputs("PA8==0 While Loop in GetPeriod\r\n");
        if(SysTick->CTRL & BIT16) return 0;
    }
    SysTick->CTRL = 0x00; // Disable Systick counter

```

```

SysTick->LOAD = 0xffffffff; // 24-bit counter reset
SysTick->VAL = 0xffffffff; // 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 (PA8!=0) // Wait for square wave to be 0
    {
        //eputs("for loop PA8!=0 While Loop in GetPeriod\r\n");
        if(SysTick->CTRL & BIT16) return 0;
    }
    while (PA8==0) // Wait for square wave to be 1
    {
        //eputs("for loop PA8==0 While Loop in GetPeriod\r\n");
        if(SysTick->CTRL & BIT16) return 0;
    }
}
SysTick->CTRL = 0x00; // Disable SysTick counter

return 0xffffffff-SysTick->VAL;
}

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

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

/*****
*****

RADIO FUNCTIONS
*****
*****/

void SendATCommand (char * s)
{
    char buff[40];
    // printf("Command: %s", s);
    eputs("Command: ");

```

```

    eputs(s);
    eputs("\r\n");
    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);
    eputs("Response: ");
    eputs(buff);
    eputs("\r\n");
}

void ReceptionOff (void)
{
    GPIOA->ODR &= ~(BIT13); // 'set' pin to 0 is 'AT' mode.
    waitms(10);
    eputs2("AT+DVID0000\r\n"); // Some unused id, so that we get nothing in RXD1.
    waitms(10);
    GPIOA->ODR |= BIT13; // 'set' pin to 1 is normal operation mode.
    while (ReceivedBytes2()>0) egetc2(); // Clear FIFO
}

/*****
 * Process ALL incoming radio data from UART2
 *   - If we see a '!' we parse a 6-char command (e.g., "Ns1515")
 *     and call ManualControl().
 *   - If we see an '@' we respond with frequency data.
 *   - Otherwise, we ignore or handle as needed.
 *****/
void processRadioData(char *buff, RemoteCommand *currentCmd)
{
    // Read until the FIFO is empty.
    while (ReceivedBytes2() > 0)
    {
        char c = egetc2();

        if (c == '!')
        {
            // We expect the next 6 characters to be the joystick/buttons, e.g. "Ns1515".
            // Read 6 chars (plus 1 for '\0').
            if (egets2(buff, 7) > 0)
            {
                eputs("Received command: ");
                eputs(buff);
                eputs("\r\n");

                // Try to parse into our RemoteCommand struct, then do manual control
                if (parseRemoteCommand(buff, currentCmd))
                {

```

```

ManualControl(currentCmd); // <--- your existing manual-control logic

if (currentCmd->b1 == START)
{
    *mode_flag = 1;
}
else if (currentCmd->b1 == SELECT)
{
    *mode_flag = 0;
}
else if (currentCmd->b1 == START_SELECT)
{
    *mode_flag = 3;
}
}
else
{
    eputs("Error: unable to parse command!\r\n");
}
}

else if (c == '@') {
    // Master is requesting frequency data and coin count.
    // Format as "F:xxxxx,C:yyy" where frequency is 5 digits and coin_count is 3
digits.

    long int freq = frequency *10000; // recalc if needed
    freq = freq / reference_frequency;

    sprintf(buff, "%05ld%02d", freq, coin_count);
    // The radio often needs a small delay
    waitms(5);
    eputs2(buff); // Send back the formatted data string
}
else {
    //eputs("NO DATA\r\n");
}
}

}

/* parseRemoteCommand() parses a string of the form:
    "LX=%d,LY=%d,RX=%d,RY=%d,B6=%d,B7=%d"
    Returns 1 on success, 0 on failure.
*/
//int parseRemoteCommand(char *str, RemoteCommand *cmd)
//{
//    int parsed = sscanf(str, "LX=%d,LY=%d,RX=%d,RY=%d,B6=%d,B7=%d",
//    //          &cmd->lx, &cmd->ly, &cmd->rx, &cmd->ry,
//    //          &cmd->b6, &cmd->b7);
//    // return (parsed == 6);
//}

```

```

//Checks if SELECT or START are Pressed
//If true, starts or stops automode based on value of autoflag and resets robot to starting
position
//Returns 1 if moving to Manual mode, otherwise returns 0
int automode_toggle_check (char button)
{
    //char * data;
    eputs("automode_toggle_check\r\n");
    if (button == 's') // If SELECT is pressed, turn off automode
    {
        eputs("Turn off Automode\r\n");
        *mode_flag = 0;

        leftWheelStop();
        rightWheelStop();
        resetArm();

        return 1;
    }
    else if (button == 'S') // If START is pressed, turn on automode
    {
        eputs("Turn on Automode\r\n");
        *mode_flag = 1;
        return 0;
    }
    else
    {
        eputs("Automode no effect\r\n");
        return 0;
    }
}

/*****
*****
MOVEMENT FUNCTIONS
*****
*****/

// Functions for controlling the wheels
// Change the wiring to match these functions
void leftWheelForward(void)
{
    PB3_1;
    PB4_0;
}

void leftWheelStop(void)
{
    PB3_0;
    PB4_0;
}

```

```

void leftWheelBackward(void)
{
    PB3_0;
    PB4_1;
}

void rightWheelForward(void)
{
    PB5_1;
    PB6_0;
}

void rightWheelStop(void)
{
    PB5_0;
    PB6_0;
}

void rightWheelBackward(void)
{
    PB5_0;
    PB6_1;
}

void turnAround(void)
{
    eputs("turnAround\r\n");
    // Change this number to make sure the robot turns 90 degrees in this amount of time
    int turning_time_ms = 200;

    // Go backwards for some time
    leftWheelBackward();
    rightWheelBackward();

    // This may need adjustment
    waitms(500);

    // The three lines of code here may not be necessary
    // leftWheelStop();
    // rightWheelStop();
    // waitms(100);

    // Turn clockwise 90 degrees
    leftWheelForward();
    rightWheelBackward();
    waitms(turning_time_ms);

    // PWM_Counter is constantly changing from 1 to 2000 in Timer2; use it for pseudorandom
    numbers
    // Turn clockwise from 0 to 180 degrees more
    waitms((turning_time_ms * PWM_Counter)/1000);
}

```

```

        // The three lines of code here may not be necessary
        // leftWheelStop();
        // rightWheelStop();
        // waitms(100);
    }

    void turn90DegreesCW(void) {
        eputs("turn90DegreesCW\r\n");

        // Turn clockwise 90 degrees
        leftWheelBackward();
        rightWheelForward();
        waitms(600);

        //Stop
        leftWheelStop();
        rightWheelStop();
    }

    void turn90DegreesCCW(void) {
        eputs("turn90DegreesCCW\r\n");

        // Turn counter clockwise 90 degrees
        leftWheelForward();
        rightWheelBackward();
        waitms(600);

        //Stop
        leftWheelStop();
        rightWheelStop();
    }

    //Inch robot forward slightly
    void aLittleForward(void)
    {
        leftWheelForward();
        rightWheelForward();
        waitms(100);
        leftWheelStop();
        rightWheelStop();
    }

    void aLittleBackward(void)
    {
        leftWheelBackward();
        rightWheelBackward();
        waitms(100);
        leftWheelStop();
        rightWheelStop();
    }

    void autodrive(RemoteCommand *currentCommand)
    {

```

```

char buff[16];
//RemoteCommand currentCommand;

eputs("autodrive\r\n");

//processRadioData(buff, &currentCommand);

// If a toggle command (SELECT: 's' or START: 'S') was received, handle it.
if (currentCommand->b1 == SELECT)
{
    *mode_flag = 0;
    coin_count = 0;
    return; // Exit autonomous mode and reset coin count to 0
}

//check if pause command was recieved
if(currentCommand->b2 == TRIANGLE)
{
    *mode_flag = 2;
    return;
}

// Drive forward.
leftWheelForward();
rightWheelForward();

// Update frequency measurement.
//frequency = (F_CPU * 60) / GetPeriod(60);

// If metal is detected, pick up the coin.
if (detectMetal())
{
    eputs("while(detectMetal())\r\n");
    collectCoin();
}

// If the perimeter is detected, perform a turn-around.
if (detectPerimeter())
{
    eputs("if(detectPerimeter())\r\n");
    turnAround();
}

// (Any frequency requests are handled within processRadioData.)
// If enough coins are collected, end automode.
if (coin_count == 20)
{
    //processRadioData(buff, &currentCommand); // Check for any pending commands.
    *mode_flag = 0; // Exit autonomous mode.
    coin_count = 0;
}

```



```

        waitms(50);

        return;
    }

//Displays countdown and lets user try to pick up as many coins as they can in a given time
//Alternatively: Has set amount of coins to pick up and lets user try to pick up as many as
they can in given time
//If we want to be able to count coins we will need a load cell
void gameMode(void)
{
    eputs("gameMode\r\n");
}

/*****
 * MANUAL MODE
 * This function processes a decoded remote command (RemoteCommand structure)
 * and drives the robot accordingly in manual mode.
 * It adjusts the speed scaling using L1 and R1, toggles the electromagnet with the
 * X button, and calculates forward/reverse speed and turning based on the left
 * joystick's Y and X axes respectively.
 * The thresholds for the deadzone are set using LEVEL4 and LEVEL5.
 *****/

// ----- Control electromagnet servo (raise/lower arm) using DPAD (buttons from set 1)
-----
// Define target servo positions for the electromagnet arm.
void ManualControl(RemoteCommand *command) {
    // ----- Static variables -----
    static int speedFactor = 50;    // Initial speed scaling factor (percentage)
    static int electromagnetOn = 0; // 0: off, 1: on.
    static int firstRun = 1;        // One-time initialization flag

    // Initialize electromagnet state on first run.
    if (firstRun) {
        PB7_0; // Turn electromagnet OFF.
        electromagnetOn = 0;
        firstRun = 0;
        eputs("Initial setup: Electromagnet OFF.\r\n");
    }

    // ----- Adjust speed factor using L1 and R1 (button set 2) -----
    //if (command->b2 == L1) {
    //    speedFactor -= 10;
    //    eputs("L1 pressed: Decreasing speed factor.\r\n");
    //}
    //if (command->b2 == R1) {
    //    speedFactor += 10;
    //    eputs("R1 pressed: Increasing speed factor.\r\n");
    //}
    //if (speedFactor < 0)    speedFactor = 0;
    //if (speedFactor > 100) speedFactor = 100;

```

```

// ----- Control electromagnet on/off (using X and O from button set 2) -----
if (command->b2 == X_BUTTON) {
    PB7_1;          // Turn electromagnet ON.
    electromagnetOn = 1;
    eputs("X pressed: Turning electromagnet ON.\r\n");
}
if (command->b2 == O_BUTTON) {
    PB7_0;          // Turn electromagnet OFF.
    electromagnetOn = 0;
    eputs("O pressed: Turning electromagnet OFF.\r\n");
}

// ----- Adjust electromagnet arm position using DPAD (button set 1) -----
// Vertical movement (up/down remains the same)
if (command->b1 == DPAD_UP) {
    if (ISR_pwm1 > MIN_ISR_PWM1) {
        ISR_pwm1 -= VERTICAL_STEP;
        waitms(VERTICAL_DELAY_MS);
        eputs("DPAD UP pressed: Moving electromagnet up.\r\n");
    }
}
if (command->b1 == DPAD_DOWN) {
    if (ISR_pwm1 < MAX_ISR_PWM1) {
        ISR_pwm1 += VERTICAL_STEP;
        waitms(VERTICAL_DELAY_MS);
        eputs("DPAD DOWN pressed: Moving electromagnet down.\r\n");
    }
}
// Inverted horizontal movement:
// DPAD_LEFT now increases ISR_pwm2 (moves right) and DPAD_RIGHT decreases ISR_pwm2 (moves
left)
if (command->b1 == DPAD_LEFT) {
    if (ISR_pwm2 < MAX_ISR_PWM2) {
        ISR_pwm2 += HORIZONTAL_STEP;
        waitms(HORIZONTAL_DELAY_MS);
        eputs("DPAD LEFT pressed: Moving electromagnet right (inverted).\r\n");
    }
}
if (command->b1 == DPAD_RIGHT) {
    if (ISR_pwm2 > MIN_ISR_PWM2) {
        ISR_pwm2 -= HORIZONTAL_STEP;
        waitms(HORIZONTAL_DELAY_MS);
        eputs("DPAD RIGHT pressed: Moving electromagnet left (inverted).\r\n");
    }
}

// ----- Compute wheel speeds from joystick positions -----
// Use the global mapValue() function (declared elsewhere) to scale joystick inputs.
int leftSpeed = 0;
if (command->ry < LEVEL4) {
    leftSpeed = mapValue(command->ry, 0, LEVEL4, MAX_SPEED, 0);
}

```

```

} else if (command->ry > LEVEL5) {
    leftSpeed = mapValue(command->ry, LEVEL5, 255, 0, -MAX_SPEED);
} else {
    leftSpeed = 0;
}

leftSpeed = (leftSpeed * speedFactor) / 100;

int rightSpeed = 0;
if (command->ly < LEVEL4) {
    rightSpeed = mapValue(command->ly, 0, LEVEL4, MAX_SPEED, 0);
} else if (command->ly > LEVEL5) {
    rightSpeed = mapValue(command->ly, LEVEL5, 255, 0, -MAX_SPEED);
} else {
    rightSpeed = 0;
}
rightSpeed = (rightSpeed * speedFactor) / 100;

// ----- Debug prints -----
eputs("ManualControl: leftSpeed=");
PrintNumber(leftSpeed, 10, 3);
eputs(", rightSpeed=");
PrintNumber(rightSpeed, 10, 3);
eputs(", speedFactor=");
PrintNumber(speedFactor, 10, 3);
eputs("\r\n");

// ----- Command the wheels -----
if (leftSpeed > SPEED_THRESHOLD)
    leftWheelForward();
else if (leftSpeed < -SPEED_THRESHOLD)
    leftWheelBackward();
else
    leftWheelStop();

if (rightSpeed > SPEED_THRESHOLD)
    rightWheelForward();
else if (rightSpeed < -SPEED_THRESHOLD)
    rightWheelBackward();
else
    rightWheelStop();

//if (decodeJoystick(command->ly, 0) > LEVEL5)
//    leftWheelForward();
//else if (decodeJoystick(command->ly, 0) < LEVEL4)
//    leftWheelBackward();
//else
//    leftWheelStop();

//if (decodeJoystick(command->ry, 0) > LEVEL5)
//    rightWheelForward();
//else if (decodeJoystick(command->ry, 0) < LEVEL4)

```

```

        //    rightWheelBackward();
    //else
        //    rightWheelStop();

    //----- More Driving Functions -----
    //Press L3 to turn robot 90 degrees counter clockwise
    if(command->b1 == L3)
    {
        turn90DegreesCCW();
    }

    //Press L3 to turn robot 90 degrees counter clockwise
    if(command->b1 == R3)
    {
        turn90DegreesCW();
    }

    //Press R1 to inch the robot backward
    if(command->b2 == R1)
    {
        aLittleBackward();
    }

    //Press R2 to inch the robot forward
    if(command->b2 == R2)
    {
        aLittleForward();
    }

    //----- Collect Coin -----
    if(command->b2 == L2)
    {
        collectCoin();
    }

    //
}

/*****
*****
ELECTROMAGNETIC FUNCTIONS
*****
*****/

// Function for detecting metal
// Returns 1 if metal is detected, else 0
int detectMetal(void)

```

```

{
    int threshold = 128;
    int voltage_threshold = 1000;

    eputs("detectMetal\r\n");
    //long int count;

    //count = GetPeriod(60);

    //frequency = (F_CPU * 60) / count;

    //printf("Count = %6d\r\n", count);

    voltage[0] = (readADC(ADC_CHSELR_CHSEL8) * 33000) / 0xffff;
    voltage[1] = (readADC(ADC_CHSELR_CHSEL9) * 33000) / 0xffff;

    // if (count > reference_count + threshold)
    // {
    //     PA6_0;
    //     return 1;
    // }

    if (voltage[0] < reference_voltage[0] - voltage_threshold || voltage[1] <
reference_voltage[1] - voltage_threshold)
    {
        PA6_0;
        return 1;
    }

    PA6_1;
    return 0;
}

// Function for detecting the perimeter
// Returns 1 if perimeter is reached, else 0
int detectPerimeter(void)
{
    int voltage_threshold = 2500;

    eputs("detectPerimeter\r\n");
    voltage[0] = (readADC(ADC_CHSELR_CHSEL8) * 33000) / 0xffff;
    voltage[1] = (readADC(ADC_CHSELR_CHSEL9) * 33000) / 0xffff;

    if (voltage[0] > reference_voltage[0] + voltage_threshold || voltage[1] >
reference_voltage[1] + voltage_threshold)
    {
        PA7_0;
        return 1;
    }

    PA7_1;
    return 0;
}

```

```

}

// Function for picking up a coin
void collectCoin(void)
{
    eputs("collectCoin\r\n");
    // Move robot backwards (if necessary)
    leftWheelBackward();
    rightWheelBackward();
    //if(automode_toggle_check()){
        //return;
    //}
    waitms(250);

    leftWheelStop();
    rightWheelStop();
    //if(automode_toggle_check()){
        //return;
    //}
    waitms(500);

    // Move arm down from starting position
    while (ISR_pwm2 < 255)
    {
        //if(automode_toggle_check()){
            //return;
        //}
        ISR_pwm2++;
        waitms(5);
    }
    while (ISR_pwm1 < 255)
    {
        //if(automode_toggle_check()){
            //return;
        //}
        ISR_pwm1++;
        waitms(5);
    }

    // Turn on electromagnet
    PB7_1;
    waitms(100);

    // Sweep left and right
    while (ISR_pwm2 > 140)
    {
        //if(automode_toggle_check()){
            //return;
        //}
        ISR_pwm2--;
        waitms(10);
    }
}

```

```

}

waitms(100);

while (ISR_pwm2 < 160)
{
    //if(automode_toggle_check()){
        //return;
    //}
    ISR_pwm2++;
    waitms(10);
}

waitms(100);

// Move arm above collection box
while (ISR_pwm1 > 130)
{
    //if(automode_toggle_check()){
        //return;
    //}
    ISR_pwm1--;
    waitms(5);
}

waitms(100);

while (ISR_pwm2 > 110)
{
    //if(automode_toggle_check()){
        //return;
    //}
    ISR_pwm2--;
    waitms(5);
}

// Move arm above collection box
while (ISR_pwm1 < 170)
{
    //if(automode_toggle_check()){
        //return;
    //}
    ISR_pwm1++;
    waitms(5);
}

// Turn off electromagnet
PB7_0;
waitms(1000);

```

```

    while (ISR_pwm1 > 60)
    {
        ISR_pwm1--;
        waitms(5);
    }

    //put coin sound function here

    // Increment the coin counter when a coin is collected
    coin_count++;

    // Return to starting position
    resetArm();

    waitms(1000);
}

void resetArm(void)
{
    // Turn off electromagnet
    PB7_0;
    waitms(100);

    // Move arm to starting position
    while (ISR_pwm2 < 255)
    {
        ISR_pwm2++;
        waitms(5);
    }
    while (ISR_pwm1 > 60)
    {
        ISR_pwm1--;
        waitms(5);
    }
}

/*****
*****

MAIN
*****
*****/

int main(void)
{
    Hardware_Init();
    initUART2(9600);

    // Just so your pointer mode_flag is valid:
    static int mode_value = 0; // 0 = manual, 1 = auto, 2 = paused
    mode_flag = &mode_value;

```



```

char buff[16];
RemoteCommand currentCommand;

// Clear screen and print startup messages.
eputs("\x1b[2J\x1b[1;1H");
eputs("\r\nSTM32L051 multi I/O example.\r\n");
eputs("Starting in MANUAL MODE.\r\n");

ReceptionOff();

// Optional: send some AT commands.
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+DVID0A13\r\n");

// Initialize wheels and electromagnet
PB3_0;
PB4_0;
PB5_0;
PB6_0;
PB7_0;
PA6_1;
PA7_1;

// Measure reference voltages.
reference_voltage[0] = (readADC(ADC_CHSELR_CHSEL8) * 33000) / 0xffff;
reference_voltage[1] = (readADC(ADC_CHSELR_CHSEL9) * 33000) / 0xffff;

while (1)
{
    reference_count = GetPeriod(60);

    if (reference_count > 0)
    {
        break;
    }

    eputs("NO SIGNAL\r\n");
}

reference_frequency = (F_CPU * 60) / reference_count;
// Main loop
while (1)
{
    count = GetPeriod(60);

    if (count > 0)

```

```

    {
        frequency = (F_CPU * 60) / count;
    }

    processRadioData(buff, &currentCommand);

    // If robot is paused and unpause command is received.
    if (*mode_flag == 2 && currentCommand.b2 == SQUARE)
    {
        *mode_flag = 1;
    }

    // Run autonomous routine if flag is set.
    if (*mode_flag == 1)
    {
        eputs("AUTONOMOUS MODE ACTIVE.\r\n");
        autodrive(&currentCommand);
        if (*mode_flag == 0)
        {
            eputs("Returning to MANUAL MODE.\r\n");
            leftWheelStop();
            rightWheelStop();
        }
    }
    else if (*mode_flag == 3)
    {
        eputs("GAME MODE ACTIVE.\r\n");
        gameMode();
    }

    waitms(2);
}

return 0;
}

```

APPENDIX IV: FULL REMOTE CODE

```

#include <XC.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/attrs.h>
#include <string.h>

// ===== CONFIGURATION BITS =====
#pragma config FOSC = FRCPLL           // Internal Fast RC oscillator (8 MHz) w/ PLL
#pragma config FPLLIDIV = DIV_2        // Divide FRC before PLL (4 MHz)
#pragma config FPLLMUL = MUL_20       // PLL Multiply (80 MHz)
#pragma config FPLLODIV = DIV_2       // Divide After PLL (40 MHz)
#pragma config FWDTEN = OFF            // Watchdog Timer Disabled
#pragma config FPBDIV = DIV_1          // PBCLK = SYSCLK
#pragma config FSOSCEN = OFF           // Secondary Oscillator off

```

```

// ===== DEFINES =====
#define SYSCLK          4000000L
#define PBCLK           SYSCLK
#define CHARS_PER_LINE  16
#define DEF_FREQ 2205L
#define Baud1BRG(desired_baud) ( (SYSCLK / (16*desired_baud))-1)

// Macro to convert desired baud rate to BRG value
#define Baud2BRG(desired_baud) ( (PBCLK / (16*desired_baud))-1 )

// ===== LCD PIN DEFINITIONS =====
#define LCD_RS          LATBbits.LATB3
#define LCD_RS_ENABLE   TRISBbits.TRISB3

#define LCD_E           LATAbits.LATA2
#define LCD_E_ENABLE    TRISAbits.TRISA2

#define LCD_D4          LATAbits.LATA3
#define LCD_D4_ENABLE   TRISAbits.TRISA3

#define LCD_D5          LATBbits.LATB4
#define LCD_D5_ENABLE   TRISBbits.TRISB4

#define LCD_D6          LATAbits.LATA4
#define LCD_D6_ENABLE   TRISAbits.TRISA4

#define LCD_D7          LATBbits.LATB5
#define LCD_D7_ENABLE   TRISBbits.TRISB5

// ===== SPEAKER PIN DEFINITIONS =====
#define SPEAKER          LATBbits.LATB12
#define SPEAKER_TRIS     TRISBbits.TRISB12

#define PIN_PERIOD PORTAbits.RA0

// ===== PS2 PIN DEFINITIONS =====
#define PS2_DATA          PORTBbits.RB0      // Input
#define PS2_DATA_TRIS     TRISBbits.TRISB0
#define PS2_CMD           LATBbits.LATB1      // Output
#define PS2_CMD_TRIS      TRISBbits.TRISB1
#define PS2_ATT           LATBbits.LATB2      // Output
#define PS2_ATT_TRIS      TRISBbits.TRISB2
#define PS2_CLK           LATBbits.LATB10     // Output
#define PS2_CLK_TRIS      TRISBbits.TRISB10

// ===== PS2 Button DEFINITIONS =====
#define LEVEL0 0
#define LEVEL1 28
#define LEVEL2 55
#define LEVEL3 82
#define LEVEL4 110
#define LEVEL5 146

```

```

#define LEVEL6 174
#define LEVEL7 201
#define LEVEL8 228
#define LEVEL9 256

// Button Values
#define NONE 255 //Verified
#define SQUARE 127 // verified
#define O_BUTTON 223 // verified
#define TRIANGLE 239 // verified
#define X_BUTTON 191 // verified
#define R1 247 // verified
#define R2 253 // verified
#define L1 251 // verified
#define L2 254 // verified

#define START 247 // verified
#define SELECT 254 // verified
#define L3 253 // verified
#define R3 251 // verified
#define DPAD_UP 239 // verified
#define DPAD_DOWN 191 // verified
#define DPAD_LEFT 127 // verified
#define DPAD_RIGHT 223 // verified

// ===== FUNCTION PROTOTYPES =====
void Timer4us(unsigned char t);
void waitms(unsigned int ms);
void waitus(unsigned int us);
void LCD_pulse(void);
void LCD_byte(unsigned char x);
void WriteData(unsigned char x);
void WriteCommand(unsigned char x);
void LCD_4BIT(void);
void LCDprint(char * string, unsigned char line, unsigned char clear);
void requestSlaveData(void);

// === ADDED/MODIFIED ===
void beepSpeaker(void);
int calcSignalStrength(int frequency);

void UART2Configure(int baud_rate);
int _mon_getc(int canblock);

void PS2_Init(void);
unsigned char PS2_TransferByte(unsigned char outByte);
void PS2_ReadData(unsigned char *d);

// ===== GLOBALS (if you need them) =====
int global_frequency = 0;
int global_coinCount = 0;

```

```

volatile unsigned int T2_overflow = 0;

// ===== LCD ROUTINES =====
void Timer4us(unsigned char t)
{
    T4CON = 0x8000; // enable Timer4, source PBCLK, 1:1 prescaler
    while(t >= 100)
    {
        t -= 100;
        TMR4 = 0;
        while(TMR4 < (SYSCLK/10000L));
    }
    while(t >= 10)
    {
        t -= 10;
        TMR4 = 0;
        while(TMR4 < (SYSCLK/100000L));
    }
    while(t > 0)
    {
        t--;
        TMR4 = 0;
        while(TMR4 < (SYSCLK/1000000L));
    }
    T4CONCLR = 0x8000;
}

void waitms(unsigned int ms)
{
    unsigned int j;
    for(j = 0; j < ms; j++)
    {
        Timer4us(250);
        Timer4us(250);
        Timer4us(250);
        Timer4us(250);
    }
}

void waitus(unsigned int us)
{
    while(us--)
        Timer4us(1);
}

void LCD_pulse(void)
{
    LCD_E = 1;
    Timer4us(40);
    LCD_E = 0;
}

```

```

void LCD_byte(unsigned char x)
{
    LCD_D7 = (x & 0x80) ? 1 : 0;
    LCD_D6 = (x & 0x40) ? 1 : 0;
    LCD_D5 = (x & 0x20) ? 1 : 0;
    LCD_D4 = (x & 0x10) ? 1 : 0;
    LCD_pulse();
    Timer4us(40);
    LCD_D7 = (x & 0x08) ? 1 : 0;
    LCD_D6 = (x & 0x04) ? 1 : 0;
    LCD_D5 = (x & 0x02) ? 1 : 0;
    LCD_D4 = (x & 0x01) ? 1 : 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_RS_ENABLE = 0;
    LCD_E_ENABLE = 0;
    LCD_D4_ENABLE = 0;
    LCD_D5_ENABLE = 0;
    LCD_D6_ENABLE = 0;
    LCD_D7_ENABLE = 0;

    LCD_E = 0;
    waitms(20);

    WriteCommand(0x33);
    WriteCommand(0x33);
    WriteCommand(0x32);

    WriteCommand(0x28);
    WriteCommand(0x0c);
    WriteCommand(0x01);
    waitms(20);
}

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

```

```

{
    int j;
    if(line == 2)
        WriteCommand(0xc0);
    else
        WriteCommand(0x80);
    waitms(5);
    for(j = 0; string[j] != 0; j++)
    {
        WriteData(string[j]);
    }
    if(clear)
    {
        for(; j < CHARS_PER_LINE; j++)
            WriteData(' ');
    }
}

// ===== JDY-40 ROUTINES =====

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

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

    U2MODESET = 0x8000;    // enable UART2
}

// Needed to by scanf() and gets()
int _mon_getc(int canblock)
{
    char c;

    if (canblock)
    {
        while( !U2STAbits.URXDA); // wait (block) until data available in RX buffer
        c=U2RXREG;
        while( U2STAbits.UTXBF);    // wait while TX buffer full
        U2TXREG = c;                // echo
        if(c=='\r') c='\n'; // When using PUTTY, pressing <Enter> sends '\r'. Ctrl-J sends
'\n'
        return (int)c;
    }
    else
    {
        if (U2STAbits.URXDA) // if data available in RX buffer

```

```

        {
            c=U2RXREG;
            if(c=='\r') c='\n';
            return (int)c;
        }
        else
        {
            return -1; // no characters to return
        }
    }
}

//Functions from Set up Two Timers

void uart_putc (unsigned char c)
{
    while( U2STAbits.UTXBF); // wait while TX buffer full
    U2TXREG = c; // send single character to transmit buffer
}

void uart_puts (char * buff)
{
    while (*buff)
    {
        uart_putc(*buff);
        buff++;
    }
}

unsigned char uart_getc (void)
{
    unsigned char c;

    while( !U2STAbits.URXDA); // wait (block) until data available in RX buffer
    c=U2RXREG;
    return c;
}

void SetupTimer1 (void)
{
    // Explanation here:
    // https://www.youtube.com/watch?v=bu6TTZHnMPY
    __builtin_disable_interrupts();
    PR1 =(SYSCLK/DEF_FREQ)-1; // since SYSCLK/FREQ = PS*(PR1+1)
    TMR1 = 0;
    T1CONbits.TCKPS = 0; // Pre-scaler: 1
    T1CONbits.TCS = 0; // Clock source
    T1CONbits.ON = 1;
    IPC1bits.T1IP = 5;
    IPC1bits.T1IS = 0;
    IFS0bits.T1IF = 0;
    IEC0bits.T1IE = 1;
}

```



```

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

volatile unsigned int Tick_Counter=0;
volatile unsigned char Second_Flag=1;

void __ISR(_TIMER_1_VECTOR, IPL5SOFT) Timer1_Handler(void)
{
    // Calculate the signal strength (clamp to 0..100)
    int strength = calcSignalStrength(global_frequency);
    if(strength > 100)
        strength = 100;
    else if(strength < 0)
        strength = 0;

    // Map strength to a beep repetition rate.
    // Here, beepFreq will vary from 5 (for low strength) to 55 (for high strength)
    int beepMinFreq = 5;
    int beepMaxFreq = 30;
    int beepFreq = beepMinFreq + (beepMaxFreq - beepMinFreq) * strength / 50;

    // Clear the timer interrupt flag
    IFS0CLR = _IFS0_T1IF_MASK;

    // Use a state machine to control the on/off period of the beep (without changing the
    speaker pitch).
    // The idea is: a fixed "on" period (250 ticks) and a variable "off" period
    (50000/beepFreq ticks).
    if(Second_Flag == 0)
    {
        SPEAKER = 0;        // Turn the speaker off during the off period.
        Tick_Counter++;
        // Use >= instead of == so that any overshoot still resets the counter.
        if(Tick_Counter >= (50000 / beepFreq))
        {
            Tick_Counter = 0;
            Second_Flag = 1;
        }
    }
    else // Second_Flag == 1
    {
        SPEAKER = !SPEAKER; // Toggle the speaker (this produces the constant pitch square
wave)
        Tick_Counter++;
        if(Tick_Counter >= 250)
        {
            Tick_Counter = 0;
            Second_Flag = 0;
        }
    }
}

```

```

}

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

    _CP0_SET_COUNT(0); // resets the core timer count
    while (PIN_PERIOD!=0) // Wait for square wave to be 0
    {
        if(_CP0_GET_COUNT() > (SYSCLK/4)) return 0;
    }

    _CP0_SET_COUNT(0); // resets the core timer count
    while (PIN_PERIOD==0) // Wait for square wave to be 1
    {
        if(_CP0_GET_COUNT() > (SYSCLK/4)) return 0;
    }

    _CP0_SET_COUNT(0); // resets the core timer count
    for(i=0; i<n; i++) // Measure the time of 'n' periods
    {
        while (PIN_PERIOD!=0) // Wait for square wave to be 0
        {
            if(_CP0_GET_COUNT() > (SYSCLK/4)) return 0;
        }
        while (PIN_PERIOD==0) // Wait for square wave to be 1
        {
            if(_CP0_GET_COUNT() > (SYSCLK/4)) return 0;
        }
    }

    return _CP0_GET_COUNT();
}

////////////////////////////////////
// UART1 functions used to communicate with the JDY40 //
////////////////////////////////////

// TXD1 is in pin 26
// RXD1 is in pin 24

int UART1Configure(int desired_baud)
{
    int actual_baud;

    // Peripheral Pin Select for UART1. These are the pins that can be used for U1RX from
    TABLE 11-1 of '60001168J.pdf':
    // 0000 = RPA2
    // 0001 = RPB6
    // 0010 = RPA4

```

```

// 0011 = RPB13
// 0100 = RPB2

ANSELB &= ~(1<<13); // Set RB13 as a digital I/O
TRISB |= (1<<13);    // configure pin RB13 as input
CNPUB |= (1<<13);    // Enable pull-up resistor for RB13
U1RXRbits.U1RXR = 3; // SET U1RX to RB13

// These are the pins that can be used for U1TX. Check table TABLE 11-2 of
'60001168J.pdf':
// RPA0
// RPB3
// RPB4
// RPB15
// RPB7

ANSELB &= ~(1<<15); // Set RB15 as a digital I/O
RPB15Rbits.RPB15R = 1; // SET RB15 to U1TX

U1MODE = 0;          // disable autobaud, TX and RX enabled only, 8N1, idle=HIGH
U1STA = 0x1400;       // enable TX and RX
U1BRG = Baud1BRG(desired_baud); // U1BRG = (FPb / (16*baud)) - 1
// Calculate actual baud rate
actual_baud = SYSCLK / (16 * (U1BRG+1));

U1MODESET = 0x8000;    // enable UART1

return actual_baud;
}

void putc1 (char c)
{
    while( U1STAbits.UTXBF); // wait while TX buffer full
    U1TXREG = c;             // send single character to transmit buffer
}

int SerialTransmit1(const char *buffer)
{
    unsigned int size = strlen(buffer);
    while(size)
    {
        while( U1STAbits.UTXBF); // wait while TX buffer full
        U1TXREG = *buffer;       // send single character to transmit buffer
        buffer++;               // transmit next character on following loop
        size--;                 // loop until all characters sent (when size = 0)
    }

    while( !U1STAbits.TRMT);     // wait for last transmission to finish

    return 0;
}

```

```

unsigned int SerialReceive1(char *buffer, unsigned int max_size)
{
    unsigned int num_char = 0;

    while(num_char < max_size)
    {
        while( !U1STAbits.URXDA);    // wait until data available in RX buffer
        *buffer = U1RXREG;            // empty contents of RX buffer into *buffer pointer

        // insert nul character to indicate end of string
        if( *buffer == '\n')
        {
            *buffer = '\0';
            break;
        }

        buffer++;
        num_char++;
    }

    return num_char;
}

```

// Copied from here: <https://forum.microchip.com/s/topic/a5C31000000MRVAEA4/t335776>

```

void delayus(uint16_t uiuSec)
{
    uint32_t ulEnd, ulStart;
    ulStart = _CP0_GET_COUNT();
    ulEnd = ulStart + (SYSCLK / 2000000) * uiuSec;
    if(ulEnd > ulStart)
        while(_CP0_GET_COUNT() < ulEnd);
    else
        while((_CP0_GET_COUNT() > ulStart) || (_CP0_GET_COUNT() < ulEnd));
}

```

```

unsigned int SerialReceive1_timeout(char *buffer, unsigned int max_size)
{
    unsigned int num_char = 0;
    int timeout_cnt;

    while(num_char < max_size)
    {
        timeout_cnt=0;
        while(1)
        {
            if(U1STAbits.URXDA) // check if data is available in RX buffer
            {
                *buffer = U1RXREG; // copy RX buffer into *buffer pointer
                break;
            }
            if(++timeout_cnt==100) // 100 * 100us = 10 ms
            {

```

```

        *buffer = '\n';
        break;
    }
    delayus(100);
}

// insert nul character to indicate end of string
if( *buffer == '\n')
{
    *buffer = '\0';
    break;
}

buffer++;
num_char++;
}

return num_char;
}

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

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

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

void ClearFIFO (void)
{
    unsigned char c;
    U1STA = 0x1400; // enable TX and RX, clear FIFO
    while(U1STAbits.URXDA) c=U1RXREG;
}

void SendATCommand (char * s)
{
    char buff[40];
    printf("Command: %s", s);
    LATB &= ~(1<<14); // 'SET' pin of JDY40 to 0 is 'AT' mode.
    delayms(10);
    SerialTransmit1(s);
    U1STA = 0x1400; // enable TX and RX, clear FIFO
    SerialReceive1(buff, sizeof(buff)-1);
    LATB |= 1<<14; // 'SET' pin of JDY40 to 1 is normal operation mode.
}

```

```

    delaysms(10);
    printf("Response: %s\n", buff);
}

void ReceptionOff (void)
{
    LATB &= ~(1<<14); // 'SET' pin of JDY40 to 0 is 'AT' mode.
    delaysms(10);
    SerialTransmit1("AT+DVID0000\r\n"); // Some unused id, so that we get nothing.
    delaysms(10);
    ClearFIFO();
    LATB |= 1<<14; // 'SET' pin of JDY40 to 1 is normal operation mode.
}

// ===== SPEAKER AND JDY-40 RECEIVING DATA ROUTINES =====

int calcSignalStrength(int frequency)
{
    int signal_strength = 10000 - frequency;

    if(signal_strength < 0) {
        return 0;
    } else if (signal_strength > 100) {
        return 100;
    }

    return (signal_strength*100)/30;
}

void beepSpeaker(void)
{
    //if(Second_Flag == 0){
    //    SPEAKER = 0;
    //}
    //else if(Second_Flag == 1){
    //    SPEAKER = 1;
    //}
    // Compute signal strength based on the global frequency
    //int strength = calcSignalStrength(global_frequency);
    //if (strength == 0)
    //    return;

    // Determine beep frequency based on strength (example linear mapping)
    //int beepMinFreq = 1;
    //int beepMaxFreq = 10;
    //int beepFreq = beepMinFreq + (beepMaxFreq - beepMinFreq) * strength / 100;

    // Calculate timer ticks for a half period.

```

```

    // The Timer1 interrupt will fire every half period so that toggling the pin creates a
square wave.
    // (SYSCLK / (2 * beepFreq)) gives the number of ticks for a half period.
    //unsigned int halfPeriodTicks = (SYSCLK / (2 * beepFreq)) - 1;

    // Reconfigure Timer1 for the desired half period.
    //if(Tick_Counter==22050/beepFreq)
    //{
    //    Tick_Counter=0;

    //    __builtin_disable_interrupts();
    //}
    //__builtin_disable_interrupts();
    //PR1 = halfPeriodTicks;    // Set period for half period timing
    //TMR1 = 0;                // Reset timer count
    //T1CONbits.TCKPS = 0;      // Pre-scaler: 1
    //T1CONbits.TCS = 0;        // Internal clock (PBCLK)
    //T1CONbits.ON = 1;         // Turn on Timer1 so that its interrupt will toggle the
speaker pin
    //__builtin_enable_interrupts();

    // Let the beep sound for ~100ms.
    // (During this time, Timer1 interrupts will toggle SPEAKER in the ISR.)
    //    waitms(100);

    // After 100ms, disable Timer1 and turn the speaker off.
    //__builtin_disable_interrupts();
    //T1CONbits.ON = 0;        // Stop Timer1
    //SPEAKER = 0;             // Ensure speaker is low (off)
    //__builtin_enable_interrupts();
}

void requestSlaveData(void)
{
    char buff[20];    // Buffer to hold the slave's response
    int timeout_cnt = 0;

    // Clear the receive FIFO so we get a fresh reply from the slave
    if(U1STAbits.URXDA)
        SerialReceive1_timeout(buff, sizeof(buff)-1);

    // Wait for a response with a timeout up to ~50ms
    timeout_cnt = 0;
    while(1)
    {
        if(U1STAbits.URXDA) break;    // Data has arrived
        if(++timeout_cnt > 100000) break; // Timeout after ~50ms
        delayus(100);                // 100us delay per iteration
    }
}

```

```

// If data is available, read it
if(U1STAbits.URXDA)
{
    SerialReceive1_timeout(buff, sizeof(buff)-1);
    if(strlen(buff) > 0)
    {
        // Debug print of the raw data received from the slave
        printf("Slave says: %s\r\n", buff);

        // === ADDED/MODIFIED ===
        // We now expect exactly 7 characters: first 5 for freq, last 2 for coin count
        // e.g. "1234512"
        if(strlen(buff) == 7)
        {
            char freqStr[6];
            char coinStr[3];

            strncpy(freqStr, buff, 5);
            freqStr[5] = '\0';
            strncpy(coinStr, &buff[5], 2);
            coinStr[2] = '\0';

            global_frequency = atoi(freqStr);
            global_coinCount = atoi(coinStr);

            // Call the speaker beep function based on frequency
            //beepSpeaker(global_frequency);

            // Display signal strength on LCD (line 1)
            char line1[20];
            int strength = calcSignalStrength(global_frequency);
            if (strength > 100){
                strength = 100;
            }
            sprintf(line1, "Strength: %d%%", strength);
            LCDprint(line1, 1, 1);

            // Display coin count on LCD (line 2)
            char line2[20];
            sprintf(line2, "Coin Count: %d", global_coinCount);
            LCDprint(line2, 2, 1);
        }
        else
        {
            // If the message isn't the proper length, just show an error:
            LCDprint("Bad data len!", 1, 1);
            LCDprint(buff, 2, 1);
        }
    }
    else
    {
        printf("No response from slave.\r\n");
    }
}

```



```

        LCDprint("Signal: NONE", 2, 1);
    }
}
else
{
    printf("No response from slave (timeout).\r\n");
    // Optionally, display a timeout message on the LCD.
    // LCDprint("Signal: TIMEOUT", 2, 1);
}

    delayms(50);
}

// ===== PS2 CONTROLLER BIT-BANGED SPI =====
static void ps2_delay(void)
{
    waitus(10);
}

unsigned char PS2_TransferByte(unsigned char outByte)
{
    unsigned char inByte = 0;
    int i;
    for(i = 0; i < 8; i++)
    {
        PS2_CMD = (outByte & 0x01) ? 1 : 0;
        outByte >>= 1;
        PS2_CLK = 0;
        ps2_delay();
        if(PS2_DATA)
            inByte |= (1 << i);
        PS2_CLK = 1;
        ps2_delay();
    }
    return inByte;
}

void PS2_Init(void)
{
    ANSELBbits.ANSB0 = 0;

    PS2_DATA_TRIS = 1;
    PS2_CMD_TRIS  = 0;
    PS2_ATT_TRIS  = 0;
    PS2_CLK_TRIS  = 0;

    PS2_CMD = 1;
    PS2_ATT = 1;
    PS2_CLK = 1;
    waitms(100);

    PS2_ATT = 0;

```

```

    PS2_TransferByte(0x01);
    PS2_TransferByte(0x43);
    PS2_TransferByte(0x00);
    PS2_TransferByte(0x01);
    PS2_TransferByte(0x00);
    PS2_ATT = 1;
    waitms(50);

    PS2_ATT = 0;
    PS2_TransferByte(0x01);
    PS2_TransferByte(0x44);
    PS2_TransferByte(0x00);
    PS2_TransferByte(0x01); // Set analog mode
    PS2_TransferByte(0x03); // Lock configuration
    PS2_ATT = 1;
    waitms(50);

    PS2_ATT = 0;
    PS2_TransferByte(0x01);
    PS2_TransferByte(0x43);
    PS2_TransferByte(0x00);
    PS2_TransferByte(0x00);
    PS2_TransferByte(0x5A);
    PS2_ATT = 1;
    waitms(50);
}

void PS2_ReadData(unsigned char *d)
{
    int i;
    PS2_ATT = 0;
    waitus(20);
    d[0] = PS2_TransferByte(0x01);
    d[1] = PS2_TransferByte(0x42);
    for(i = 2; i < 9; i++)
    {
        d[i] = PS2_TransferByte(0x00);
    }
    PS2_ATT = 1;
}

void Encode_Data(unsigned char * data, char * dest)
{
    unsigned char buttons1;
    unsigned char buttons2;
    unsigned char rx;
    unsigned char ry;
    unsigned char lx;
    unsigned char ly;

    //Encode buttons1 value
    if ((data[3]) == NONE)

```

```

{
    dest[0] = 'N';
}
else if ((data[3]) == SELECT)
{
    dest[0] = 's';
}
else if ((data[3]) == START)
{
    dest[0] = 'S';
}
else if ((data[3]) == DPAD_UP)
{
    dest[0] = 'U';
}
else if ((data[3]) == DPAD_DOWN)
{
    dest[0] = 'D';
}
else if ((data[3]) == DPAD_LEFT)
{
    dest[0] = 'L';
}
else if ((data[3]) == DPAD_RIGHT)
{
    dest[0] = 'R';
}
else if ((data[3]) == L3)
{
    dest[0] = 'l';
}
else if ((data[3]) == R3)
{
    dest[0] = 'r';
}
else
{
    dest[0] = 'E';
}

//Encode buttons2 value
if ((data[4]) == NONE)
{
    dest[1] = 'N';
}
else if ((data[4]) == X_BUTTON)
{
    dest[1] = 'X';
}
else if ((data[4]) == SQUARE)
{
    dest[1] = 'S';
}

```

```

}
else if ((data[4]) == O_BUTTON)
{
    dest[1] = 'O';
}
else if ((data[4]) == TRIANGLE)
{
    dest[1] = 'T';
}
else if ((data[4]) == R1)
{
    dest[1] = 'r';
}
else if ((data[4]) == R2)
{
    dest[1] = 'R';
}
else if ((data[4]) == L1)
{
    dest[1] = 'l';
}
else if ((data[4]) == L2)
{
    dest[1] = 'L';
}
else
{
    dest[1] = 'E';
}

//Encode rx value
if ((data[5]) >= LEVEL0 && (data[5]) < LEVEL1)
{
    dest[2] = '1';
}
else if ((data[5]) >= LEVEL1 && (data[5]) < LEVEL2)
{
    dest[2] = '2';
}
else if ((data[5]) >= LEVEL2 && (data[5]) < LEVEL3)
{
    dest[2] = '3';
}
else if ((data[5]) >= LEVEL3 && (data[5]) < LEVEL4)
{
    dest[2] = '4';
}
else if ((data[5]) >= LEVEL4 && (data[5]) < LEVEL5)
{
    dest[2] = '5';
}
else if ((data[5]) >= LEVEL5 && (data[5]) < LEVEL6)

```

```

{
    dest[2] = '6';
}
else if ((data[5]) >= LEVEL6 && (data[5]) < LEVEL7)
{
    dest[2] = '7';
}
else if ((data[5]) >= LEVEL7 && (data[5]) < LEVEL8)
{
    dest[2] = '8';
}
else if ((data[5]) >= LEVEL8 && (data[5]) < LEVEL9)
{
    dest[2] = '9';
}
else
{
    dest[2] = 'E';
}

//Encode ry value
if ((data[6]) >= LEVEL8 && (data[6]) < LEVEL9)
{
    dest[3] = '1';
}
else if ((data[6]) >= LEVEL7 && (data[6]) < LEVEL8)
{
    dest[3] = '2';
}
else if ((data[6]) >= LEVEL6 && (data[6]) < LEVEL7)
{
    dest[3] = '3';
}
else if ((data[6]) >= LEVEL5 && (data[6]) < LEVEL6)
{
    dest[3] = '4';
}
else if ((data[6]) >= LEVEL4 && (data[6]) < LEVEL5)
{
    dest[3] = '5';
}
else if ((data[6]) >= LEVEL3 && (data[6]) < LEVEL4)
{
    dest[3] = '6';
}
else if ((data[6]) >= LEVEL2 && (data[6]) < LEVEL3)
{
    dest[3] = '7';
}
else if ((data[6]) >= LEVEL1 && (data[6]) < LEVEL2)
{
    dest[3] = '8';
}

```

```

}
else if ((data[6]) >= LEVEL0 && (data[6]) < LEVEL1)
{
    dest[3] = '9';
}
else
{
    dest[3] = 'E';
}

//Encode lx value
if ((data[7]) >= LEVEL0 && (data[7]) < LEVEL1)
{
    dest[4] = '1';
}
else if ((data[7]) >= LEVEL1 && (data[7]) < LEVEL2)
{
    dest[4] = '2';
}
else if ((data[7]) >= LEVEL2 && (data[7]) < LEVEL3)
{
    dest[4] = '3';
}
else if ((data[7]) >= LEVEL3 && (data[7]) < LEVEL4)
{
    dest[4] = '4';
}
else if ((data[7]) >= LEVEL4 && (data[7]) < LEVEL5)
{
    dest[4] = '5';
}
else if ((data[7]) >= LEVEL5 && (data[7]) < LEVEL6)
{
    dest[4] = '6';
}
else if ((data[7]) >= LEVEL6 && (data[7]) < LEVEL7)
{
    dest[4] = '7';
}
else if ((data[7]) >= LEVEL7 && (data[7]) < LEVEL8)
{
    dest[4] = '8';
}
else if ((data[7]) >= LEVEL8 && (data[7]) < LEVEL9)
{
    dest[4] = '9';
}
else
{
    dest[4] = 'E';
}

```

```

//Encode ly value
if ((data[8]) >= LEVEL8 && (data[8]) < LEVEL9)
{
    dest[5] = '1';
}
else if ((data[8]) >= LEVEL7 && (data[8]) < LEVEL8)
{
    dest[5] = '2';
}
else if ((data[8]) >= LEVEL6 && (data[8]) < LEVEL7)
{
    dest[5] = '3';
}
else if ((data[8]) >= LEVEL5 && (data[8]) < LEVEL6)
{
    dest[5] = '4';
}
else if ((data[8]) >= LEVEL4 && (data[8]) < LEVEL5)
{
    dest[5] = '5';
}
else if ((data[8]) >= LEVEL3 && (data[8]) < LEVEL4)
{
    dest[5] = '6';
}
else if ((data[8]) >= LEVEL2 && (data[8]) < LEVEL3)
{
    dest[5] = '7';
}
else if ((data[8]) >= LEVEL1 && (data[8]) < LEVEL2)
{
    dest[5] = '8';
}
else if ((data[8]) >= LEVEL0 && (data[8]) < LEVEL1)
{
    dest[5] = '9';
}
else
{
    dest[5] = 'E';
}

//Set last char to terminate data
dest[6] = '\n';

}

// ===== MAIN PROGRAM =====
int main(void) {

    unsigned char ps2Data[9];
    char buff[80];

```

```

int timeout_cnt = 0;

// === 1) Basic PIC32 setup ===
DDPCON = 0;
CFGCON = 0;

TRISBbits.TRISB12 = 0;
LATBbits.LATB12 = 0;
INTCONbits.MVEC = 1;

SetupTimer1(); // Setup timer 1 and its interrupt
// Configure UART2 for debugging at 115200 baud
UART2Configure(115200);

Second_Flag = 0;
while(!Second_Flag);

PR2 = 0xffff; // When TMR2 hits 0xffff resets back to zero
T2CONbits.TCKPS = 0; // Pre-scaler: 1.
T2CONbits.TCS = 1; // External clock

// Configure UART1 for JDY40 communications at 9600 baud
UART1Configure(9600);

delayms(500); // Allow time for terminal startup

printf("JDY40 PS2 Master Demo. PIC32 as Master.\r\n");

// === 2) Configure JDY40 SET pin (RB14) ===
ANSELB &= ~(1 << 14); // Set RB14 as digital
TRISB &= ~(1 << 14); // Configure RB14 as output
LATB |= (1 << 14); // 'SET' = 1 --> normal operation mode

ANSELB &= ~(1<<6); // Set RB3 as a digital I/O
TRISB |= (1<<6); // configure pin RB6 as input
CNPUB |= (1<<6); // Enable pull-up resistor for RB6
T2CKRbits.T2CKR = 1; // Use RPB6 as input clock

ReceptionOff();
// === 3) Send AT commands to verify/configure JDY40 (optional) ===
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");

// Set an example unique device ID (0xABBA)
SendATCommand("AT+RFC013\r\n");
SendATCommand("AT+DVID0A13\r\n");

```



```

// === 4) Initialize LCD and PS2 controller ===
LCD_4BIT();
waitms(50);
LCDprint("PS2 Remote Demo", 1, 1);
LCDprint("Initializing...", 2, 1);
printf("Initializing PS2 and LCD...\r\n");

PS2_Init();
waitms(100);
LCDprint("PS2 Ready!", 1, 1);
LCDprint("Reading data...", 2, 1);

// Configure the speaker pin (RB12) as output
SPEAKER_TRIS = 0; // Set RB12 as output
SPEAKER = 0;      // Initialize the speaker output low

// === 5) Main loop: Read PS2, update LCD, send data via JDY40 ===
while(1) {
    //TMR2 = 0; // Reset timer count
    //T2_overflow=0; // Reset overflow count
    //T2CONbits.ON = 1; // Start timer
    //Second_Flag=0;
    // Check for overflow of timer 2
    //if(IFS0&_IFS0_T2IF_MASK)
    //{
    //    IFS0CLR=_IFS0_T2IF_MASK; // Clear overflow flag
    //    T2_overflow++; // Increment overflow counter
    //}
    //T2CONbits.ON = 0; // Stop Timer
    // Call the speaker beep function based on frequency
    //beepSpeaker();
    // 5a) Read the PS2 data packet (9 bytes)
    PS2_ReadData(ps2Data);

    // 5e) Construct the PS2 data string to send to the slave JDY module.
    Encode_Data(ps2Data, buff);

    // 5f) Transmit the PS2 data over UART1 (JDY40)
    putc1('!'); // Send the attention character first
    delayms(5); // Short delay
    SerialTransmit1(buff);

    delayms(20);

    // Send the request character '@' to the slave
    putc1('@');
    delayms(5);
    // Request data from the slave (this updates the LCD and speaker)
    requestSlaveData();

    // 5h) Small delay to set the pace of communication

```

```
        waitms(5);  
    }  
  
    return 0;  
}
```