Dr. Phillips 12/10/2024

Design Document

Trent Chipman
A02324069
Isaac Stoll
A02324729

1. Introduction

The device we have built is a small robotic arm with an electromagnet. It's built with 3 stepper motors and 3-D printed parts. It runs with an STM32 ARM microcontroller and is powered by a 7V battery. The robot arm is controlled with a Wii nunchuck using I2C communication.

2. Scope

This document contains information on how the robot arm operates, its design details, how it was created, and how it was tested.

3. Design Overview

3.1 Requirements:

3.1.1 The robot arm will not need external power connected

3.1.2 The arm will be able to be turned on and off easily

3.1.3 The robot arm will have 3 positions of movement

3.1.3.1 The Arm will be able to swivel at the base

3.1.3.2 The arm will have a main arm extension that can move up and down

3.1.3.3 The arm will have a secondary arm that can move up and down

3.1.4 The arm will be able to be controlled by a Wii nunchuck

3.1.4.1 The base swivel axis will move according to the x axis of the joystick

3.1.4.2 The main arm will move up and down according to the y axis of the joystick when no other buttons are pressed

3.1.4.3 When the Z button is held, the secondary arm with the electromagnet will move up and down according to the y axis on the joystick

3.1.4.4 When the C button is pushed the electromagnet will toggle on/off

3.1.5 The arm will be able to remember and repeat movements

3.2 Dependencies:

3.2.1 The arm will depend on a 7 volt external battery for power.

3.2.2 The electromagnet will depend on a 1.5 battery source for power.

3.2.3 The arm will depend on an electromagnet to pick metal objects up.

3.2.4 The arm will depend on 3 stepper motors for its movement.

3.2.5 The arm will depend on the ARM microcontroller for interfacing all the required components.

3.2.6 The motors will depend on motor drivers to controller current and data.

3.2.7 The arm will depend on a Wii nunchuck controller for user input.

3.2.8 The electromagnet will depend on a relay to toggle the magnet on/off.

3.2.9 The arm will depend on a switch to toggle the power on/off.

3.3 Theory of Operation:

3.3.1 Wii Nunchuck:

The Wii nunchuck is a device made by Nintendo for the Wii game system. It is a device with buttons and a joystick designed to interface with the Wii controller via I2C communication protocol. We communicate to the Wii nunchuck by connecting to its pins, configuring I2C with the Microcontroller, and requesting data. After the nunchuck register is initialized and the request number sent, it will reply with 6 bytes of data. The first two bytes are the x and y axis data, and the last register includes the C and Z buttons in its last 2 bits.

3.3.2 Electromagnet:

The electromagnet is made with a steel, three inch nail, 22 gauge insulated copper wire, and a D cell battery that is 1.5 volts. The way an electromagnet is made is by wrapping the insulated wire around the nail. The more coils you have the stronger it will be. Once we connect it to the 1.5 volt battery a current gets sent through the wire. This creates a magnetic field which will be used to pick the metal objects

with the arm. We send one end of the coiled wire to the relay which allows us to toggle power on or off (grab/release) to the magnet.

3.3.3 Relay:

The relay is a switching device we are using to turn on the electromagnet. When 2Vs is applied through the primary terminals the secondary terminals connect together and can pass up to 2 amps of current. In our device this allows us to connect the electromagnet to a high powered battery using a small switching voltage from a GPIO pin.

3.3.4 Motor drivers:

These devices receive 4 input data lines and 5V power from the microcontroller. They output higher voltage through 4 wires to control a stepper motor in the exact pattern as received through the 4 input pins. This can output the correct pattern to turn on the electromagnets in a stepper motor so it will spin. The MCU will have to output the correct pattern of signals to the motor drivers.

3.3.5 Stepper Motors:

Stepper motors typically have a rotor surrounded by 4 internal electromagnets that can each be turned on by running current through the right pins. In order to spin these motors require a high voltage of around 5V minimum to turn on the electromagnets one after another at the right speed to “step” through the rotor positions. This will be achieved with the motor drivers we are using for the device.

3.3.6 STM32 Microcontroller

The STM32L476RGTx microcontroller (MCU) brings all of the other components together. It has GPIO pins that are configured for I2C communication and it runs code that allows it to communicate with the Wii nunchuck and read its data. It has GPIO pins for motor controller outputs that turn on in a pattern of full stepping to tell the motor drivers how to run the stepper motors. There are functions written for each motor direction to step in the correct way. The MCU runs code that takes the nunchuck input and then decides if certain thresholds are met and calls the relevant motor output function or turns on the electromagnetic relay to turn on the magnet.

3.4 Design Alternatives:

3.4.1 Electromagnet Battery:

Originally, we were using a 9 volt battery for the electromagnet. However we had major problems with generating enough current to create a strong enough magnetic field. It was later changed to the 1.5 volt D cell battery to solve that problem.

3.4.2 Main Battery

We decided on using a 7V battery because it should provide enough power to run the motors and the electromagnet at the same time, and because we already had it.

3.4.3 Primary Arm

We went through different variations of the primary arm. The key problem to overcome was that the stepper motor does not have enough torque to lift a long arm with an electromagnet on the end. We had to decide on the right length of arm to use and eventually decided on 100mm of length. We also decided on using a counter weight on the opposite end of the arm to balance out the weight.

3.4.4 Motor Output Locking

Another design choice was whether or not to use motor locking, meaning that after turning a motor, the motor output would not turn off but keep outputting the latest "step" so that the arm would not be able to fall. The downside of this is that it draws more power, and also heats up the motor. We decided to implement motor locking because it would allow the motor to move more smoothly and have increased torque.

3.4.5 Relay

We had a few different relays to choose between. The first one we tested required at least 7 volts across the primary terminal to switch the secondary. This would have required a mosfet or transistor to switch the voltage required to switch the relay. The other relay we tested required only 2 volts to switch, so we used that because it would be simpler and allow us to control it directly from a GPIO pin.

4. Design Details

4.1 Circuit Setup

Our device uses the STM32L476RGTx microcontroller as the main power board and controller. We connected a 7V battery to Vin and ground of the microcontroller with a switch in series with the positive line. In order to configure the microcontroller to use an external power source we needed to remove a jumper on pins JP1 and shifted the jumper to the right on JP5. We connected 3 groups of 4 pins(PB0-3, PB8-11, PB12-15) each to their own motor driver input. The motor drivers were all connected to ground and 5V power. The relay was connected to pin PB5 and ground with its primary terminals. Its secondary terminals connected to the positive output of the 1.5V battery and the electromagnet. The other end of the electromagnet connected to the negative terminal of the 1.5V battery. Pins PB6 and PB7 were the clock and data lines respectively for the I2C communication and connected into the Wii nunchuck along with ground and 3.3V to power it. The nunchuck pinout was followed to connect to the correct pins. Refer to schematic below to see a basic diagram of the circuit setup (Figure 1).

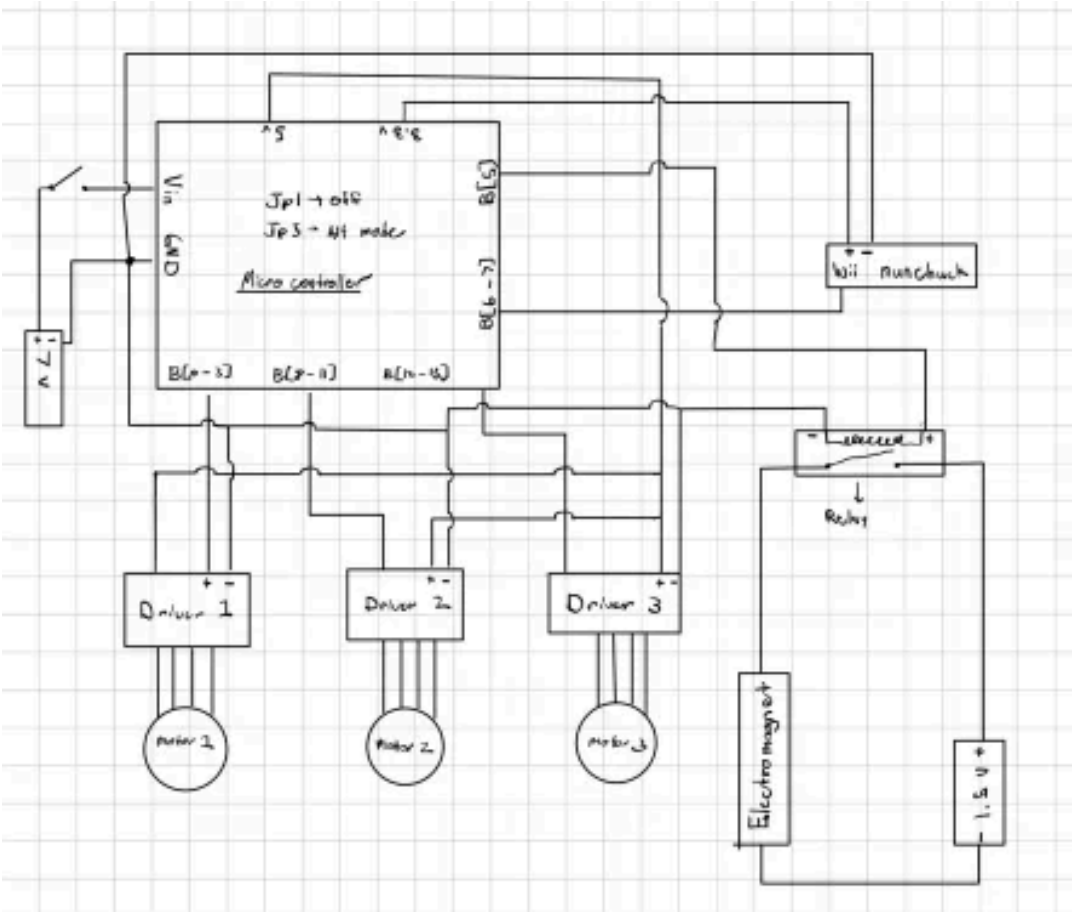


Figure 1

4.2 Electromagnet

To make the electromagnet you will need preferably an iron, steel, zinc nail/screw. About eight feet of enameled or insulated copper wire, around 20 to 22 gauge. Along with a 1.5 volt D cell battery. First step is to wrap the wire around the nail creating coils from tip to tip. Once you reach the end, loop the wire back around to allow for a stronger magnetic field. Next is to strip both ends of the wire and attach it to the battery on the positive and negative terminals. With this the magnet will be complete next we attached the magnet to the secondary arm of the robot with electrical tape. With the spare wire we soldered on two pieces to the secondary pins of the relay to allow for a switching mechanism. From the relay we have a wire connecting the GPIOB pin 5 to the data pin of the relay. The electromagnetic should look something Figure 2 down below.

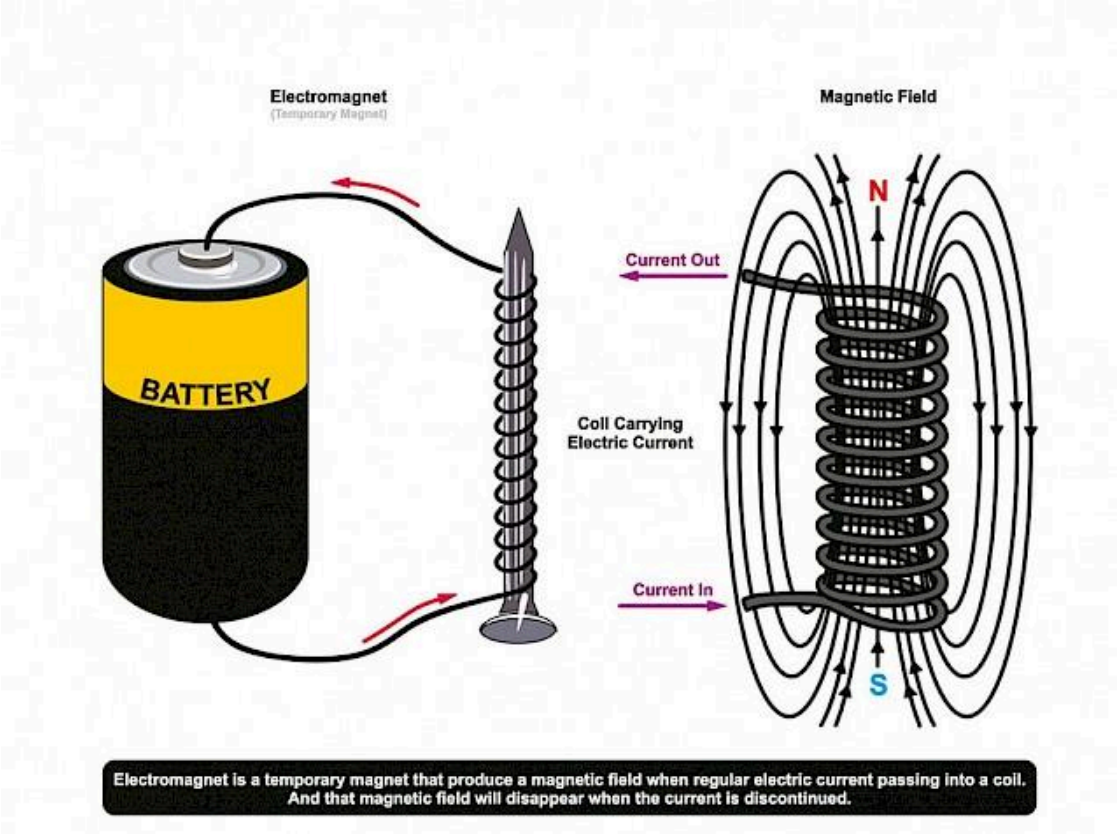


Figure 2

4.2 Physical Design

The robot arm was made out of 3D printed parts. We decided on a general size and shape for the base, the tower, and the two arms. We measured the size of the stepper motor housing and shaft. Then we got help from a friend to design the parts in SolidWorks and printed each of them. We fit the motors into these parts and put it all together. The electromagnet was connected to the arm with electrical tape. The whole arm and its box base were fitted over the microcontroller and the battery and the base motor driver. The two arm motor drivers and the relay were taped to the top of the base. The battery for the electromagnet was fitted into the first arm as a counter weight.

4.3 Wii nunchuck

The wii nunchuck uses four pins for its pin out; it contains a data line, clock line, power line, and ground line. The clock line is connected to the microcontroller through GPIOB pin 6 and the data line is connected through GPIOB pin 7. The ground line is connected to the ground pin on the microcontroller, and the power line is connected to the 3.3 volts pin on the microcontroller. To see which line is which refer to the diagram below (Figure 3).

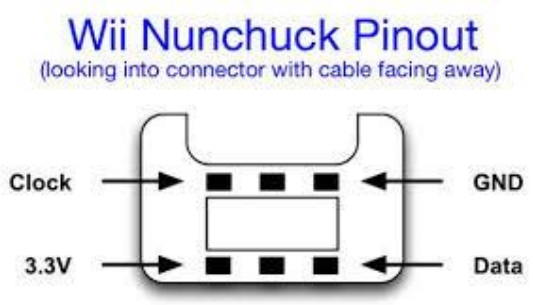


Figure (3)

4.4 Stepper motors/drivers

For all of the motor drivers we have them connected to the ground pin and 5 volt pin on the microcontroller. From the microcontroller we have GPIOB pins [0-3] connected to the first motor driver and the driver is connected to the motor. For the second driver we have GPIOB pins [8-11] connected. For the third driver we have GPIOB pins [12-15] connected. Refer to the appendices for GPIO pin layout on the microcontroller.

4.5 Programming

We used the CubeMX STM program to generate files to help us connect to I2C. We set up the GPIOB pins 0-5,8-15 all as output pins. We set up GPIOB 6-7 to their respective I2C ports, and configured them to be open-drain, alternate mode, and fast speed. We generated a starting file with this program. Then we wrote code to use the I2C, control the motors, check the Wii nunchuck data, and implement a storage and repeat function. The code roughly follows this pseudocode:

```
Start
        Start clocks
        Set motor output pins and other GPIO
        Initialize I2C
        Start WiiChuck registers
        Forever:
                Ask for Wiichuck data
                Get Wiichuck register values
                If buttons pressed/joystick toggled:
                        Activate appropriate motor function
                        Keep track of each move amount of time in an array
                        Debounce C button
                        When C button pressed toggle relay
                If copy mode activated:
                        Follow each move stored
End
```

We wrote functions for running each of the motors in both directions. We set the 4 output pins used by that motor in ODR to a hex value to go through each of the steps while waiting a certain amount of time in between. This steps the stepper motors so they turn. We used the HAL_Delay function and waited between 5 and 20 milliseconds depending on if we needed more or less torque for that motor.

```
void RunMotorBaseRight(uint32_t steps){
  //Run continuously for time
  int wait = 5;
  //int steps = time*10;
  for(int i = 0; i < steps; i++){
    //Mask output 0011
    GPIOB->ODR |= 0x0000000F;
    GPIOB->ODR &= 0xFFFFFFF3;
    //Delay for step to happen
    HAL_Delay(wait);
    //Mask output 0110
    GPIOB->ODR |= 0x0000000F;
    GPIOB->ODR &= 0xFFFFFFF6;
    //Delay for step to happen
    HAL_Delay(wait);
    //Mask output 1100
```

Then we wrote a function for toggling the electromagnet on and off by checking whether it is already on or off and turning it on or off accordingly.

```
void MagnetToggle(void){
        //Toggle Solenoid
        if(SolenoidOn){
          //If on, turn off
          GPIOB->ODR &= 0xFFFFFFCF;
          SolenoidOn = 0;
        }else{
          //If off, turn on
          GPIOB->ODR |= 0x00000030;
          SolenoidOn = 1;
        }
}
```

For the I2C code we used some functions in the Hal files. We used the MX_I2C1_Init() function to set the timing parameters and other important settings for I2C.

```
void MX_I2C1_Init(void)
]{
  hi2c1.Instance = I2C1;
  hi2c1.Init.Timing = 0x0010061A;
  hi2c1.Init.OwnAddress1 = 0;
  hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c1.Init.OwnAddress2 = 0;
  hi2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
  hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
```

We also used the HAL_I2C_Master_Receive and HAL_I2C_Master_Transmit functions (see appendix). The Wii nunchuck has the address of 0x52, which, shifted left by one bit and modified is 0xA4 and 0xA5 for sending and receiving. Once I2C parameters are set, we call these functions to initialize the Wii nunchuck data registers by sending 0xF0, 0x55 followed by 0xFB, 0x00.

```
//Device addresses
uint16_t SendAdd = 0xA4;
uint16_t RecAdd = 0xA5;
//Data registers to send and receive
uint8_t Data1[2] = {0xF0, 0x55};
uint8_t Data2[2] = {0xFB, 0x00};
uint8_t WiiData[6];
//Start Wii nunchuck registers
ReturnedVal = HAL_I2C_Master_Transmit(&hi2c1, SendAdd, Data1, 2, Timeout);
HAL_Delay(10);
ReturnedVal = HAL_I2C_Master_Transmit(&hi2c1, SendAdd, Data2, 2, Timeout);
HAL_Delay(10);
```

Then in our main loop function we continuously poll the nunchuck by sending the command to tell it to send data, and then calling the command to receive that data.

```
while (1)
{
  //Ask for wii nunchuck data
  ReturnedVal = HAL_I2C_Master_Transmit(&hi2c1, SendAdd, DataRec, 1, Timeout);
  HAL_Delay(2);
  //Receive wii nunchuck data
  ReturnedVal = HAL_I2C_Master_Receive(&hi2c1, RecAdd, WiiData, 6, Timeout);
  HAL_Delay(2);
```

This gives us 6 bytes with all the most recent data from the nunchuck. Then our code uses that data with a bunch of if statements to check for all of the conditions in the X and Y axis and Z and C buttons. The Z and C data is inverted and anded with the correct bit spot.

```
//Z and C
Button1 = ~WiiData[5] & 0x00000001;
Button2 = ~WiiData[5] & 0x00000002;
```

The X axis value is in the first register or first byte, and the Y axis data is in the second. We check if these are greater than 0xB0 or less than 0x40 to see if the joystick is pushed to the end or bottom of its range. Based on those conditions we call the correct motor or magnet functions. The X axis is used to call the BaseMotor functions and then the Z button decides whether the Y axis calls the main arm motor or the magnet arm motor (see appendix for full logic, Main.c).

```
//Main base right or left
if(WiiData[0] > 0xB0){
  TrackStep(0x02);
  RunMotorBaseRight(1);
}else if(WiiData[0] < 0x40){
  TrackStep(0x03);
  RunMotorBaseLeft(1);
```

Each of these conditions also calls a TrackStep function with a value for the type of function being called. We created variable arrays to track each step type and amount for the first 10 movements taken. This tracking function adds that function to a list to remember or if already on that type it increments the amount that function is called.

```
//Variables for tracking what movements made
uint8_t IMove = 0;//Index of what move your on
uint8_t MoveType[10];//Arrays of types and steps of each move
uint32_t MoveSteps[10];
```

```
void TrackStep(uint8_t type){
  //Track step
  if(MoveType[IMove] == type){
    MoveSteps[IMove] ++;
  }else{
    if(IMove < 9){
      IMove ++;
      MoveType[IMove] = type;
      MoveSteps[IMove] = 1;
    }
  }
}
```

Then on the condition that the Z button is pressed while the toggle is all the way to the right, it activates the RetraceMovements() function which loops through all the steps in the array calling the correct function for that step's amount of time.

5. Testing

5.1 Wii nunchuck:

We hooked up the two data lines on a logic analyzer to see if we could get the clock working. Originally we had an extremely difficult time getting the clock enabled. What that told us is that our prescaler was wrong and the clock was wrong. Through a lot of experimenting we finally found the right timing and what told it was correct was the clock signal we were receiving on the oscilloscope.

5.2 Motors:

With the nunchuck working we tested if we could get the motors to turn with user input. The way we tested it was simply by providing input and seeing if it behaved correctly.

5.3 External power:

We had only used one motor before and didn’t know how many the MCU could handle. We tested that all 3 motor drivers could be plugged in at the same time and that all of them could be controlled individually by the GPO output pins. This test was successful, and we also made sure that the board being powered by the 7V battery could power them all.

5.4 Remember feature:

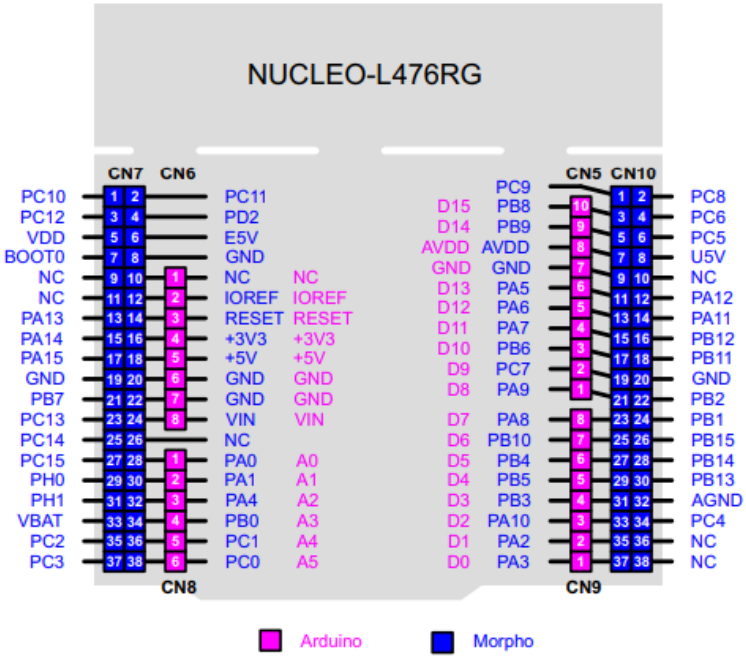
The remember feature was a more complicated function that was being added to the current design without messing up the original functionality. Testing this mostly consisted of manually checking the code and calculating what it would do. After checking that all functions worked it was able to remember and repeat movements from the user. Different movements and toggling of the relay were tested and it successfully recreated them.

6. Conclusion

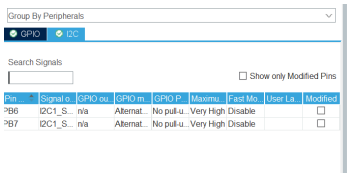
When we originally set out to do this project our goal was to learn about the I2C communication protocol and implement it into a robotic arm that would be controlled with a Wii nunchuck controller. We successfully got the motors working correctly along with interfacing the nunchuck with the microcontroller. It works as we intended and can even pick up metal objects using an electromagnet. As all things it's not perfect, ideally we would get stronger motors because frankly the stepper motors we have are not very powerful and it really limits how much weight it can pick up. To solve this, we shortened the main arm and gave it a counter weight to help the torque issue. On top of that we could redesign the 3D printed parts to have it hide the exposed wires better and just have a better overall look. We tested I2C communication with a logic analyzer, along with checking the behavior of the motors. We changed the battery source of both the electromagnet and microcontroller to make sure we had enough power. Overall this project was a success and very enjoyable to learn about.

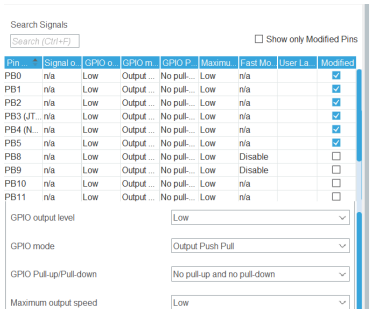
Appendices

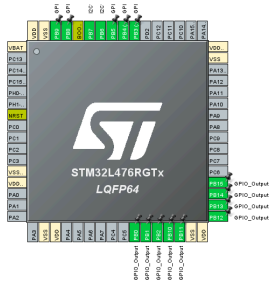
(GPIO pin layout)

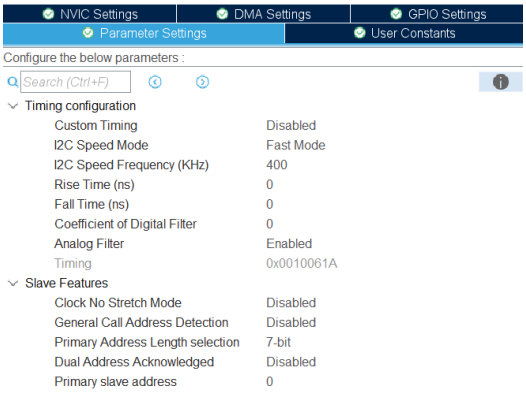


(Pin Setup in CubeMX and I2C parameters)









(Registers for Wii Nunchuck Data)

Data byte receive								Address
Joystick X								0x00
Joystick Y								0x01
Accelerometer X (bit 9 to bit 2 for 10-bit resolution)								0x02
Accelerometer Y (bit 9 to bit 2 for 10-bit resolution)								0x03
Accelerometer Z (bit 9 to bit 2 for 10-bit resolution)								0x04
Accel. Z bit 1	Accel. Z bit 0	Accel. Y bit 1	Accel. Y bit 0	Accel. X bit 1	Accel. X bit 0	C-button	Z-button	0x05

Byte 0x00 : X-axis data of the joystick

Byte 0x01 : Y-axis data of the joystick

Byte 0x02 : X-axis data of the accellerometer sensor

Byte 0x03 : Y-axis data of the accellerometer sensor

Byte 0x04 : Z-axis data of the accellerometer sensor

Byte 0x05 : bit 0 as Z button status - 0 = pressed and 1 = release

bit 1 as C button status - 0 = pressed and 1 = release

bit 2 and 3 as 2 lower bit of X-axis data of the accellerometer sensor

bit 4 and 5 as 2 lower bit of Y-axis data of the accellerometer sensor

bit 6 and 7 as 2 lower bit of Z-axis data of the accellerometer sensor

Figure 3: Wii Nunchuk I²C Output

Main.c

```
#include "main.h"
#include "i2c.h"
#include "gpio.h"
//StepMotor functions
void StepMotor_Initialize(void);
void MagnetToggle(void);
int SolenoidOn = 0;
void RunMotor(uint8_t, uint8_t);
void RunMotorBaseRight(uint32_t);
void RunMotorBaseLeft(uint32_t);
void RunMotorArmUp(uint32_t);
void RunMotorArmDown(uint32_t);
void RunMotorMagUp(uint32_t);
void RunMotorMagDown(uint32_t);
void RetraceMovements(void);
void TrackStep(uint8_t);
//Variables for tracking what movements made
uint8_t IMove = 0;//Index of what move your on
```

```
uint8_t MoveType[10];//Arrays of types and steps of each move
uint32_t MoveSteps[10];

void SystemClock_Config(void);

/**
 * @brief  The application entry point.
 * @retval int
 */
int main(void)
{
		//Initialize Hal stuff
 HAL_Init();
		//System clock set HSI
 SystemClock_Config();
		//Config GPIOs
 MX_GPIO_Init();
		//Init I2C params
 MX_I2C1_Init();

		//Device addresses
		uint16_t SendAdd = 0xA4;
		uint16_t RecAdd = 0xA5;
		//Data arryas to send and receive
		uint8_t Data1[2] = {0xF0, 0x55};
		uint8_t Data2[2] = {0xFB, 0x00};
		uint8_t Data3[2] = {0x40, 0x00};
		uint8_t Data4[2] = {0x00, 0x00};
		uint8_t WiiData[6];
		uint8_t DataRec[1] = {0x00};
		HAL_StatusTypeDef ReturnedVal;
		uint32_t Timeout = 0xFFFFFFFF;

		//Bariables for debouncing of button 2 and toggling of solenoid
		int Button1 = 0;
		int Button2 = 0;
		int Button2Pressed = 0;

		//Start Wii nunchuck registers
		ReturnedVal = HAL_I2C_Master_Transmit(&hi2c1, SendAdd, Data1, 2, Timeout);
		HAL_Delay(10);
		ReturnedVal = HAL_I2C_Master_Transmit(&hi2c1, SendAdd, Data2, 2, Timeout);
		HAL_Delay(10);
 while (1)
 {
			//Ask for wii nunchuck data
			ReturnedVal = HAL_I2C_Master_Transmit(&hi2c1, SendAdd, DataRec, 1, Timeout);
			HAL_Delay(2);
			ReturnedVal = HAL_I2C_Master_Receive(&hi2c1, RecAdd, WiiData, 6, Timeout);
			HAL_Delay(2);
			//Z and C
			Button1 = ~WiiData[5] & 0x00000001;
			Button2 = ~WiiData[5] & 0x00000002;

			//Main base right or left
			if(WiiData[0] > 0xB0){
				TrackStep(0x02);
				RunMotorBaseRight(1);
			}else if(WiiData[0] < 0x40){
				TrackStep(0x03);
				RunMotorBaseLeft(1);
```

```
                        //Arm segments based on Button1
                }else if(Button1 == 1){
                        //Debounce Button
                        //Magnet arm
                        if(WiiData[1] > 0xB0){
                                TrackStep(0x06);
                                RunMotorMagUp(1);
                        }else if(WiiData[1] < 0x40){
                                TrackStep(0x07);
                                RunMotorMagDown(1);
                        }
                        }else {
                        //Main arm
                        if(WiiData[1] > 0xB0){
                                TrackStep(0x04);
                                RunMotorArmUp(1);
                        }else if(WiiData[1] < 0x40){
                                TrackStep(0x05);
                                RunMotorArmDown(1);
                        }
                }
                if(Button2 == 2){
                        //Debounce Button2
                        //Activate/Deactivate Magnet
                        if(Button2Pressed == 0){
                                //If button pressed when not pressed already
                                //Toggle Solenoid
                                MagnetToggle();
                                //Track toggle on next step
                                        IMove ++;
                                        MoveType[IMove] = 0x01;
                                //Set flag that button was just pressed
                                Button2Pressed = 1;
                                //Also a bit of delay
                                HAL_Delay(100);
                        }
                }else{
                        //Once button not pressed 4 times, flag is reset
                        Button2Pressed = 0;
                }

                if(WiiData[0] > 0xB0 && Button1 == 1){
                        //Activate Recall mode
                        RetraceMovements();
                        HAL_Delay(1000);
                }
}

}

void TrackStep(uint8_t type){
        //Track step
        if(MoveType[IMove] == type){
                MoveSteps[IMove] ++;
        }else{
                if(IMove < 9){
                        IMove ++;
                        MoveType[IMove] = type;
                        MoveSteps[IMove] = 1;
                }
        }
```

```
}

void RetraceMovements(void){
	for(uint8_t i = 1; i < IMove; i++){
		//For each step recorded
		//Call that function
		//For the recorded duration
		if(MoveType[i] == 0x00){
			//Nothing
		}else if(MoveType[i] == 0x01){
			MagnetToggle();
		}else if(MoveType[i] == 0x02){
			RunMotorBaseRight(MoveSteps[i]);
		}else if(MoveType[i] == 0x03){
			RunMotorBaseLeft(MoveSteps[i]);
		}else if(MoveType[i] == 0x04){
			RunMotorArmUp(MoveSteps[i]);
		}else if(MoveType[i] == 0x05){
			RunMotorArmDown(MoveSteps[i]);
		}else if(MoveType[i] == 0x06){
			RunMotorMagUp(MoveSteps[i]);
		}else if(MoveType[i] == 0x07){
			RunMotorMagDown(MoveSteps[i]);
		}
	}
}

void MagnetToggle(void){
		//Toggle Solenoid
		if(SolenoidOn){
			//If on, turn off
			GPIOB->ODR &= 0xFFFFFFCF;
			SolenoidOn = 0;
		}else{
			//If off, turn on
			GPIOB->ODR |= 0x00000030;
			SolenoidOn = 1;
		}
}

void RunMotorBaseRight(uint32_t steps){
	//Run continuously for time
	int wait = 5;
	//int steps = time*10;
	for(int i = 0; i < steps; i++){
		//Mask output 0011
		GPIOB->ODR |= 0x0000000F;
		GPIOB->ODR &= 0xFFFFFFF3;
		//Delay for step to happen
		HAL_Delay(wait);
		//Mask output 0110
		GPIOB->ODR |= 0x0000000F;
		GPIOB->ODR &= 0xFFFFFFF6;
		//Delay for step to happen
		HAL_Delay(wait);
		//Mask output 1100
		GPIOB->ODR |= 0x0000000F;
		GPIOB->ODR &= 0xFFFFFFFC;
		//Delay for step to happen
		HAL_Delay(wait);
		//Mask output 1001
```

```
                GPIOB->ODR |= 0x0000000F;
                GPIOB->ODR &= 0xFFFFFFF9;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask to turn off
                GPIOB->ODR &= 0xFFFFFFF0;
                //HAL_Delay(delay*10);
        }
}
void RunMotorBaseLeft(uint32_t steps){
        //Run continuously for time
        int wait = 5;
        //int steps = time*10;
        for(int i = 0; i < steps; i++){
                //Mask output 0011
                GPIOB->ODR |= 0x0000000F;
                GPIOB->ODR &= 0xFFFFFFF9;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 0110
                GPIOB->ODR |= 0x0000000F;
                GPIOB->ODR &= 0xFFFFFFFC;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 1100
                GPIOB->ODR |= 0x0000000F;
                GPIOB->ODR &= 0xFFFFFFF6;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 1001
                GPIOB->ODR |= 0x0000000F;
                GPIOB->ODR &= 0xFFFFFFF3;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask to turn off
                GPIOB->ODR &= 0xFFFFFFF0;
                //HAL_Delay(delay*10);
        }
}
void RunMotorArmUp(uint32_t steps){
        //Run continuously for time
        int wait = 20;
        //int steps = time*10;
        for(int i = 0; i < steps; i++){
                //Mask output 0011
                GPIOB->ODR |= 0x00000F00;
                GPIOB->ODR &= 0xFFFFF3FF;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 0110
                GPIOB->ODR |= 0x00000F00;
                GPIOB->ODR &= 0xFFFFF6FF;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 1100
                GPIOB->ODR |= 0x00000F00;
                GPIOB->ODR &= 0xFFFFFCFF;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 1001
                GPIOB->ODR |= 0x00000F00;
```

```
                GPIOB->ODR &= 0xFFFFF9FF;
                //Delay for step to happen
                HAL_Delay(wait-5);
                //Mask to turn off
                //GPIOB->ODR &= 0xFFFFF0FF;
                //HAL_Delay(delay*10);
        }
}
void RunMotorArmDown(uint32_t steps){
        //Run continuously for time
        int wait = 20;
        //int steps = time*10;
        for(int i = 0; i < steps; i++){
                //Mask output 0011
                GPIOB->ODR |= 0x00000F00;
                GPIOB->ODR &= 0xFFFFF9FF;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 0110
                GPIOB->ODR |= 0x00000F00;
                GPIOB->ODR &= 0xFFFFFCFF;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 1100
                GPIOB->ODR |= 0x00000F00;
                GPIOB->ODR &= 0xFFFFF6FF;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 1001
                GPIOB->ODR |= 0x00000F00;
                GPIOB->ODR &= 0xFFFFF3FF;
                //Delay for step to happen
                HAL_Delay(wait-5);
                //Mask to turn off
                //GPIOB->ODR &= 0xFFFFF0FF;
                //HAL_Delay(delay*10);
        }
}
void RunMotorMagUp(uint32_t steps){
        //Run continuously for time
        int wait = 5;
        //int steps = time*10;
        for(int i = 0; i < steps; i++){
                //Mask output 0011
                GPIOB->ODR |= 0x0000F000;
                GPIOB->ODR &= 0xFFFF3FFF;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 0110
                GPIOB->ODR |= 0x0000F000;
                GPIOB->ODR &= 0xFFFF6FFF;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 1100
                GPIOB->ODR |= 0x0000F000;
                GPIOB->ODR &= 0xFFFFCFFF;
                //Delay for step to happen
                HAL_Delay(wait);
                //Mask output 1001
                GPIOB->ODR |= 0x0000F000;
                GPIOB->ODR &= 0xFFFF9FFF;
```

```
                        //Delay for step to happen
                        HAL_Delay(wait-2);
                        //Mask to turn off
                        //GPIOB->ODR &= 0xFFFF0FFF;
                        //HAL_Delay(delay*10);
                }
}
void RunMotorMagDown(uint32_t steps){
                //Run continuously for time
                int wait = 5;
                //int steps = time*10;
                for(int i = 0; i < steps; i++){
                        //Mask output 0011
                        GPIOB->ODR |= 0x0000F000;
                        GPIOB->ODR &= 0xFFFF9FFF;
                        //Delay for step to happen
                        HAL_Delay(wait);
                        //Mask output 0110
                        GPIOB->ODR |= 0x0000F000;
                        GPIOB->ODR &= 0xFFFFCFFF;
                        //Delay for step to happen
                        HAL_Delay(wait);
                        //Mask output 1100
                        GPIOB->ODR |= 0x0000F000;
                        GPIOB->ODR &= 0xFFFF6FFF;
                        //Delay for step to happen
                        HAL_Delay(wait);
                        //Mask output 1001
                        GPIOB->ODR |= 0x0000F000;
                        GPIOB->ODR &= 0xFFFF3FFF;
                        //Delay for step to happen
                        HAL_Delay(wait-2);
                        //Mask to turn off
                        //GPIOB->ODR &= 0xFFFF0FFF;
                        //HAL_Delay(delay*10);
                }
}

void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

 /** Configure the main internal regulator output voltage
 */
 if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
 {
  Error_Handler();
 }

 /** Initializes the RCC Oscillators according to the specified parameters
 * in the RCC_OscInitTypeDef structure.
 */
 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSIState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
  Error_Handler();
 }
```

```
 /** Initializes the CPU, AHB and APB buses clocks
 */
 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                       |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
 {
  Error_Handler();
 }
}
```

Hal Functions used for I2C

```
#include "main.h"
#include "i2c.h"


I2C_HandleTypeDef hi2c1;

/* I2C1 init function */
void MX_I2C1_Init(void)
{
//Configure instance of I2C with timing and parameters for Wii Nunchuck
 hi2c1.Instance = I2C1;
 hi2c1.Init.Timing = 0x0010061A;
 hi2c1.Init.OwnAddress1 = 0;
 hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
 hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
 hi2c1.Init.OwnAddress2 = 0;
 hi2c1.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
 hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
 hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
 if (HAL_I2C_Init(&hi2c1) != HAL_OK)
 {
  Error_Handler();
 }
}

HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData,
                                          uint16_t Size, uint32_t Timeout)
{
 uint32_t tickstart;
 uint32_t xfermode;

 if (hi2c->State == HAL_I2C_STATE_READY)
 {
  /* Process Locked */
  __HAL_LOCK(hi2c);

  /* Init tickstart for timeout management*/
  tickstart = HAL_GetTick();
```

```
if (I2C_WaitOnFlagUntilTimeout(hi2c, I2C_FLAG_BUSY, SET, I2C_TIMEOUT_BUSY, tickstart) != HAL_OK)
{
 return HAL_ERROR;
}

hi2c->State     = HAL_I2C_STATE_BUSY_TX;
hi2c->Mode      = HAL_I2C_MODE_MASTER;
hi2c->ErrorCode = HAL_I2C_ERROR_NONE;

/* Prepare transfer parameters */
hi2c->pBuffPtr  = pData;
hi2c->XferCount = Size;
hi2c->XferISR   = NULL;

if (hi2c->XferCount > MAX_NBYTE_SIZE)
{
 hi2c->XferSize = MAX_NBYTE_SIZE;
 xfermode = I2C_RELOAD_MODE;
}
else
{
 hi2c->XferSize = hi2c->XferCount;
 xfermode = I2C_AUTOEND_MODE;
}

if (hi2c->XferSize > 0U)
{
 /* Preload TX register */
 /* Write data to TXDR */
 hi2c->Instance->TXDR = *hi2c->pBuffPtr;

 /* Increment Buffer pointer */
 hi2c->pBuffPtr++;

 hi2c->XferCount--;
 hi2c->XferSize--;

 /* Send Slave Address */
 /* Set NBYTES to write and reload if hi2c->XferCount > MAX_NBYTE_SIZE and generate RESTART */
 I2C_TransferConfig(hi2c, DevAddress, (uint8_t)(hi2c->XferSize + 1U), xfermode,
                    I2C_GENERATE_START_WRITE);
}
else
{
 /* Send Slave Address */
 /* Set NBYTES to write and reload if hi2c->XferCount > MAX_NBYTE_SIZE and generate RESTART */
 I2C_TransferConfig(hi2c, DevAddress, (uint8_t)hi2c->XferSize, xfermode,
                    I2C_GENERATE_START_WRITE);
}

while (hi2c->XferCount > 0U)
{
 /* Wait until TXIS flag is set */
 if (I2C_WaitOnTXISFlagUntilTimeout(hi2c, Timeout, tickstart) != HAL_OK)
 {
   return HAL_ERROR;
 }
 /* Write data to TXDR */
 hi2c->Instance->TXDR = *hi2c->pBuffPtr;

 /* Increment Buffer pointer */
```

```
    hi2c->pBuffPtr++;

    hi2c->XferCount--;
    hi2c->XferSize--;

    if ((hi2c->XferCount != 0U) && (hi2c->XferSize == 0U))
    {
      /* Wait until TCR flag is set */
      if (I2C_WaitOnFlagUntilTimeout(hi2c, I2C_FLAG_TCR, RESET, Timeout, tickstart) != HAL_OK)
      {
        return HAL_ERROR;
      }

      if (hi2c->XferCount > MAX_NBYTE_SIZE)
      {
        hi2c->XferSize = MAX_NBYTE_SIZE;
        I2C_TransferConfig(hi2c, DevAddress, (uint8_t)hi2c->XferSize, I2C_RELOAD_MODE,
                           I2C_NO_STARTSTOP);
      }
      else
      {
        hi2c->XferSize = hi2c->XferCount;
        I2C_TransferConfig(hi2c, DevAddress, (uint8_t)hi2c->XferSize, I2C_AUTOEND_MODE,
                           I2C_NO_STARTSTOP);
      }
    }
  }

  /* No need to Check TC flag, with AUTOEND mode the stop is automatically generated */
  /* Wait until STOPF flag is set */
  if (I2C_WaitOnSTOPFlagUntilTimeout(hi2c, Timeout, tickstart) != HAL_OK)
  {
    return HAL_ERROR;
  }

  /* Clear STOP Flag */
  __HAL_I2C_CLEAR_FLAG(hi2c, I2C_FLAG_STOPF);

  /* Clear Configuration Register 2 */
  I2C_RESET_CR2(hi2c);

  hi2c->State = HAL_I2C_STATE_READY;
  hi2c->Mode  = HAL_I2C_MODE_NONE;

  /* Process Unlocked */
  __HAL_UNLOCK(hi2c);

  return HAL_OK;
 }
 else
 {
  return HAL_BUSY;
 }
}

/**
 * @brief  Receives in master mode an amount of data in blocking mode.
 * @param  hi2c Pointer to a I2C_HandleTypeDef structure that contains
 *               the configuration information for the specified I2C.
 * @param  DevAddress Target device address: The device 7 bits address value
 *         in datasheet must be shifted to the left before calling the interface
```

```
  * @param  pData Pointer to data buffer
  * @param  Size Amount of data to be sent
  * @param  Timeout Timeout duration
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData,
                                         uint16_t Size, uint32_t Timeout)
{
  uint32_t tickstart;

  if (hi2c->State == HAL_I2C_STATE_READY)
  {
    /* Process Locked */
    __HAL_LOCK(hi2c);

    /* Init tickstart for timeout management*/
    tickstart = HAL_GetTick();

    if (I2C_WaitOnFlagUntilTimeout(hi2c, I2C_FLAG_BUSY, SET, I2C_TIMEOUT_BUSY, tickstart) != HAL_OK)
    {
      return HAL_ERROR;
    }

    hi2c->State     = HAL_I2C_STATE_BUSY_RX;
    hi2c->Mode      = HAL_I2C_MODE_MASTER;
    hi2c->ErrorCode = HAL_I2C_ERROR_NONE;

    /* Prepare transfer parameters */
    hi2c->pBuffPtr  = pData;
    hi2c->XferCount = Size;
    hi2c->XferISR   = NULL;

    /* Send Slave Address */
    /* Set NBYTES to write and reload if hi2c->XferCount > MAX_NBYTE_SIZE and generate RESTART */
    if (hi2c->XferCount > MAX_NBYTE_SIZE)
    {
      hi2c->XferSize = 1U;
      I2C_TransferConfig(hi2c, DevAddress, (uint8_t)hi2c->XferSize, I2C_RELOAD_MODE,
                         I2C_GENERATE_START_READ);
    }
    else
    {
      hi2c->XferSize = hi2c->XferCount;
      I2C_TransferConfig(hi2c, DevAddress, (uint8_t)hi2c->XferSize, I2C_AUTOEND_MODE,
                         I2C_GENERATE_START_READ);
    }

    while (hi2c->XferCount > 0U)
    {
      /* Wait until RXNE flag is set */
      if (I2C_WaitOnRXNEFlagUntilTimeout(hi2c, Timeout, tickstart) != HAL_OK)
      {
        return HAL_ERROR;
      }

      /* Read data from RXDR */
      *hi2c->pBuffPtr = (uint8_t)hi2c->Instance->RXDR;

      /* Increment Buffer pointer */
      hi2c->pBuffPtr++;
```

```
    hi2c->XferSize--;
    hi2c->XferCount--;

    if ((hi2c->XferCount != 0U) && (hi2c->XferSize == 0U))
    {
      /* Wait until TCR flag is set */
      if (I2C_WaitOnFlagUntilTimeout(hi2c, I2C_FLAG_TCR, RESET, Timeout, tickstart) != HAL_OK)
      {
        return HAL_ERROR;
      }

      if (hi2c->XferCount > MAX_NBYTE_SIZE)
      {
        hi2c->XferSize = MAX_NBYTE_SIZE;
        I2C_TransferConfig(hi2c, DevAddress, (uint8_t)hi2c->XferSize, I2C_RELOAD_MODE,
                           I2C_NO_STARTSTOP);
      }
      else
      {
        hi2c->XferSize = hi2c->XferCount;
        I2C_TransferConfig(hi2c, DevAddress, (uint8_t)hi2c->XferSize, I2C_AUTOEND_MODE,
                           I2C_NO_STARTSTOP);
      }
    }
  }

  /* No need to Check TC flag, with AUTOEND mode the stop is automatically generated */
  /* Wait until STOPF flag is set */
  if (I2C_WaitOnSTOPFlagUntilTimeout(hi2c, Timeout, tickstart) != HAL_OK)
  {
    return HAL_ERROR;
  }

  /* Clear STOP Flag */
  __HAL_I2C_CLEAR_FLAG(hi2c, I2C_FLAG_STOPF);

  /* Clear Configuration Register 2 */
  I2C_RESET_CR2(hi2c);

  hi2c->State = HAL_I2C_STATE_READY;
  hi2c->Mode  = HAL_I2C_MODE_NONE;

  /* Process Unlocked */
  __HAL_UNLOCK(hi2c);

  return HAL_OK;
 }
 else
 {
  return HAL_BUSY;
 }
}
```