Ardiuno I2C

Shawn Hagler
EEL-4713
Florida State University
Panama City, United States
sth20u@fsu.edu

Jaehyun Lee
EEL-4713
Florida State University
Panama City, United States
jl21bd@fsu.edu

Abstract

The integration of Arduino and the Inter-integrated Circuit (I2C) protocol presents an efficient and minimalistic approach to complex electronic project development. This paper explores the Arduino platform, focusing on its open-source environment, programming structure, and the Integrated Development Environment (IDE). It highlights the procedural distinction and function of the setup() and loop() constructs in Arduino programming. Furthermore, we discuss the implementation of the I2C protocol through the Wire library, emphasizing its importance in simplifying device connectivity and the ease of incorporation into Arduino projects. Results from a demonstration project involving an Arduino Uno R3 as a peripheral device and an Arduino Nano as a controller illustrate the practical application and effectiveness of I2C communication through a simple LED control task.

I. Introduction

The Arduino ecosystem represents a paradigm shift in the way individuals from various backgrounds approach electronics and embedded systems. Initially conceived as a tool for artists and designers to create interactive prototypes without a steep learning curve in electronics, Arduino's reach has extensively permeated the spheres of education, hobbyist projects, and professional product development. Its open-source hardware and software philosophy empowers users to contribute, modify, and personalize their experiences, which not only democratizes access to technology but also fosters a global community of innovation. At the heart of the Arduino platform is the pairing of accessible hardware, consisting of a variety of microcontroller boards, with user-friendly software, namely, the Arduino Integrated Development Environment (IDE). This environment is unique in its approach—a confluence of simplicity and functionality, offering a low barrier of entry for novices without sacrificing the depth required by advanced users. Through this IDE, users can write, validate, compile, and upload code to their Arduino hardware.

Delving deeper into the inner workings of the Arduino programming model, one encounters the structural foundation of every Arduino sketch—the setup() and loop() functions. This dichotomy establishes a clear and intuitive framework for controlling the flow of the program. The setup() function is called once at the beginning of a sketch; it is the initialization phase where configurations such as pin modes, libraries, and communications protocols are defined. In stark contrast, the loop() function encapsulates the dynamic portion of the sketch, where sensory data is read, logic is processed, and output is generated in a perpetual cycle. Together, they embody a balance of preparation and execution that streamlines the programming process. Yet, the prowess of Arduino does not cease at the level of individual board programming. Connected systems and the burgeoning field of the Internet of Things (IoT) necessitate communication protocols that can link together multiple devices efficiently. It is here that the Inter-integrated Circuit (I2C) protocol takes center stage, bringing with it a practical, two-wire solution for serial communication. The I2C protocol is masterfully integrated into the Arduino platform through the Wire library, a staple in Arduino's communication toolkit. With minimal wiring—using only a Serial Data Line (SDA) and a Serial Clock Line (SCL)—multiple devices can connect to a single controller. Each device on the I2C bus can be individually addressed by the controller, soliciting data or issuing commands as needed. The beauty of the I2C protocol lies in its ability to handle complex communication scenarios with relative ease, all while conserving valuable input/output (I/O) pins on the Arduino microcontroller.

Exploring the Wire library reveals a suite of functions that abstract away the intricacies of the I2C protocol. These functions—read(), write(), and onReceive()—provide a streamlined interface for transmitting and receiving data, registering callbacks, and managing the interactions between devices. Significantly, this allows users to focus on the higher-level aspects of their projects without being bogged down by the low-level mechanics of device communication. To illustrate the practical application of these concepts, this paper describes a demonstration project in which an Arduino Uno R3, acting as a peripheral device, and an Arduino Nano, serving as the controller, communicate over an I2C connection. The nuances of this interaction are examined, and the performance of the Arduino system, in conjunction with the I2C protocol, is evaluated through the simple yet illustrative task of controlling an LED's blinking pattern. In expanding on the significance of Arduino and I2C, this introduction has set the stage for a comprehensive exploration of an ecosystem that has not only revolutionized how we approach embedded systems but also continues to be at the vanguard of user-centric design in electronics. The subsequent sections will delve into the specific architectural and technical aspects, the related work in the field, the project's outcomes, and the avenues for future exploration, painting a complete picture of Arduino's integral role in the landscape of modern electronics and communication.

II. Existing Works

Prior studies have documented the versatility of the Arduino platform and its applicability to a diverse range of projects. The literature shows a wealth of information on the use of the Arduino IDE and programming syntax for various applications. Past work has also focused on the integration of I2C in microcontroller-based systems, often highlighting the Wire library's role in abstracting the complexities of the protocol. Comparisons with other communication protocols such as SPI and UART have also been documented to underscore the benefits of using I2C in certain scenarios.

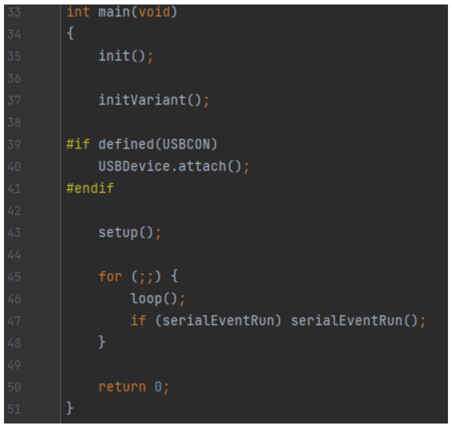
III. Implementation

A. Ardiuno Programming

The setup and loop() functions are essential in Arduino programming. The setup() function is responsible for initializing the environment and preparing for the main program, while the loop() function contains the actual main program functionalities. By separating the initialization and core functionalities, the Arduino code becomes easier to maintain and modify, enabling reusability of functions and clearer separation of concerns. It's important to carefully handle delays in the loop in order to prevent blocking the entire program, especially if multi-threading or other parallel processing is anticipated.[3]

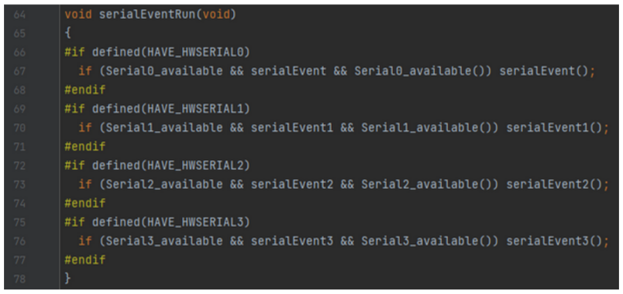
The setup() function is called once at the beginning of the program and is used to initialize variables, configure initial settings, set up communication protocols like Serial, specify pin modes for digital pins, and initialize hardware components such as sensors or actuators. Any time-consuming setup operations, such as waiting for sensor stabilization, are also included in the setup() function. Its primary purpose is to prepare the environment for the main program functionalities. It's important to focus the setup() function solely on initialization and to avoid executing core program functionalities within it. On the other hand, the loop() function runs continuously after the setup() function has finished its execution. It contains the main program functionalities and is continuously executed in a loop. Variables declared inside the loop() function are only valid within the loop, with their values being lost when the loop ends. To preserve data between loops, variables should be declared in a more global scope. The last line of the loop() function is followed by the first line, allowing for a continuous flow of the program. [3] As shown in the figure 1, setup() function is called once within the main() function of the Arduino program, then the loop() function is running consistently. Within the loop it is also consistently checking for the serial event.

Figure 2 shows the SerialEvent() function which is called at the end of the loop() function when data is available. It can use Serial.read() to capture the data.



```
int main(void)
{
    init();

    initVariant();

#if defined(USBCON)
    USBDevice.attach();
#endif

    setup();

    for (;;) {
        loop();
        if (serialEventRun) serialEventRun();
    }

    return 0;
}
```

Fig. 1. main() function located in main.cpp file in the core directory of Arduino program packages



```
void serialEventRun(void)
{
#if defined(HAVE_HWSERIAL0)
  if (Serial0_available && serialEvent && Serial0_available()) serialEvent();
#endif
#if defined(HAVE_HWSERIAL1)
  if (Serial1_available && serialEvent1 && Serial1_available()) serialEvent1();
#endif
#if defined(HAVE_HWSERIAL2)
  if (Serial2_available && serialEvent2 && Serial2_available()) serialEvent2();
#endif
#if defined(HAVE_HWSERIAL3)
  if (Serial3_available && serialEvent3 && Serial3_available()) serialEvent3();
#endif
}
```

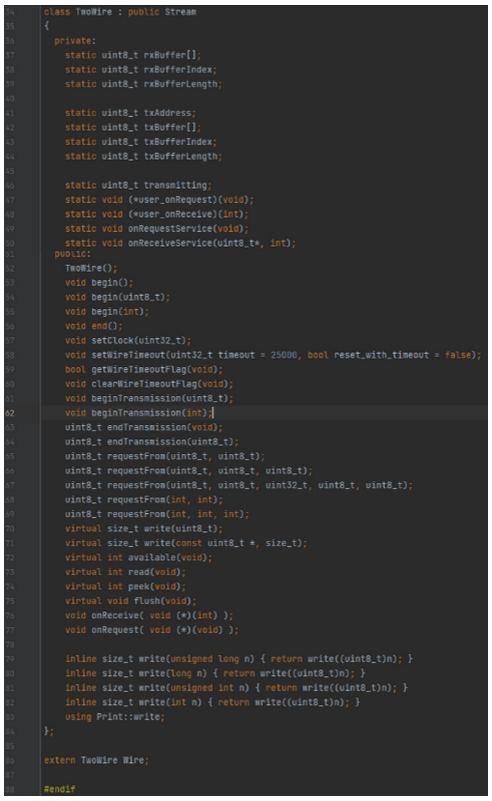
Fig. 2. The serialEventRun() function located in the HardwareSerial.cpp

B. I2C

The Inter-integrated circuit (I2C) protocol is a useful way to add complex features to Arduino projects without adding complexity to wiring. It allows connecting multiple peripheral devices with just a few wires, such as sensors, displays, and motor drivers. The I2C protocol involves using two lines to send and receive data: a serial clock pin (SCL) and a serial data pin (SDA). The controller sends out instructions through the I2C bus on the data pin (SDA), and the instructions are prefaced with the address, so that only the correct device listens. Each device in the I2C bus is functionally independent from the controller, but will respond with information when prompted by the controller. Because the I2C protocol allows for each enabled device to have its own unique address, and for both the controller and peripheral devices to take turns communicating over a single line, it is possible for an Arduino board to communicate with many devices or other boards using just two pins on the microcontroller. The Wire library in the Arduino ecosystem handles the complex aspects of the I2C protocol, making it easy for users to implement and use I2C devices in their projects. [4]

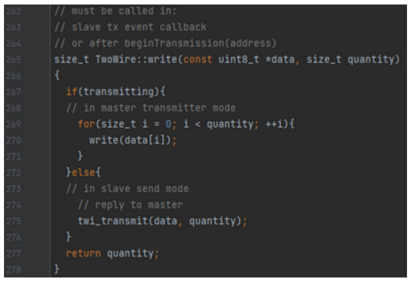
C. Wire Library

The Wire library in Arduino is used for communication with I2C devices. I2C, or Inter-Integrated Circuit, is a serial communication protocol used to connect multiple microcontrollers and peripheral devices to the same bus. The Wire library provides functions for transmitting and receiving data over the I2C bus. One key feature of the Wire library is that it provides a consistent interface for communication with I2C devices, making it easier to work with different I2C components. It allows for easy integration and communication with various sensors, displays, and other I2C devices commonly used in Arduino projects. Additionally, the Wire library can be used to communicate with multiple I2C devices connected to the same bus, by addressing each device with its unique 7-bit address. The library's implementation also includes a 32-byte buffer for communication, and any communication should stay within this limit to ensure data integrity. [5] Key functions that are used in this project are write(), read(), and onReceive() functions. The write() function in the Wire library is used to send data from a peripheral device to a controller device or to queue bytes for transmission from a controller to a peripheral device. It can take different parameters, including a single byte value, a string to be sent as a series of bytes, or an array of data to be sent as bytes. The length parameter specifies the number of bytes to transmit. The function returns the number of bytes written, although this value is optional and does not need to be read. Figure 4 shows that the write() function calls the twi_transmit() function the parameters of (const uint8_t* data, uint8_t length) and figure 5 contains the function definition of the twi_transmit() function which writes the data to the transmit buffer (txBuffer).



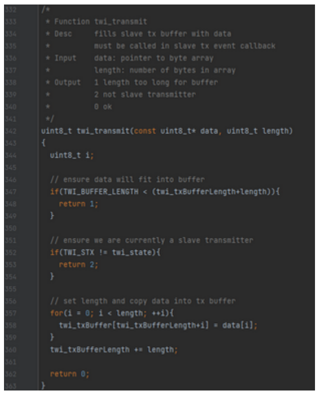
```
class TwoWire : public Stream
{
  private:
    static uint8_t rxBuffer[];
    static uint8_t rxBufferIndex;
    static uint8_t rxBufferLength;

    static uint8_t txAddress;
    static uint8_t txBuffer[];
    static uint8_t txBufferIndex;
    static uint8_t txBufferLength;

    static uint8_t transmitting;
    static void (*user_onRequest)(void);
    static void (*user_onReceive)(int);
    static void onRequestService(void);
    static void onReceiveService(uint8_t*, int);
  public:
    TwoWire();
    void begin();
    void begin(uint8_t);
    void begin(int);
    void end();
    void setClock(uint32_t);
    void setWireTimeout(uint32_t timeout = 25000, bool reset_with_timeout = false);
    bool getWireTimeoutFlag(void);
    void clearWireTimeoutFlag(void);
    void beginTransmission(uint8_t);
    void beginTransmission(int);
    uint8_t endTransmission(void);
    uint8_t endTransmission(uint8_t);
    uint8_t requestFrom(uint8_t, uint8_t);
    uint8_t requestFrom(uint8_t, uint8_t, uint8_t);
    uint8_t requestFrom(uint8_t, uint8_t, uint32_t, uint8_t, uint8_t);
    uint8_t requestFrom(int, int);
    uint8_t requestFrom(int, int, int);
    virtual size_t write(uint8_t);
    virtual size_t write(const uint8_t *, size_t);
    virtual int available(void);
    virtual int read(void);
    virtual int peek(void);
    virtual void flush(void);
    void onReceive( void (*)(int) );
    void onRequest( void (*)(void) );

    inline size_t write(unsigned long n) { return write((uint8_t)n); }
    inline size_t write(long n) { return write((uint8_t)n); }
    inline size_t write(unsigned int n) { return write((uint8_t)n); }
    inline size_t write(int n) { return write((uint8_t)n); }
    using Print::write;
};

extern TwoWire Wire;

#endif
```

Fig. 3. TwoWire class, data members and member functions



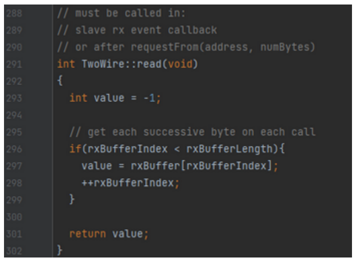
```
// must be called in:
// slave tx event callback
// or after beginTransmission(address)
size_t TwoWire::write(const uint8_t *data, size_t quantity)
{
  if(transmitting){
  // in master transmitter mode
    for(size_t i = 0; i < quantity; ++i){
      write(data[i]);
    }
  }else{
  // in slave send mode
    // reply to master
    twi_transmit(data, quantity);
  }
  return quantity;
}
```

Fig. 4. write() function definition



```
/*
 * Function twi_transmit
 * Desc     fills slave tx buffer with data
 *          must be called in slave tx event callback
 * Input    data: pointer to byte array
 *          length: number of bytes in array
 * Output   1 length too long for buffer
 *          2 not slave transmitter
 *          0 ok
 */
uint8_t twi_transmit(const uint8_t* data, uint8_t length)
{
  uint8_t i;

  // ensure data will fit into buffer
  if(TWI_BUFFER_LENGTH < (twi_txBufferLength+length)){
    return 1;
  }

  // ensure we are currently a slave transmitter
  if(TWI_STX != twi_state){
    return 2;
  }

  // set length and copy data into tx buffer
  for(i = 0; i < length; ++i){
    twi_txBuffer[twi_txBufferLength+i] = data[i];
  }
  twi_txBufferLength += length;

  return 0;
}
```

The read() function in the wire library is used to read a byte that was transmitted from a controller device to a peripheral device. This function inherits from the Stream utility class and has no parameters. It simply returns the next byte received from the communication.



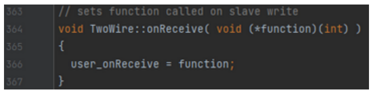
```
// must be called in:
// slave rx event callback
// or after requestFrom(address, numBytes)
int TwoWire::read(void)
{
  int value = -1;

  // get each successive byte on each call
  if(rxBufferIndex < rxBufferLength){
    value = rxBuffer[rxBufferIndex];
    ++rxBufferIndex;
  }

  return value;
}
```

Fig. 6. read() function definition

The onReceive() function in the wire library is used to register a function that will be called when a peripheral device receives a transmission from a controller device. The syntax for using this function is Wire.onReceive(handler), where the parameter 'handler' is the function that will be called when data is received. This function should take a single int parameter, which represents the number of bytes read from the controller device, and should return nothing. There is no return value for this function.

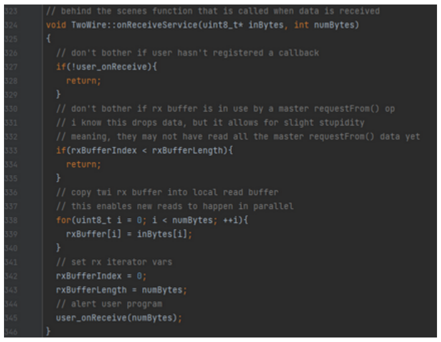
D. I2C Connection

For this project Arduino Uno R3 and Arduino Nano are used to perform the I2C connection. Arduino Nano being the controller device and Arduino Uno R3 being the peripheral



```
// sets function called on slave write
void TwoWire::onReceive( void (*function)(int) )
{
  user_onReceive = function;
}
```

Fig. 7. onReceive() function definition



```
// behind the scenes function that is called when data is received
void TwoWire::onReceiveService(uint8_t* inBytes, int numBytes)
{
  // don't bother if user hasn't registered a callback
  if(!user_onReceive){
    return;
  }
  // don't bother if rx buffer is in use by a master requestFrom() op
  // i know this drops data, but it allows for slight stupidity
  // meaning, they may not have read all the master requestFrom() data yet
  if(rxBufferIndex < rxBufferLength){
    return;
  }
  // copy twi rx buffer into local read buffer
  // this enables new reads to happen in parallel
  for(uint8_t i = 0; i < numBytes; ++i){
    rxBuffer[i] = inBytes[i];
  }
  // set rx iterator vars
  rxBufferIndex = 0;
  rxBufferLength = numBytes;
  // alert user program
  user_onReceive(numBytes);
}
```

Fig. 8. onReceiveService() function definition

device. For both of the devices the I2C pins are located in A4 (SDA) and A5 (SCL). Figure 9 shows the I2C connection of those two devices.

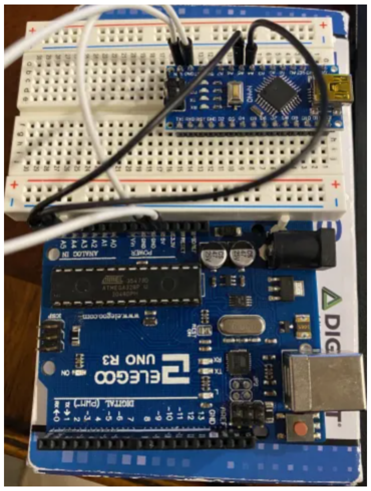
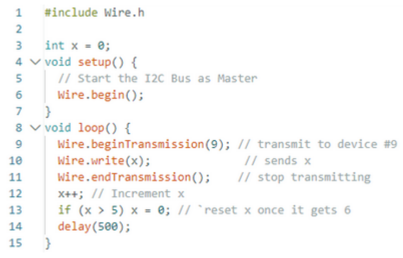


Fig. 9. Arduino Uno R3 and Arduino Nano I2C connection

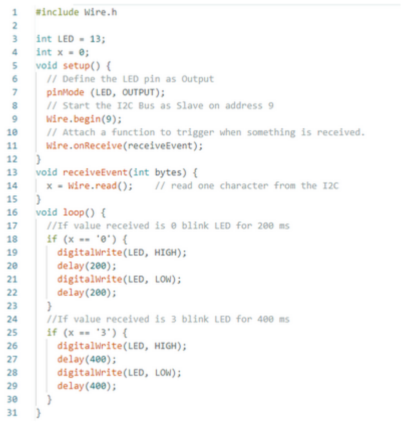


```
#include Wire.h

int x = 0;
void setup() {
  // Start the I2C Bus as Master
  Wire.begin();
}
void loop() {
  Wire.beginTransmission(9); // transmit to device #9
  Wire.write(x);              // sends x
  Wire.endTransmission();    // stop transmitting
  x++; // Increment x
  if (x > 5) x = 0; // `reset x once it gets 6
  delay(500);
}
```

Fig. 10. Arduino code for the controller device

As shown in the figure 10, the Arduino Nano is continuously sending an integer value from 0 to 5 with a delay of 500 ms.

Figure 11 shows that the Arduino Nano R3 receives the integer send from the Arduino Nano and blinks the LED for



```
#include Wire.h

int LED = 13;
int x = 0;
void setup() {
  // Define the LED pin as Output
  pinMode (LED, OUTPUT);
  // Start the I2C Bus as Slave on address 9
  Wire.begin(9);
  // Attach a function to trigger when something is received.
  Wire.onReceive(receiveEvent);
}
void receiveEvent(int bytes) {
  x = Wire.read();    // read one character from the I2C
}
void loop() {
  //If value received is 0 blink LED for 200 ms
  if (x == '0') {
    digitalWrite(LED, HIGH);
    delay(200);
    digitalWrite(LED, LOW);
    delay(200);
  }
  //If value received is 3 blink LED for 400 ms
  if (x == '3') {
    digitalWrite(LED, HIGH);
    delay(400);
    digitalWrite(LED, LOW);
    delay(400);
  }
}
```

Fig. 11. Arduino code for the peripheral device

200ms if the integer received is 0 and for 400 ms if the integer received is 3.

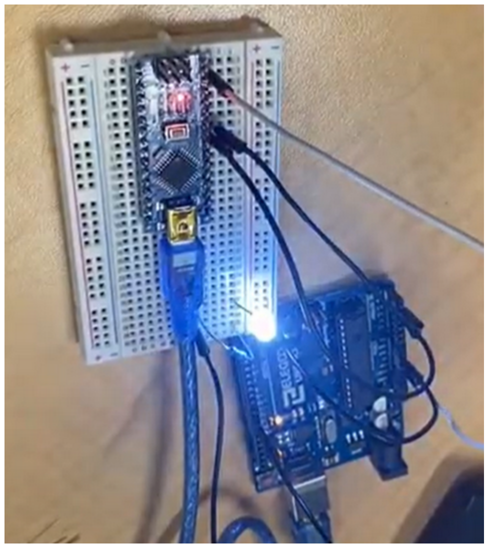


Fig. 12. LED controlled by the Arduino Uno R3

IV. Results

The demonstration project aimed at showcasing the implementation and functionality of the I2C protocol within the Arduino environment yielded substantial insights. Utilizing an Arduino Uno R3 as the peripheral device and an Arduino Nano as the controller, the performance of their I2C communication via the Wire library was meticulously evaluated. The controller device (Arduino Nano) was programmed to transmit a sequence of integer values ranging from 0 to 5, with a pause of 500 milliseconds between each transmission. This simulates a simplified control signal that might be used in a larger, more complex system to issue commands or trigger events. The code for the controller device was carefully crafted to ensure stable and repetitive transmission of these control integers, leveraging the Wire library's write() function to queue the data for transmission. In the counterpart setup, the peripheral device

(Arduino Uno R3) was designed to receive the incoming integers and perform a conditional response depending on the value. The key values of interest, 0 and 3, were set to trigger LED blinking on the Uno for 200 milliseconds and 400 milliseconds, respectively. Such conditional behavior is representative of a typical scenario where a device responds to discrete signals with varied outputs. Throughout the duration of the demonstration, the connection between Arduino Uno R3 and Arduino Nano exhibited high reliability. Data integrity was verified at each transition, with the LED blink responses corresponding accurately to the transmitted integer values. No transmission errors or missed integers were recorded, which underscores the stability of the I2C communication via the Wire library. A critical observation was the ease and correctness of the implementation of the onReceive() function within the Wire library. As data arrived at the peripheral device, the corresponding callback function was executed without fail, effectively handling the incoming information and initiating the appropriate response in the form of LED blinks. Through consistent observation and logging, the system demonstrated seamless interfacing between the controller and peripheral device. The Wire library's functions for data transmission (write) and reception (read), along with the registering of the receive event handler (onReceive), performed as documented without requiring additional troubleshooting or complex configuration. In conclusion, the results of this demonstration project emphatically validate the Arduino platform and the Wire library as robust tools for implementing I2C communication. The Wire library's functionality effectively abstracted the complexity of I2C transactions, enabling novice and expert users alike to deploy interactions between microcontrollers with minimal setup and reliable performance. These results affirm the Wire library's practicality in facilitating microcontroller communication and highlight the potential for Arduino-based projects to integrate multiple devices in a simple, effective manner.

V. Conclusion

The exploration of interfacing Arduino boards via I2C, as examined in this paper, established the Arduino platform and Wire library as potent resources for developing interconnected microcontroller systems. Within the electronics community, the utility and reliability of the Arduino ecosystem are often touted, and the results of the demonstration project lend further credence to these claims. By employing an Arduino Uno R3 and Arduino Nano to illustrate a straightforward yet illustrative application of I2C communication, we were able to demonstrate the seamless integration and functionality of these tools. Our analysis focused on the ease of implementing I2C using the Wire library, the successful transmission of data between a controller and peripheral device, and the accurate execution of conditional behaviors on the peripheral device. Throughout the project, the robustness of the I2C protocol within the Arduino environment was made evident—the designed system was not only stable but also maintained data integrity without fail across all tests.

The applicability of the Arduino and I2C protocol extends beyond the boundaries of hobbyist tinkering; it can serve as a foundation for educational purposes, enabling students to grasp complex electronic concepts through hands-on experience. Moreover, it paves the way for professionals to prototype rapidly and iterate designs without delving into the intricacies of lower-level embedded systems programming. The success of our demonstration project, characterized by a noticeable lack of complexity and high reliability, showcases the potential for Arduino-based I2C applications across a spectrum of uses, including but not limited to IoT devices, sensor networks, and robotic controls. Moving forward, the durability, affordability, and accessibility of Arduino, coupled with the flexibility and simplicity of the I2C protocol, set the stage for continued innovation and expansion in the realm of digital electronics. By furthering user education and creating more inclusive development tools, the Arduino platform stands as a beacon of open-source collaboration, continually inviting users to learn, create, and share their technical explorations with the global community.

In conclusion, the strategic alliance between Arduino's accessible programming model and I2C's efficient communication standard presents a powerful combination for electronics developers. The success of the demonstration project underscores the potential of this partnership and encourages the continued adoption and application of Arduino and I2C in both educational and professional settings. As such, these results contribute to reaffirming the Arduino platform's standing as an invaluable asset for innovators and creators around the world.

A. Future Work

Building on the successes and lessons learned from the current study, future work in this domain should consider several avenues for further exploration. At a technical level, the expansion of the Wire library to handle larger data buffers and higher-speed I2C communication could open doors to more complex and demanding applications, enhancing versatility for power users. Moreover, investigating mechanisms for error detection and correction could greatly increase robustness in noisy environments or at extended distances. Additionally, exploring multi-master I2C configurations within the Arduino ecosystem could lead to advancements in distributed control systems. On the educational and usability front, there is potential for developing more comprehensive and interactive tutorials, as well as user-friendly debugging tools within the Arduino IDE that cater specifically to common challenges encountered during I2C communication. Finally, there is much room for bridging the gap between Arduino prototyping and the production of commercial-grade electronics, especially in the context of IoT devices, by creating standardized modules and practices that can transition from prototype to product seamlessly. This future work will continue to solidify the role of Arduino as an integral part of the electronics and maker communities, promoting innovation and simplifying the sophisticated processes that drive the world of embedded systems.

References

[1] Arduino. (n.d.). Introduction to Arduino. Retrieved from https://www.arduino.cc/en/Guide/Introduction

[2] Device Interactions Blog. (2018, April). Behind the Scenes of Arduino IDE. Retrieved from https://blog.device-interactions.com/2018/04/behind-scenes-of-arduino-ide.html

[3] Robotics Backend. (n.d.). Arduino setup() and loop() Functions Explained. Retrieved from https://roboticsbackend.com/arduino-setup-loop-functions-explained/Arduino_void_setup

[4] Arduino Documentation. (n.d.). Wire Library. Retrieved from https://docs.arduino.cc/learn/communication/wirewire-library

[5] Arduino Reference. (n.d.). Wire Library Functions. Retrieved from https://www.arduino.cc/reference/en/language/functions/communication/wire/