HAMMING CODE IMPLEMENTATION

A

Project Report

Submitted in Partial Fulfillment of the Bangalore University

for the Degree

of

Bachelor of Technology

in

Electronics and Communication Engineering

by

1. **Meghana G N**
U03NM21T043041 VI SEM, B.Tech,

2. **Meghashyama N Aithal**
U03NM21T043042 VI SEM, B.Tech,

3. **Mohammed Daanish Hassain**
U03NM21T043043 VI SEM, B.Tech,

4. **Mohammed Hamza**
U03NM21T043044 VI SEM, B.Tech,

Under the supervision of

Dr. Harsha MV

Department of Electronics and Communication Engineering

Universtiy of Vivesvaraya College of Engineering

Contents

Acknowledgment

We sincerely express our gratitude to our guide, **Dr. Harsha MV**, for his invaluable guidance, support, and encouragement throughout this project. We would also extend our gratitude towards Our Department's Esteemed HoD **Dr. Kiran K**. We are thankful to the Department of Electronics and Communication Engineering, UVCE, for providing us with the resources and opportunity to work on this project. Special thanks to our peers and family for their continuous motivation and assistance. We would also like to acknowledge the support of the administrative staff, whose help ensured smooth progress of our work. Additionally, we are grateful to the authors and researchers whose work has significantly contributed to our understanding and implementation. Finally, we recognize the inspiration and insights gained from discussions and feedback provided by our friends and colleagues.

Introduction

The Hamming Code is an error-detecting and error-correcting code used in telecommunications, computer systems, and data storage. It was developed in the late 1940s by Richard W. Hamming, an American mathematician and computer scientist, while working at Bell Labs. Hamming was frustrated with the frequent errors in data transmission and storage, particularly in early computer systems, which often required manual error correction. To address this, he devised a mathematical method that could automatically detect and correct errors, leading to the development of the Hamming Code. [**?**]

1.1 Background Information

The Hamming Code is based on binary linear error-correcting codes and is particularly known for its ability to detect and correct single-bit errors in transmitted data. It uses redundant bits, known as parity bits, to create a code that can check for errors and identify which bit is incorrect if an error is detected. The original Hamming Code (referred to as (7,4) Hamming Code) uses seven bits, where four are data bits, and three are parity bits.

The code operates using a method called Hamming Distance, which is the number of bit positions in which two code words differ. For error detection and correction, the Hamming Code relies on a minimum Hamming Distance of three between any two code words, allowing it to detect up to two-bit errors and correct one-bit errors.

Key Principles

1. **Parity Bit Calculation**:
   Parity bits are added at specific positions in the code word. Each parity bit covers a certain set of data bits and ensures that the number of 1s in that subset is even (even parity) or odd (odd parity).
2. **Error Detection and Correction**:
   When a code word is received, parity checks are performed. If the parity doesn't match, it indicates an error, and the position of the error can be determined using a binary index derived from the parity checks.

1.2 Current Trends and Application

While technology has evolved significantly since the Hamming Code's invention, it remains a foundational concept in digital error correction. Its simplicity and efficiency in correcting single-bit errors make it popular in applications where minimal redundancy and simplicity are key.
Some modern applications and trends in Low-Power Embedded Systems and in Network Communications

Literature Survey

A 15-bit Hamming Code, such as (15,11) Hamming Code, was used to encode 11 data bits into 15 bits by adding four parity bits. This allows for single-bit error correction and double-bit error detection. Implementing the Hamming code encoding and correction between two communicating Micro controller.[**?**] [**?**]

2.1 Mathematical principles

Theorem 1 (Hamming code)**.** *Hamming codes are a class of binary linear code. For each integer*

$$r \geq 2 \tag{2.1}$$

there is a code-word with block length

$$[n = 2 * r - 1] \tag{2.2}$$

message length

$$[k = 2 * r - r - 1] \tag{2.3}$$

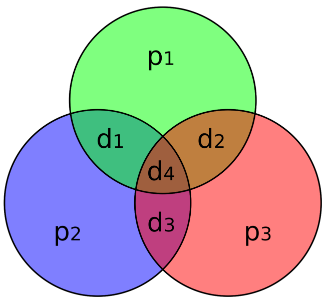


Figure 2.1: The Hamming(7,4) code (with r = 3)

Objective ,Methodology and Implementation

3.1 OBJECTIVE-I : Hamming Code Algorithm

3.1.1 Code Layout

Encode 8-bit data into a 12-bit code word.

Compute parity bits to ensure error detection and correction.

Include error detection and single-bit error correction for the receiving Arduino.

Bit positions: The 12-bit Hamming code is structured as follows:

Table 3.1: Hamming (12,8) Code Layout

Bit Position	12	11	10	9	8	7	6	5	4	3	2	1
Bit Type	D8	D7	D6	D5	P8	D4	D3	D2	P4	D1	P2	P1

D1 to D8 are the 8 data bits.

P1, P2, P4, and P8 are the 4 parity bits. [**?**]

3.1.2 Designing the Hamming Code

- Step 1: Determine the number of parity bits. Given a data bit sequence of length m, the number of parity bits r is determined such that:

$$2^r \geq m + r + 1 \tag{3.1}$$

- Step 2: Position the parity bits Place the parity bits at positions that are powers of 2 (i.e., positions 1, 2, 4, 8, etc.). The remaining positions are reserved for data bits.

- Step 3: Calculate parity bit values, The value of each parity bit is calculated based on specific bits in the data sequence. For each parity bit, cover bits are determined using binary representation. For example, parity bit P_1 covers bits at positions where the least significant bit of their binary representation is 1.

  ExampleConsider a 4-bit data sequence: `1011`.

  1. Calculate the number of parity bits needed:

  $$2^r \geq 4 + r + 1 \Longrightarrow r = 3 \tag{3.2}$$

  2. Insert parity bits at positions 1, 2, and 4: `P1, P2, 1, P4, 0, 1, 1`.

  3. Calculate values for each parity bit:

     – P_1 covers positions 1, 3, 5, 7.

  - P_2 covers positions 2, 3, 6, 7.
  - P_4 covers positions 4, 5, 6, 7.

A (12,8) Hamming code uses 8 data bits and 4 parity bits to form a 12-bit code word. The parity bits are strategically placed in positions that are powers of two (1, 2, 4, and 8), while the data bits are placed in the remaining positions.[**?**]

3.1.3 Types of Hamming Code Algorithms

Hamming code is a family of error-detecting and error-correcting codes that can correct single-bit errors and detect two-bit errors. Below are the common types of Hamming codes:

1. **(7,4) Hamming Code**

   The (7,4) Hamming Code is one of the simplest and most commonly used Hamming codes. It takes 4 data bits and adds 3 parity bits to form a 7-bit codeword. The extra bits enable error detection and correction. The algorithm for encoding and decoding is as follows:

2. **(15,11) Hamming Code**

   The (15,11) Hamming Code is an extended version that encodes 11 data bits into a 15-bit codeword by adding 4 parity bits. This increases error detection and correction capabilities.

3. **Generalized Hamming Code**

   The generalized Hamming code is a more flexible version of the Hamming code that can be used for different block sizes and error-correcting capabilities.

4. **Extended Hamming Code**

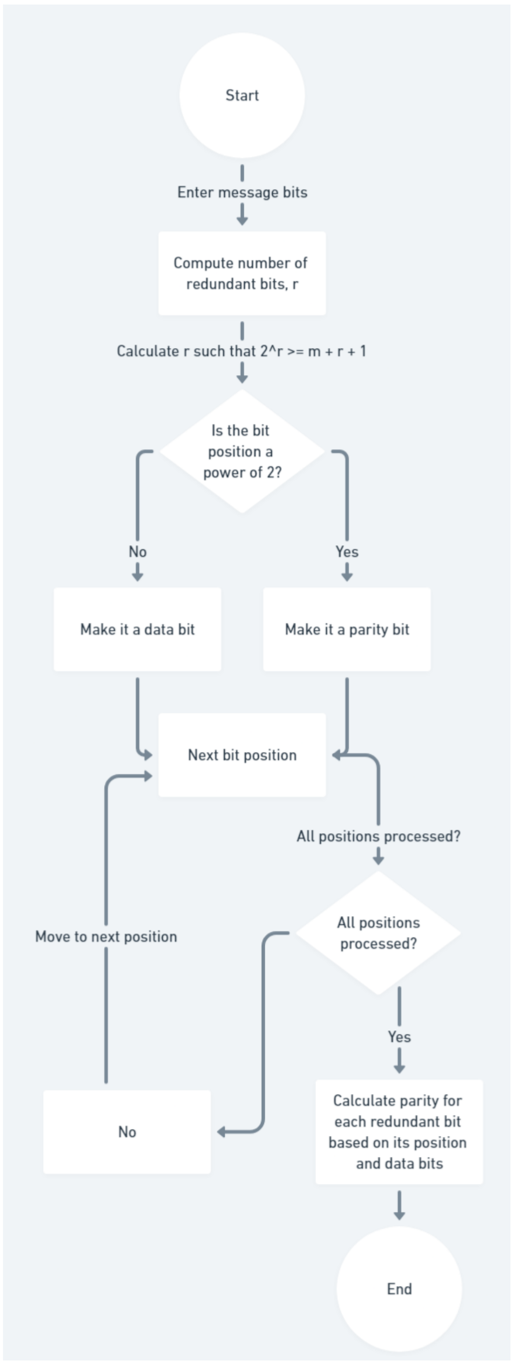
   Extended Hamming codes are derived by adding an extra parity bit to the standard Hamming codes, enabling detection of two-bit errors.

3.2 OBJECTIVE-II : Implementing on software

Function to Encode and Decode Hamming code

Below is a simple C++ function to implement a Hamming code encoding and decoding algorithm:[**?**]

Encoding Algorithm


Start
Enter message bits
Compute number of redundant bits, r
Calculate r such that 2^r >= m + r + 1
Is the bit position a power of 2?
No
Yes
Make it a data bit
Make it a parity bit
Next bit position
All positions processed?
Move to next position
All positions processed?
Yes
No
Calculate parity for each redundant bit based on its position and data bits
End

Decoding Algorithm

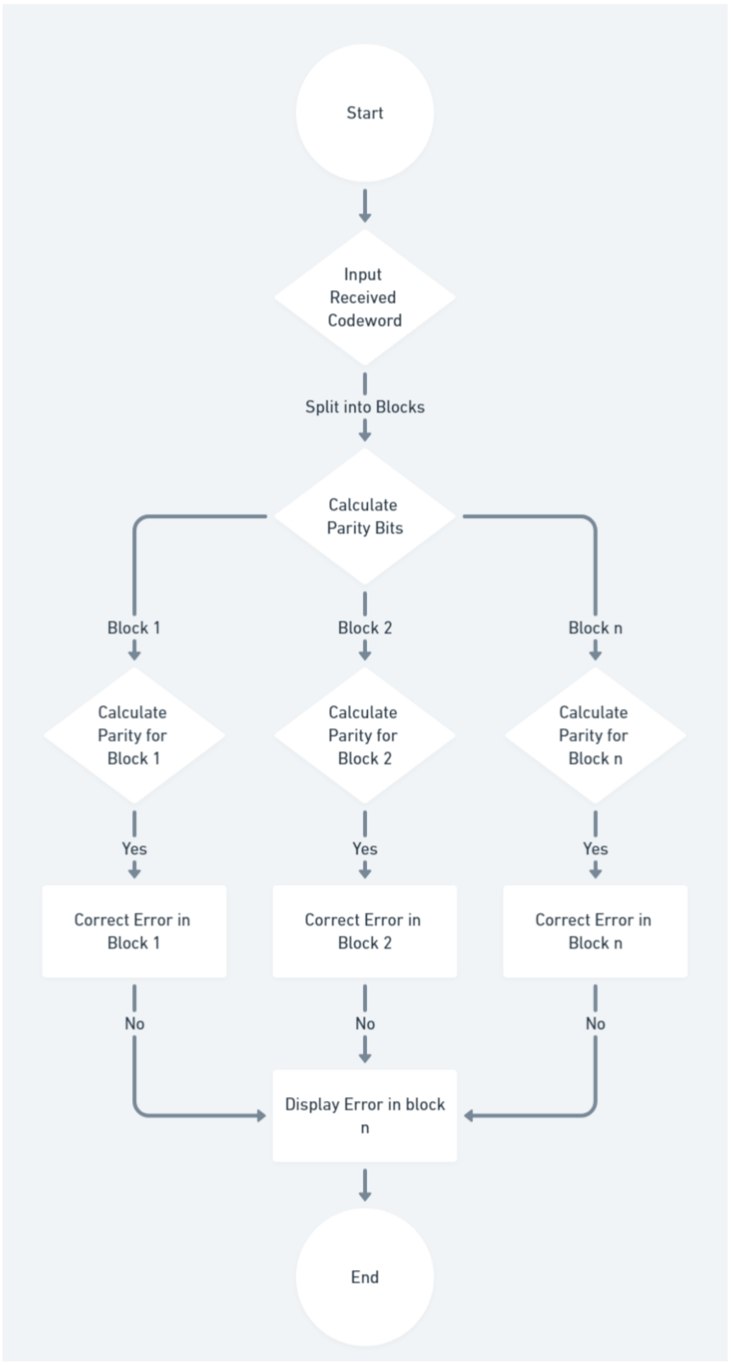


Figure 4.1(a): Encoding flow diagram & 4.1(b): Decoding flow diagram

3.3 OBJECTIVE-III : Implementing on Microprocessor

Implement the Hamming (12,8) code encoding and decoding on a microprocessor, such as an Arduino. This implementation will allow real-time error detection and correction in data transmitted between devices.

1. Setting Up the Microprocessor
   - Use an Arduino (or similar micro controller) to send and receive data.
   - Connect the Arduinos via Serial or I2C communication, which allows them to send and receive data reliably.
2. Programming the Encoder and Decoder
   - Write functions to encode 8-bit data into a 12-bit Hamming code.
   - Write functions to decode the 12-bit Hamming code back into 8-bit data and correct single-bit errors if necessary.
3. Integrating the Hamming Encoder and Decoder Code
   - Import the encoder and decoder header files into your Arduino sketch.
   - Designate one Arduino as the **transmitter** (encoding and sending data) and the other as the **receiver** (decoding and error-checking the received data).

Tools for Hamming Code Implementation on an Arduino

Table 3.2: Tools for Hamming code Implementation

Tool	**Description**
Arduino Board	The microcontroller board ie, Arduino nano using ATmega32P used for coding and testing the Hamming Code implementation.
Arduino IDE	Software used to write, compile, and upload code to the Arduino board. It provides an environment for developing Hamming code logic in C/C++.
Breadboard	Used for building and testing the circuit without soldering, connecting the Arduino pins with peripherals like LEDs and switches.
Jumper Wires	Used to make connections between the Arduino, breadboard, and other components in the circuit.
LEDs	Can be used as output indicators for testing Hamming code output such as error detection/-correction results.
Resistors	Typically used with LEDs to limit current and protect components during testing and debug-ging.
Oscilloscope	A device used to observe and analyze the electrical signals in the circuit, which is helpful for debugging timing and signal integrity issues.
PS/2 Keyboard	A peripheral device that can be used for input testing and interfacing with the Arduino to send data streams that may require Hamming code error checking.

3.4 Implementation Steps

1. **Software Implementation**

   Writing a program to encode data into Hamming code and decode the same. This involves:

   - Implementing algorithms for data encoding, parity bit generation, and error detection.
   - Introducing controlled random bit flips to simulate data corruption during transmission.
   - Decoding the corrupted data and applying error correction techniques to restore the original message.
   - Testing with various message lengths and error scenarios to validate robustness.

   The software implementation ensures that the Hamming code performs well under simulated transmission errors, making the data error-free.

2. **Implementation Between Two Communicating Arduinos**

   Establishing a reliable communication link between two Arduino microcontrollers involves:

   - Configuring one Arduino as a transmitter and the other as a receiver.
   - Using serial communication protocols (e.g., UART or I2C) to transmit data.
   - Encoding data at the transmitter using Hamming code and decoding it at the receiver to detect and correct errors.
   - Simulating environmental noise or interference to induce random errors during transmission.
   - Validating data integrity at the receiver and displaying results on an LCD or serial monitor.

   This implementation demonstrates practical error correction in embedded systems.

3. **Implementation on a Microprocessor**

   Building a transmitter and a receiver on a microprocessor platform includes:

   - Developing separate routines for encoding data into Hamming code at the transmitter and decoding it at the receiver.
   - Interfacing input devices (e.g., keyboards or sensors) to provide data to the transmitter.
   - Connecting output devices (e.g., LCDs or LEDs) to display decoded data and error correction results.
   - Optimizing the microprocessor code to handle large data sets and ensure minimal latency in encoding/decoding processes.
   - Testing performance under different operating conditions, such as high data rates or noisy environments.

   The microprocessor-based implementation showcases the adaptability of Hamming code for real-world applications.

3.5 Methodology

3.5.1 Thought Train Mapping

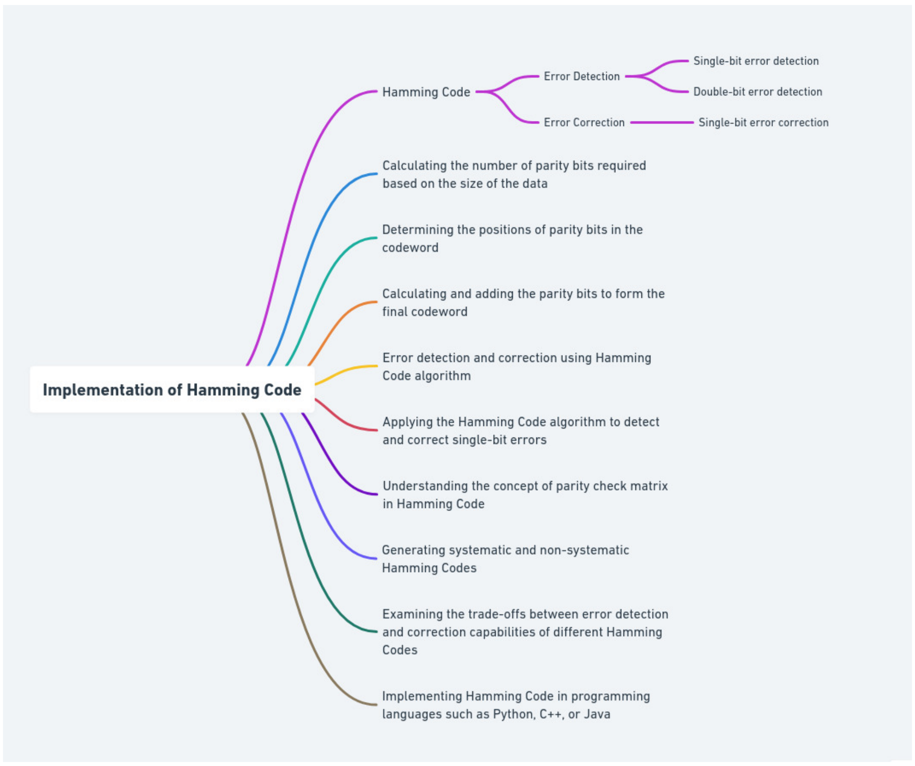

Implementation of Hamming Code
Hamming Code
Error Detection
Single-bit error detection
Double-bit error detection
Error Correction
Single-bit error correction
Calculating the number of parity bits required based on the size of the data
Determining the positions of parity bits in the codeword
Calculating and adding the parity bits to form the final codeword
Error detection and correction using Hamming Code algorithm
Applying the Hamming Code algorithm to detect and correct single-bit errors
Understanding the concept of parity check matrix in Hamming Code
Generating systematic and non-systematic Hamming Codes
Examining the trade-offs between error detection and correction capabilities of different Hamming Codes
Implementing Hamming Code in programming languages such as Python, C++, or Java

Figure 3.1: Methodology Mindmap

3.5.2 System design Mapping

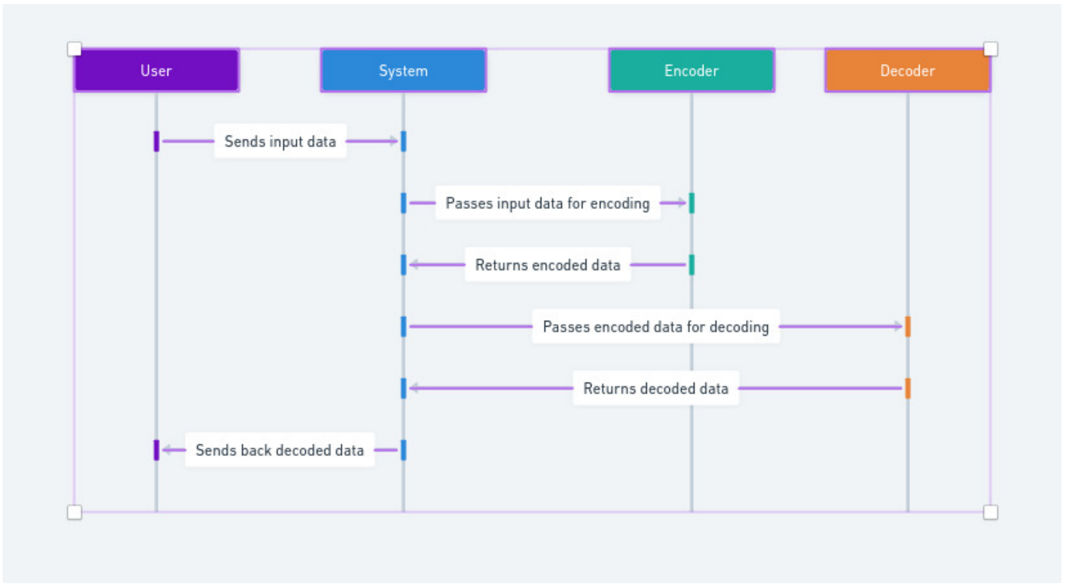


Figure 3.2: Algorithim Sequence flow

3.5.3 Technologies Used

3.5.4 Overview of PS 2

The PS/2 keyboard is a type of input device that connects to a computer through the PS/2 port, a standard interface developed by IBM in 1987. This keyboard and connector were widely used in earlier computer systems but have been largely replaced by USB interfaces.[**?**]

PS/2 Connector

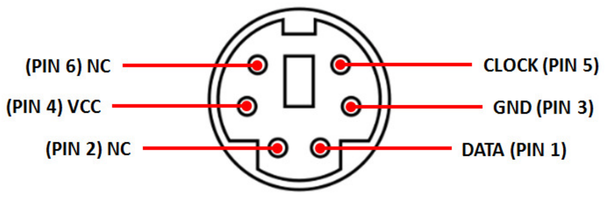
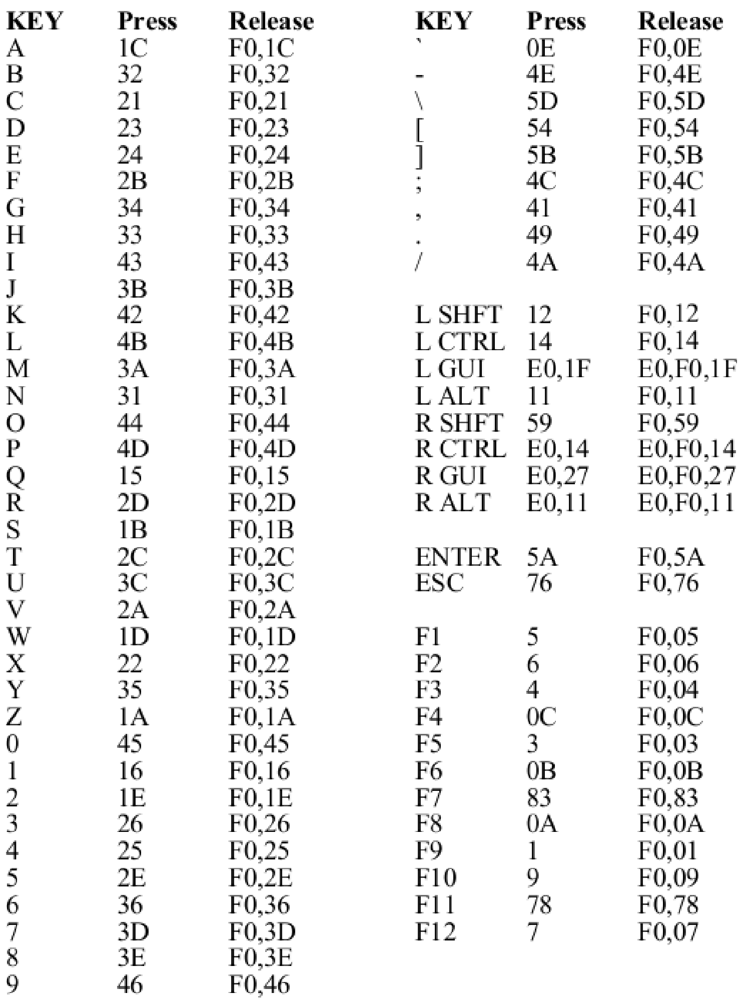


Figure 3.3: PS/2 connector Pin diagram

The PS/2 connector is a 6-pin mini-DIN connector with the following characteristics:

- **Round Design**: The connector has a circular shape with six metal pins and a keyway for orientation.
- **Pin Configuration**: Each pin has a specific function:
  - **Pin 1**: Data
  - **Pin 2**: Reserved
  - **Pin 3**: Ground
  - **Pin 4**: +5V (Power)
  - **Pin 5**: Clock
  - **Pin 6**: Reserved
- **Data Transmission**: Communication occurs synchronously, with the keyboard generating clock signals for data transmission.

Table for keyboard scan codes



KEY	Press	Release	KEY	Press	Release
A	1C	F0,1C	`	0E	F0,0E
B	32	F0,32	-	4E	F0,4E
C	21	F0,21	\	5D	F0,5D
D	23	F0,23	[	54	F0,54
E	24	F0,24	]	5B	F0,5B
F	2B	F0,2B	;	4C	F0,4C
G	34	F0,34	,	41	F0,41
H	33	F0,33	.	49	F0,49
I	43	F0,43	/	4A	F0,4A
J	3B	F0,3B			
K	42	F0,42	L SHFT	12	F0,12
L	4B	F0,4B	L CTRL	14	F0,14
M	3A	F0,3A	L GUI	E0,1F	E0,F0,1F
N	31	F0,31	L ALT	11	F0,11
O	44	F0,44	R SHFT	59	F0,59
P	4D	F0,4D	R CTRL	E0,14	E0,F0,14
Q	15	F0,15	R GUI	E0,27	E0,F0,27
R	2D	F0,2D	R ALT	E0,11	E0,F0,11
S	1B	F0,1B			
T	2C	F0,2C	ENTER	5A	F0,5A
U	3C	F0,3C	ESC	76	F0,76
V	2A	F0,2A			
W	1D	F0,1D	F1	5	F0,05
X	22	F0,22	F2	6	F0,06
Y	35	F0,35	F3	4	F0,04
Z	1A	F0,1A	F4	0C	F0,0C
0	45	F0,45	F5	3	F0,03
1	16	F0,16	F6	0B	F0,0B
2	1E	F0,1E	F7	83	F0,83
3	26	F0,26	F8	0A	F0,0A
4	25	F0,25	F9	1	F0,01
5	2E	F0,2E	F10	9	F0,09
6	36	F0,36	F11	78	F0,78
7	3D	F0,3D	F12	7	F0,07
8	3E	F0,3E			
9	46	F0,46			

Figure 3.4: Scan Codes

Advantages and Disadvantages **Advantages**

- Supports lower latency compared to USB keyboards.
- Allows “n-key rollover,” enabling simultaneous key presses without ghosting.

Disadvantages

- Does not support hot-plugging. Connecting or disconnecting a device while the system is powered on may cause issues.
- The interface is becoming obsolete as USB has become the new standard.

3.5.5 Overview of interrupts in Arduino

Interrupts in Arduino allow the microcontroller to stop its current task and execute a specific block of code when a certain event occurs. This is particularly useful for time-sensitive tasks or when dealing with asynchronous inputs like sensors or user inputs.

Interrupts in Arduino allow the microcontroller to temporarily halt its current task and execute a predefined block of code in response to specific events or conditions. When an interrupt occurs, the microcontroller's normal operation is paused, and it jumps to a special function called an interrupt service routine (ISR) to handle the event. After the ISR completes, the microcontroller returns to its previous task. This mechanism is crucial for handling time-sensitive tasks or responding to asynchronous events in real-time without continuously checking for them in the main program loop.

Types of Interrupts in Arduino Arduino supports the following types of interrupts:

- **External Interrupts:** Triggered on specific pins (e.g., `INT0`, `INT1`) on most Arduino boards.
- **Pin Change Interrupts:** Available on all pins, but their implementation is less direct.

Commonly Used Interrupt Modes

- `LOW`: Triggered when the pin is low.
- `CHANGE`: Triggered when the pin changes state.
- `RISING`: Triggered when the pin transitions from LOW to HIGH.
- `FALLING`: Triggered when the pin transitions from HIGH to LOW.

Syntax for Using Interrupts The function `attachInterrupt()` is used to configure and enable interrupts on Arduino.

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);
```

Listing 1: Syntax for attachInterrupt

`pin` : The pin number to attach the interrupt to.

`ISR` : The Interrupt Service Routine, a function that runs when the interrupt is triggered.

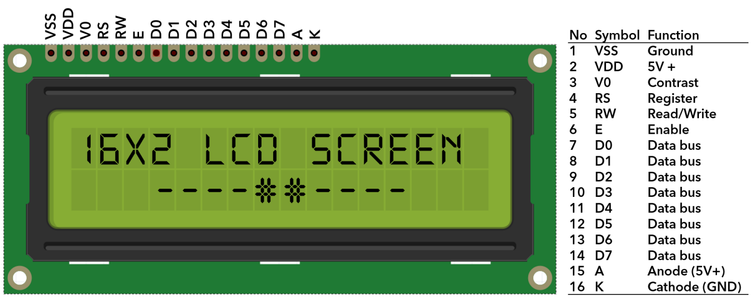
`mode` : The condition that triggers the interrupt (e.g., `RISING`, `FALLING`, `CHANGE`, `LOW`).

Important Notes

- The **Interrupt Service Routine (ISR)** should be kept as short and efficient as possible to avoid delays in other tasks.
- Functions like `delay()`, `Serial.print()`, or other time-consuming operations are not recommended inside an ISR.
- Use the `volatile` keyword for variables shared between the ISR and the main program to prevent compiler optimizations.

3.5.6 Overview of LCD

Liquid Crystal Displays (LCDs) are flat-panel displays that utilize the light-modulating properties of liquid crystals combined with a backlight or reflector to produce images. They are widely used in devices such as televisions, computer monitors, mobile phones, and more.[**?**]



No	Symbol	Function
1	VSS	Ground
2	VDD	5V +
3	V0	Contrast
4	RS	Register
5	RW	Read/Write
6	E	Enable
7	D0	Data bus
8	D1	Data bus
9	D2	Data bus
10	D3	Data bus
11	D4	Data bus
12	D5	Data bus
13	D6	Data bus
14	D7	Data bus
15	A	Anode (5V+)
16	K	Cathode (GND)

Figure 3.5: Scan Codes

Working Principle The operation of an LCD is based on the ability of liquid crystals to control the passage of light. Key components include:

- **Liquid Crystal Material:** Changes its optical properties when an electric field is applied.
- **Polarizing Filters:** Ensure that only light aligned in certain directions passes through.
- **Electrodes:** Create the electric field to manipulate liquid crystals.

The combination of these components allows LCDs to display images by selectively blocking or allowing light to pass through.

Types of LCDs There are various types of LCDs, categorized based on their technology and application:

- **Twisted Nematic (TN):** Known for fast response times and low cost.
- **In-Plane Switching (IPS):** Offers better color reproduction and viewing angles.
- **Vertical Alignment (VA):** Provides high contrast ratios.

Advantages

- Low power consumption compared to older display technologies.
- Lightweight and slim design.
- High resolution and image quality.

Applications LCDs are used in a wide range of devices, including:

- Smartphones
- Laptops
- Televisions
- Digital clocks
- Instrument panels

Challenges and Limitations While LCDs are popular, they also have certain drawbacks:

- Limited viewing angles (improved in IPS technology).
- Dependency on backlight reduces contrast compared to OLEDs.
- Potential for motion blur in fast-moving images.

3.6 USART Overview

The Universal Asynchronous Receiver-Transmitter (USART) protocol is one of the simplest and most widely used methods for serial data communication. USART operates asynchronously, meaning it does not require a clock signal for data transmission. Instead, it relies on a pre-defined baud rate (speed of transmission) and synchronization between the transmitter and receiver.

Although USART is often considered an older communication method, it continues to play a crucial role in modern electronics. Its simple implementation, low cost, and widespread support make it suitable for a wide range of applications, from microcontroller-based systems to peripheral communication in computers and industrial systems.

In this document, we explore the various aspects of the USART protocol, from its basic functioning to its use in modern devices. We will cover its data format, the components involved, and the communication process in detail.[**?**]

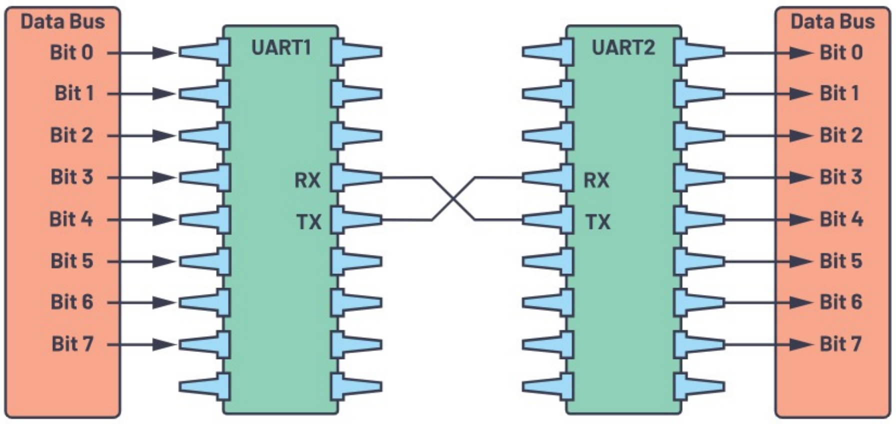


Figure 3.6: USART diagram

Basic Operation of USART At the core of USART is the process of transmitting and receiving data bit-by-bit through a single communication line. Data is sent asynchronously, meaning there is no need for a clock signal to synchronize the data transmission. Instead, both the transmitter and receiver must operate at the same baud rate.

A basic USART communication cycle involves the following stages:

1. **Start Bit:** The start bit marks the beginning of data transmission. It is a low logic level (0) and is sent before the actual data bits to alert the receiver that new data is coming. The start bit ensures that the receiver is ready to receive the incoming data.
2. **Data Bits:** After the start bit, the data bits follow. The number of data bits is usually 8, though USART supports configurations with 5 to 9 bits per character. These bits represent the actual data being transmitted. The data bits are sent one by one in sequence from the least significant bit (LSB) to the most significant bit (MSB).
3. **Parity Bit (Optional):** The parity bit is an optional error-checking mechanism. It can be configured to be even, odd, or none. When used, the parity bit is calculated based on the number of 1s in the data bits. In even parity, the number of 1s in the data bits, plus the parity bit, must be even, while in odd parity, the total number of 1s must be odd.
4. **Stop Bits:** Stop bits signal the end of the data transmission. These bits are set to a high logic level (1) and ensure that the receiver knows when the transmission of one byte is complete. One or two stop bits are typically used in most configurations.

Baud Rate: The baud rate determines the speed at which data is transmitted and received. It is typically expressed in bits per second (bps). Common baud rates include 9600, 19200, 38400, and 115200 bps. Both the transmitter and receiver must operate at the same baud rate for successful communication.

USART Data Format The USART protocol defines a standard format for sending and receiving data. As mentioned earlier, the basic USART data format consists of the start bit, data bits, an optional parity bit, and stop bits. Below is a breakdown of the typical USART frame structure:

- **Start Bit:** A single low bit (0) indicating the beginning of data transmission.
- **Data Bits:** The data being transmitted. This usually consists of 8 bits, but can range from 5 to 9 bits.
- **Parity Bit (optional):** An optional bit used for error checking. The parity bit helps detect transmission errors by ensuring that the number of 1s in the data is either even or odd, depending on the configuration.
- **Stop Bits:** One or more bits (usually 1 or 2) set to a high logic level (1) to indicate the end of the transmission. These bits give the receiver enough time to process the data and get ready for the next transmission.

The configuration of these parameters (data bits, parity, and stop bits) can be customized depending on the requirements of the communication system. For instance, the number of data bits can be adjusted to fit different types of data, and parity can be enabled to ensure error detection.

Error Detection in USART While USART is a simple protocol, it does provide basic mechanisms for error detection. These mechanisms help ensure that data is transmitted correctly and that errors can be identified and corrected.

Parity Bit The parity bit is the most basic form of error detection in USART. It is used to verify whether the transmitted data has been received correctly. There are two main types of parity:

- **Even Parity:** The parity bit ensures that the total number of 1s in the data (including the parity bit) is even.
- **Odd Parity:** The parity bit ensures that the total number of 1s in the data (including the parity bit) is odd.

If the number of 1s in the data does not match the expected parity (even or odd), an error is detected, and the data is flagged as corrupt.

Stop Bit Monitoring Another basic form of error detection is the monitoring of stop bits. If the stop bit is not at a high logic level (1), the receiver may interpret this as an error and discard the data.

Flow Control in USART Flow control is used to manage the rate at which data is transmitted to ensure that the receiver can process the incoming data without losing any information. USART supports two types of flow control:

Hardware Flow Control (RTS/CTS) Hardware flow control uses additional wires, typically *Request to Send* (RTS) and *Clear to Send* (CTS), to signal when the receiver or transmitter is ready to transmit or receive data. This method ensures that data is only transmitted when the receiver is ready, preventing data loss.

Software Flow Control (XON/XOFF) Software flow control uses control characters (XON and XOFF) to manage data transmission. When the receiver's buffer is full, it sends the XOFF character to instruct the transmitter to pause transmission. Once the receiver is ready, it sends the XON character to resume data flow.

Applications of USART USART is a versatile protocol used in a variety of applications. Some common uses include:

- **Microcontrollers:** USART is widely used for communication between microcontrollers and external peripherals like sensors, displays, and actuators.
- **Computer Peripherals:** USART is used in devices like modems, GPS receivers, Bluetooth modules, and other communication peripherals.
- **Embedded Systems:** USART is a standard communication protocol in embedded systems for serial communication with external devices.
- **Networking:** Although it is not used for high-speed networking, USART can be used for lower-speed data communication, such as in RS-232 and RS-485 networks.

Advantages of USART The USART protocol offers several advantages:

- **Simplicity:** USART is a simple protocol that requires minimal hardware and software support.
- **Low Cost:** The protocol's minimal hardware requirements make it cost-effective for many applications.
- **Asynchronous Communication:** USART does not require a clock signal, simplifying system design and reducing the number of required pins.
- **Wide Adoption:** USART is widely supported by microcontrollers and other embedded devices, making it easy to integrate into a variety of systems.

Results & Discussion

Results of the above objective were sequentially achieved and Realized.

4.1 Result of Objective-I

Implementation of the Hamming code encoder in C++ steps:

Algorithim for Hamming code Encoder

1. **Input Data Bits**: Take k data bits from the user.
2. **Determine Number of Parity Bits**: Calculate r such that $2^r \geq k + r + 1$.
3. **Position Data and Parity Bits**:
    - Create a combined array of size $n = k + r$.
    - Place parity bits at positions that are powers of 2 $(1, 2, 4, 8, \ldots)$.
    - Place data bits in the remaining positions.
4. **Calculate Parity Bits**:
    - For each parity bit position 2^{i-1}, compute parity using XOR of all covered positions.
5. **Output Hamming Code**: Print the combined array as the encoded Hamming code.

Algorithim for Hamming code Encoder

1. **Input the Received Code**: Take the n-bit received Hamming code from the user.
2. **Determine the Number of Parity Bits**:
    - Calculate r (number of parity bits) such that $2^r \geq n + 1$.
3. **Check Parity Bits**:
    - Compute the parity at each 2^{i-1} position by XORing all bits covered by that parity bit.
    - Store the results in a binary number called the *syndrome*.
4. **Detect and Correct Errors**:
    - If the syndrome is non-zero, it indicates an error at the position corresponding to the binary value of the syndrome.

- Flip the bit at the error position to correct the code.

5. **Extract the Data Bits**:

    - Remove all parity bit positions (2^{i-1}) from the corrected code to retrieve the original data bits.

4.2 Results of Objective-II

The implementation of the Hamming Code Encoder and Decoder functions has been successfully completed. The system is designed using two microprocessors, where one operates as the transmitter and the other as the receiver. To ensure efficient and reliable communication, an I2C (Inter-Integrated Circuit) connection is established between the two microprocessors, facilitating seamless data transfer.

The encoder function on the transmitter side is responsible for generating Hamming codes. It takes the input data bits and calculates the necessary parity bits based on the Hamming code algorithm. These parity bits are then appended to the original data bits to form a complete Hamming code. This process ensures that the transmitted data can support error detection and correction, making the communication robust against single-bit errors.

On the receiver side, the decoder function verifies the integrity of the received Hamming code. It checks the parity bits to identify any discrepancies that may have occurred during transmission. In the event of a single-bit error, the decoder determines the exact location of the error using the syndrome and corrects it, ensuring data accuracy. Finally, the decoder extracts the original data bits from the corrected Hamming code, delivering the intended information as transmitted.

This implementation demonstrates the effectiveness of Hamming codes in achieving error-resilient data transmission, especially in environments where data integrity is critical. By leveraging the capabilities of two microprocessors and the I2C protocol, the system showcases an efficient and practical solution for reliable communication.

The implementation of the Hamming Code Encoder and Decoder functions has been successfully completed, resulting in a robust error-correction system based on the Hamming code algorithm. This system involves two microprocessors: one functioning as the transmitter and the other as the receiver. These two microprocessors are interconnected via an I2C (Inter-Integrated Circuit) communication protocol, which is a widely-used, efficient, and low-power data transfer method that enables reliable communication between the transmitter and receiver. The I2C connection facilitates the smooth and synchronous transmission of data, ensuring that the encoded data is correctly sent and received between the two microprocessors.

On the transmitter side, the microprocessor is responsible for implementing the encoder function. The encoder takes the input data bits, typically a series of binary values representing the information to be transmitted. Using the Hamming code algorithm, the encoder calculates the required parity bits, which are additional bits added to the data to detect and correct errors that might occur during transmission. The calculation of the parity bits follows the Hamming code's systematic procedure, where the parity bits are inserted at specific positions in the data sequence, typically at powers of 2. These parity bits are generated by performing XOR operations on selected data bits, ensuring that the transmitted Hamming code will have the properties required for error correction. Once the parity bits are calculated, they are appended to the original data, forming the complete Hamming code that will be transmitted over the communication link. This process ensures that the transmitted data can be validated for integrity and corrected if necessary, making the system resilient to single-bit errors that might occur during transmission.

On the receiver side, the microprocessor implements the decoder function, which plays a crucial role in verifying the received Hamming code's integrity. Upon receiving the transmitted data through the I2C connection, the decoder first checks the parity bits to assess whether the received data has been corrupted. The decoder performs the same XOR operations as in the encoding process to check the consistency of the parity bits. By comparing the received parity bits with the expected values based on the data, the decoder generates a syndrome—a binary value that indicates whether an error has occurred, and if so, at which position in the data.

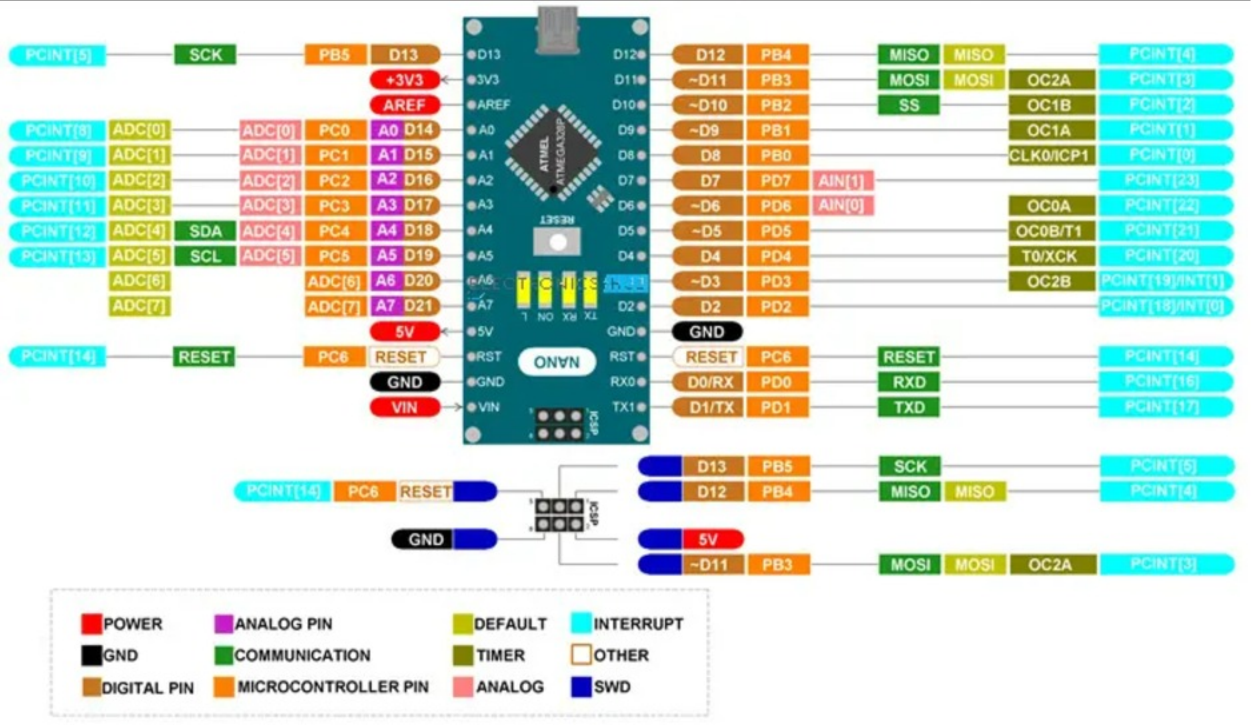

PCINT[5]
SCK
PB5
D13
+3V3
AREF
PCINT[8] ADC[0] ADC[0] PC0 A0 D14
PCINT[9] ADC[1] ADC[1] PC1 A1 D15
PCINT[10] ADC[2] ADC[2] PC2 A2 D16
PCINT[11] ADC[3] ADC[3] PC3 A3 D17
PCINT[12] ADC[4] SDA ADC[4] PC4 A4 D18
PCINT[13] ADC[5] SCL ADC[5] PC5 A5 D19
ADC[6] ADC[6] A6 D20
ADC[7] ADC[7] A7 D21
5V
PCINT[14] RESET PC6 RESET
GND
VIN
D12 PB4 MISO MISO PCINT[4]
~D11 PB3 MOSI MOSI OC2A PCINT[3]
~D10 PB2 SS OC1B PCINT[2]
~D9 PB1 OC1A PCINT[1]
D8 PB0 CLKO/ICP1 PCINT[0]
D7 PD7 AIN[1] PCINT[23]
~D6 PD6 AIN[0] OC0A PCINT[22]
~D5 PD5 OC0B/T1 PCINT[21]
D4 PD4 T0/XCK PCINT[20]
~D3 PD3 OC2B PCINT[19]/INT[1]
D2 PD2 PCINT[18]/INT[0]
GND
RESET PC6 RESET PCINT[14]
D0/RX PD0 RXD PCINT[16]
D1/TX PD1 TXD PCINT[17]
ATMEL ATMEGA328P
NANO
ICSP
D13 PB5 SCK PCINT[5]
D12 PB4 MISO MISO PCINT[4]
PCINT[14] PC6 RESET
GND
5V
~D11 PB3 MOSI MOSI OC2A PCINT[3]
POWER
GND
DIGITAL PIN
ANALOG PIN
COMMUNICATION
MICROCONTROLLER PIN
DEFAULT
TIMER
ANALOG
INTERRUPT
OTHER
SWD

Figure 4.1: Arduino Nano connector Pin diagram

4.3 Results of Objective-III

Circuit Diagram of Hardware

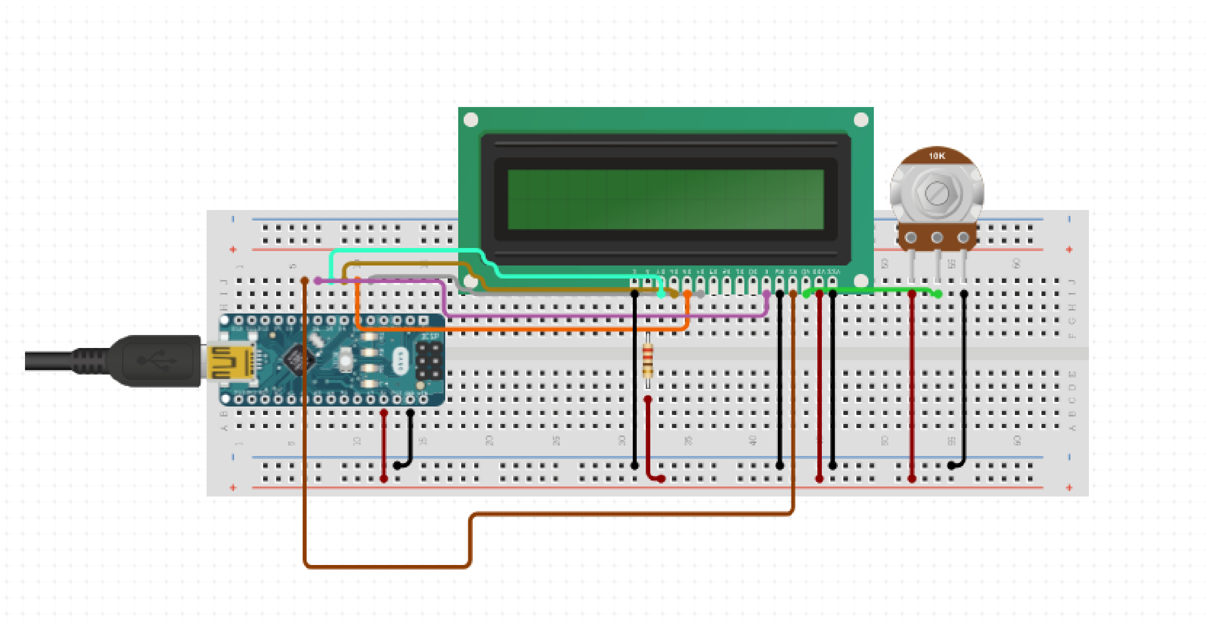


Figure 4.2: Circuit of Tx

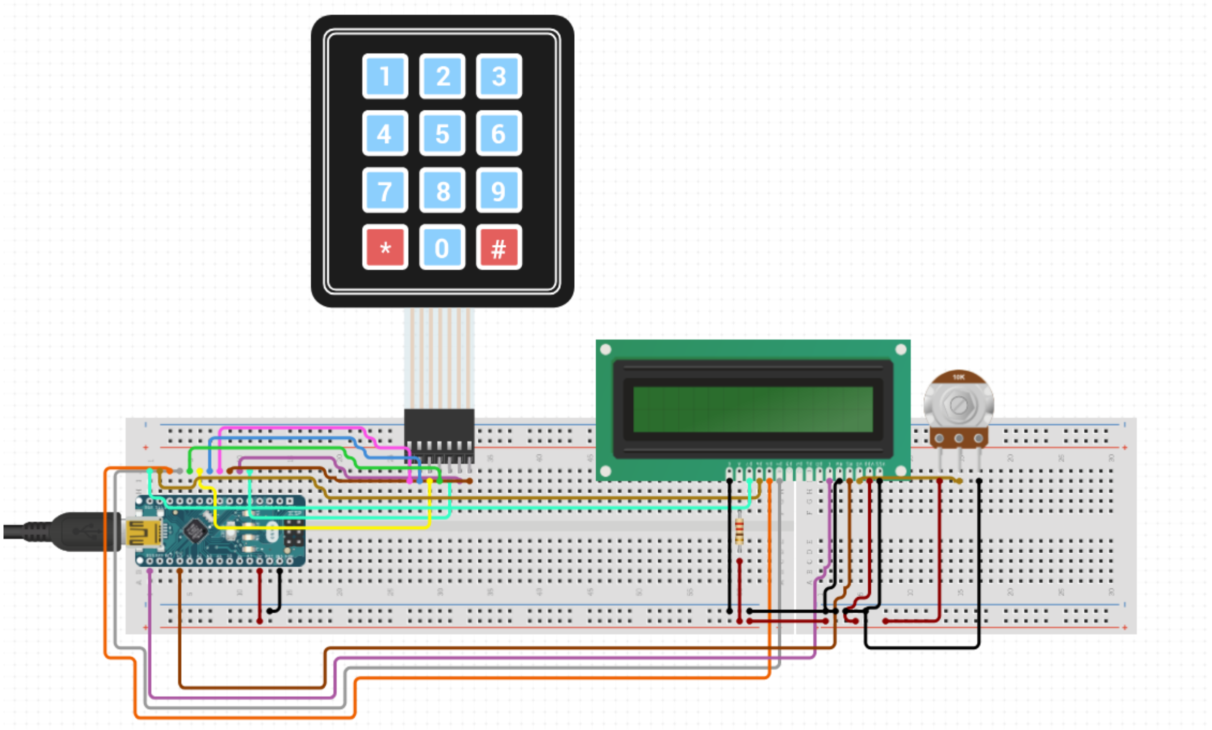


Figure 4.3: circuit of transmitter

Interfacing PS-2 keyboard with the Transmitter Program to interpret the PS-2 Input

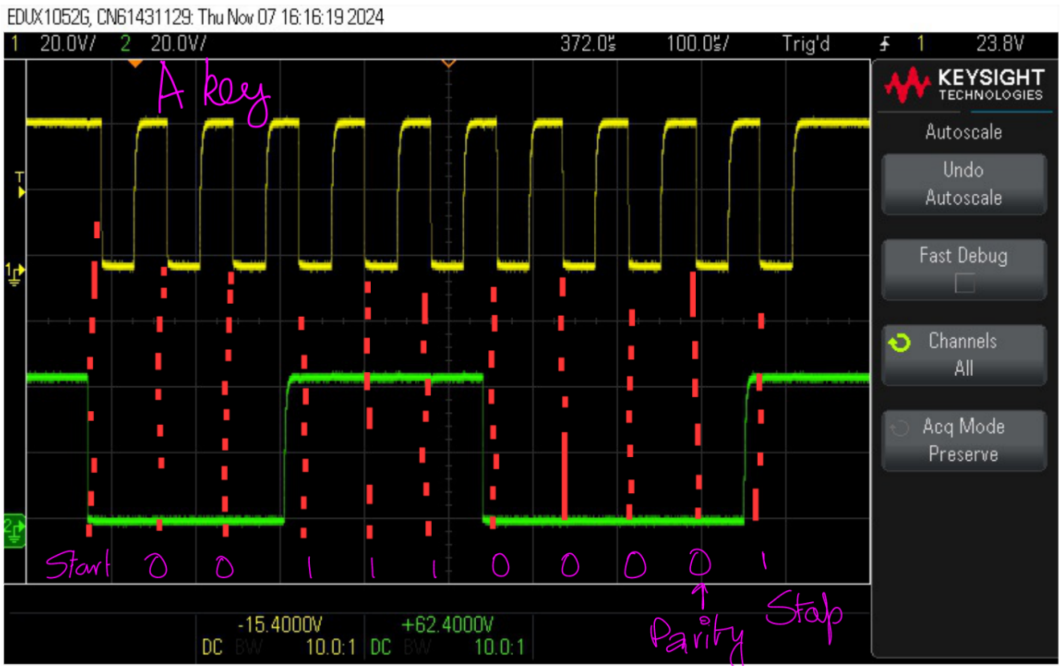


Figure 4.4: PS-2 interface input wave

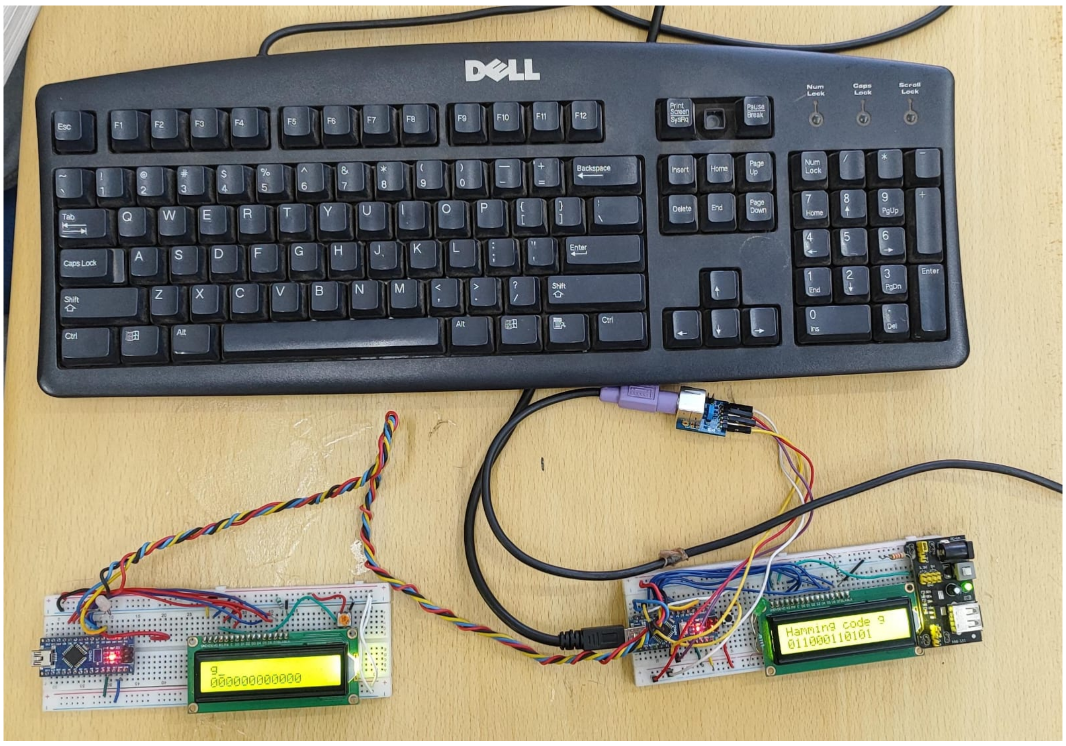


Figure 4.5: Project image

4.4 Observation

Table 4.1: Observation

Character	G
Bit Type	D8,D7,D6,D5,P8,D4,D3,D2,P4,D1,P2,P1
Binary	01000111
Hamming encoded	0100110111

Table 4.2: Observation

Character	H
Bit Type	D8,D7,D6,D5,P8,D4,D3,D2,P4,D1,P2,P1
Binary	01001000
Hamming encoded	0100110111

Table 4.3: Observation

Character	M
Bit Type	D8,D7,D6,D5,P8,D4,D3,D2,P4,D1,P2,P1
Binary	01001101
Hamming encoded	0100110111

Table 4.4: Observation

Character	A
Bit Type	D8,D7,D6,D5,P8,D4,D3,D2,P4,D1,P2,P1
Binary	01000001
Hamming encoded	100010010001

Table 4.5: Observation

Character	B
Bit Type	D8,D7,D6,D5,P8,D4,D3,D2,P4,D1,P2,P1
Binary	01000010
Hamming encoded	010110010010

Table 4.6: Observation

Character	C
Bit Type	D8,D7,D6,D5,P8,D4,D3,D2,P4,D1,P2,P1
Binary	01000011
Hamming encoded	010010000011

4.5 Test Scenarios

Hamming code provides 1 bit Error Correction and 2 bits of Error Detection. The Test Scenarios are given below

4.5.1 Single-bit Error Detection and Correction

Purpose: Verify that the code detects and corrects single-bit errors.

Test Case: Inject a single-bit error at various positions in the encoded message.

Example: For a 7-bit Hamming Code (`0111000`), flip one bit (e.g., `1111000`) and check if the original message is recovered.

4.5.2 Double-bit Error Detection

Purpose: Ensure the code identifies that it cannot correct double-bit errors but can detect them.

Test Case: Flip two bits in the encoded message and observe the failure to decode accurately.

Example: Inject errors into positions 2 and 5 of `0111000`.

4.5.3 Larger Message Sizes

Purpose: Verify scalability of the implementation.

Test Case: Test with increasingly large input sizes (e.g., 16, 32, 64-bit messages).

Example: Use a 64-bit input and validate that encoding and error correction work correctly.

4.5.4 No Errors

Purpose: Ensure that no errors in transmission do not alter the message.

Test Case: Encode and decode a message without any injected errors.

Example: Input `1011` → Encode → Decode → Verify `1011`.

4.5.5 Random Error Injection

Purpose: Simulate real-world scenarios where errors occur randomly.

Test Case: Randomly flip bits in the encoded message across multiple runs.

Conclusions and Scope for Future Works

Key findings of Hamming code implementation include:

- **Error Detection and Correction Capability:** Hamming codes detect and correct single-bit errors, which enhances communication system reliability.
- **Low Complexity:** The encoding and decoding processes are computationally lightweight, suitable for real-time applications.
- **Hardware Efficiency:** Hamming codes can be effectively implemented in both software and hardware, including FPGA and ASIC designs.

Scope for Future Work While Hamming codes provide a strong foundation for error detection and correction, further enhancements can extend their capabilities for evolving applications. Future work can focus on the following aspects:

- Hardware Optimization

  Optimizing the hardware implementation of Hamming code, including faster encoding and decoding circuits, can benefit real-time communication and memory systems. This involves reducing power consumption, increasing speed, and enhancing scalability.
- Integration with Modern Protocols

  As communication protocols continue to evolve, integrating Hamming codes into modern network and communication standards, including wireless networks, 5G systems, and IoT applications.
- Quantum Error Correction

  Exploring the application of Hamming code principles in quantum computing environments, where quantum error correction is essential, can open new research directions and lead to novel hybrid quantum-classical solutions.

In conclusion, Hamming code remains a fundamental approach for error detection and correction. Continued research and development can expand its applications, making it more relevant for emerging technologies and critical communication systems.

References

[1] Ad. Attarian. *Algebraic Coding Theory*. Self-published, 2006. 12 pages.

[2] Jo Author Unknown or Hall. Hamming codes notes, Access Year, e.g., 2024. Accessed: December 5, 2024.

[3] Beneater. Error detection video series. `https://github.com/beneater/error-detection-videos`, 2018. Accessed: 2024-11-07.

[4] Ali El-Mousa, Anssari Nasser, Ashraf Suyyagh, and Al-Zubi Hamzah. The design and implementation of a reconfigurable usart ip core for embedded computing with support for networks. *Lecture Notes in Engineering and Computer Science*, 2170, 07 2008.

[5] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[6] Irene Ndanu John, Peter Waweru Kamaku, Dishon Kahuthu Macharia, and Nicholas Muthama Mutua. Error detection and correction using hamming and cyclic codes in a communication channel. *Pure and Applied Mathematics Journal*, 5(6):220–231, 2017.

[7] Hirohisa Kawamoto. The history of liquid-crystal display and its industry. In *2012 Third IEEE HISTory of ELectro-technology CONference (HISTELCON)*, pages 1–6, Sep. 2012.

[8] Omar Khadir. A simple coding-decoding algorithm for the hamming code, 01 2022.

[9] Stephen. *PS2 keyboard and mouse interface*. http://www.cs.columbia.edu/ sedwards/classes/2011/4840/ps2-keyboard.pdf, 2011. Accessed: 2024-12-05.