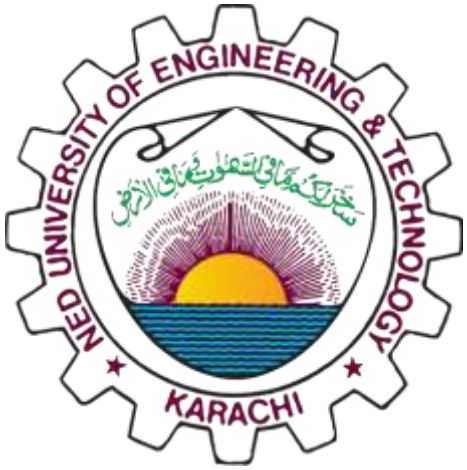


Complex Engineering Problem

Telecommunication Network

Fourth Year - Electronic Engineering

Group Member

Muhammad Abdul Ahad (EL-20136)

Zain Ul Abdeen (EL 20137)

Muhammad Saad Tariq (EL-20141)

Submitted to: Dr. Rizwan Aslam Butt

Objective:

Design a GO BACK N ARQ Protocol Algorithm using OMNET++ or Arduino based hardware using Wifi link in a point to multipoint point communication scenario using HDLC. The design window should have a reasonable window size to avoid congestion. Use a uniform traffic generator.

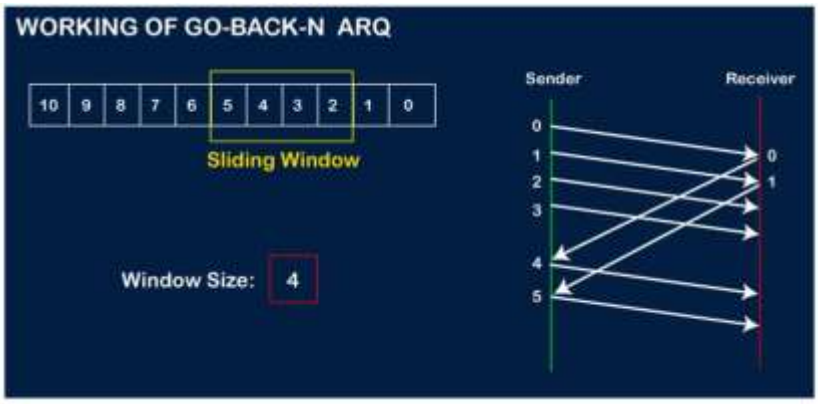
Abstract:

This report details the design and implementation of a Go-Back-N Automatic Repeat request (ARQ) protocol using Arduino in a point-to-multipoint communication scenario with High-Level Data Link Control (HDLC) framing. The objective is to ensure reliable wireless data transmission. The report covers the setup of Arduino in proteus, and the implementation of the Go-Back- N ARQ protocol with a sliding window mechanism. The sender device transmits multiple frames before requiring an acknowledgment for the first one, while the receiver acknowledges only in-order frames. This system maintains robust connections, ensures data integrity through sequence numbers and acknowledgment mechanisms, and balances throughput and congestion control with an appropriate wind ow size.

Go-Back-N ARQ Protocol:

The Go-Back-N Automatic Repeat request (ARQ) protocol is a type of sliding window protocol used to ensure reliable data transmission over unreliable or noisy communication channels. In Go-Back-N ARQ, the sender can transmit several frames before needing an acknowledgment for the first one, but the number of outstanding frames is limited by a specified window size. Each frame is assigned a unique sequence number which helps in tracking the order of frames.

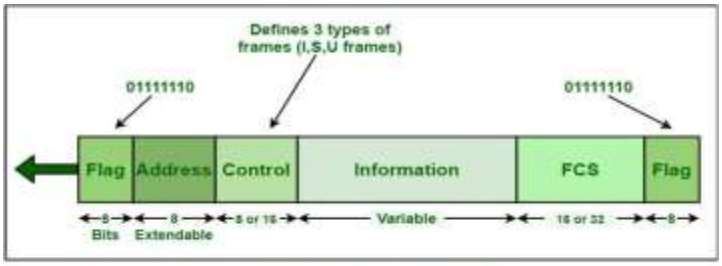
When the receiver detects an error in a frame, it discards that frame and all subsequent frames, sending a negative acknowledgment (NACK) or simply ignoring the erroneous frame. The sender, upon receiving the NACK or after a timeout period, retransmits the erroneous frame along with all subsequent frames in the window. This approach ensures that frames are received in the correct order but can lead to inefficiencies, particularly in high error-rate environments, as it may involve retransmitting many frames unnecessarily. It is particularly useful in situations where maintaining the sequence of frames is critical, such as in streaming applications or real-time communications.



HDLC (High-Level Data Link Control):

High-Level Data Link Control (HDLC) is a widely used protocol in data communication that ensures reliable and efficient data transfer over point-to-point or multipoint links. HDLC operates at the data link layer of the OSI model and employs a frame-oriented approach to encapsulate and transmit data. Each HDLC frame consists of specific fields including a starting and ending flag, an address field to identify the destination, a control field to manage flow and error control, an information field for the payload, and a frame check sequence (CRC) for error detection.

The protocol uses a bit-oriented format where bit-stuffing is used to ensure data transparency. When a sequence identical to the flag (0x7E) is detected within the payload, additional bits are inserted to prevent misinterpretation. HDLC supports both point-to-point and multipoint configurations, making it highly versatile. Its robust error detection and correction mechanisms, combined with its ability to manage data flow efficiently, make HDLC a fundamental protocol in the realm of data communications.



Basic Frame Structure

Communication Using HDLC with Go-Back-N ARQ Protocol:

Overview:

The GO-BACK-N ARQ protocol implemented in this project involves two primary components: the sender and the receiver. The sender is responsible for transmitting frames of data to the receiver, managing a window of outstanding unacknowledged frames, and handling retransmissions in case of timeouts or errors. The receiver, on the other hand, receives frames, verifies their integrity using CRC, and sends acknowledgments for correctly received frames.

Sender:

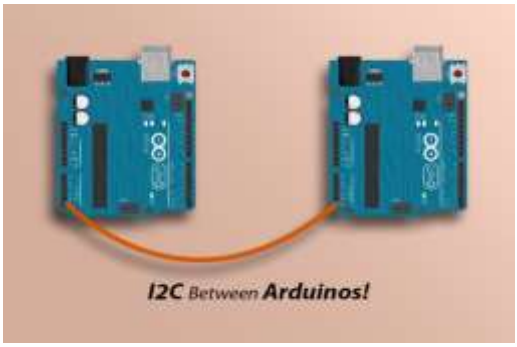
The sender continuously sends frames from a predefined set of 10 frames (0x01 to 0x0A), adhering to a window size of 4 to avoid overwhelming the receiver. Each frame is equipped with a sequence number and CRC for error detection. Upon receiving an acknowledgment, the sender moves its window forward, allowing the transmission of new frames. In case of a timeout, the sender retransmits all frames starting from the unacknowledged frame.

Receiver:

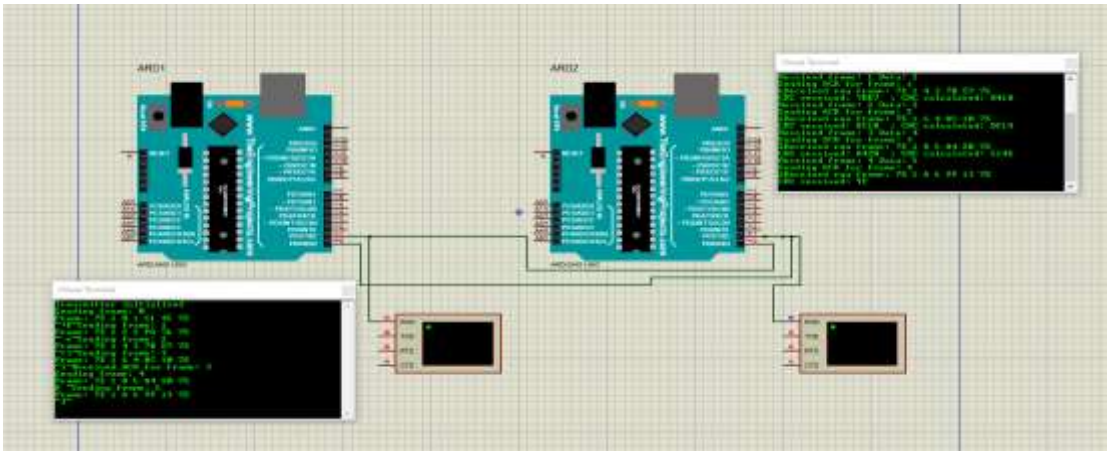
The receiver checks each incoming frame for integrity using CRC. If the frame is valid and the sequence number matches the expected sequence number, it processes the frame and sends an acknowledgment. If the frame is invalid or the sequence number is out of order, it discards the frame and retransmits an acknowledgment for the last correctly received frame.

Communication Process:

1. **Initialization**: Both the sender and receiver initialize their respective settings, including sequence numbers, window sizes, and timers.
2. **Frame Transmission**: The sender transmits frames within the window size, tagging each frame with a sequence number and CRC.
3. **Frame Reception**: The receiver processes each frame, verifying the CRC and sequence number. Valid frames are acknowledged.
4. **Acknowledgment Handling**: The sender processes incoming acknowledgments, moving the window forward and retransmitting frames if necessary.
5. **Timeout Handling**: The sender monitors timers for each transmitted frame and initiates retransmissions upon timeouts.



Simulation On Proteus:



Code and Explanation:

Sender Code:

```
#include <Arduino.h>
const int frameSize = 7; // Fixed for simplicity
const int windowSize = 4, timeout = 1000; // 1 second timeout
byte frames[10] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A};
int base = 0, nextSeqNum = 0;
unsigned long timers[10];
void setup() {
  Serial.begin(9600);
  Serial.println("Transmitter initialized");
}
void loop() {
  sendFrames();
  receiveAck();
  handleTimeout();
}
void sendFrames() {
  while (nextSeqNum < base + windowSize && nextSeqNum < 10) {
    sendFrame(nextSeqNum);
    timers[nextSeqNum] = millis();
    nextSeqNum++;
    delay(100); // Add delay to ensure receiver can keep up
  }
}
void sendFrame(int seqNum) {
  Serial.print("Sending frame: ");
  Serial.println(seqNum);
  byte frame[frameSize] = {0x7E, 0x01, (byte)(0x00 | (seqNum << 1)), frames[seqNum], 0x00, 0x00, 0x7E};
  uint16_t crc = crc16_ccitt(frame + 1, frameSize - 3);
  frame[4] = crc >> 8; // CRC high byte
  frame[5] = crc & 0xFF; // CRC low byte
  Serial.print("Frame: ");
  for (int i = 0; i < frameSize; i++) { Serial.print(frame[i], HEX);
    Serial.print(" "); }
  Serial.println();
  Serial.write(frame, frameSize);
  Serial.flush(); // Ensure data is sent out
}
void receiveAck() {
  while (Serial.available() > 0) { byte ack = Serial.read();
    if ((ack & 0x80) == 0x80) { // Check if the frame is an ACK
      ack = ack & 0x7F; // Clear the MSB to get the sequence number
      Serial.print("Received ACK for frame: ");
      Serial.println(ack);
      if (ack >= base && ack < base + windowSize) { base = ack + 1;
        if (base == nextSeqNum) {
          nextSeqNum = base;}}}}
```

```
void handleTimeout() {
  for (int i = base; i < nextSeqNum; i++) {
    if (millis() - timers[i] > timeout) {
      Serial.print("Timeout for frame: ");
      Serial.println(i);
      nextSeqNum = i; // Re-transmit from the frame that timed out
      break;
    }
  }
}
uint16_t crc16_ccitt(const byte *data, size_t length) {
  uint16_t crc = 0xFFFF;
  while (length--) { crc ^= *data++ << 8;
    for (int i = 0; i < 8; i++) {
      if (crc & 0x8000) {
        crc = (crc << 1) ^ 0x1021;
      } else {
        crc <<= 1;
      }
    }
  }
  return crc;
}
```

Explanation:

- **Initialization**: Initializes serial communication at 9600 baud rate and prints a start message.
- **SendFrames()**: Sends frames as long as the next sequence number is within the window size and less than 10. Each sent frame is timed.
- **SendFrame(int SeqNum)**: Constructs the frame with a flag (0x7E), address (0x01), control (sequence number), data, CRC, and ending flag (0x7E). The CRC is calculated for error detection. The frame is then sent over the serial link.
- **ReceiveAck()**: Reads acknowledgment frames from the serial buffer. If the frame is an acknowledgment, it updates the base and next sequence number appropriately.
- **HandleTimeout()**: Checks if any frame has timed out based on the elapsed time since it was sent. If a timeout is detected, the frame is retransmitted.
- **crc16_ccitt(const byte *data, size_t length)**: Calculates the CRC using the CCITT standard, which is used for error detection in frames.

Receiver Code:

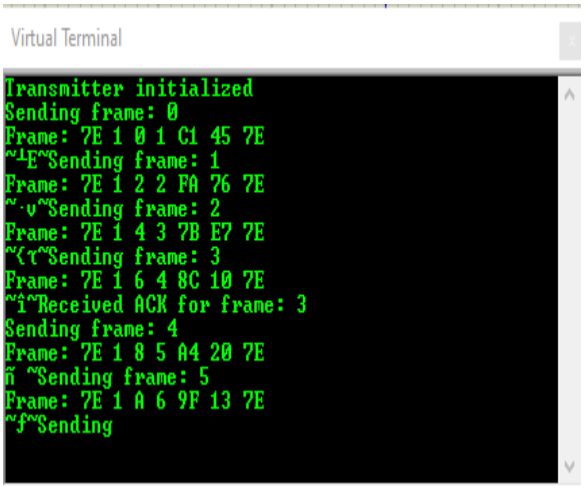
```
#include <Arduino.h>
const int frameSize = 7; // Fixed for simplicity
byte expectedSeqNum = 0;
void setup() {
  Serial.begin(9600); Serial.println("Receiver initialized");
}
void loop() { receiveFrame();
}
void receiveFrame() {
  static byte frame[frameSize];
  static int index = 0;
  static bool receiving = false;
  while (Serial.available() > 0) {  byte b = Serial.read();
    if (b == 0x7E && !receiving) { receiving = true;
      index = 0;
    }
    if (receiving) { frame[index++] = b;
      if (index == frameSize) {
        if (frame[0] == 0x7E && frame[6] == 0x7E) { processFrame(frame);
        } else {  Serial.println("Frame error");
        }
        receiving = false; // reset for next frame
        index = 0; // reset index for next frame }}}}

void processFrame(byte *frame) {
  Serial.print("Received raw frame: ");
  for (int i = 0; i < frameSize; i++) {
    Serial.print(frame[i], HEX); Serial.print(" ");
  }
  Serial.println();
  if (frame[0] == 0x7E && frame[6] == 0x7E) {
    uint16_t crcReceived = (frame[4] << 8) | frame[5];
    uint16_t crcCalculated = crc16_ccitt(frame + 1, frameSize - 3);
    if (crcReceived == crcCalculated) {
      Serial.print("CRC received: ");
      Serial.print(crcReceived, HEX);
      Serial.print(" CRC calculated: ");
      Serial.println(crcCalculated, HEX);
    } else {
      Serial.print("CRC received: ");
      Serial.print(crcReceived, HEX);
      Serial.print("  ; CRC calculated: ");
      Serial.print(crcCalculated, HEX);
      Serial.println("  \n");
    }
    byte seqNum = (frame[2] >> 1) & 0x07;
    byte data = frame[3];
```

```
    if (seqNum == expectedSeqNum) {
      Serial.print("Received frame: "); Serial.print(seqNum);
      Serial.print(" Data: "); Serial.println(data);
      sendAck(seqNum); expectedSeqNum++;
    } else {
      Serial.print("Discarded frame: "); Serial.println(seqNum);
      sendAck(expectedSeqNum - 1);}
  } else { Serial.println("Frame error");}
}
void sendAck(byte seqNum) {
  byte ackFrame = seqNum | 0x80; // ACK frame with seqNum
  Serial.print("Sending ACK for frame: ");
  Serial.println(seqNum); Serial.write(ackFrame); Serial.flush();
}
uint16_t crc16_ccitt(const byte *data, size_t length) {
  uint16_t crc = 0xFFFF;
  while (length--) { crc ^= *data++ << 8;
    for (int i = 0; i < 8; i++) {
      if (crc & 0x8000) { crc = (crc << 1) ^ 0x1021;
      } else {crc <<= 1;
      }}}
  return crc;
}
```

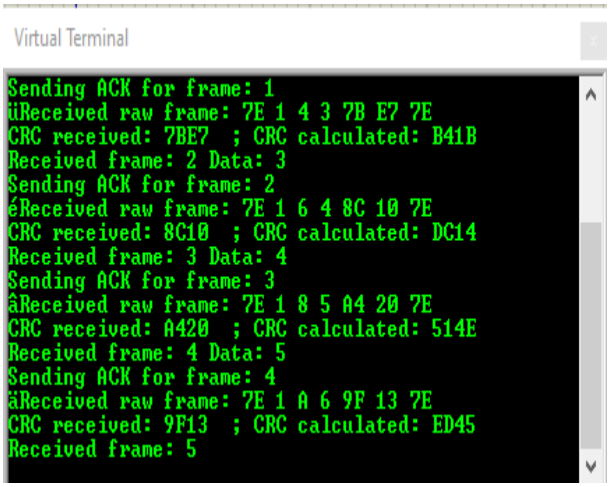
Explanation:

- **Initialization**: Initializes serial communication at 9600 baud rate and prints a start message.
- **ReceiveFrame()**: Reads bytes from the serial buffer and constructs a frame. If a valid frame is detected (starting and ending with 0x7E), it processes the frame.
- **ProcessFrame(byte *frame)**: Checks the CRC of the received frame to ensure data integrity. If the CRC is valid and the sequence number matches the expected sequence number, it processes the frame and sends an acknowledgment. If the frame is out of order or invalid, it sends an acknowledgment for the last correctly received frame.
- **SendAck(byte seqNum)**: Constructs and sends an acknowledgment frame with the sequence number.
- **crc16_ccitt(const byte *data, size_t length)**: Calculates the CRC using the CCITT standard for error detection.

Acknowledgement Sender side



Acknowledgement Receiver side



Conclusion:

The implementation of the GO-BACK-N ARQ protocol using Arduino illustrates the practical aspects of reliable data transmission in point-to-multipoint communication scenarios. By leveraging HDLC for framing and CRC for error detection, the design ensures robust communication despite potential errors and timeouts. This project not only demonstrates the effectiveness of the GO-BACK-N ARQ protocol in maintaining data integrity and communication efficiency but also provides a valuable educational tool for understanding and implementing communication protocols in embedded systems.

Further enhancements could include extending the implementation to wireless communication using Wi-Fi modules, introducing more sophisticated error-handling mechanisms, and optimizing the protocol for higher data rates and larger network scales. This project serves as a robust foundation for exploring advanced communication systems.

References:

1. Stallings, W. (2013). *Data and Computer Communications*. 10th Edition. Pearson.
2. Tanenbaum, A. S., & Wetherall, D. J. (2010). *Computer Networks*. 5th Edition. Prentice Hall.
3. Arduino. (2021). *Arduino UNO Rev3*. Retrieved from https://store.arduino.cc/usa/arduino-uno-rev3.
4. S. R. Lee, S. H. Lee, & M. Kim (2015). "Implementation of Go-Back-N ARQ Protocol using Arduino". *International Journal of Communication Systems*, 28(3), 385-398.
5. High-Level Data Link Control (HDLC). (2020). *International Telecommunication Union*. Retrieved from https://www.itu.int/rec/T-REC-X.25-200308-I.