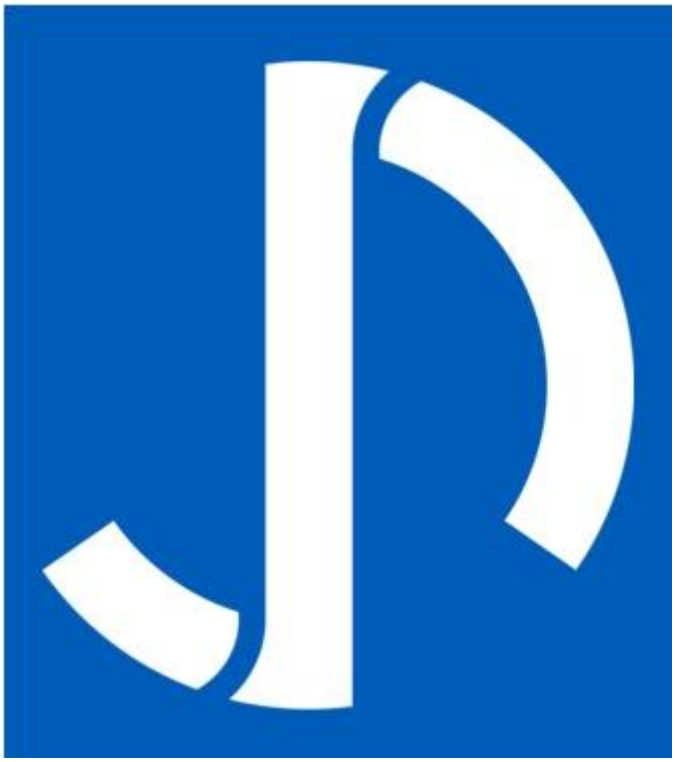

DEICO ENGINEERING

INTERNSHIP REPORT

OGUZHAN OGUZ

Contents

1. TASKS PERFORMED DURING THE INTERNSHIP

1.1 Introduction

The internship started on January 20. The internship generally focused on communication protocols. The first days were theoretical studies. Serial communication protocols were investigated. The differences between RS232, RS422, RS485 were investigated. UART, SPI, I2C, CAN, MODBUS and LIN protocols were investigated. In addition, the differences between TCP and UDP, OSI model layer 7, IP address and DNS were investigated.

Later, the focus was on interface development using the C# language. A TCP protocol application was developed for a single computer. A UDP application was developed for communication between two computers. Later, an interface was developed to control the KEYSIGHT 33500B Waveform Generator and KEYSIGHT InfiniiVision MSOX310T Mixed Signal Oscilloscope devices from the computer with SCPI commands. The development of a test application using the developed interface was started. An application was developed that tests the accuracy of the devices by measuring the parameters of the signals generated from the waveform generator on the oscilloscope. Approximately two weeks were spent on developing this application.

1.2 Communication Protocols

Communication protocols are used to exchange data between devices in modern systems. According to the data transmission method, it is divided into two as serial and parallel communication. Serial communication allows data to be sent bit by bit over a single channel. Parallel communication transmits data over multiple boilers at the same time. Parallel communication provides higher data over short distances. However, signal distortions occur more over long distances. It is also more costly and complex to implement because it requires more physical connections. In contrast, serial communication provides less cable usage and better noise immunity. It offers more reliable data transmission over long distances. Therefore, it is a more preferred method in embedded systems and long-distance communication.

Serial communication is divided into two basic categories: synchronous and asynchronous. In synchronous communication, data is sent with a clock signal. In this way, synchronization is achieved between the receiver and the transmitter. It is more efficient because it minimizes the delay. In asynchronous communication, there is no clock signal. Instead, packet start and stop bits are used. It is a more easily applicable method. It may create additional load due to time discrepancies. Common serial protocols will be discussed below.

RS-232, RS-422 and RS-485 are some of the most commonly used serial communication protocols. These protocols can be used with the device in Figure 1. Detailed information is given in Table 1.

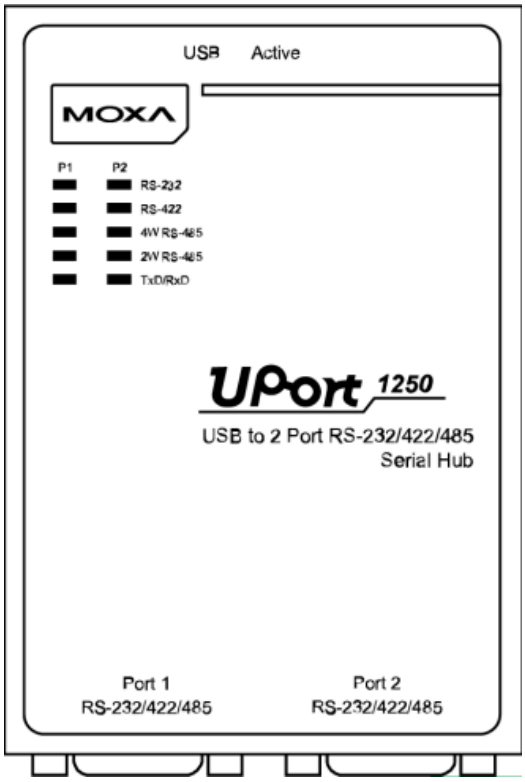

Figure 1. UPort 1250 device [1]

Table 1. RS-232, RS-422 and RS-485 protocols

Feature	RS-232	RS-422	RS-485
Device Number	1 driver, 1 receivers	1 driver, 10 receivers	Up to 32 drivers, 32 receivers
Signal Type	Single-ended	Differential	Differential
Maximum Distance	15 meters	1200 meters	1200 meters
Maximum Speed	115 Kbps	10 Mbps	Mbpss
Noise Resistance	Low	High	High
Application Areas	Computers and peripheral devices	Industrial automation	Industrial control, SCADA

To briefly mention other serial communication protocols, also comparative features can be seen in Table 2.

- **UART** stands for Universal Asynchronous Receiver/Transmitter. As its name suggests, it is an asynchronous communication method. It is widely used in microcontroller-based systems. The reason for this is that its hardware requirements are low.
- **SPI** stands for Serial Peripheral Interface. It is a high-speed and synchronous protocol. It is used between microcontrollers and peripherals.

- **I²C** stands for Inter-Integrated Circuit. It is designed for communication between devices operating at low speed. It is a synchronous protocol. It allows multiple devices to communicate over a single data path.
- **CAN** stands for Controller Area Network. It is used to provide reliable communication in noisy environments. It has error detection and collision avoidance mechanisms. It is a robust protocol with multi-node support. It is widely used in automotive and industrial applications.
- **MODBUS** protocol generally uses master-slave architecture. It is a synchronous protocol. A device (master) sends commands, slave devices (slaves) respond. It supports RTU and ASCII data formats. It is a lightweight and open protocol. Therefore, its cost is low. It is widely used in industrial automation systems, PLC, sensor and actuator data transmission.
- **LIN** stands for Local Interconnect Network. It uses a master-slave architecture. It provides communication over a single cable. It has low bandwidth. It offers low data rate and simplicity, so it is low cost. It is generally used as a complement to CAN. It is frequently used in automotive subsystems.

Table 2. Serial communication protocols

Feature	UART	SPI	I²C	CAN	MODBUS	LIN
Communication Type	Asynchronous	Synchronous	Synchronous	Multi-master	Master-slave	Master-slave
Number of Wires	2 (TX - RX)	4 (MOSI, MISO, SCK, CS)	2 (SDA, SCL)	2 (CANH, CANL)	Varies	1
Speed	Up to 1 Mbps	Up to 50 Mbps	Up to 5 Mbps	Up to 1 Mbps	Up to 10 Mbps	Up to 20 Kbps
Distance	Varies depending on RS type	PCB level communication (less than 1 meter)	Up to 1 meter (standart mode), up to 30 meters (low-speed mode)	Up to 40 meters at 1 Mbps, up to 1 km at lower speeds	Varies depending on RS type	Up to 40 meters
Number of Devices	1-to-1	1 master, multiple slaves	Multiple masters, multiple slaves	Multiple nodes (up to 127)	Multiple slaves	1 master and up to 16 slaves
Error Handling	No built-in error checking	No built-in error checking	Acknowledgment (ACK/NACK)	CRC & Arbitration	Error detection in some modes	Basic error detection
Application Areas	Microcontrollers	High-speed data transfer	Low-speed sensor networks	Automative	Industrial automation	Automative subsystems

1.3 OSI Model

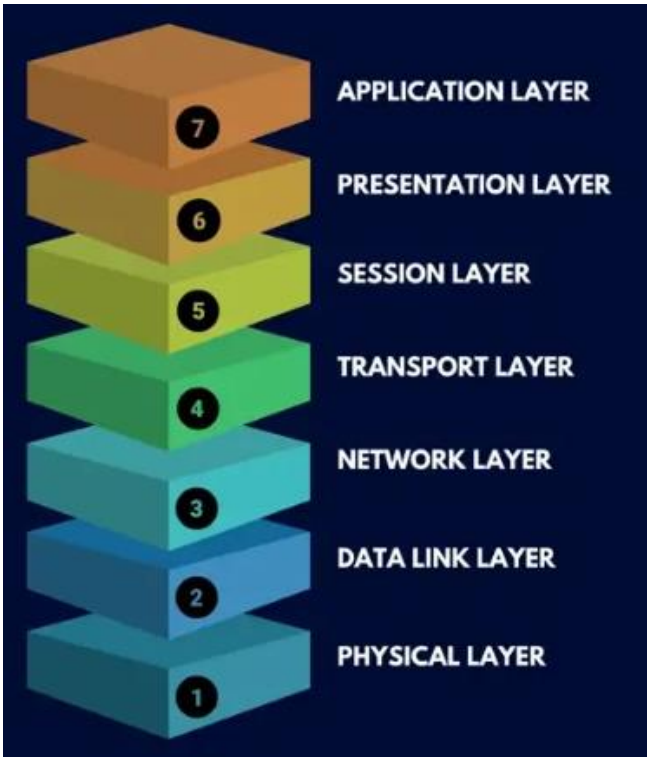

Figure 2. Seven layers of OSI model [2]

The OSI (Open System Interconnection) model standardizes network communication by dividing it into seven layers. Each layer is responsible for a specific function and interacts with the lower and upper layers. These layers can be seen in Figure 2.

1. **Physical Layer:** It is the lowest layer of the model. It provides the transmission of raw data over the physical medium. Electrical and mechanical properties are defined. In this way, it determines how data signals will be transmitted via cables, fiber optic lines or radio waves. Things like voltage levels, signal timing, data transmission speeds, connector types are defined in this layer.

2. **Data Link Layer:** It performs error detection and correction functions in data transmission over the physical layer. In this layer, communication between directly connected devices is managed and data collisions are prevented. It is divided into MAC and LLC sublayers. MAC layer means media access control. By assigning MAC addresses to devices, it is determined which device the data packets will go to. Ethernet works here. LLC stands for logical link control. Synchronization and flow

control of data frames are managed. Devices operating at this layer include switches and NICs.

3. **Network Layer:** In this layer, the data is directed between different networks, or routing. The most suitable path between the source and destination is determined. Data packets are directed to the determined path. The most important protocol working in this layer is Internet Protocol (IP). In addition, error messages are transmitted in this layer with Internet Control Message Protocol (ICMP). Address Resolution Protocol (ARP) is used to translate IP addresses into MAC addresses. Routers work in this layer.

4. **Transport Layer:** This layer provides end-to-end communication, ensuring that data is transmitted completely and accurately. Data segmentation is managed and reassembled. This layer contains two main protocols. The first is Transmission Control Protocol (TCP). It is a connection-oriented protocol. It provides error control, sequential data transmission, and retransmission mechanisms. It is suitable for web, e-mail, and file transfer. The second protocol is User Datagram Protocol. It is a connectionless protocol. It does not perform error correction. However, it is used in real-time applications such as video streaming and online games because it provides lower latency. This layer also provides data routing between different applications on the same device using port numbers.

5. **Session Layer:** This layer manages communication sessions between two devices or applications. It performs the tasks of establishing, maintaining, and terminating the connection. It provides authentication and data synchronization.

6. **Presentation Layer:** In this layer, data is converted into a format that the sending and receiving systems can understand. Data encryption, compression and encoding processes are performed.

7. **Application Layer:** It is the top layer where users interact directly. Programs such as web browsers and email clients run on this layer. The most common protocols are HTTP/HTTPS, ETP, DNS.

1.4 Transport Layer Applications

Communication systems and interfaces of these systems have been developed for the TCP and UDP protocols mentioned in the OSI Model section with using C#.

1.4.1 TCP Protocol

One server interface and two client interfaces were made. The code can be seen in the appendix. The client connects to the server using TcpClient. The server accepts the connection using TcpListener.AcceptTcpClient(). The client first sends its name to the server. The server reads the client's name and logs it. The client sends messages. The server logs them and sends a response. The client receives the response and displays it on the screen. Finally, the client disconnects, the server detects this and logs it.

To explain more detailed:

- **Starting the server:**

  server = new TcpListener(IPAddress.Any, 12345);
  server.Start();

  With this code, TcpListener is created and connections are accepted on port 12345. IPAddress.Any allows the server to accept connections from all network interfaces. The created server GUI can be seen in Figure 3. After the server is started, the start button is disabled, preventing multiple server starts.

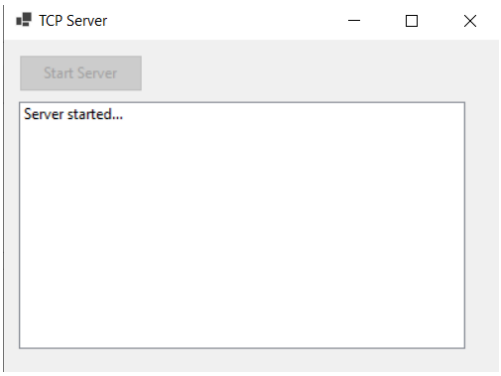

Figure 3. TCP server

- **Multithreading:**

```
while (true)
{
    TcpClient client = server.AcceptTcpClient();
    Thread clientThread = new Thread(() => HandleClient(client));
    clientThread.IsBackground = true;
    clientThread.Start();
}
```

The server.AcceptTcpClient() code waits for a client connection. A separate thread is started for each client. The IsBackground property ensures that the threads are automatically terminated when the application closes. The clients and their connections can be seen in Figure 4 and 5.

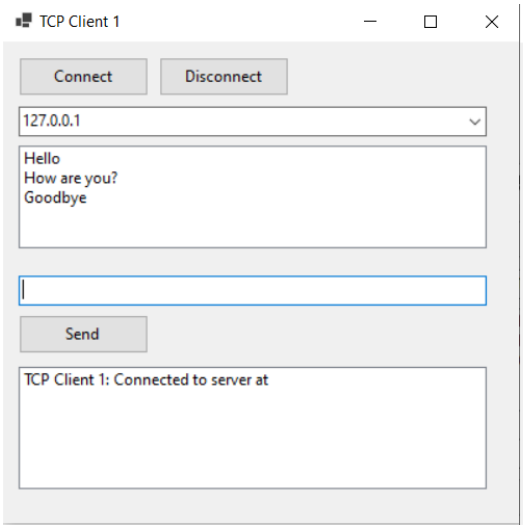

Figure 4. TCP client 1 connection

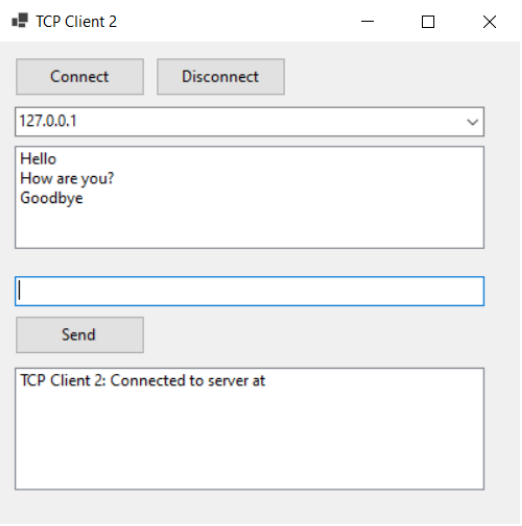

Figure 5. TCP client 2 connection

- **Client Communication:**

Reading Client Name:

```
NetworkStream stream = client.GetStream();
byte[] buffer = new byte[1024];
int nameBytesRead = stream.Read(buffer, 0, buffer.Length);
string clientName = Encoding.UTF8.GetString(buffer, 0, nameBytesRead);
```

A NetworkStream object is created to receive client data. A 1024-byte buffer is allocated and the client name is read. The incoming byte sequence is converted to a string in UTF-8 format.

Processing Incoming Messages:

```
while (true)
{
```

```
    int bytesRead = stream.Read(buffer, 0, buffer.Length);
    if (bytesRead == 0) break;

    string message = Encoding.UTF8.GetString(buffer, 0, bytesRead);

    logs.Invoke((MethodInvoker)delegate
    {
        logs.Items.Add($"Message from {clientName}: {message}");
    });

    string response = $"Server received: {message}";
    byte[] responseBytes = Encoding.UTF8.GetBytes(response);
    stream.Write(responseBytes, 0, responseBytes.Length);
}
```

In a loop, the server waits for client messages. When the message is received, it is decoded as a UTF-8 string and added to the log. The server received message is sent to the client in response. The messages of the clients can be seen in Figure 6 and 7.

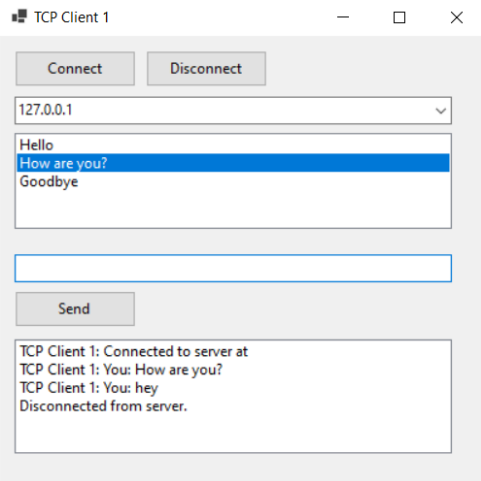

Figure 6. TCP client 1 messages

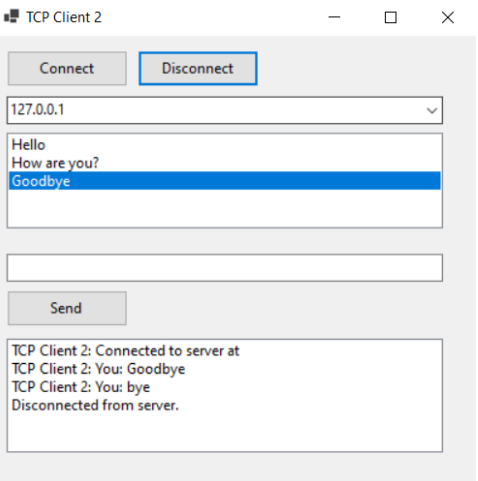

Figure 7. TCP client 2 messages

Client Disconnection:

```
catch (IOException)
{
    logs.Invoke((MethodInvoker)delegate
    {
        logs.Items.Add($"Client disconnected: {clientName}");
    });
}
finally
{
```

```
    client.Close();
}
```

When the client closes the connection, an IOException error is thrown and logged. client.Close() is called to release the resources.

- **Creating Client App:**

```
string ipAddress = ipAddressBox.Text;
client = new TcpClient(ipAddress, 12345);
stream = client.GetStream();

string clientName = this.Text;
byte[] nameBytes = Encoding.UTF8.GetBytes(clientName);
stream.Write(nameBytes, 0, nameBytes.Length);
```

The client opens a TCP connection to the server. A NetworkStream object is created and data is sent. First, the client's name is passed to the server.

```
byte[] messageBytes = Encoding.UTF8.GetBytes(message);
stream.Write(messageBytes, 0, messageBytes.Length);
```

The message is converted to a UTF-8 byte sequence and sent to the server with stream.Write().

```
if (client != null && client.Connected)
{
    client.Close();
    logs.Items.Add("Disconnected from server.");
}
```

The connection is closed and the client is added to the log. Each client operates independently and opens a separate connection to the server. The server manages each client in a separate thread. Clients can send messages and receive responses independently of each other. Also, if clients try to send messages without connecting to the server, they will receive a "server not connected" error. It can be seen in Figure 8. In Figure 9, server logs can be seen.

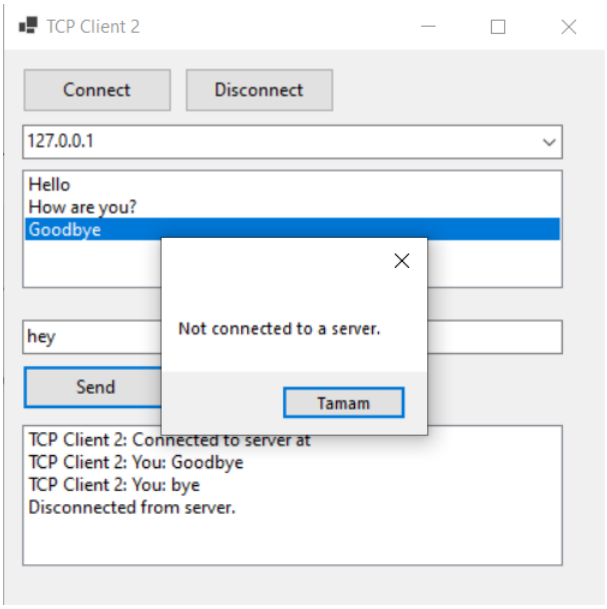

Figure 8. Not connected to a server error

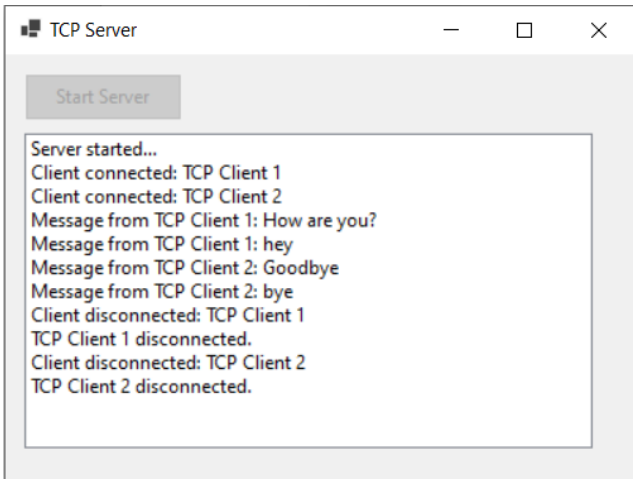

Figure 9. Server logs

1.4.2 UDP Protocol

This time, a server and a client were created to communicate on two different computers. A LAN cable was used to connect two computers to ensure communication. The IP address for the server device was set to 1.1.2.3, and the IP address for the client was set to 1.1.2.4.

- **Server Initialization and Start:**

```
private void startButton_Click(object sender, EventArgs e)
{
   serverThread = new Thread(StartServer);
   serverThread.IsBackground = true;
   serverThread.Start();
   logs.Items.Add("UDP Server started...");
   startButton.Enabled = false;
}
```

A background thread is created and the UDP server is started. The StartServer() function is run in a separate thread to avoid blocking the MAIN UI thread. After the server is started, the start button is disabled, preventing multiple server starts.

- **Listening to Incoming UDP Packets:**

```
private void StartServer()
{
   udpServer = new UdpClient(12345);
   IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, 0);
```

The UDP server (UdpClient) listens on port 12345. IPAddress.Any allows the server to accept packets from all network interfaces.

- **Processing Incoming Messages:**

```
while (true)
{
   byte[] receivedBytes = udpServer.Receive(ref remoteEP);
   string receivedMessage = Encoding.UTF8.GetString(receivedBytes);
   string clientIdentifier = $"{remoteEP.Address}:{remoteEP.Port}";
```

The udpServer.Receive(ref remoteEP) function acts as a blocker. In other words, it waits until data arrives from the client. The incoming byte array is converted to a string in UTF-8 format. A new ID is created in IP:Port format to identify the client.

- **Client Connection Management:**

```
if (receivedMessage == "CONNECT")
{
   if (!connectedClients.Contains(clientIdentifier))
   {
      connectedClients.Add(clientIdentifier);
      logs.Invoke((MethodInvoker)delegate
      {
         logs.Items.Add($"Client connected: {clientIdentifier}");
      });
   }
}
```

If a connect message is received from the client, the client is added to the list of connected clients. It uses HashSet for this. It uses logs.invoke() for UI updates.

- **Client Disconnection:**

```
else if (receivedMessage == "DISCONNECT")
{
   if (connectedClients.Contains(clientIdentifier))
   {
      connectedClients.Remove(clientIdentifier);
      logs.Invoke((MethodInvoker)delegate
      {
         logs.Items.Add($"Client disconnected: {clientIdentifier}");
      });
   }
}
```

If the client sends a disconnect message, the connection is removed from the list of clients.

- **Error Management and Server Shutdown:**

```
catch (Exception ex)
{
```

```
    logs.Invoke((MethodInvoker)delegate
    {
        logs.Items.Add($"Error: {ex.Message}");
    });
}
finally
{
    udpServer.Close();
}
```

Errors are caught and logged. When the server is stopped, the UDP connection is closed. The created server can be seen in Figure 10.

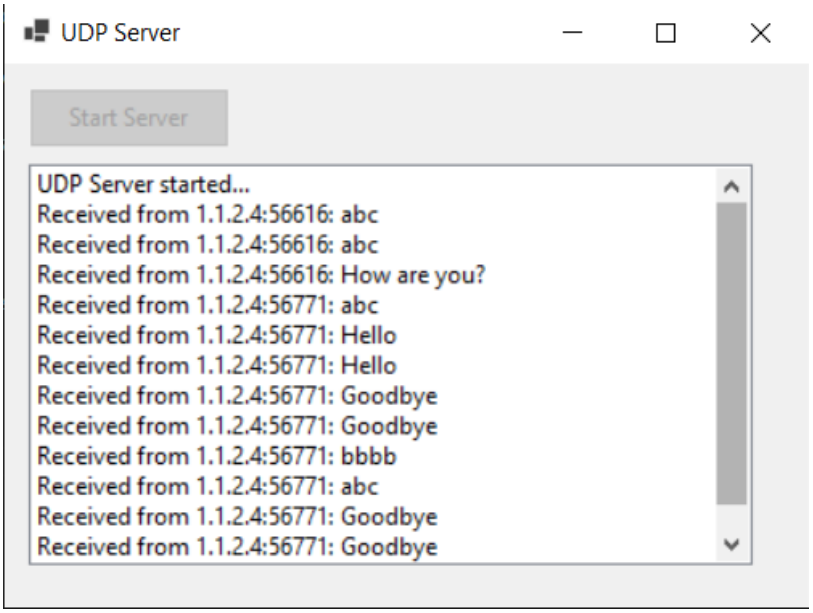

Figure 10. UDP server

- **Connecting the Client to the Server:**

```
private void connectButton_Click(object sender, EventArgs e)
{
    if (isConnected)
    {
        MessageBox.Show("You are already connected to the server.");
        return;
    }


try
{
    udpClient = new UdpClient();
```

```
    string connectMessage = "CONNECT";
    byte[] connectBytes = Encoding.UTF8.GetBytes(connectMessage);
    udpClient.Send(connectBytes, connectBytes.Length, ipAddressBox.Text, 12345);
    logs.Items.Add($"Connected to server at {ipAddressBox.Text}");
    isConnected = true;
}
```

If the client has connected before, it is not allowed to connect again. A new UdpClient is created. The Connect message is sent to the server. The connection status is also updated with isConnected. If the client has connected before, it is not allowed to reconnect.

- **Disconnect from Server:**

```
private void disconnectButton_Click(object sender, EventArgs e)
{
    if (udpClient != null)
    {
        try
        {
            string disconnectMessage = "DISCONNECT";
            byte[] disconnectBytes = Encoding.UTF8.GetBytes(disconnectMessage);
            udpClient.Send(disconnectBytes,  disconnectBytes.Length,  ipAddressBox.Text,
12345);
            logs.Items.Add("Disconnected from server.");
            isConnected = false;
        }
```

The Disconnect message is sent to the server. The RDP client is closed and set to null.

- **Sending Message:**

```
private void sendButton_Click(object sender, EventArgs e)
{
    if (udpClient == null || !isConnected)
    {
        MessageBox.Show("Not connected to a server.");
        return;
    }

byte[] messageBytes = Encoding.UTF8.GetBytes(message);
udpClient.Send(messageBytes, messageBytes.Length, ipAddressBox.Text, 12345);
```

logs.Items.Add($"Message sent to {ipAddressBox.Text}: {message}");

A connection check is performed, if the client is not connected, the message sending is blocked. The entered message is converted into a byte array and sent to the server. The created UDP client can be seen in Figure 11.

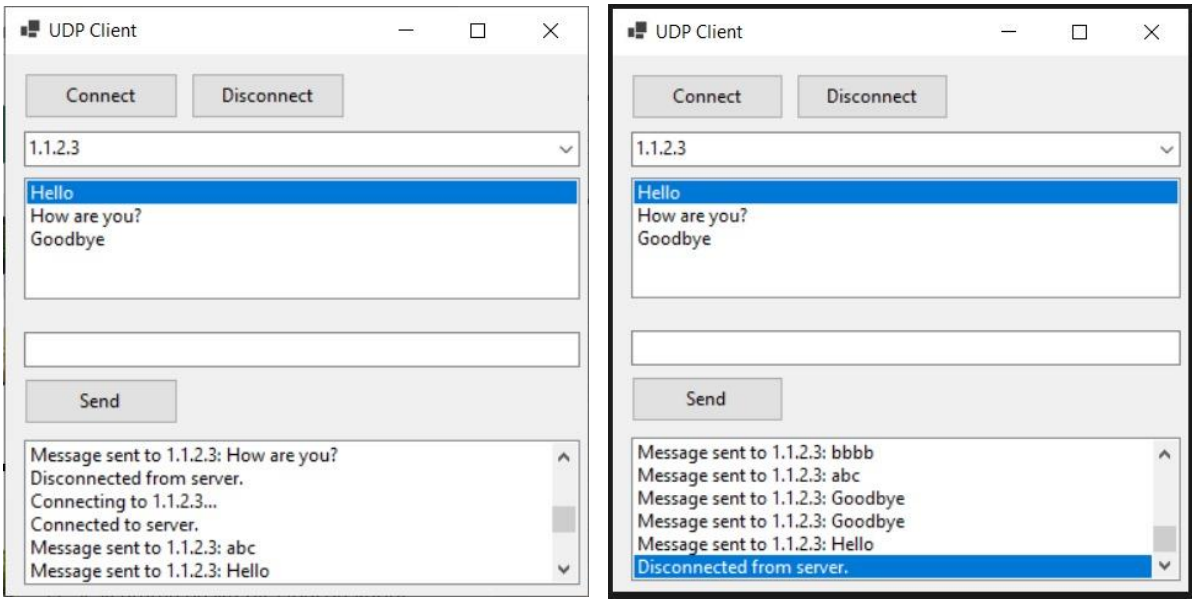

Figure 11. UDP client

In fact, UDP protocol is connectionless. In the developed application, it was actually developed as if two devices were connected. The differences between UDP and TCP can be seen in Table 3.

Table 3. UDP and TCP protocols

Feature	UDP Protocol	TCP Protocol
Connection	Datagram based (connectionless)	Connection-oriented
Message Delivery	Unreliable	Reliable
Speed	Faster since no handshake	Slower due to overhead
Use Cases	Real time applications	File transfer, HTTP, email

1.5 Oscilloscope and Waveform Generator Control Interface

An application was created using C# to control the KEYSIGHT InfiniiVision MSOX3104T Mixed Signal Oscilloscope and KEYSIGHT 33500B Waveform Generator from a computer. SCPI commands were used to create this application.

SCPI stands for Standard Commands for Programmable Instruments. It is a standard command protocol used to control and communicate with test and measurement devices. It is used to communicate with devices such as oscilloscopes, multimeters, signal generators, power supplies. It uses ASCII-based text commands and can operate over GPIB, USB, RS232, or Ethernet. Being an ASCII-based syntax makes it easy to read and debug. These commands are generally in the form of

<Subsystem>:<Command> <Parameter>

The developed application uses USB method for communication with the oscilloscope. It uses LAN method for communication with the waveform generator. In Figure 12, different code sections of the application can be seen. The logic used in creating these sections will be discussed.

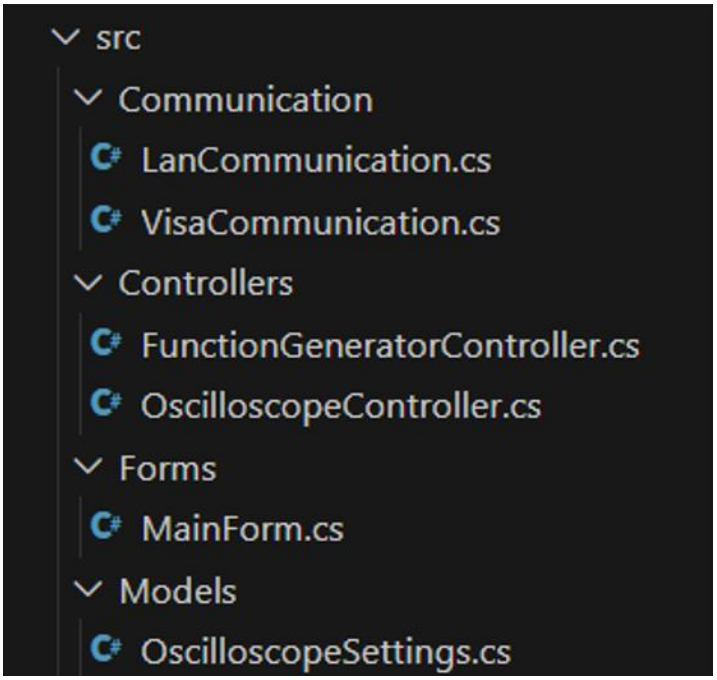

Figure 12. Application codes

LAN Communication

The *LanCommunication.cs* class provides a simple communication mechanism for sending and receiving SCPI commands over TCP/IP. This class makes communication with an waveform

generator possible. The class manages the opening of a TCP connection, sending a command, reading the response, and the proper release of resources.

- **Structure and Members of the Class:**

  The class is used to exchange SCPI commands with the waveform generator over TCP/IP.

  It is defined in the Keysight33500BApp namespace. Implement the *IDisposable* interface. This interface is used to manage resources. It is usually used to clean up unmanaged resources. Class basically communicates using a *TcpClient* and a *NetworkStream* attached to it. There are also fields that keep track of the IP address, port, and connection status.

```
public class LanCommunication : IDisposable

{

    private TcpClient? _tcpClient;

    private NetworkStream? _stream;

    private readonly string _ipAddress;

    private readonly int _port;

    private bool _connected;



    public LanCommunication(string ipAddress, int port = 5025)

    {

        _ipAddress = ipAddress ?? throw new
ArgumentNullException(nameof(ipAddress));

        _port = port;

        _connected = false;

    }

    // Other methods…

}
```

TCP connection is established with *_tcpClient* and the *TcpClient* object is created to maintain it. *NetworkStream* object is created with _stream, where TCP client data is sent and received. *_ipAdress* and *_port* hold the IP address and port number of the waveform generator. The default port is 5025. *_connected* indicates whether a connection is established to the device.

- **Constructor:**

Initializes the connection parameter required to communicate with the device.
The IP address parameter is mandatory. If null is sent, it throws "*ArgumentNullException*". The port number is optional and 5025 is used by default. Initially, the *_connected* value is set to false.

```
public LanCommunication(string ipAddress, int port = 5025)
{
   _ipAddress = ipAddress ?? throw new ArgumentNullException(nameof(ipAddress));
   _port = port;
   _connected = false;
}
```

- **Establishing a Connection: ConnectAsync()**

It establishes a TCP connection with the device asynchronously.
If the connection is already established, that is, if *_connected = True*, the method does not need to reconnect. It creates a new *TcpClient* instance. It establishes a connection via the IP and port specified by the *ConnectAsync* method. If the connection is successful, the relevant *NetworkStream* is obtained. Read and write timeouts are set. The connection status is marked. If any error occurs during the connection, the *Dispose* method is called to release the resources and a detailed error message is displayed.

```
public async Task ConnectAsync()
{
   if (_connected) return;

   try
   {
      _tcpClient = new TcpClient();
      await _tcpClient.ConnectAsync(_ipAddress, _port);
      _stream = _tcpClient.GetStream();
      _stream.ReadTimeout  = 3000;
      _stream.WriteTimeout = 3000;
      _connected = true;
   }
   catch (Exception ex)
   {
```

```
        Dispose();
        throw new Exception($"Could not connect to {_ipAddress}:{_port}.
{ex.Message}", ex);
    }
}
```

- **Sending Data: WriteAsync(string command):**

Sends SCPI commands to the device.
First, it checks whether the connection is active or not. If it is not active, *InvalidOperationException* is thrown. The command to be sent is checked to see if it is empty or null. If it is null, *ArgumentException* is thrown. Since SCPI commands are generally expected to end with \n, it adds \n to the end of the command if it is not already added. The command is converted to a byte array using ASCII encoding. The byte array is written asynchronously to the network stream. The stream is then flushed so that the data can be sent immediately. If any error occurs during writing, an *IOException* is thrown along with the relevant command information.

```
public async Task WriteAsync(string command)
{
    if (!_connected || _stream == null)
        throw new InvalidOperationException("Not connected to instrument.");

    if (string.IsNullOrEmpty(command))
        throw new ArgumentException("Command cannot be null or empty.");

    try
    {
        string cmd = command.EndsWith("\n") ? command : command + "\n";
        byte[] data = Encoding.ASCII.GetBytes(cmd);
        await _stream.WriteAsync(data, 0, data.Length);
        await _stream.FlushAsync();
    }
    catch (Exception ex)
    {
        throw new IOException($"Failed to write command '{command}': {ex.Message}",
ex);
    }
}
```

- **Data Retrieval: QueryAsync(string command) and ReadLineAsync():**

  First, the command is sent using the *WriteAsync* method. Then, the *ReadLineAsync* method is called to read the response from the device line by line. A *MemoryStream* is created to accumulate incoming bytes. Bytes are read asynchronously from the network stream. The read byte is processed until the \n character is detected. If the network stream is closed unexpectedly, an *IOException* is thrown. The accumulated bytes are converted to a string using ASCII encoding. Leading and trailing whitespace characters are stripped.

```
public async Task<string> QueryAsync(string command)
{
   await WriteAsync(command);
   return await ReadLineAsync();
}

private async Task<string> ReadLineAsync()
{
   if (_stream == null)
      throw new InvalidOperationException("Stream is not open.");

   using var ms = new MemoryStream();
   byte[] buffer = new byte[1];

   while (true)
   {
      int read = await _stream.ReadAsync(buffer, 0, 1);
      if (read < 1)
         throw new IOException("Socket closed unexpectedly.");

      if (buffer[0] == (byte)'\n')
         break;

      ms.Write(buffer, 0, read);
   }

   return Encoding.ASCII.GetString(ms.ToArray()).Trim();
}
```

- **Releasing Resources: Dispose():**

  The network stream and TCP client are properly shut down and cleared from memory. If the *_stream* element is present, the *Dispose* method is called. The *_tcpClient* is closed and the references for both objects are set to null. *_connected* is updated to false.

```
public void Dispose()
{
    _stream?.Dispose();
    _tcpClient?.Close();
    _tcpClient = null;
    _stream = null;
    _connected = false;
}
```

Visa Communication

It provides communication with the oscilloscope via VISA (Virtual Instrument Software Architecture) protocol using IVI.Visa.Interop library. It performs operations such as communication, querying, writing, reading and connection verification via SCPI commands. It provides proper release of resources using *IDisposable* interface.

- **Structure and Members of the Class:**

  The basic members of the class are a string holding the VISA address, the *ResourceManager* used for the connection, the *FormattedIO488* object used to communicate with the oscilloscope, and marks the control the connection.

```
public class VisaCommunication : IDisposable
{
    private readonly string visaAddress;
    private ResourceManager? resourceManager;
    private FormattedIO488? session;
    private bool isConnected;
    private bool disposed;

    // ...
}
```

- **Constructor:**

  The constructor gets the VISA address of the oscilloscope. If an empty or null address is given, an ArgumentException is thrown. It also initializes the connection state and marks the disposion.

```
public VisaCommunication(string address)
{
   if (string.IsNullOrEmpty(address))
      throw new ArgumentException("VISA address cannot be empty", nameof(address));

   visaAddress = address;
   isConnected = false;
   disposed = false;
}
```

- **Establishing a Connection: ConnectAsync()**

The VISA ResourceManager is created asynchronously with the ConnectAsync method. The session is opened and the basic device startup commands are sent. Thus, communication is established with the oscilloscope. Errors that occur during the connection are caught. Resources are released and an exception is thrown with an error message.

```
public async Task<bool> ConnectAsync()
{
   ThrowIfDisposed();

   try
   {
      await Task.Run(() =>
      {
         Console.WriteLine("Creating VISA ResourceManager...");
         resourceManager = new ResourceManager();
         session = new FormattedIO488
         {
            IO = (IMessage)resourceManager.Open(visaAddress)
         };

         if (session.IO == null)
            throw new Exception("Failed to open session with VISA address: " + visaAddress);

         Console.WriteLine("Session opened successfully.");
         session.IO.Timeout = 5000;

         session.WriteString("*RST", true);
         session.WriteString("*CLS", true);

         isConnected = true;
```

```
        });

        return true;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Connection failed: " + ex.Message);
        Dispose();
        throw new Exception($"Failed to connect to oscilloscope: {ex.Message}", ex);
    }
}
```

- **Data Reading and Writing:**

The *QueryAsync* method sends the SCPI command to the data and reads the response from the device. In this method, first the *WriteString* operation is performed, then the response is read with *ReadString*.

```
public async Task<string> QueryAsync(string command)
{
    ThrowIfDisposed();
    EnsureConnected();

    if (string.IsNullOrEmpty(command))
        throw new ArgumentException("Command cannot be empty", nameof(command));

    try
    {
        return await Task.Run(() =>
        {
            Console.WriteLine($"Sending command: {command}");
            session!.WriteString(command, true);
            return session.ReadString();
        });
    }
    catch (Exception ex)
    {
        throw new Exception($"Failed to execute query '{command}': {ex.Message}", ex);
    }
}
```

The *WriteAsync* method sends the given command to the device asynchronously. *SendCommandAsync* simply calls the *WriteAsync* method. Thus, a single command is sent.

```
public async Task WriteAsync(string command)
{
    ThrowIfDisposed();
    EnsureConnected();

    if (string.IsNullOrEmpty(command))
        throw new ArgumentException("Command cannot be empty", nameof(command));

    try
    {
        await Task.Run(() =>
        {
            Console.WriteLine($"Writing command: {command}");
            session!.WriteString(command, true);
        });
    }
    catch (Exception ex)
    {
        throw new Exception($"Failed to write command '{command}': {ex.Message}",
ex);
    }
}

public async Task SendCommandAsync(string command)
{
    await WriteAsync(command);
}
```

The *ReadBinaryDataAsync* method is used to read data from the device in binary format. First, the relevant SCPI command is sent, then the binary data is read with *ReadIEEEBlock*.

```
public async Task<byte[]> ReadBinaryDataAsync()
{
    ThrowIfDisposed();
    EnsureConnected();

    try
    {
        return await Task.Run(() =>
        {
            session!.WriteString(":DISP:DATA? PNG", true);
```

```
            return (byte[])session.ReadIEEEBlock(IEEEBinaryType.BinaryType_UI1,
false, true);
        });
    }
    catch (Exception ex)
    {
        throw new Exception($"Failed to read binary data from oscilloscope:
{ex.Message}", ex);
    }
}
```

The *ReadResponseAsync* method reads the response received after the command sent to the device asynchronously. It basically runs the *ReadString* method.

```
public async Task<string> ReadResponseAsync()
{
    ThrowIfDisposed();
    EnsureConnected();

    try
    {
        return await Task.Run(() =>
        {
            return session!.ReadString();
        });
    }
    catch (Exception ex)
    {
        throw new Exception($"Failed to read response from oscilloscope: {ex.Message}",
ex);
    }
}
```

- **Connection Verification:**

The *VerifyConnectionAsync* method checks whether the connection to the device is active. The device ID is read by sending the "*IDN?" query. If a response is received, the connection is confirmed; otherwise, the connection is considered unsuccessful.

```
public async Task<bool> VerifyConnectionAsync()
{
    if (disposed || !isConnected || session == null)
```

```
        return false;

    try
    {
        string response = await QueryAsync("*IDN?");
        Console.WriteLine("Connection verified. Device ID: " + response);
        return true;
    }
    catch
    {
        isConnected = false;
        return false;
    }
}
```

- **Helper Methods:**

  EnsureConnected() checks if the connection to the device is active. If not, it throws *InvalidOperationException.*

```
private void EnsureConnected()
{
    if (!isConnected || session == null)
        throw new InvalidOperationException("Not connected to oscilloscope");
}
```

  ThrowIfDisposed() checks if the object has been disposed. If the object has already been disposed, it throws *ObjectDisposedException*.

```
private void ThrowIfDisposed()
{
    if (disposed)
        throw new ObjectDisposedException(nameof(VisaCommunication));
}
```

- **Releasing Resources:**

  The Dispose method closes the opened session and ResourceManager objects to free up system resources. The Dispose operation is performed once and then ensure Dispose is set to True.

```
public void Dispose()
{
```

```
    if (disposed) return;

    try
    {
        session?.IO.Close();
        session = null;
        resourceManager = null;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error during session cleanup: " + ex.Message);
    }
    finally
    {
        isConnected = false;
        disposed = true;
    }
}
```

Function Generator Controller

- **Structure and Members of the Class:**

  This class allows to set the function generator parameters using various SCPI commands on the waveform generator device. It uses the LanCommunication class for communication. A dictionary is also used to keep track of which waveform is set for which channel.

```
public class FunctionGeneratorController
{
    private readonly LanCommunication _comm;

    private readonly Dictionary<string, string> _channelShape;

    public FunctionGeneratorController(LanCommunication communication)
    {
        _comm = communication ?? throw new ArgumentNullException(nameof(communication));

        _channelShape = new Dictionary<string, string>
        {
            { "CH1", "SIN" },
            { "CH2", "SIN" }
        };
```

```
    }

    // ...
}
```

- **Initialization and Definition Methods:**

  InitializeAsync() sends basic SCPI commands to reset and clear the device. This sets the initial state of the device.

```
public async Task InitializeAsync()
{
    await _comm.WriteAsync("*RST");
    await Task.Delay(1000);
    await _comm.WriteAsync("*CLS");
}
```

  The *IdentifyAsync()* method sends the "*IDN?" query to get the device's identity information. The device's response is obtained through this method.

```
public async Task<string> IdentifyAsync()
{
    return await _comm.QueryAsync("*IDN?");
}
```

- **Helper Methods: Check Waveform Properties:**

  Some SCPI commands may be valid only for certain waveforms. Therefore, the following helper methods check which features the current waveform supports.

```
private bool SupportsPhase(string shape)
    => shape == "SIN" || shape == "SQU" || shape == "RAMP" || shape == "PULS";

private bool SupportsPulseEdges(string shape)
    => shape == "PULS";

private bool SupportsSquareDuty(string shape)
    => shape == "SQU";

private bool SupportsRampSymmetry(string shape)
    => shape == "RAMP";
```

- **Basic Parameter Adjustment Methods:**

SetWaveformAsync() sets the selected waveform on the specified channel using the SCPI command. After the command is sent, the dictionary is updated to show which waveform is being set.

```
public async Task SetWaveformAsync(string channel, string shape)
{
   shape = shape.ToUpperInvariant();
   if (channel == "CH1")
   {
      await _comm.WriteAsync($":SOURce1:FUNCtion {shape}");
   }
   else if (channel == "CH2")
   {
      await _comm.WriteAsync($":SOURce2:FUNCtion {shape}");
   }
   else
   {
      throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
   }

   _channelShape[channel] = shape;
}
```

This is how the parameters are set. The parameter codes used can be seen in Table 4.

Table 4. Parameter codes

Function	Description	Supported Waveforms	SCPI Command
SetWaveformAsync	Sets the waveform shape for the selected channel.	All waveforms	:SOURceX:FUNCtion {shape}
SetFrequencyAsync	Sets the frequency for the selected channel.	All waveforms	:SOURceX:FREQuency {frequency}
SetAmplitudeAsync	Sets the amplitude for the selected channel.	All waveforms	:SOURceX:VOLTage {amplitude}

SetOffsetAsync	Sets the voltage offset for the selected channel.	All waveforms	:SOURceX:VOLTage:OFFSet {offset}
SetPhaseAsync	Sets the phase value if the current waveform supports phase adjustment.	SIN, SQU, RAMP, PULS	:SOURceX:PHASe {phase}
SetSquareDutyCycleAsync	Sets the duty cycle for square waves.	SQU	:SOURceX:FUNCtion:SQUare:DCYCle {dutyCycle}
SetRampSymmetryAsync	Sets the symmetry parameter for ramp waves.	RAMP	:SOURceX:FUNCtion:RAMP:SYMMetry {symmetry}
SetPulseWidthAsync	Sets the pulse width for pulse waveforms.	PULS	:SOURceX:FUNCtion:PULSe:WIDTh {width}
SetPulseLeadingEdgeAsync	Sets the leading edge duration for pulse waveforms.	PULS	:SOURceX:FUNCtion:PULSe:TRANsition:LEADing {leadEdge}
SetPulseTrailingEdgeAsync	Sets the trailing edge duration for pulse waveforms.	PULS	:SOURceX:FUNCtion:PULSe:TRANsition:TRAiling {trailEdge}
SetNoiseBandwidthAsync	Sets the noise bandwidth.	NOIS	:SOURceX:FUNCtion:NOISe:BWIDth {bandwidth}

Afterwards, the output is opened and closed on the specified channel with the *EnableChannelAsync()* and *SetOutputStateAsync()* functions. The *"$":OUTPut1:STATe {onOff}"* code is used for this. In *the SendSoftwareTriggerAsync()* function, the trigger

mode is activated with the *":TRIGger:IMMediate*" code. Finally, the *DisposeAsync()* method closes the device's outputs and frees the communication object's resources.

Oscilloscope Controller

It allows controlling the oscilloscope. The code logic is similar to that explained above.

- **Horizontal Scale (Timebase) Adjustment:**

  Used to adjust the timebase scaling of the oscilloscope. The SCPI commands used can be seen in Table 5. In Figure 13, the GUI section created for this section can be seen.

Table 5. Timebase scaling commands

Function	Description	SCPI Command
SetTimebaseScaleAsync	Sets the timebase scale per division	:TIMebase:SCALe <secondsPerDivision>
GetTimebaseScaleAsync	Queries the current timebase scale	:TIMebase:SCALe?
SetTimebaseReferenceAsync	Sets the horizontal reference point	:TIMebase:REFerence <LEFT/CENTER/RIGHT>

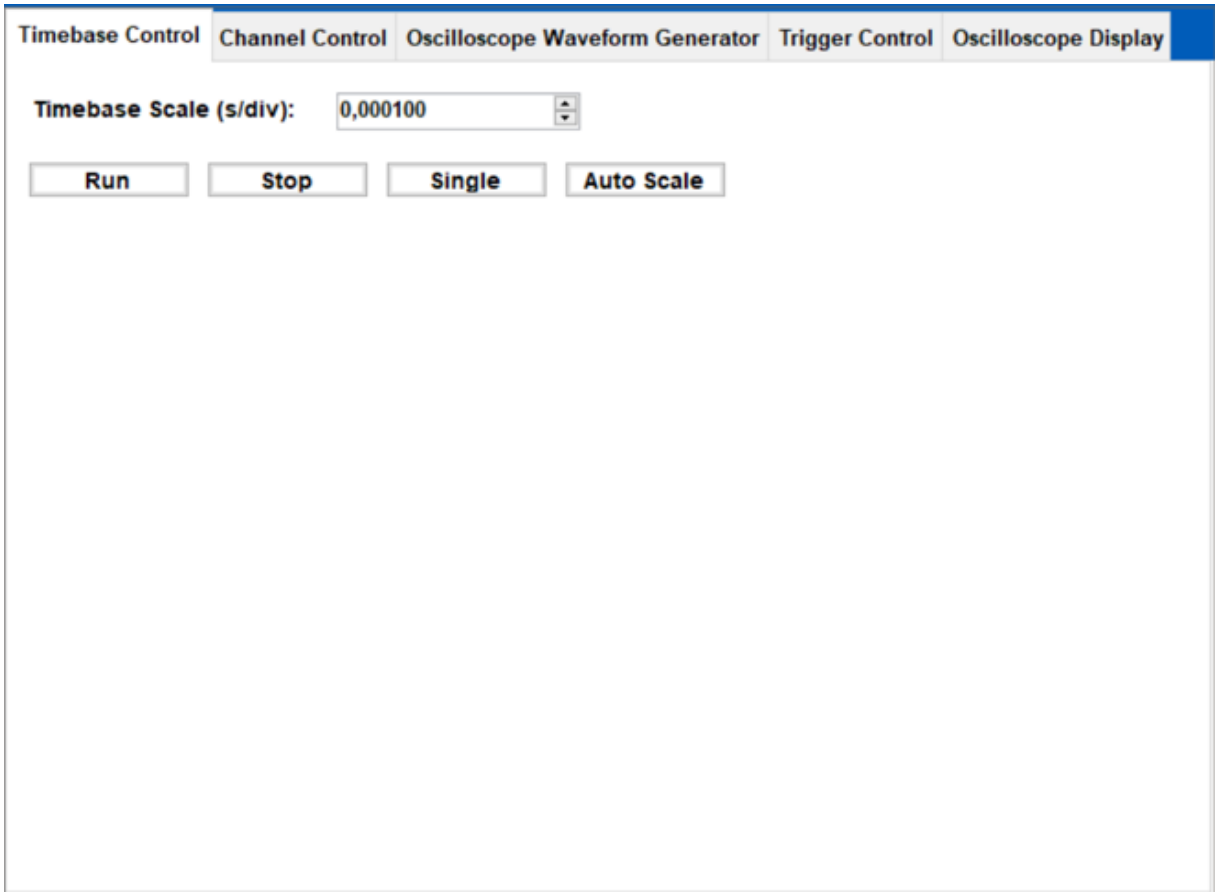

Figure 13. Osilloscope timebase control panel

- **Vertical Scale (Voltage) Adjusment:**

  Used to adjust the voltage scaling of the oscilloscope. The SCPI commands used can be seen in Table 6. In Figure 14, the GUI section created for this section can be seen.

Table 6. Voltage scaling commands

Function	Description	SCPI Command
SetVerticalScaleAsync	Sets the vertical scale	:CHANnelX:SCALe <voltsPerDivision>
GetVerticalScaleAsync	Queries the current vertical scale	:CHANnelX:SCALe?
SetVerticalOffsetAsync	Sets the vertical offset	:CHANnelX:OFFSet <offsetVolts>
SetChannelStateAsync	Enables or disables the display	:CHANnelX:DISPlay <ON/OFF>

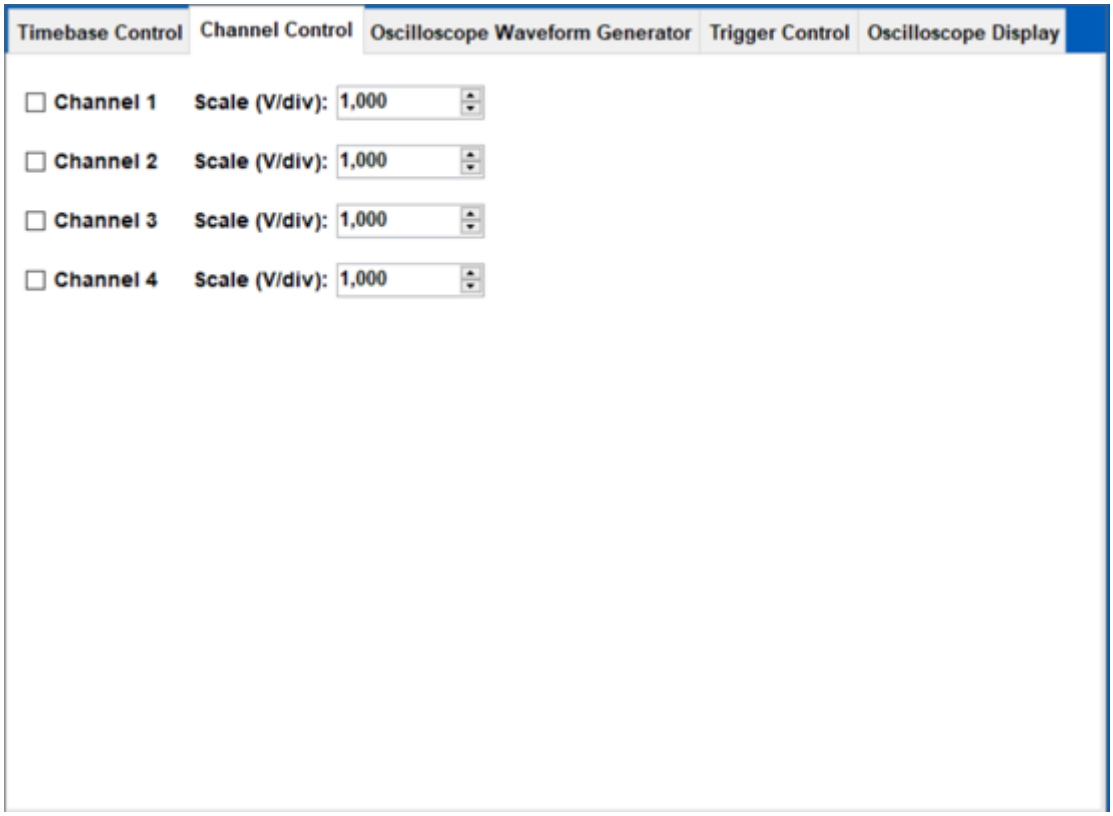

Figure 14. Osilloscope voltage control panel

- **Waveform Generator Commands :**

  This part allows the oscilloscope to generate signals with wave gen mode. The SCPI commands used can be seen in Table 7.

Table 7. Waveform generator commands

Function	Description	SCPI Command
ConfigureWaveformGeneratorAsync	Sets the waveform type	:WGEN:FUNCtion <waveformType>
	Sets the frequency	:WGEN:FREQuency <frequency>
	Sets the amplitude	:WGEN:VOLTage <amplitude>
	Sets the DC offset	:WGEN:VOLTage:OFFSet <offset>
	Sets the duty cycle for square waves	:WGEN:FUNCtion:SQUare:DCYCle <dutyCycle>
	Sets the symmetry parameter for ramp waves	:WGEN:FUNCtion:RAMP:SYMMetry <symmetry>
	Sets the pulse width in seconds	:WGEN:FUNCtion:PULSe:WIDTh <widthSec>
	Enables the output of the waveform generator	:WGEN:OUTPut ON

- **Measurement Commands:**

  This part allows measuring the parameters of signals produced by a waveform generator or oscilloscope. The SCPI commands used can be seen in Table 8. In Figure 15, the GUI section created for this section can be seen. Sample code usage is as follows:

  public async Task<double> MeasureVppAsync(int channel)

  {

  ValidateChannel(channel);

  await _communication.WriteAsync($":MEASure:SOURce CHANnel{channel}");

  string response = await _communication.QueryAsync(":MEASure:VPP?");

  return double.Parse(response, CultureInfo.InvariantCulture);

  }

  With the code "$":MEASure:SOURce CHANnel{channel}" the measurement of the desired channel is taken and printed. Then the desired parameter is asked. For example, with (":MEASure:VPP?) the Vpp value is queried and the answer is read.

Table 8. Measurement commands

Function	Description	SCPI Command
MeasureVppAsync	Measures the peak-to-peak voltage	:MEASure:VPP?
MeasureVrmsAsync	Measures the RMS voltage	:MEASure:VRMS?
MeasureFrequencyAsync	Measures the frequency	:MEASure:FREQuency?
MeasurePeriodAsync	Measures the period of the waveform	:MEASure:PERiod?
MeasureAmplitudeAsync	Measures the amplitude	:MEASure:VAMPlitude?
MeasureMeanVoltageAsync	Measures the mean voltage	:MEASure:VAVerage?
MeasurePhaseAsync	Measures the phase	:MEASure:PHASe?
MeasureDutyCycleAsync	Measures the duty cyle	:MEASure:DUTYcycle?
MeasurePulseWidthAsync	Measures the pulse width	:MEASure:PWIDth?
MeasureRiseTimeAsync	Measures the rise time	:MEASure:RISetime?
MeasureFallTimeAsync	Measures the fall time	:MEASure:FALLtime?
MeasureOvershootAsync	Measures the overshoot	:MEASure:OVERshoot?
MeasurePreshootAsync	Measures the preshoot	:MEASure:PREShoot?
MeasureSlewRateAsync	Measures the slew rate	:MEASure:SLEWrate?
MeasureSignalDelayAsync	Estimates the signal delay based on period and rise time measurements	Combination of :MEASure:PERiod? and :MEASure:RISetime?
MeasureSymmetryAsync	Calculates symmetry as (rise time / period) * 100	Derived from :MEASure:RISetime? and :MEASure:PERiod?
MeasureLeadEdgeTimeAsync	Measures the leading edge time	:MEASure:TRANsition?
MeasureTrailEdgeTimeAsync	Measures the trailing edge time	:MEASure:TRANsition?
MeasureBandwidthAsync	Measures the bandwidth	:MEASure:BANDwidth?

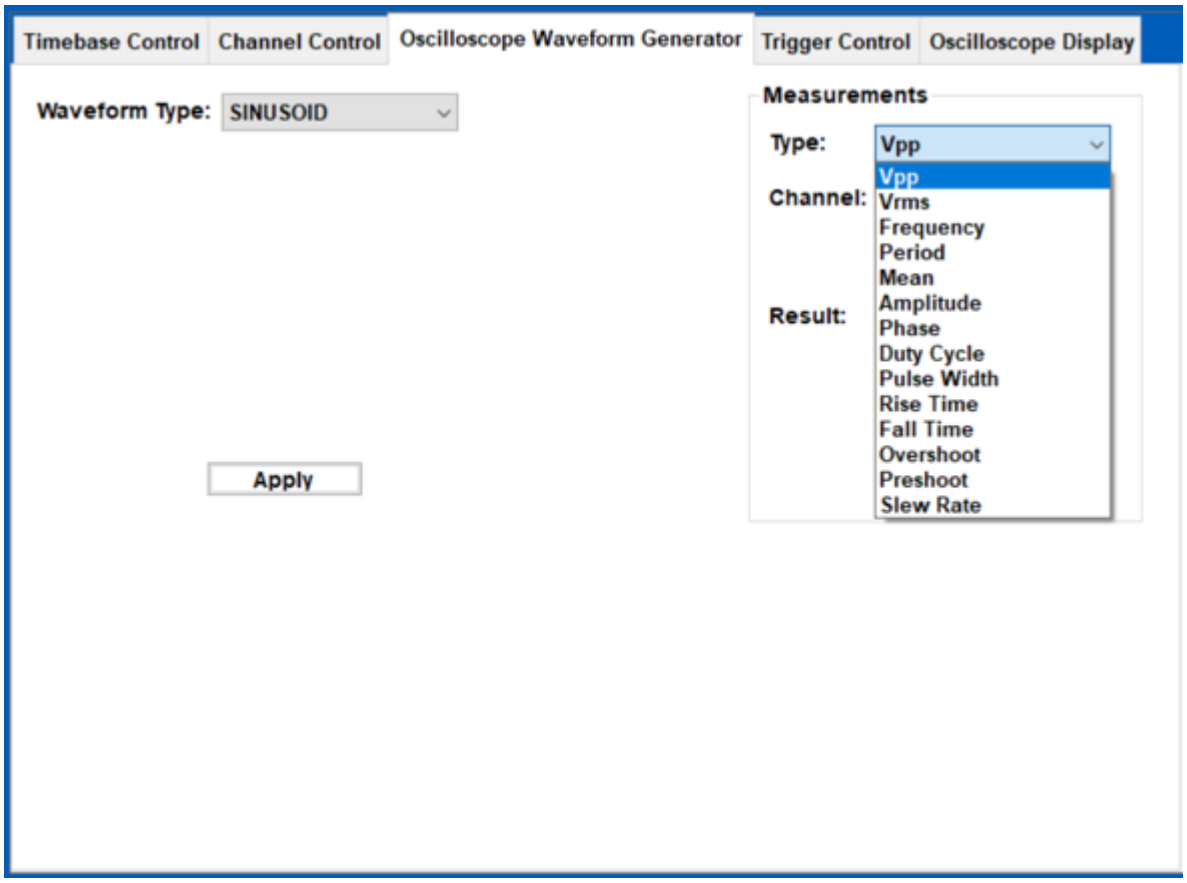

Figure 15. Osilloscope measurements panel

- **Trigger Commands:**

  It allows the use of the trigger mode of the oscilloscope. Helps to set different trigger modes and trigger level. The SCPI commands used can be seen in Table 9. In Figure 16, the GUI section created for this section can be seen.

Table 9. Trigger commands

Function	Description	SCPI Command
SetTriggerSweepModeAsync	Sets the oscilloscope sweep mode	:TRIGger:SWEep <AUTO/NORMal/SINGle>
SetTriggerEdgeSourceAsync	Sets the trigger source	:TRIGger:EDGE:SOURce <source>
SetTriggerEdgeSlopeAsync	Sets the edge slope for triggering	:TRIGger:EDGE:SLOPe <POSitive/NEGative>
SetTriggerLevelAsync	Sets the trigger level	:TRIGger:EDGE:LEVel <level>
SetTriggerTypeAsync	Sets the trigger mode	:TRIGger:MODE <EDGE/PULSe/PATTern>

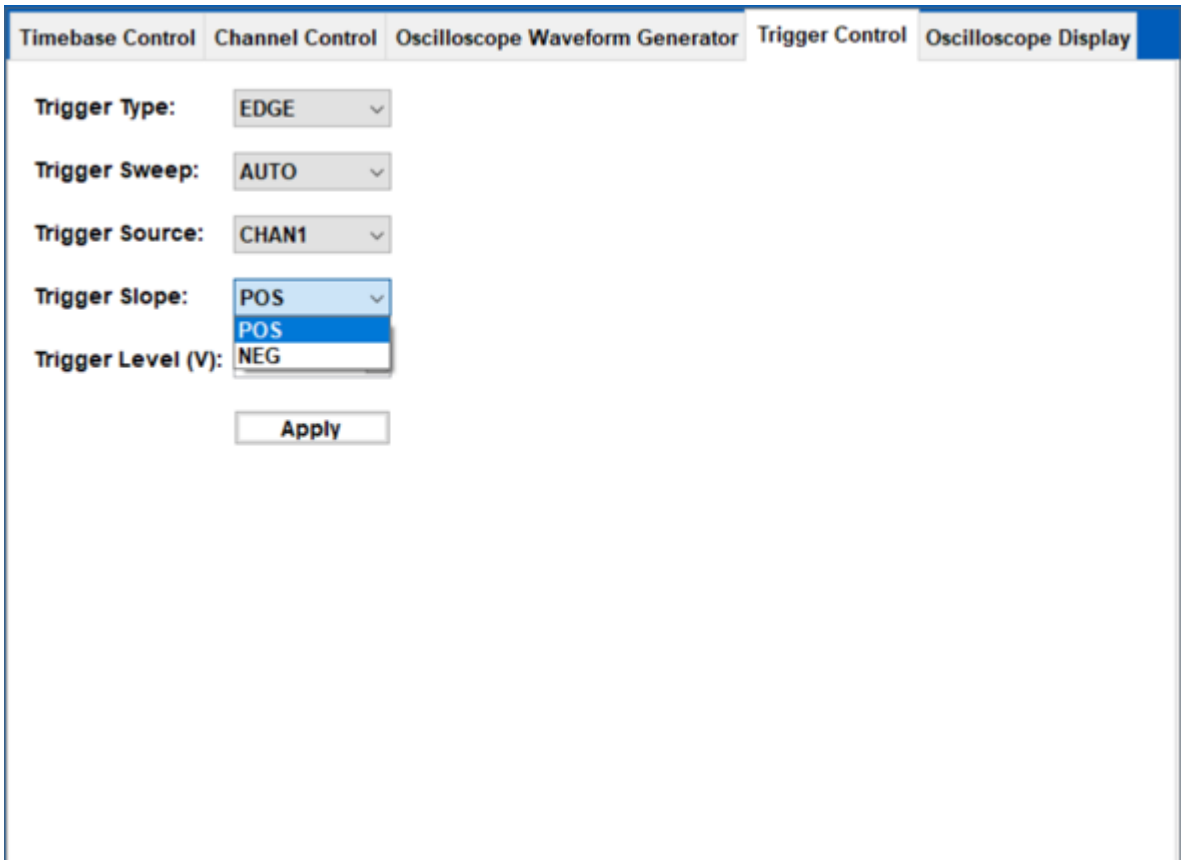

Figure 16. Osilloscope trigger control panel

As a result, it allows the oscilloscope to be controlled by the above commands. It has functions such as scaling adjustment, signal generation, and measuring the parameters of the signal. In addition, it can take a screenshot of the oscilloscope screen with *:DISP:DATA? PNG*. It can also use the autoscale feature with *:AUTOSCALE*.

Mainform

In this section, the features of all other classes created are brought together and used. In addition, a GUI was created for easy use of all these features. The necessary data was obtained from the oscilloscope and a display feature as close to real time as possible was added. Apart from this, a completely automatic test procedure was developed and added as an additional tab.

- **General Structure and UI Layout:**

  It creates all UI components with the InitializeComponent() method. After setting the main window title, dimensions, background color and position in this method, a TabControl with two main tabs is created. "Main Control" tab: This is the area where the main control panel is located. "Test Screen" tab: This is the area where test results are displayed, test logs and test statistics are shown. A general view of the application can be seen in Figure 17.

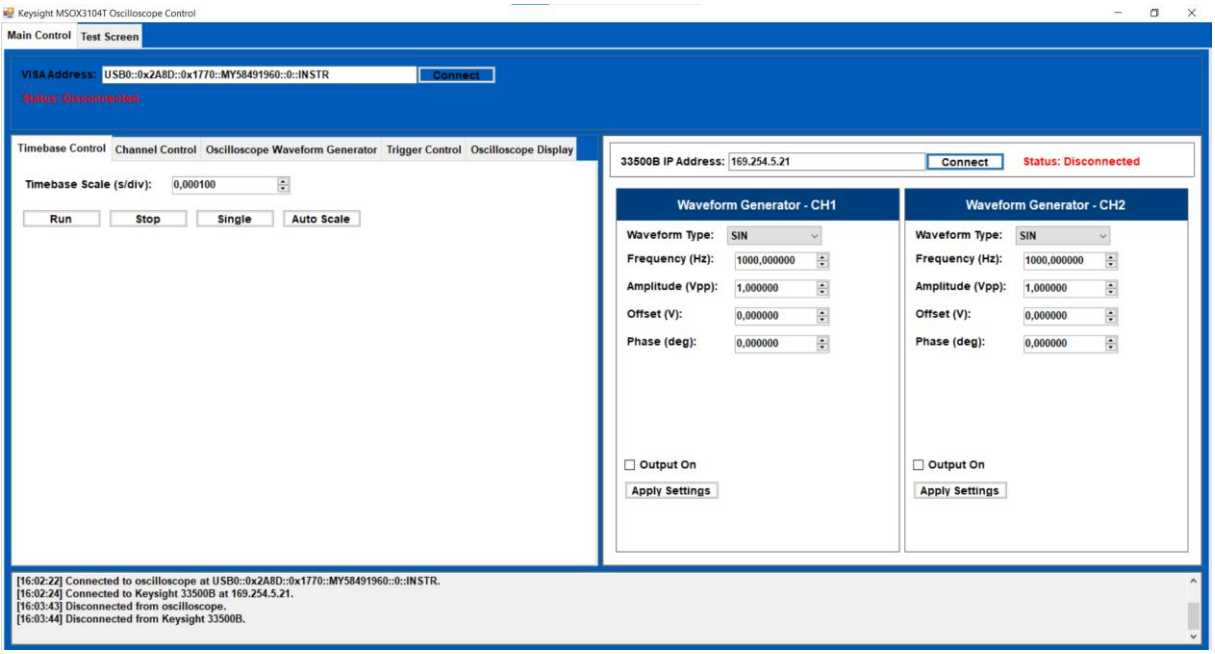

Figure 17. Interface GUI

In the Main control tab, there is the oscilloscope connection panel at the top, the split container in the middle, and the status panel at the bottom. The right side of the screen contains the 33500B waveform generator panel, and the left side contains the MSOX3104T oscilloscope panel.

- **Connection Management:**

  The user establishes a connection by entering the VISA address in the section in Figure 18 and clicking the connect button. The *ConnectButtonClick()* method is called. Using the entered VISA address, an instance of *VisaCommunication* is created and connected asynchronously. Then, the *OscilloscopeController* object is created and made ready to use the SCPI commands defined in the previous codes. If the connection is successful, the address box is disabled, and the color and text of the status label are updated.

Figure 18. VISA connection

Similar logic is applied to the function generator. The LAN connection is established using the section in Figure 19. Here, the IP is taken from the "33500B IP Address" box and the asynchronous connection is established via the *LanCommunication* instance. Once the connection is established, the Function Generator Controller is created, initialize commands are sent, and status labels are updated.

Figure 19. LAN connection

- **Parameter Setting and Command Execution:**

  For oscilloscope, time scale, channel scale and trigger settings are made via various buttons and controls such as NumericUpDown and ComboBox on the MainForm. For these settings, the values in the relevant controls are converted to SCPI commands via OscilloscopeController using asynchronous methods and sent depending on user changes. For waveform generator, waveform type, frequency, amplitude, offset, and extended parameters are set via the user interface. These values are read from the relevant controls and applied to the device by sending SCPI commands via the FunctionGeneratorController methods. The parameter panel is dynamically created according to the selected waveform type. For this, the *UpdateParameterPanel()* and *UpdateWaveformParameterInputs()* methods are used. In these methods, NumericUpDown controls are added according to the relevant waveform type; the minimum, maximum, increment amount, and default values of each control are determined.

- **Real Time Display:**

  The Display section is for displaying the waveform data received from the oscilloscope in real time in a graphical form on the PictureBox. This section allows the user to visually monitor the signal. The display section can be seen in Figure 20.

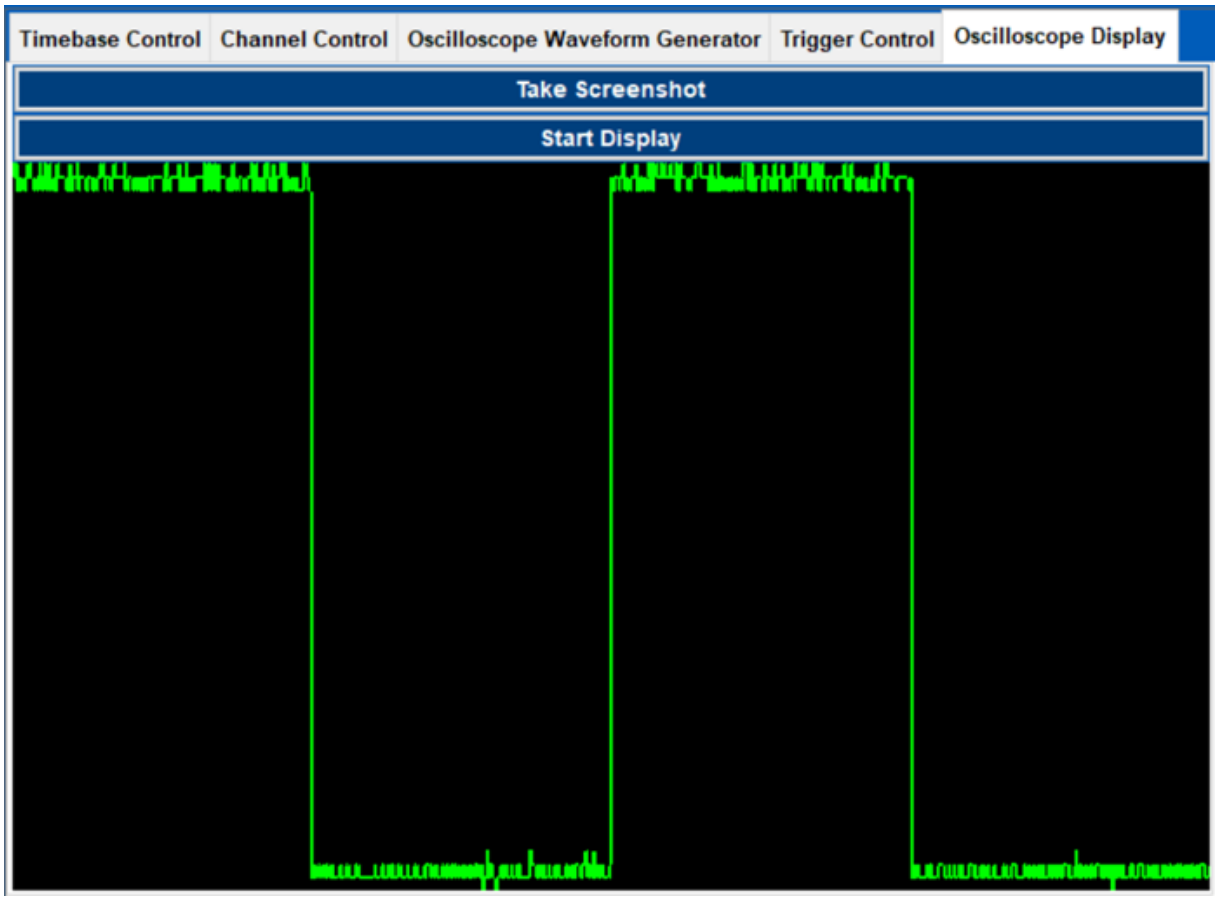

Figure 20. Display section

:DISP:DATA? PNG code can be used to capture an image of the oscilloscope screen. However, this image will not be instantaneous, it will be updated approximately every 5-10 seconds. Instead, a different method was applied for a more real-time display.

First, the necessary data is collected. For this, the *UpdateOscilloscopeDisplay()* method runs continuously in an asynchronous loop (as long as the display is active). This method first sends the RUN command to the oscilloscope. Then, the desired channel is selected as the measurement source. Then, the data format is set to ASCII and the number of data points is determined. Then, the X increment *(:WAVeform:XINCrement?),* X origin *(:WAVeform:XORigin*?), Y increment *(:WAVeform:YINCrement*?), Y origin *(:WAVeform:YORigin*?), Y reference (:*WAVeform:YREFerence*?) values are queried via SCPI commands. Finally, the data is received by sending the *:WAVeform:DATA?* command containing the waveform data. The received data is in comma-separated ASCII format. This data is obtained as a string. If the received string data is not empty or erroneous, it is converted to an array by separating it with a comma. Each data point is first converted to a double; during the conversion, the reference value from the device, the y-increment and y-origin are used to calculate the actual voltage value. Thus, a double array called yData is obtained. At the same time, an xData array is created using the initial value (xOrigin) and increment (xIncrement) information for the x-axis. xData represents the temporal location of each data point. Finally, the graphic is drawn. For this, the DisplayWaveform(double[] xData, double[] yData) method creates a Bitmap using the obtained x and y data. The width and height of the PictureBox are taken. A

new Bitmap object is created. Drawing is done on the Bitmap using the Graphics object. First, the screen is cleared to black (Clear(Color.Black)). The line color and thickness are determined with a pencil. A line is drawn between two consecutive points in the xData and yData arrays. The x coordinates are normalized from the beginning of the data and calculated according to the Bitmap width. The y coordinates are normalized according to the voltage values of the signal. Here, the signal height is scaled to the Bitmap height using the minimum and maximum values of yData. Since the y axis works in the opposite direction, it is subtracted from the height after the calculation is made. After all the data points are drawn, the created Bitmap is assigned to the Image property of the PictureBox. The UpdateOscilloscopeDisplay() method loops every 50 ms (await Task.Delay(50)) to update the PictureBox. This provides a near real-time signal display to the user. If the display is closed, the loop stops and the Image property of the PictureBox is cleared. It can also be saved as a screenshot at any time as can be seen in Figure 21.

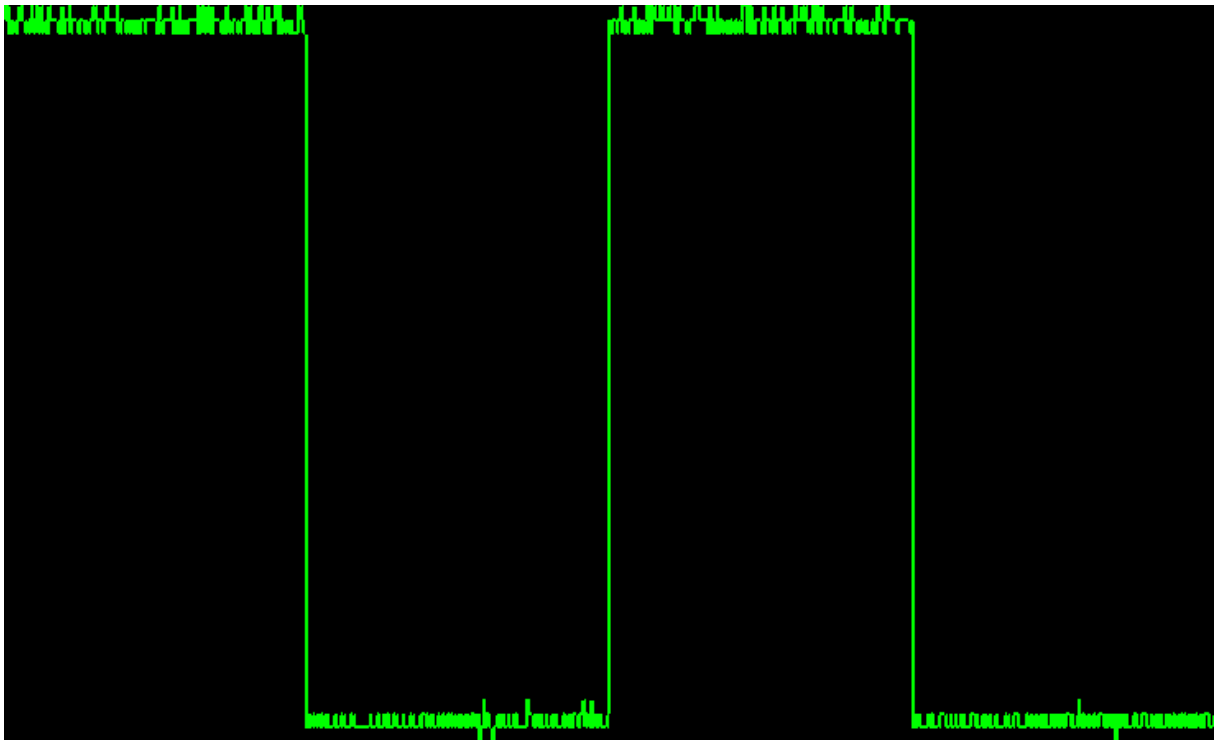

Figure 21. Screenshot of display

- **Fully Automatic Testing:**

The purpose of the test section is to measure the signals generated by the function generator for different wave types via the oscilloscope and check whether parameters such as frequency, amplitude, duty cycle or pulse width are within a certain tolerance of the expected values. During these processes, in each test step; settings are applied, measurements are taken, measured values are compared and results are logged.

RunImprovedTest method is used for testing. In this method, first the initial preparations are made. The test results and logs for the selected channel, the relevant RichTextBox are cleared. A Stopwatch (_testStopwatch) and a Timer (_testTimer) are started. In this way, the duration of the test is updated on the screen. First, the oscilloscope connection is checked. If there is no connection, a connection is tried to be established similar to the ConnectButtonClick() method. Then, the function generator connection is also checked; if there is no connection, a connection is also provided for the function generator. Then the test parameters are determined. The wave types to be used in the test, frequency steps, amplitude steps, duty cycle steps for square waves and pulse width steps for pulse waves are defined as fixed sequences. It is determined on which channel the test will be performed. In addition, the dictionary named _paramStats is reset. This dictionary will keep track of the total number of measured parameters for each wave type and the number of successful measurements. Then the test cycle is started. A separate outer loop is run for each wave type. For each wave type, an inner loop is started on the defined frequency steps. For each frequency value, another inner loop is run on the defined amplitude steps. If the wave type is square (SQU), a separate loop is run on the duty cycle steps after the frequency and amplitude are set. If the wave type is pulse (PULS), a separate inner loop is applied for the pulse width steps. During testing, the following happens:

According to the selected wave type, the relevant parameters are set via the function generator. For this process, the relevant FunctionGeneratorController methods are called, such as SetFrequencyAsync(). Depending on the test step, the time and voltage scales are set on the oscilloscope. The time scale is set with the :TIMebase:SCALe command and the voltage scale is determined with the :CHANnelX:SCALe command. These settings are important for the accuracy of the test. Then, the MeasureWaveform() method is called. In this method, the frequency, amplitude and, if necessary, duty cycle or pulse width measurements are taken from the oscilloscope for the relevant channel. The measured values are compared with the expected values. The comparison is made with a tolerance such as ±15% using the CheckWithinTolerance() method. Each measurement result is determined as "PASS" or "FAIL". The measurement results are added to the log screen (testResultsBox or testResultsBoxCH2) and also to the _paramStats dictionary. After each test step, the measurement results are generated as a text message and written to the screen using the AppendTestResult() method. This method displays the results up-to-date by adding a timestamp to the message and coloring it accordingly. When all wave types, frequency, amplitude and special parameter steps are completed, the test cycle ends. Stopwatch is stopped, Timer is stopped and total test time is calculated and added to the result text. All test results obtained are transferred to the relevant TextBoxes with summary information. All these operations are applied separately for CH1 and CH2 of the waveform generator. The test section can be seen in Figures 22 and 23. Also, the tests and results conducted for both channels can be seen.

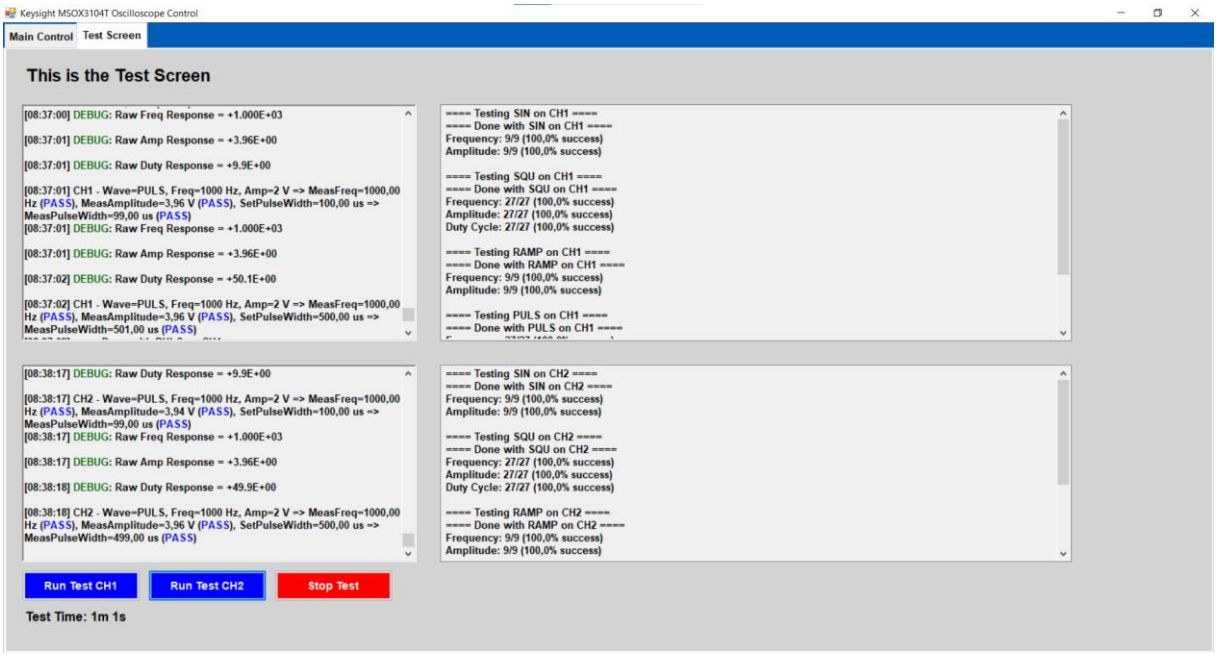

Figure 22. Test GUI

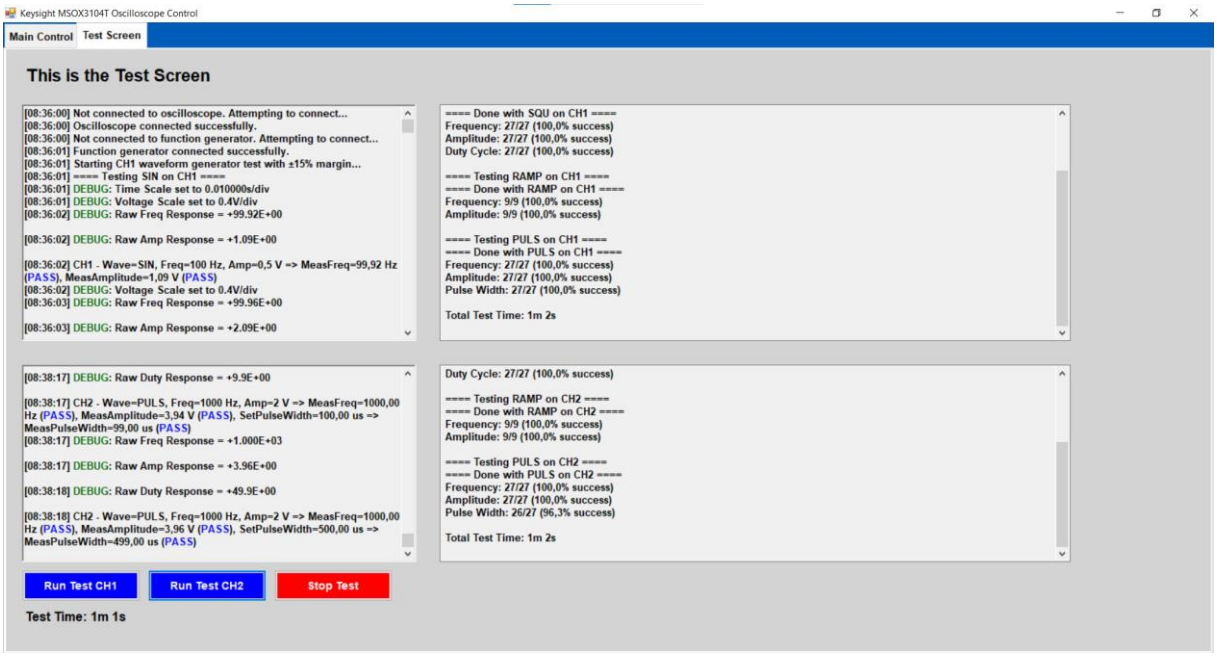

Figure 23. Test GUI

1.6 UPort RS-232 Communication Application

An interface was developed to provide communication over RS-232 using the UPort 1250 device. With this interface, communication is provided between two ports of the UPort 1250 device, or between the UPort 1250 device and another device such as oscilloscope.

The development process of the application started with the need to provide reliable data exchange via RS-232 with the UPort 1250 device. In order to implement basic serial port communication functions, the developers created a structure that performs data sending, receiving and error management operations via COM1 and COM2 ports using the SerialPort class of the .NET Framework. At this stage, the basic data communication mechanism of the application was established and how the communication between the two ports would be provided and which messages would be sent and received were determined.

In the following stages, visual components were developed using Windows Forms in order to create a user-friendly interface. The developed interface includes controls such as separate group boxes for each port, send and receive text boxes, status labels, and connection open/close buttons. In addition, a global log screen and areas where test summaries are displayed were added, providing the user with instant information about port connection statuses and sent and received data. In this way, a comprehensive user interface was created for both manual data exchange and automatic testing processes.

The test automation part was one of the most important development stages of the application. In this stage, a test procedure was developed in two phases to check whether certain test messages were sent between two ports and received correctly. In the first phase, it was checked whether the messages sent from Port1 were received correctly by Port2, while in the second phase, it was ensured that the messages sent from Port2 were received by Port1. In both stages, the message sent for each test step was compared with the message received, and if a match was found, the step was reported as “PASS”, otherwise “FAIL”. During these processes, special event handlers were created for each step and data waiting processes were provided using TaskCompletionSource. Timeout, pause and cancellation mechanisms were also integrated asynchronously, aiming to ensure that the test processes were carried out uninterruptedly and reliably.

In addition, Stopwatch and Windows Forms Timer were used to measure the test duration and present it to the user in real time. Thus, the total duration of the test was continuously updated and the user was able to follow the test duration instantly. The results of the test steps were logged in detail and displayed on the interface in terms of both the success or failure of each step and the calculation of the overall success rates and presentation in a summary report.

First, a communication test was performed between port2 of the UPort device and the MSOX3104T oscilloscope. The sent message can be seen in Figure 24, and the response received via the oscilloscope can be seen in Figure 25. Ground (pin 5) was connected to ground from two jumpers on the oscilloscope, and the other jumper was connected to pin2 (RX). In this way, a signal was sent to the oscilloscope from port 2, and the signal was recorded to the USB.

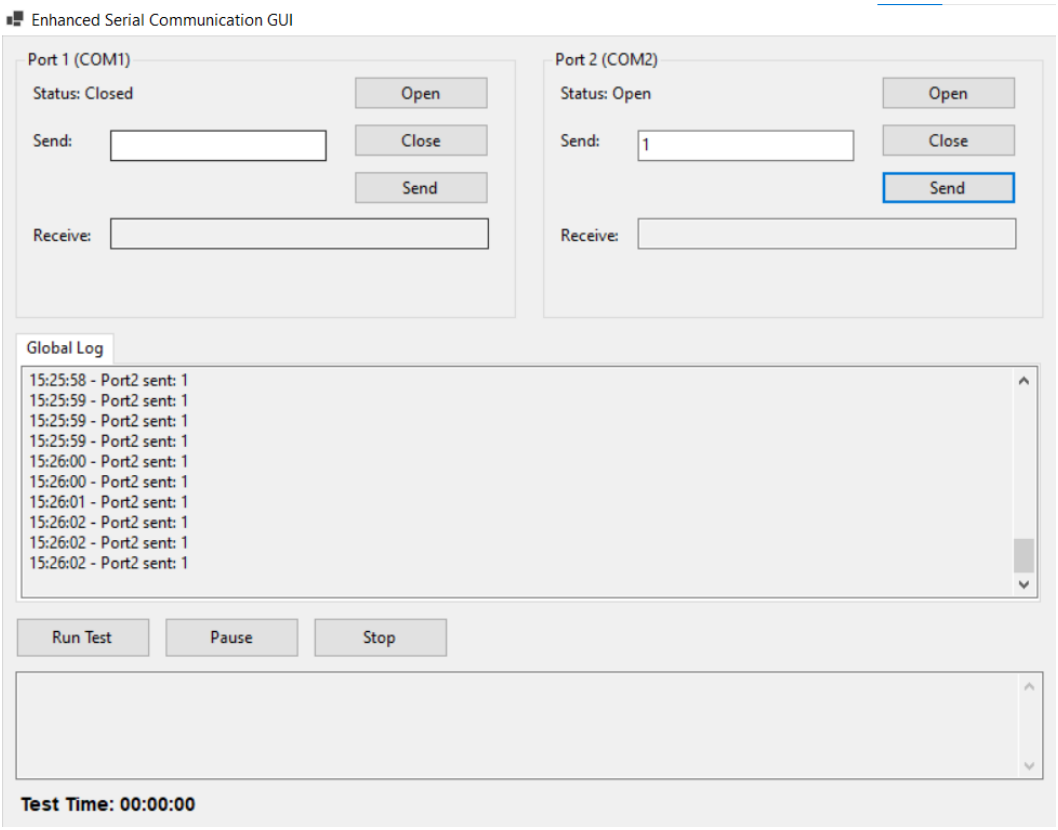

Figure 24. Message sent to the oscilloscope from port 2

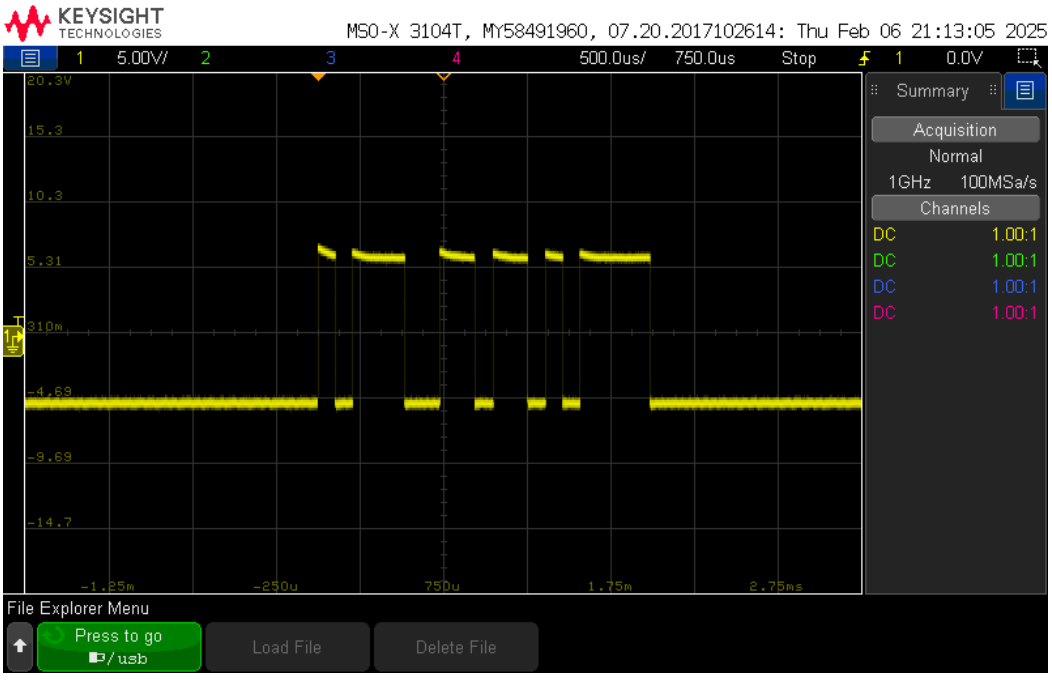

Figure 25. Message received on the oscilloscope from port 2

Later, port 1 and port 2 were connected to each other. For this, pin2 of port 1 was connected to pin 3 of port 2, pin3 of port 1 was connected to pin 2 of port 2, pin5 of port 1 was connected to pin5 of port 2. The test performed can be seen in Figure 26.

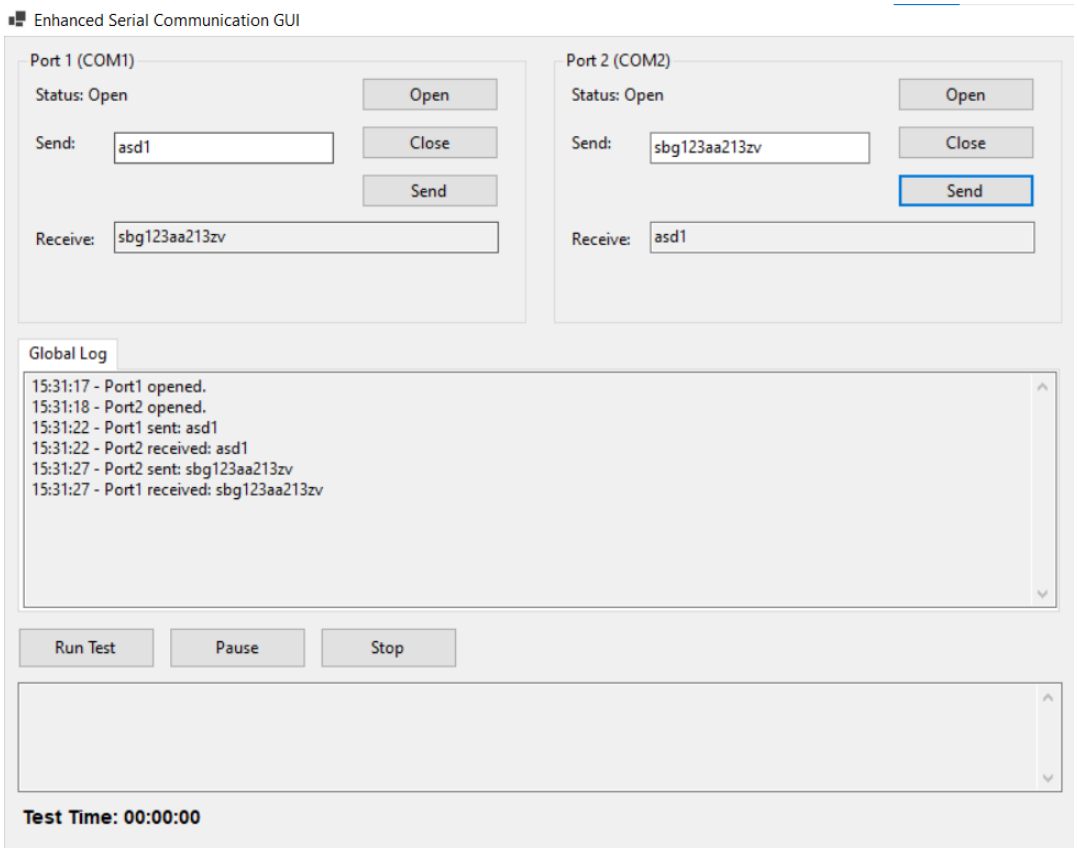

Figure 26. Port 1 and port 2 communication

The automatic test procedure can be seen in Figures 27 and 28. When the Run button is pressed, 25 messages are sent from port1 to port 2 in order and the received message is read at port2. It is checked whether they are the same and the PASS or FAIL result is printed. Then the same process is done from port2 to port 1. And as a result, the overall success rate and test duration are shown.

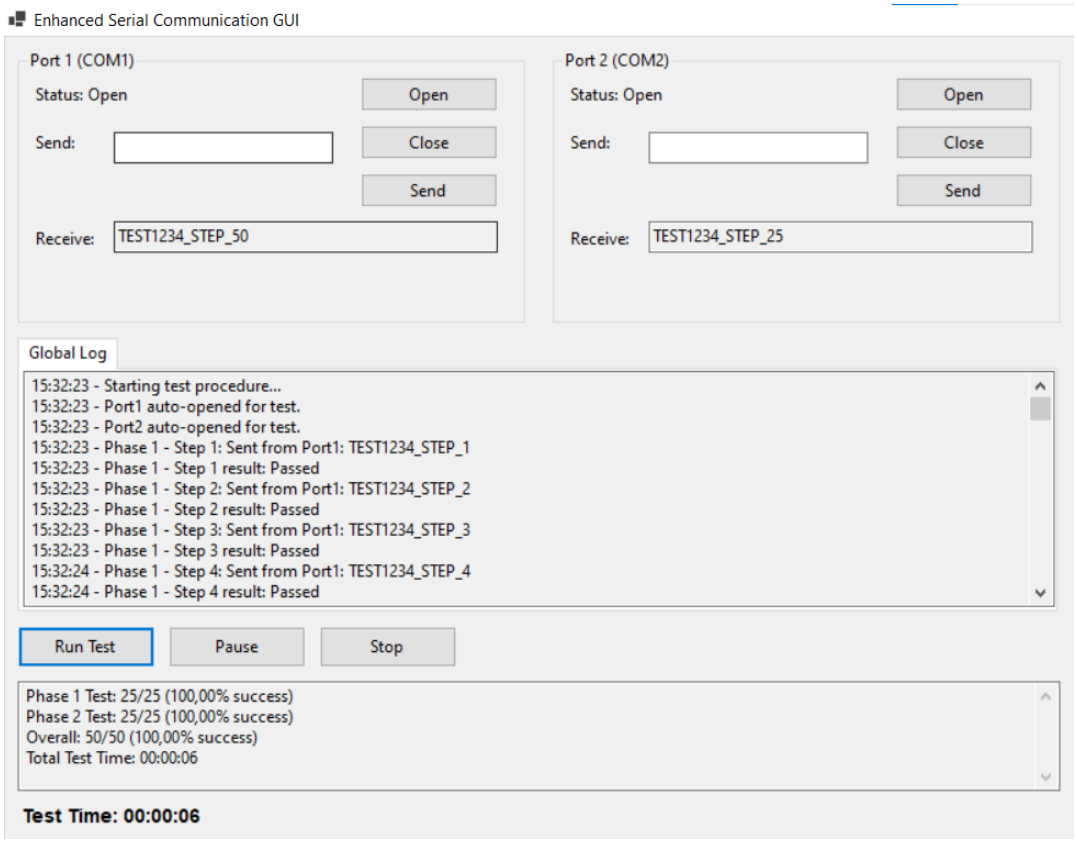

Figure 27. Automated test procedure

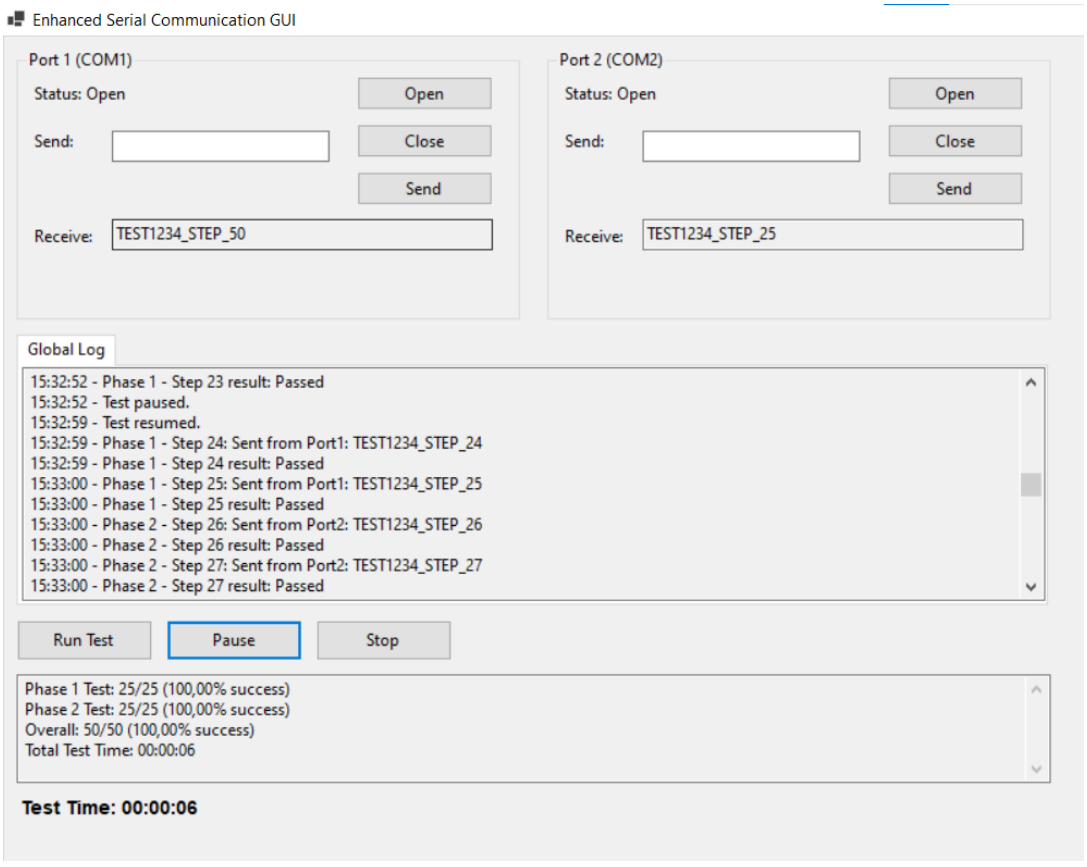

Figure 28. Automated test procedure

2. CONCLUSION

During the internship, communication protocols were studied intensively. First, widely used serial communication protocols were studied theoretically and reported. Later, theoretical work was done on OSI Model 7 layer. TCP and UDP protocols in the Transport layer were focused on. An application was made using C# for both protocols. Communication was provided via one computer with the TCP protocol application, and via two computers with the UDP protocol application. The code written and the application developed during this process were added to the report as details. Later, an application was developed that allows the control of KEYSIGHT InfiniiVision MSOX3104T Mixed Signal Oscilloscope and KEYSIGHT 33500B Waveform Generator devices from a computer. SCPI commands were used in the development of the application. A test application was developed using the functionality of the developed application. With the test application, the accuracy of the parameters of the signals generated from the waveform generator was tested using an oscilloscope and the results were printed. This entire process was fully automated. This entire application development process was also explained in detail in the report. Finally, an application was developed that enables RSR-232 communication using UPort 1250. The application was tested using only UPort and using UPort and oscilloscope. Then, a fully automated test section was developed that tests the accuracy of the UPort 1250 device. All processes were added to the report.

3. REFERENCES

[1] Moxa Inc., *UPort 1200/1400/1600 Series Datasheet v2.0*, Moxa Inc., May 17, 2023.

[2] WhatIsMyIP.com, “OSI Model.” Available: https://www.whatismyip.com/osi-model/.

4. APPENDIX

tcp_protocol\Server\ServerForm.cs

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Windows.Forms;

public partial class ServerForm : Form
{
    private TcpListener server;
    private Thread serverThread;

    public ServerForm()
    {
        InitializeComponent();
    }

    private void InitializeComponent()
    {
        this.startButton = new System.Windows.Forms.Button();
        this.logs = new System.Windows.Forms.ListBox();
        this.SuspendLayout();
        //
        // startButton
        //
        this.startButton.Location = new System.Drawing.Point(12, 12);
        this.startButton.Name = "startButton";
        this.startButton.Size = new System.Drawing.Size(100, 30);
        this.startButton.TabIndex = 0;
        this.startButton.Text = "Start Server";
        this.startButton.UseVisualStyleBackColor = true;
        this.startButton.Click += new System.EventHandler(this.startButton_Click);
        //
        // logs
        //
        this.logs.FormattingEnabled = true;
        this.logs.ItemHeight = 16;
        this.logs.Location = new System.Drawing.Point(12, 50);
        this.logs.Name = "logs";
        this.logs.Size = new System.Drawing.Size(360, 200);
        this.logs.TabIndex = 1;
        //
        // ServerForm
        //
        this.ClientSize = new System.Drawing.Size(400, 270);
        this.Controls.Add(this.logs);
        this.Controls.Add(this.startButton);
        this.Name = "ServerForm";
        this.Text = "TCP Server";
        this.ResumeLayout(false);
    }
```

```
52
53      private System.Windows.Forms.Button startButton;
54      private System.Windows.Forms.ListBox logs;
55
56      private void startButton_Click(object sender, EventArgs e)
57      {
58          serverThread = new Thread(StartServer);
59          serverThread.IsBackground = true;
60          serverThread.Start();
61          logs.Items.Add("Server started...");
62          startButton.Enabled = false;
63      }
64
65      private void StartServer()
66      {
67          server = new TcpListener(IPAddress.Any, 12345);
68          server.Start();
69
70          while (true)
71          {
72              try
73              {
74                  TcpClient client = server.AcceptTcpClient();
75                  Thread clientThread = new Thread(() => HandleClient(client));
76                  clientThread.IsBackground = true;
77                  clientThread.Start();
78              }
79              catch (Exception ex)
80              {
81                  logs.Invoke((MethodInvoker)delegate
82                  {
83                      logs.Items.Add("Error: " + ex.Message);
84                  });
85              }
86          }
87      }
88
89      private void HandleClient(TcpClient client)
90      {
91          NetworkStream stream = client.GetStream();
92          byte[] buffer = new byte[1024];
93
94          // Read client name
95          int nameBytesRead = stream.Read(buffer, 0, buffer.Length);
96          string clientName = Encoding.UTF8.GetString(buffer, 0, nameBytesRead);
97
98          logs.Invoke((MethodInvoker)delegate
99          {
100             logs.Items.Add($"Client connected: {clientName}");
101         });
102
103         try
104         {
105             while (true)
```

```
106             {
107                 int bytesRead = stream.Read(buffer, 0, buffer.Length); // Try to read data
108                 if (bytesRead == 0) break; // Client disconnected gracefully
109 
110                 string message = Encoding.UTF8.GetString(buffer, 0, bytesRead); // Decode
    the message
111                 logs.Invoke((MethodInvoker)delegate
112                 {
113                     logs.Items.Add($"Message from {clientName}: {message}");
114                 });
115 
116                 // Respond to the client
117                 string response = $"Server received: {message}";
118                 byte[] responseBytes = Encoding.UTF8.GetBytes(response);
119                 stream.Write(responseBytes, 0, responseBytes.Length);
120             }
121         }
122         catch (IOException) // Catch only disconnection-related exceptions
123         {
124             logs.Invoke((MethodInvoker)delegate
125             {
126                 logs.Items.Add($"Client disconnected: {clientName}"); // Log disconnection
    as normal
127             });
128         }
129         catch (Exception ex)
130         {
131             logs.Invoke((MethodInvoker)delegate
132             {
133                 logs.Items.Add($"Error with {clientName}: {ex.Message}"); // Log unexpected
    errors
134             });
135         }
136         finally
137         {
138             logs.Invoke((MethodInvoker)delegate
139             {
140                 logs.Items.Add($"{clientName} disconnected.");
141             });
142             client.Close(); // Ensure resources are released
143         }
144     }
145 }
146 
```

tcp_protocol\Client1\ClientForm.cs

```
using System;
using System.Net.Sockets;
using System.Text;
using System.Windows.Forms;

public partial class ClientForm : Form
{
    private TcpClient client;
    private NetworkStream stream;

    public ClientForm()
    {
        InitializeComponent();
    }

    private void InitializeComponent()
    {
        this.connectButton = new System.Windows.Forms.Button();
        this.disconnectButton = new System.Windows.Forms.Button();
        this.ipAddressBox = new System.Windows.Forms.ComboBox();
        this.defaultMessages = new System.Windows.Forms.ListBox();
        this.customMessageBox = new System.Windows.Forms.TextBox();
        this.sendButton = new System.Windows.Forms.Button();
        this.logs = new System.Windows.Forms.ListBox();
        this.SuspendLayout();
        //
        // connectButton
        //
        this.connectButton.Location = new System.Drawing.Point(12, 12);
        this.connectButton.Name = "connectButton";
        this.connectButton.Size = new System.Drawing.Size(100, 30);
        this.connectButton.TabIndex = 0;
        this.connectButton.Text = "Connect";
        this.connectButton.UseVisualStyleBackColor = true;
        this.connectButton.Click += new System.EventHandler(this.connectButton_Click);
        //
        // disconnectButton
        //
        this.disconnectButton.Location = new System.Drawing.Point(120, 12);
        this.disconnectButton.Name = "disconnectButton";
        this.disconnectButton.Size = new System.Drawing.Size(100, 30);
        this.disconnectButton.TabIndex = 1;
        this.disconnectButton.Text = "Disconnect";
        this.disconnectButton.UseVisualStyleBackColor = true;
        this.disconnectButton.Click += new System.EventHandler(this.disconnectButton_Click);
        //
        // ipAddressBox
        //
        this.ipAddressBox.FormattingEnabled = true;
        this.ipAddressBox.Items.AddRange(new object[] {
        "127.0.0.1"});
```

```
52          this.ipAddressBox.Location = new System.Drawing.Point(12, 50);
53          this.ipAddressBox.Name = "ipAddressBox";
54          this.ipAddressBox.Size = new System.Drawing.Size(360, 24);
55          this.ipAddressBox.TabIndex = 2;
56          //
57          // defaultMessages
58          //
59          this.defaultMessages.FormattingEnabled = true;
60          this.defaultMessages.Items.AddRange(new object[] {
61          "Hello",
62          "How are you?",
63          "Goodbye"});
64          this.defaultMessages.Location = new System.Drawing.Point(12, 80);
65          this.defaultMessages.Name = "defaultMessages";
66          this.defaultMessages.Size = new System.Drawing.Size(360, 84);
67          this.defaultMessages.TabIndex = 3;
68          //
69          // customMessageBox
70          //
71          this.customMessageBox.Location = new System.Drawing.Point(12, 180);
72          this.customMessageBox.Name = "customMessageBox";
73          this.customMessageBox.Size = new System.Drawing.Size(360, 22);
74          this.customMessageBox.TabIndex = 4;
75          //
76          // sendButton
77          //
78          this.sendButton.Location = new System.Drawing.Point(12, 210);
79          this.sendButton.Name = "sendButton";
80          this.sendButton.Size = new System.Drawing.Size(100, 30);
81          this.sendButton.TabIndex = 5;
82          this.sendButton.Text = "Send";
83          this.sendButton.UseVisualStyleBackColor = true;
84          this.sendButton.Click += new System.EventHandler(this.sendButton_Click);
85          //
86          // logs
87          //
88          this.logs.FormattingEnabled = true;
89          this.logs.ItemHeight = 16;
90          this.logs.Location = new System.Drawing.Point(12, 250);
91          this.logs.Name = "logs";
92          this.logs.Size = new System.Drawing.Size(360, 100);
93          this.logs.TabIndex = 6;
94          //
95          // ClientForm
96          //
97          this.ClientSize = new System.Drawing.Size(400, 370);
98          this.Controls.Add(this.logs);
99          this.Controls.Add(this.sendButton);
100         this.Controls.Add(this.customMessageBox);
101         this.Controls.Add(this.defaultMessages);
102         this.Controls.Add(this.ipAddressBox);
103         this.Controls.Add(this.disconnectButton);
104         this.Controls.Add(this.connectButton);
105         this.Name = "ClientForm";
```

```
106         this.Text = "TCP Client 1"; // Change to "TCP Client 2" for Client2
107         this.ResumeLayout(false);
108         this.PerformLayout();
109     }
110 
111     private System.Windows.Forms.Button connectButton;
112     private System.Windows.Forms.Button disconnectButton;
113     private System.Windows.Forms.ComboBox ipAddressBox;
114     private System.Windows.Forms.ListBox defaultMessages;
115     private System.Windows.Forms.TextBox customMessageBox;
116     private System.Windows.Forms.Button sendButton;
117     private System.Windows.Forms.ListBox logs;
118 
119     private void connectButton_Click(object sender, EventArgs e)
120     {
121         try
122         {
123             string ipAddress = ipAddressBox.Text;
124             client = new TcpClient(ipAddress, 12345);
125             stream = client.GetStream();
126 
127             // Send client name
128             string clientName = this.Text;
129             byte[] nameBytes = Encoding.UTF8.GetBytes(clientName);
130             stream.Write(nameBytes, 0, nameBytes.Length);
131 
132             logs.Items.Add($"{clientName}: Connected to server at {ipAddress}");
133         }
134         catch (Exception ex)
135         {
136             MessageBox.Show("Error: " + ex.Message);
137         }
138     }
139 
140     private void disconnectButton_Click(object sender, EventArgs e)
141     {
142         if (client != null && client.Connected)
143         {
144             client.Close();
145             logs.Items.Add("Disconnected from server.");
146         }
147     }
148 
149     private void sendButton_Click(object sender, EventArgs e)
150     {
151         if (client == null || !client.Connected)
152         {
153             MessageBox.Show("Not connected to a server.");
154             return;
155         }
156 
157         string message = customMessageBox.Text.Trim();
158         if (string.IsNullOrEmpty(message))
159         {
```

```
160             message = defaultMessages.SelectedItem?.ToString();
161             if (string.IsNullOrEmpty(message))
162             {
163                 MessageBox.Show("Please select or enter a message.");
164                 return;
165             }
166         }
167 
168         byte[] messageBytes = Encoding.UTF8.GetBytes(message);
169         stream.Write(messageBytes, 0, messageBytes.Length);
170 
171         logs.Items.Add($"{this.Text}: You: {message}");
172         customMessageBox.Clear();
173     }
174 }
175 
```

tcp_protocol\Client\ClientForm.cs

```
using System;
using System.Net.Sockets;
using System.Text;
using System.Windows.Forms;

public partial class ClientForm : Form
{
    private TcpClient client;
    private NetworkStream stream;

    public ClientForm()
    {
        InitializeComponent();
    }

    private void InitializeComponent()
    {
        this.connectButton = new System.Windows.Forms.Button();
        this.disconnectButton = new System.Windows.Forms.Button();
        this.ipAddressBox = new System.Windows.Forms.ComboBox();
        this.defaultMessages = new System.Windows.Forms.ListBox();
        this.customMessageBox = new System.Windows.Forms.TextBox();
        this.sendButton = new System.Windows.Forms.Button();
        this.logs = new System.Windows.Forms.ListBox();
        this.SuspendLayout();
        //
        // connectButton
        //
        this.connectButton.Location = new System.Drawing.Point(12, 12);
        this.connectButton.Name = "connectButton";
        this.connectButton.Size = new System.Drawing.Size(100, 30);
        this.connectButton.TabIndex = 0;
        this.connectButton.Text = "Connect";
        this.connectButton.UseVisualStyleBackColor = true;
        this.connectButton.Click += new System.EventHandler(this.connectButton_Click);
        //
        // disconnectButton
        //
        this.disconnectButton.Location = new System.Drawing.Point(120, 12);
        this.disconnectButton.Name = "disconnectButton";
        this.disconnectButton.Size = new System.Drawing.Size(100, 30);
        this.disconnectButton.TabIndex = 1;
        this.disconnectButton.Text = "Disconnect";
        this.disconnectButton.UseVisualStyleBackColor = true;
        this.disconnectButton.Click += new System.EventHandler(this.disconnectButton_Click);
        //
        // ipAddressBox
        //
        this.ipAddressBox.FormattingEnabled = true;
        this.ipAddressBox.Items.AddRange(new object[] {
        "127.0.0.1"});
```

```
52          this.ipAddressBox.Location = new System.Drawing.Point(12, 50);
53          this.ipAddressBox.Name = "ipAddressBox";
54          this.ipAddressBox.Size = new System.Drawing.Size(360, 24);
55          this.ipAddressBox.TabIndex = 2;
56          //
57          // defaultMessages
58          //
59          this.defaultMessages.FormattingEnabled = true;
60          this.defaultMessages.Items.AddRange(new object[] {
61          "Hello",
62          "How are you?",
63          "Goodbye"});
64          this.defaultMessages.Location = new System.Drawing.Point(12, 80);
65          this.defaultMessages.Name = "defaultMessages";
66          this.defaultMessages.Size = new System.Drawing.Size(360, 84);
67          this.defaultMessages.TabIndex = 3;
68          //
69          // customMessageBox
70          //
71          this.customMessageBox.Location = new System.Drawing.Point(12, 180);
72          this.customMessageBox.Name = "customMessageBox";
73          this.customMessageBox.Size = new System.Drawing.Size(360, 22);
74          this.customMessageBox.TabIndex = 4;
75          //
76          // sendButton
77          //
78          this.sendButton.Location = new System.Drawing.Point(12, 210);
79          this.sendButton.Name = "sendButton";
80          this.sendButton.Size = new System.Drawing.Size(100, 30);
81          this.sendButton.TabIndex = 5;
82          this.sendButton.Text = "Send";
83          this.sendButton.UseVisualStyleBackColor = true;
84          this.sendButton.Click += new System.EventHandler(this.sendButton_Click);
85          //
86          // logs
87          //
88          this.logs.FormattingEnabled = true;
89          this.logs.ItemHeight = 16;
90          this.logs.Location = new System.Drawing.Point(12, 250);
91          this.logs.Name = "logs";
92          this.logs.Size = new System.Drawing.Size(360, 100);
93          this.logs.TabIndex = 6;
94          //
95          // ClientForm
96          //
97          this.ClientSize = new System.Drawing.Size(400, 370);
98          this.Controls.Add(this.logs);
99          this.Controls.Add(this.sendButton);
100         this.Controls.Add(this.customMessageBox);
101         this.Controls.Add(this.defaultMessages);
102         this.Controls.Add(this.ipAddressBox);
103         this.Controls.Add(this.disconnectButton);
104         this.Controls.Add(this.connectButton);
105         this.Name = "ClientForm";
```

```
106         this.Text = "TCP Client 2"; // Change to "TCP Client 2" for Client2
107         this.ResumeLayout(false);
108         this.PerformLayout();
109     }
110 
111     private System.Windows.Forms.Button connectButton;
112     private System.Windows.Forms.Button disconnectButton;
113     private System.Windows.Forms.ComboBox ipAddressBox;
114     private System.Windows.Forms.ListBox defaultMessages;
115     private System.Windows.Forms.TextBox customMessageBox;
116     private System.Windows.Forms.Button sendButton;
117     private System.Windows.Forms.ListBox logs;
118 
119     private void connectButton_Click(object sender, EventArgs e)
120     {
121         try
122         {
123             string ipAddress = ipAddressBox.Text;
124             client = new TcpClient(ipAddress, 12345);
125             stream = client.GetStream();
126 
127             // Send client name
128             string clientName = this.Text;
129             byte[] nameBytes = Encoding.UTF8.GetBytes(clientName);
130             stream.Write(nameBytes, 0, nameBytes.Length);
131 
132             logs.Items.Add($"{clientName}: Connected to server at {ipAddress}");
133         }
134         catch (Exception ex)
135         {
136             MessageBox.Show("Error: " + ex.Message);
137         }
138     }
139 
140     private void disconnectButton_Click(object sender, EventArgs e)
141     {
142         if (client != null && client.Connected)
143         {
144             client.Close();
145             logs.Items.Add("Disconnected from server.");
146         }
147     }
148 
149     private void sendButton_Click(object sender, EventArgs e)
150     {
151         if (client == null || !client.Connected)
152         {
153             MessageBox.Show("Not connected to a server.");
154             return;
155         }
156 
157         string message = customMessageBox.Text.Trim();
158         if (string.IsNullOrEmpty(message))
159         {
```

```
160             message = defaultMessages.SelectedItem?.ToString();
161             if (string.IsNullOrEmpty(message))
162             {
163                 MessageBox.Show("Please select or enter a message.");
164                 return;
165             }
166         }
167 
168         byte[] messageBytes = Encoding.UTF8.GetBytes(message);
169         stream.Write(messageBytes, 0, messageBytes.Length);
170 
171         logs.Items.Add($"{this.Text}: You: {message}");
172         customMessageBox.Clear();
173     }
174 }
175 
```

udp_protocol\UDPServer\Form1.Designer.cs

```
namespace UDPServer
{
    partial class ServerForm
    {
        private System.ComponentModel.IContainer components = null;

        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        private void InitializeComponent()
        {
            this.startButton = new System.Windows.Forms.Button();
            this.logs = new System.Windows.Forms.ListBox();
            this.SuspendLayout();
            //
            // startButton
            //
            this.startButton.Location = new System.Drawing.Point(12, 12);
            this.startButton.Name = "startButton";
            this.startButton.Size = new System.Drawing.Size(100, 30);
            this.startButton.TabIndex = 0;
            this.startButton.Text = "Start Server";
            this.startButton.UseVisualStyleBackColor = true;
            this.startButton.Click += new System.EventHandler(this.startButton_Click);
            //
            // logs
            //
            this.logs.FormattingEnabled = true;
            this.logs.Location = new System.Drawing.Point(12, 50);
            this.logs.Name = "logs";
            this.logs.Size = new System.Drawing.Size(360, 200);
            this.logs.TabIndex = 1;
            //
            // ServerForm
            //
            this.ClientSize = new System.Drawing.Size(400, 270);
            this.Controls.Add(this.logs);
            this.Controls.Add(this.startButton);
            this.Name = "ServerForm";
            this.Text = "UDP Server";
            this.ResumeLayout(false);
        }

        private System.Windows.Forms.Button startButton;
        private System.Windows.Forms.ListBox logs;
```

```
52          }
53      }
54
```

udp_protocol\UDPServer\Form1.cs

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Windows.Forms;

namespace UDPServer
{
    public partial class ServerForm : Form
    {
        private UdpClient udpServer;
        private Thread serverThread;
        private HashSet<string> connectedClients; // Track connected clients

        public ServerForm()
        {
            InitializeComponent();
            connectedClients = new HashSet<string>();
        }

        private void startButton_Click(object sender, EventArgs e)
        {
            serverThread = new Thread(StartServer);
            serverThread.IsBackground = true;
            serverThread.Start();
            logs.Items.Add("UDP Server started...");
            startButton.Enabled = false;
        }

        private void StartServer()
        {
            udpServer = new UdpClient(12345); // Listen on port 12345
            IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, 0);

            try
            {
                while (true)
                {
                    byte[] receivedBytes = udpServer.Receive(ref remoteEP); // Receive data
                    string receivedMessage = Encoding.UTF8.GetString(receivedBytes);
                    string clientIdentifier = $"{remoteEP.Address}:{remoteEP.Port}";

                    // Check if the message is a "CONNECT" or "DISCONNECT" signal
                    if (receivedMessage == "CONNECT")
                    {
                        if (!connectedClients.Contains(clientIdentifier))
                        {
                            connectedClients.Add(clientIdentifier);
                            logs.Invoke((MethodInvoker)delegate
```

```
                            {
                                logs.Items.Add($"Client connected: {clientIdentifier}");
                            });
                        }
                    }
                    else if (receivedMessage == "DISCONNECT")
                    {
                        if (connectedClients.Contains(clientIdentifier))
                        {
                            connectedClients.Remove(clientIdentifier);
                            logs.Invoke((MethodInvoker)delegate
                            {
                                logs.Items.Add($"Client disconnected: {clientIdentifier}");
                            });
                        }
                    }
                    else
                    {
                        // Log the received message
                        logs.Invoke((MethodInvoker)delegate
                        {
                            logs.Items.Add($"Received from {clientIdentifier}:
{receivedMessage}");
                        });

                        // Send a response back to the client
                        string responseMessage = $"Server received: {receivedMessage}";
                        byte[] responseBytes = Encoding.UTF8.GetBytes(responseMessage);
                        udpServer.Send(responseBytes, responseBytes.Length, remoteEP);
                    }
                }
            }
            catch (Exception ex)
            {
                logs.Invoke((MethodInvoker)delegate
                {
                    logs.Items.Add($"Error: {ex.Message}");
                });
            }
            finally
            {
                udpServer.Close();
            }
        }
    }
}

```

udp_protocol\UDPClient\Form1.cs

```
using System;
using System.Net.Sockets;
using System.Text;
using System.Windows.Forms;

namespace UDPClient
{
    public partial class ClientForm : Form
    {
        private UdpClient udpClient;
        private bool isConnected = false; // Track connection status

        public ClientForm()
        {
            InitializeComponent();
        }

        private void connectButton_Click(object sender, EventArgs e)
        {
            if (isConnected)
            {
                MessageBox.Show("You are already connected to the server.", "Connection Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                return;
            }

            if (string.IsNullOrEmpty(ipAddressBox.Text))
            {
                MessageBox.Show("Please enter or select a valid IP address.");
                return;
            }

            try
            {
                udpClient = new UdpClient();
                string connectMessage = "CONNECT";
                byte[] connectBytes = Encoding.UTF8.GetBytes(connectMessage);
                udpClient.Send(connectBytes, connectBytes.Length, ipAddressBox.Text, 12345);
                logs.Items.Add($"Connected to server at {ipAddressBox.Text}");
                isConnected = true; // Update connection status
            }
            catch (Exception ex)
            {
                MessageBox.Show($"Error: {ex.Message}");
            }
        }

        private void disconnectButton_Click(object sender, EventArgs e)
        {
            if (udpClient != null)
            {
                try
```

```
52                 {
53                     string disconnectMessage = "DISCONNECT";
54                     byte[] disconnectBytes = Encoding.UTF8.GetBytes(disconnectMessage);
55                     udpClient.Send(disconnectBytes, disconnectBytes.Length,
    ipAddressBox.Text, 12345);
56                     logs.Items.Add("Disconnected from server.");
57                     isConnected = false; // Reset connection status
58                 }
59                 catch (Exception ex)
60                 {
61                     MessageBox.Show($"Error: {ex.Message}");
62                 }
63                 finally
64                 {
65                     udpClient.Close();
66                     udpClient = null;
67                 }
68             }
69         }
70 
71         private void sendButton_Click(object sender, EventArgs e)
72         {
73             if (udpClient == null || !isConnected)
74             {
75                 MessageBox.Show("Not connected to a server.");
76                 return;
77             }
78 
79             string message;
80 
81             if (useCustomMessageCheckBox.Checked)
82             {
83                 // Use custom message
84                 message = customMessageBox.Text.Trim();
85                 if (string.IsNullOrEmpty(message))
86                 {
87                     MessageBox.Show("Please enter a custom message.");
88                     return;
89                 }
90             }
91             else
92             {
93                 // Use default message
94                 message = defaultMessages.SelectedItem?.ToString();
95                 if (string.IsNullOrEmpty(message))
96                 {
97                     MessageBox.Show("Please select a default message.");
98                     return;
99                 }
100             }
101 
102             // Send the message
103             byte[] messageBytes = Encoding.UTF8.GetBytes(message);
104             udpClient.Send(messageBytes, messageBytes.Length, ipAddressBox.Text, 12345);
```

```
105             logs.Items.Add($"Message sent to {ipAddressBox.Text}: {message}");
106         }
107     }
108 }
109
```

udp_protocol\UDPClient\Form1.Designer.cs

```
namespace UDPClient
{
    partial class ClientForm
    {
        private System.ComponentModel.IContainer components = null;

        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        private void InitializeComponent()
        {
            this.connectButton = new System.Windows.Forms.Button();
            this.disconnectButton = new System.Windows.Forms.Button();
            this.ipAddressBox = new System.Windows.Forms.ComboBox();
            this.defaultMessages = new System.Windows.Forms.ListBox();
            this.customMessageBox = new System.Windows.Forms.TextBox();
            this.useCustomMessageCheckBox = new System.Windows.Forms.CheckBox();
            this.sendButton = new System.Windows.Forms.Button();
            this.logs = new System.Windows.Forms.ListBox();
            this.SuspendLayout();
            //
            // connectButton
            //
            this.connectButton.Location = new System.Drawing.Point(12, 12);
            this.connectButton.Name = "connectButton";
            this.connectButton.Size = new System.Drawing.Size(100, 30);
            this.connectButton.TabIndex = 0;
            this.connectButton.Text = "Connect";
            this.connectButton.UseVisualStyleBackColor = true;
            this.connectButton.Click += new System.EventHandler(this.connectButton_Click);
            //
            // disconnectButton
            //
            this.disconnectButton.Location = new System.Drawing.Point(120, 12);
            this.disconnectButton.Name = "disconnectButton";
            this.disconnectButton.Size = new System.Drawing.Size(100, 30);
            this.disconnectButton.TabIndex = 1;
            this.disconnectButton.Text = "Disconnect";
            this.disconnectButton.UseVisualStyleBackColor = true;
            this.disconnectButton.Click += new
System.EventHandler(this.disconnectButton_Click);
            //
            // ipAddressBox
            //
            this.ipAddressBox.FormattingEnabled = true;
            this.ipAddressBox.Items.AddRange(new object[] {
```

```
            "127.0.0.1",       // Localhost
            "192.168.1.10",    // Example local IP
            "10.0.0.5"         // Example custom IP
            });
            this.ipAddressBox.Location = new System.Drawing.Point(12, 50);
            this.ipAddressBox.Name = "ipAddressBox";
            this.ipAddressBox.Size = new System.Drawing.Size(360, 24);
            this.ipAddressBox.TabIndex = 2;
            //
            // defaultMessages
            //
            this.defaultMessages.FormattingEnabled = true;
            this.defaultMessages.Items.AddRange(new object[] {
            "Hello",
            "How are you?",
            "Goodbye"});
            this.defaultMessages.Location = new System.Drawing.Point(12, 80);
            this.defaultMessages.Name = "defaultMessages";
            this.defaultMessages.Size = new System.Drawing.Size(360, 84);
            this.defaultMessages.TabIndex = 3;
            //
            // customMessageBox
            //
            this.customMessageBox.Location = new System.Drawing.Point(12, 180);
            this.customMessageBox.Name = "customMessageBox";
            this.customMessageBox.Size = new System.Drawing.Size(360, 22);
            this.customMessageBox.TabIndex = 4;
            //
            // useCustomMessageCheckBox
            //
            this.useCustomMessageCheckBox.AutoSize = true;
            this.useCustomMessageCheckBox.Location = new System.Drawing.Point(12, 210);
            this.useCustomMessageCheckBox.Name = "useCustomMessageCheckBox";
            this.useCustomMessageCheckBox.Size = new System.Drawing.Size(174, 20);
            this.useCustomMessageCheckBox.TabIndex = 5;
            this.useCustomMessageCheckBox.Text = "Use Custom Message";
            this.useCustomMessageCheckBox.UseVisualStyleBackColor = true;
            //
            // sendButton
            //
            this.sendButton.Location = new System.Drawing.Point(12, 240);
            this.sendButton.Name = "sendButton";
            this.sendButton.Size = new System.Drawing.Size(100, 30);
            this.sendButton.TabIndex = 6;
            this.sendButton.Text = "Send";
            this.sendButton.UseVisualStyleBackColor = true;
            this.sendButton.Click += new System.EventHandler(this.sendButton_Click);
            //
            // logs
            //
            this.logs.FormattingEnabled = true;
            this.logs.Location = new System.Drawing.Point(12, 280);
            this.logs.Name = "logs";
            this.logs.Size = new System.Drawing.Size(360, 100);
```

```
106             this.logs.TabIndex = 7;
107             //
108             // ClientForm
109             //
110             this.ClientSize = new System.Drawing.Size(400, 400);
111             this.Controls.Add(this.logs);
112             this.Controls.Add(this.sendButton);
113             this.Controls.Add(this.useCustomMessageCheckBox);
114             this.Controls.Add(this.customMessageBox);
115             this.Controls.Add(this.defaultMessages);
116             this.Controls.Add(this.ipAddressBox);
117             this.Controls.Add(this.disconnectButton);
118             this.Controls.Add(this.connectButton);
119             this.Name = "ClientForm";
120             this.Text = "UDP Client";
121             this.ResumeLayout(false);
122             this.PerformLayout();
123         }
124 
125         private System.Windows.Forms.Button connectButton;
126         private System.Windows.Forms.Button disconnectButton;
127         private System.Windows.Forms.ComboBox ipAddressBox;
128         private System.Windows.Forms.ListBox defaultMessages;
129         private System.Windows.Forms.TextBox customMessageBox;
130         private System.Windows.Forms.CheckBox useCustomMessageCheckBox;
131         private System.Windows.Forms.Button sendButton;
132         private System.Windows.Forms.ListBox logs;
133     }
134 }
135 
```

src\Communication\LanCommunication.cs

```
using System;
using System.IO;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;

namespace Keysight33500BApp
{
    public class LanCommunication : IDisposable
    {
        private TcpClient? _tcpClient;
        private NetworkStream? _stream;
        private readonly string _ipAddress;
        private readonly int _port;
        private bool _connected;



        public LanCommunication(string ipAddress, int port = 5025)
        {
            _ipAddress = ipAddress ?? throw new ArgumentNullException(nameof(ipAddress));
            _port = port;
            _connected = false;
        }

        public async Task ConnectAsync()
        {
            if (_connected) return;

            try
            {
                _tcpClient = new TcpClient();
                await _tcpClient.ConnectAsync(_ipAddress, _port);
                _stream = _tcpClient.GetStream();
                _stream.ReadTimeout  = 3000; // 3s timeout
                _stream.WriteTimeout = 3000;
                _connected = true;
            }
            catch (Exception ex)
            {
                Dispose();
                throw new Exception($"Could not connect to {_ipAddress}:{_port}.
{ex.Message}", ex);
            }
        }

        public async Task WriteAsync(string command)
        {
            if (!_connected || _stream == null)
                throw new InvalidOperationException("Not connected to instrument.");

            if (string.IsNullOrEmpty(command))
```

```
52                 throw new ArgumentException("Command cannot be null or empty.");
53
54             try
55             {
56                 // SCPI commands typically end with "\n"
57                 string cmd = command.EndsWith("\n") ? command : command + "\n";
58                 byte[] data = Encoding.ASCII.GetBytes(cmd);
59                 await _stream.WriteAsync(data, 0, data.Length);
60                 await _stream.FlushAsync();
61             }
62             catch (Exception ex)
63             {
64                 throw new IOException($"Failed to write command '{command}': {ex.Message}",
    ex);
65             }
66         }
67
68         public async Task<string> QueryAsync(string command)
69         {
70             await WriteAsync(command);
71
72             // Read the response line from the instrument
73             // 33500B typically terminates with "\n"
74             return await ReadLineAsync();
75         }
76
77         private async Task<string> ReadLineAsync()
78         {
79             if (_stream == null)
80                 throw new InvalidOperationException("Stream is not open.");
81
82             using var ms = new MemoryStream();
83             byte[] buffer = new byte[1];
84
85             // Read until we hit '\n'
86             while (true)
87             {
88                 int read = await _stream.ReadAsync(buffer, 0, 1);
89                 if (read < 1)
90                     throw new IOException("Socket closed unexpectedly.");
91
92                 if (buffer[0] == (byte)'\n')
93                     break;
94
95                 ms.Write(buffer, 0, read);
96             }
97
98             return Encoding.ASCII.GetString(ms.ToArray()).Trim();
99         }
100
101         public void Dispose()
102         {
103             _stream?.Dispose();
104             _tcpClient?.Close();
```

```
105             _tcpClient = null;
106             _stream = null;
107             _connected = false;
108         }
109     }
110 }
111
```

src\Communication\VisaCommunication.cs

```
using System;
using System.Threading.Tasks;
using Ivi.Visa.Interop;
using System.Globalization;

namespace Oscilloscope.Communication
{
    public class VisaCommunication : IDisposable
    {
        private readonly string visaAddress;
        private ResourceManager? resourceManager;
        private FormattedIO488? session;
        private bool isConnected;
        private bool disposed;

        public VisaCommunication(string address)
        {
            if (string.IsNullOrEmpty(address))
                throw new ArgumentException("VISA address cannot be empty",
nameof(address));

            visaAddress = address;
            isConnected = false;
            disposed = false;
        }

        public async Task<bool> ConnectAsync()
        {
            ThrowIfDisposed();

            try
            {
                await Task.Run(() =>
                {
                    Console.WriteLine("Creating VISA ResourceManager...");
                    resourceManager = new ResourceManager();
                    session = new FormattedIO488
                    {
                        IO = (IMessage)resourceManager.Open(visaAddress)
                    };

                    if (session.IO == null)
                        throw new Exception("Failed to open session with VISA address: " +
visaAddress);

                    Console.WriteLine("Session opened successfully.");
                    session.IO.Timeout = 5000; // 5-second timeout

                    // Basic initialization
                    session.WriteString("*RST", true);
                    session.WriteString("*CLS", true);

```

```
51                      isConnected = true;
52                  });
53
54                  return true;
55              }
56              catch (Exception ex)
57              {
58                  Console.WriteLine("Connection failed: " + ex.Message);
59                  Dispose();
60                  throw new Exception($"Failed to connect to oscilloscope: {ex.Message}", ex);
61              }
62          }
63
64          public async Task<string> QueryAsync(string command)
65          {
66              ThrowIfDisposed();
67              EnsureConnected();
68
69              if (string.IsNullOrEmpty(command))
70                  throw new ArgumentException("Command cannot be empty", nameof(command));
71
72              try
73              {
74                  return await Task.Run(() =>
75                  {
76                      Console.WriteLine($"Sending command: {command}");
77                      session!.WriteString(command, true);
78                      return session.ReadString();
79                  });
80              }
81              catch (Exception ex)
82              {
83                  throw new Exception($"Failed to execute query '{command}': {ex.Message}",
    ex);
84              }
85          }
86
87          public async Task WriteAsync(string command)
88          {
89              ThrowIfDisposed();
90              EnsureConnected();
91
92              if (string.IsNullOrEmpty(command))
93                  throw new ArgumentException("Command cannot be empty", nameof(command));
94
95              try
96              {
97                  await Task.Run(() =>
98                  {
99                      Console.WriteLine($"Writing command: {command}");
100                     session!.WriteString(command, true);
101                 });
102             }
103             catch (Exception ex)
```

```
            {
                throw new Exception($"Failed to write command '{command}': {ex.Message}", ex);
            }
        }

        public async Task SendCommandAsync(string command)
        {
            await WriteAsync(command);
        }

        public async Task<byte[]> ReadBinaryDataAsync()
        {
            ThrowIfDisposed();
            EnsureConnected();

            try
            {
                return await Task.Run(() =>
                {
                    session!.WriteString(":DISP:DATA? PNG", true);
                    return (byte[])session.ReadIEEEBlock(IEEEBinaryType.BinaryType_UI1, false, true);
                });
            }
            catch (Exception ex)
            {
                throw new Exception($"Failed to read binary data from oscilloscope: {ex.Message}", ex);
            }
        }

        public async Task<bool> VerifyConnectionAsync()
        {
            if (disposed || !isConnected || session == null)
                return false;

            try
            {
                string response = await QueryAsync("*IDN?");
                Console.WriteLine("Connection verified. Device ID: " + response);
                return true;
            }
            catch
            {
                isConnected = false;
                return false;
            }
        }

        private void EnsureConnected()
        {
            if (!isConnected || session == null)
                throw new InvalidOperationException("Not connected to oscilloscope");
```

```
155         }
156
157         private void ThrowIfDisposed()
158         {
159             if (disposed)
160                 throw new ObjectDisposedException(nameof(VisaCommunication));
161         }
162
163         public void Dispose()
164         {
165             if (disposed) return;
166
167             try
168             {
169                 session?.IO.Close();
170                 session = null;
171                 resourceManager = null;
172             }
173             catch (Exception ex)
174             {
175                 Console.WriteLine("Error during session cleanup: " + ex.Message);
176             }
177             finally
178             {
179                 isConnected = false;
180                 disposed = true;
181             }
182         }
183
184         public bool IsConnected => isConnected;
185
186         public async Task<string> ReadResponseAsync()
187         {
188             ThrowIfDisposed();
189             EnsureConnected();
190
191             try
192             {
193                 return await Task.Run(() =>
194                 {
195                     return session!.ReadString();
196                 });
197             }
198             catch (Exception ex)
199             {
200                 throw new Exception($"Failed to read response from oscilloscope:
    {ex.Message}", ex);
201             }
202         }
203
204     }
205 }
206
```

src\Controllers\FunctionGeneratorController.cs

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Threading.Tasks;

namespace Keysight33500BApp
{
    public class FunctionGeneratorController
    {
        private readonly LanCommunication _comm;

        // Track which waveform shape is currently set on each channel, e.g. "SIN", "PULS",
...
        // This helps us skip invalid commands for DC or NOIS, etc.
        private readonly Dictionary<string, string> _channelShape;

        public FunctionGeneratorController(LanCommunication communication)
        {
            _comm = communication ?? throw new ArgumentNullException(nameof(communication));

            // Default both channels to SIN, or unknown, etc.
            _channelShape = new Dictionary<string, string>
            {
                { "CH1", "SIN" },
                { "CH2", "SIN" }
            };
        }

        public async Task InitializeAsync()
        {
            await _comm.WriteAsync("*RST");
            await Task.Delay(1000);
            await _comm.WriteAsync("*CLS");
        }

        public async Task<string> IdentifyAsync()
        {
            return await _comm.QueryAsync("*IDN?");
        }

        // ──────────────────────────────────────────
        //         HELPER: IsCommandValid?
        // ──────────────────────────────────────────
        // We check if the current shape supports phase, pulse edges, etc.
        private bool SupportsPhase(string shape)
            => shape == "SIN" || shape == "SQU" || shape == "RAMP" || shape == "PULS";

        private bool SupportsPulseEdges(string shape)
            => shape == "PULS"; // Only pulses can set leading/trailing edge times

        private bool SupportsSquareDuty(string shape)
            => shape == "SQU";
```

```
52
53          private bool SupportsRampSymmetry(string shape)
54              => shape == "RAMP";
55
56          // ──────────────────────────────────────────
57          //  BASIC PARAMETERS (Shared)
58          // ──────────────────────────────────────────
59          public async Task SetWaveformAsync(string channel, string shape)
60          {
61              shape = shape.ToUpperInvariant();
62              if (channel == "CH1")
63              {
64                  await _comm.WriteAsync($":SOURce1:FUNCtion {shape}");
65              }
66              else if (channel == "CH2")
67              {
68                  await _comm.WriteAsync($":SOURce2:FUNCtion {shape}");
69              }
70              else
71              {
72                  throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
73              }
74
75              // Remember which shape we set for this channel
76              _channelShape[channel] = shape;
77          }
78
79          public async Task SetFrequencyAsync(string channel, double frequency)
80          {
81              if (channel == "CH1")
82              {
83                  await _comm.WriteAsync($":SOURce1:FREQuency
    {frequency.ToString(CultureInfo.InvariantCulture)}");
84              }
85              else if (channel == "CH2")
86              {
87                  await _comm.WriteAsync($":SOURce2:FREQuency
    {frequency.ToString(CultureInfo.InvariantCulture)}");
88              }
89              else
90              {
91                  throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
92              }
93          }
94
95          public async Task SetAmplitudeAsync(string channel, double amplitude)
96          {
97              if (channel == "CH1")
98              {
99                  await _comm.WriteAsync($":SOURce1:VOLTage
    {amplitude.ToString(CultureInfo.InvariantCulture)}");
100             }
101             else if (channel == "CH2")
102             {
```

```
103                 await _comm.WriteAsync($":SOURce2:VOLTage
{amplitude.ToString(CultureInfo.InvariantCulture)}");
104             }
105             else
106             {
107                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
108             }
109         }
110 
111         public async Task SetOffsetAsync(string channel, double offset)
112         {
113             if (channel == "CH1")
114             {
115                 await _comm.WriteAsync($":SOURce1:VOLTage:OFFSet
{offset.ToString(CultureInfo.InvariantCulture)}");
116             }
117             else if (channel == "CH2")
118             {
119                 await _comm.WriteAsync($":SOURce2:VOLTage:OFFSet
{offset.ToString(CultureInfo.InvariantCulture)}");
120             }
121             else
122             {
123                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
124             }
125         }
126 
127         // ──────────────────────────────────────────
128         //  EXTENDED PARAMETERS
129         // ──────────────────────────────────────────
130         public async Task SetPhaseAsync(string channel, double phase)
131         {
132             string shape = _channelShape[channel]; // Check current shape
133             if (!SupportsPhase(shape))
134             {
135                 // We skip this command to avoid instrument error
136                 return;
137             }
138 
139             if (channel == "CH1")
140             {
141                 await _comm.WriteAsync($":SOURce1:PHASe
{phase.ToString(CultureInfo.InvariantCulture)}");
142             }
143             else if (channel == "CH2")
144             {
145                 await _comm.WriteAsync($":SOURce2:PHASe
{phase.ToString(CultureInfo.InvariantCulture)}");
146             }
147             else
148             {
149                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
150             }
151         }
```

```
152
153         public async Task SetSquareDutyCycleAsync(string channel, double dutyCycle)
154         {
155             string shape = _channelShape[channel];
156             if (!SupportsSquareDuty(shape))
157             {
158                 return; // skip if shape isn't SQU
159             }
160
161             if (channel == "CH1")
162             {
163                 await _comm.WriteAsync($":SOURce1:FUNCtion:SQUare:DCYCle
    {dutyCycle.ToString(CultureInfo.InvariantCulture)}");
164             }
165             else if (channel == "CH2")
166             {
167                 await _comm.WriteAsync($":SOURce2:FUNCtion:SQUare:DCYCle
    {dutyCycle.ToString(CultureInfo.InvariantCulture)}");
168             }
169             else
170             {
171                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
172             }
173         }
174
175         public async Task SetRampSymmetryAsync(string channel, double symmetry)
176         {
177             string shape = _channelShape[channel];
178             if (!SupportsRampSymmetry(shape))
179             {
180                 return; // skip if shape isn't RAMP
181             }
182
183             if (channel == "CH1")
184             {
185                 await _comm.WriteAsync($":SOURce1:FUNCtion:RAMP:SYMMetry
    {symmetry.ToString(CultureInfo.InvariantCulture)}");
186             }
187             else if (channel == "CH2")
188             {
189                 await _comm.WriteAsync($":SOURce2:FUNCtion:RAMP:SYMMetry
    {symmetry.ToString(CultureInfo.InvariantCulture)}");
190             }
191             else
192             {
193                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
194             }
195         }
196
197         public async Task SetPulseWidthAsync(string channel, double width)
198         {
199             string shape = _channelShape[channel];
200             if (!SupportsPulseEdges(shape))
201             {
```

```
202                 return; // skip if shape isn't PULS
203             }
204 
205             if (channel == "CH1")
206             {
207                 await _comm.WriteAsync($":SOURce1:FUNCtion:PULSe:WIDTh
    {width.ToString(CultureInfo.InvariantCulture)}");
208             }
209             else if (channel == "CH2")
210             {
211                 await _comm.WriteAsync($":SOURce2:FUNCtion:PULSe:WIDTh
    {width.ToString(CultureInfo.InvariantCulture)}");
212             }
213             else
214             {
215                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
216             }
217         }
218 
219         public async Task SetPulseLeadingEdgeAsync(string channel, double leadEdge)
220         {
221             string shape = _channelShape[channel];
222             if (!SupportsPulseEdges(shape))
223             {
224                 return;
225             }
226 
227             if (channel == "CH1")
228             {
229                 await _comm.WriteAsync($":SOURce1:FUNCtion:PULSe:TRANsition:LEADing
    {leadEdge.ToString(CultureInfo.InvariantCulture)}");
230             }
231             else if (channel == "CH2")
232             {
233                 await _comm.WriteAsync($":SOURce2:FUNCtion:PULSe:TRANsition:LEADing
    {leadEdge.ToString(CultureInfo.InvariantCulture)}");
234             }
235             else
236             {
237                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
238             }
239         }
240 
241         public async Task SetPulseTrailingEdgeAsync(string channel, double trailEdge)
242         {
243             string shape = _channelShape[channel];
244             if (!SupportsPulseEdges(shape))
245             {
246                 return;
247             }
248 
249             if (channel == "CH1")
250             {
```

```
251                 await _comm.WriteAsync($":SOURce1:FUNCtion:PULSe:TRANsition:TRAiling
{trailEdge.ToString(CultureInfo.InvariantCulture)}");
252             }
253             else if (channel == "CH2")
254             {
255                 await _comm.WriteAsync($":SOURce2:FUNCtion:PULSe:TRANsition:TRAiling
{trailEdge.ToString(CultureInfo.InvariantCulture)}");
256             }
257             else
258             {
259                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
260             }
261         }
262 
263         public async Task SetNoiseBandwidthAsync(string channel, double bandwidth)
264         {
265             string shape = _channelShape[channel];
266             // It's valid for shape == "NOIS". If shape != NOIS, skip to avoid SCPI error.
267             if (!shape.StartsWith("NOIS"))
268             {
269                 return;
270             }
271 
272             if (channel == "CH1")
273             {
274                 await _comm.WriteAsync($":SOURce1:FUNCtion:NOISe:BWIDth
{bandwidth.ToString(CultureInfo.InvariantCulture)}");
275             }
276             else if (channel == "CH2")
277             {
278                 await _comm.WriteAsync($":SOURce2:FUNCtion:NOISe:BWIDth
{bandwidth.ToString(CultureInfo.InvariantCulture)}");
279             }
280             else
281             {
282                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
283             }
284         }
285 
286         // ──────────────────────────────────────────
287         //  DISPOSE
288         // ──────────────────────────────────────────
289         public async Task DisposeAsync()
290         {
291             // Turn off both channels before disconnecting if you want
292             await _comm.WriteAsync(":OUTPut1:STATe OFF");
293             await _comm.WriteAsync(":OUTPut2:STATe OFF");
294 
295             _comm.Dispose();
296         }
297 
298         // ──────────────────────────────────────────
299         //  OPTIONAL: SetChannelAsync
300         // ──────────────────────────────────────────
```

```
301         public async Task SetChannelAsync(string channel)
302         {
303             // Some 33500B firmware might accept :INSTrument:SELect 1 or 2
304             // But if it triggers an error, you can skip calling this entirely
305             // since all your commands are direct to :SOURce1: or :SOURce2: anyway.
306             if (channel == "CH1")
307             {
308                 await _comm.WriteAsync(":INSTrument:SELect 1");
309             }
310             else if (channel == "CH2")
311             {
312                 await _comm.WriteAsync(":INSTrument:SELect 2");
313             }
314             else
315             {
316                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
317             }
318         }
319 
320         // ──────────────────────────────────────────
321         //  Enable/Disable Output
322         // ──────────────────────────────────────────
323         public async Task EnableChannelAsync(string channel, bool enable = true)
324         {
325             // CH1 => :OUTPut1:STATe ON|OFF
326             // CH2 => :OUTPut2:STATe ON|OFF
327             string onOff = enable ? "ON" : "OFF";
328             if (channel == "CH1")
329             {
330                 await _comm.WriteAsync($":OUTPut1:STATe {onOff}");
331             }
332             else if (channel == "CH2")
333             {
334                 await _comm.WriteAsync($":OUTPut2:STATe {onOff}");
335             }
336             else
337             {
338                 throw new ArgumentException("Invalid channel. Use 'CH1' or 'CH2'.");
339             }
340         }
341 
342         /// <summary>
343         /// If you want a simple “set output ON/OFF” for the *currently selected* channel
    only,
344         /// you can do:
345         /// </summary>
346         public async Task SetOutputStateAsync(bool enable)
347         {
348             string onOff = enable ? "ON" : "OFF";
349             await _comm.WriteAsync($":OUTPut:STATe {onOff}");
350         }
351 
352 
353         // ──────────────────────────────────────────
```

```
354         //  (Optional) Utility to Force a Trigger
355         // ──────────────────────────────────────────
356         public async Task SendSoftwareTriggerAsync()
357         {
358             // e.g. :TRIGger:IMMediate
359             await _comm.WriteAsync(":TRIGger:IMMediate");
360         }
361     }
362 }
363
```

src\Controllers\OscilloscopeController.cs

```
199 public async Task<double> MeasureVppAsync(int channel)
200         {
201             ValidateChannel(channel);
202             await _communication.WriteAsync($":MEASure:SOURce CHANnel{channel}");
203             string response = await _communication.QueryAsync(":MEASure:VPP?");
204             return double.Parse(response, CultureInfo.InvariantCulture);
205         }
```

src\Forms\MainForm.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading.Tasks;
using Oscilloscope.Communication;
using Oscilloscope.Controllers;
using Oscilloscope.Models;
using System.Globalization;
using Keysight33500BApp;
using System.Text;


namespace Oscilloscope.Forms
{
    public partial class MainForm : Form
    {
        private readonly OscilloscopeSettings settings;
        private OscilloscopeController? oscilloscope;
        private bool isConnected ;


        private TabControl resultTabs = null!;

        private TabControl mainTabs = null!;
        private PictureBox oscilloscopeDisplay = null!;
        // Tracks test duration
        private System.Diagnostics.Stopwatch _testStopwatch = null!;

        // Periodic timer that updates the timeLabel in real time
        private System.Windows.Forms.Timer _testTimer = null!;

        // Remove or comment out the old single-line dictionary
        // private Dictionary<string, (int total, int pass)> _waveStats;

        // Instead declare the nested dictionary
        private Dictionary<string, Dictionary<string, (int total, int pass)>> _paramStats
            = new Dictionary<string, Dictionary<string, (int total, int pass)>>();

        // Fields for standard function generator params
        private ComboBox _waveformCombo = null!;
        private NumericUpDown _freqInput = null!;
        private NumericUpDown _amplitudeInput = null!;
        private NumericUpDown _offsetInput = null!;
        private CheckBox _outputEnableCheck = null!;

        private Label summaryLabel = new Label();
        private TextBox ch1Results = null!;
        private Label timeLabel = null!;     // To show how long the test took

        // Fields for extended parameters
        private Panel _parameterPanel = null!;
```

```
52          private NumericUpDown? _phaseInput = null!;
53          private NumericUpDown? _dutyInput = null!;
54          private NumericUpDown? _symmetryInput = null!;
55
56          private TextBox? _resultsBox = null!;
57
58          private NumericUpDown? _pulseWidthInput = null!;
59          private NumericUpDown? _leadEdgeInput = null!;
60          private NumericUpDown? _trailEdgeInput = null!;
61          private NumericUpDown? _noiseBandwidthInput = null!;
62
63          private ComboBox _channelCombo = null!;
64
65
66          private RichTextBox testResultsBoxCH2 = null!;
67          private TextBox ch2Results = null!;
68
69
70
71          private string _selectedChannel = "CH1";
72
73          private void UpdateSelectedChannel()
74          {
75              _selectedChannel = _channelCombo.SelectedItem?.ToString() ?? "CH1";
76          }
77
78
79
80
81          private TabPage CreateOscilloscopeTab(string tabName)
82          {
83              return new TabPage(tabName);
84          }
85
86          public MainForm()
87          {
88              InitializeComponent();
89              settings = new OscilloscopeSettings();
90              isConnected = false;
91          }
92
93          private Panel CreateStatusPanel()
94          {
95              Panel panel = new Panel
96              {
97                  Dock = DockStyle.Fill,
98                  BorderStyle = BorderStyle.FixedSingle
99              };
100
101             TextBox statusBox = new TextBox
102             {
103                 Multiline = true,
104                 ReadOnly = true,
105                 ScrollBars = ScrollBars.Vertical,
```

```
106                 Dock = DockStyle.Fill,
107                 Name = "statusBox"
108             };
109 
110             panel.Controls.Add(statusBox);
111             return panel;
112         }
113 
114         private void InitializeComponent()
115         {
116             // Overall window setup
117             this.Text = "Keysight MSOX3104T Oscilloscope Control";
118             this.Size = new Size(1200, 700);
119             this.StartPosition = FormStartPosition.CenterScreen;
120             this.BackColor = ColorTranslator.FromHtml("#005cb9");
121 
122             // Create a main TabControl to switch between "Main Control" and "Test Screen"
123             TabControl mainTabControl = new TabControl
124             {
125                 Dock = DockStyle.Fill,
126                 Font = new Font("Arial", 10, FontStyle.Bold),
127                 ItemSize = new Size(160, 30)
128             };
129 
130             // Create the two tabs
131             TabPage screen1Tab = new TabPage("Main Control")
132             {
133                 BackColor = Color.White
134             };
135             TabPage screen2Tab = new TabPage("Test Screen")
136             {
137                 BackColor = Color.White
138             };
139 
140             // Add your main panel (the big UI) to the first tab
141             screen1Tab.Controls.Add(CreateMainPanel());
142 
143             // Add a placeholder panel for the second screen or your test logic
144             screen2Tab.Controls.Add(CreateTestScreenPanel());
145 
146             // Add both tabs to mainTabControl
147             mainTabControl.TabPages.Add(screen1Tab);
148             mainTabControl.TabPages.Add(screen2Tab);
149 
150             // Finally, add the mainTabControl to this form
151             this.Controls.Add(mainTabControl);
152 
153             // -- If your _waveformCombo is created in CreateChannelSettingsPanel for
    "CH1",
154             //    you must ensure you store that control in _waveformCombo:
155             //    e.g. inside CreateChannelSettingsPanel("CH1") do:
156             //      _waveformCombo = waveformCombo;
157 
158             // -- Then AFTER _waveformCombo is assigned, hook the event:
```

```
159             if (_waveformCombo != null)
160             {
161                 _waveformCombo.SelectedIndexChanged += (s, e) =>
162                 {
163                     string selectedWaveform =
_waveformCombo.SelectedItem?.ToString()?.ToUpper() ?? "SIN";
164                     UpdateParameterPanel(_parameterPanel, selectedWaveform); // Corrected
function call
165                 };
166             }
167 
168         }
169 
170 
171 
172         private void UpdateStatus(string message)
173         {
174             if (InvokeRequired)
175             {
176                 Invoke(new Action<string>(UpdateStatus), message);
177                 return;
178             }
179 
180             Control[] foundControls = Controls.Find("statusBox", true);
181             if (foundControls.Length == 0 || foundControls[0] is not TextBox statusBox)
182             {
183                 MessageBox.Show("Error: statusBox not found!", "Error",
MessageBoxButtons.OK, MessageBoxIcon.Error);
184                 return;
185             }
186 
187             statusBox.AppendText($"[{DateTime.Now:HH:mm:ss}] {message}
{Environment.NewLine}");
188             statusBox.SelectionStart = statusBox.TextLength;
189             statusBox.ScrollToCaret();
190         }
191 
192 
193 
194         private bool _isDisplayEnabled = false; // Track display status
195 
196 
197         private async void ToggleDisplay(object sender, EventArgs e)
198         {
199             _isDisplayEnabled = !_isDisplayEnabled;
200 
201             Button button = sender as Button;
202             if (button != null)
203             {
204                 button.Text = _isDisplayEnabled ? "Stop Display" : "Start Display";
205             }
206 
207             if (_isDisplayEnabled)
208             {
```

```
209                 await Task.Run(UpdateOscilloscopeDisplay);
210             }
211             else
212             {
213                 oscilloscopeDisplay.Image = null; // Clear screen when stopped
214             }
215         }
216 
217 
218         private void SaveScreenshot(object sender, EventArgs e)
219         {
220             if (oscilloscopeDisplay.Image == null)
221             {
222                 MessageBox.Show("No image to save! Start the display first.", "Warning",
    MessageBoxButtons.OK, MessageBoxIcon.Warning);
223                 return;
224             }
225 
226             try
227             {
228                 // Define the project folder path
229                 string projectPath = AppDomain.CurrentDomain.BaseDirectory;
230                 string filePath = Path.Combine(projectPath, $"Oscilloscope_Screens↩
    hot_{DateTime.Now:yyyyMMdd_HHmmss}.png");
231 
232                 // Save the image
233                 oscilloscopeDisplay.Image.Save(filePath,
    System.Drawing.Imaging.ImageFormat.Png);
234 
235                 MessageBox.Show($"Screenshot saved: {filePath}", "Success",
    MessageBoxButtons.OK, MessageBoxIcon.Information);
236             }
237             catch (Exception ex)
238             {
239                 MessageBox.Show($"Failed to save screenshot: {ex.Message}", "Error",
    MessageBoxButtons.OK, MessageBoxIcon.Error);
240             }
241         }
242 
243 
244 
245 
246         private FunctionGeneratorController? _functionGenerator;
247         private bool _isConnectedToFunctionGenerator = false;
248 
249         private RichTextBox testResultsBox = null!;
250 
251 
252         private Panel CreateTestScreenPanel()
253         {
254             Panel testPanel = new Panel
255             {
256                 Dock = DockStyle.Fill,
257                 BackColor = Color.LightGray
```

```
258             };
259
260             Label testLabel = new Label
261             {
262                 Text = "This is the Test Screen",
263                 Font = new Font("Arial", 16, FontStyle.Bold),
264                 Location = new Point(20, 20),
265                 AutoSize = true
266             };
267             testPanel.Controls.Add(testLabel);
268
269             // CH1 Logs
270             testResultsBox = new RichTextBox
271             {
272                 Location = new Point(20, 70),
273                 Width = 500,
274                 Height = 300,
275                 Multiline = true,
276                 ReadOnly = true,
277                 ScrollBars = RichTextBoxScrollBars.Vertical,
278                 DetectUrls = false
279             };
280             testPanel.Controls.Add(testResultsBox);
281
282             // CH2 Logs
283             testResultsBoxCH2 = new RichTextBox
284             {
285                 Location = new Point(20, 400),
286                 Width = 500,
287                 Height = 250,
288                 Multiline = true,
289                 ReadOnly = true,
290                 ScrollBars = RichTextBoxScrollBars.Vertical,
291                 DetectUrls = false
292             };
293             testPanel.Controls.Add(testResultsBoxCH2);
294
295             // CH1 Results
296             ch1Results = new TextBox
297             {
298                 Location = new Point(550, 70),
299                 Width = 800,
300                 Height = 300,
301                 Multiline = true,
302                 ReadOnly = true,
303                 ScrollBars = ScrollBars.Vertical,
304                 Name = "ch1Results"
305             };
306             testPanel.Controls.Add(ch1Results);
307
308             // CH2 Results
309             ch2Results = new TextBox
310             {
311                 Location = new Point(550, 400),
```

```
312                 Width = 800,
313                 Height = 250,
314                 Multiline = true,
315                 ReadOnly = true,
316                 ScrollBars = ScrollBars.Vertical,
317                 Name = "ch2Results"
318             };
319             testPanel.Controls.Add(ch2Results);
320 
321             // CH1 Test Button
322             Button testButtonCH1 = new Button
323             {
324                 Text = "Run Test CH1",
325                 Location = new Point(20, 660),
326                 Width = 150,
327                 Height = 40,
328                 BackColor = Color.Blue,
329                 ForeColor = Color.White
330             };
331             testButtonCH1.Click += async (s, e) => await RunImprovedTest("CH1");
332             testPanel.Controls.Add(testButtonCH1);
333 
334             // CH2 Test Button
335             Button testButtonCH2 = new Button
336             {
337                 Text = "Run Test CH2",
338                 Location = new Point(180, 660),
339                 Width = 150,
340                 Height = 40,
341                 BackColor = Color.Blue,
342                 ForeColor = Color.White
343             };
344             testButtonCH2.Click += async (s, e) => await RunImprovedTest("CH2");
345             testPanel.Controls.Add(testButtonCH2);
346 
347             // Stop Test Button
348             Button stopButton = new Button
349             {
350                 Text = "Stop Test",
351                 Location = new Point(340, 660),
352                 Width = 150,
353                 Height = 40,
354                 BackColor = Color.Red,
355                 ForeColor = Color.White
356             };
357             stopButton.Click += (s, e) => StopTest();
358             testPanel.Controls.Add(stopButton);
359 
360             timeLabel = new Label
361             {
362                 Text = "Test Time: 0s",
363                 Location = new Point(20, 710),
364                 AutoSize = true,
365                 Font = new Font("Arial", 12, FontStyle.Bold)
```

```
366             };
367             testPanel.Controls.Add(timeLabel);
368 
369             _testTimer = new System.Windows.Forms.Timer();
370             _testTimer.Interval = 500;
371             _testTimer.Tick += (s, e) =>
372             {
373                 if (_testStopwatch != null && _testStopwatch.IsRunning)
374                 {
375                     var ts = _testStopwatch.Elapsed;
376                     timeLabel.Text = $"Test Time: {ts.Minutes}m {ts.Seconds}s";
377                 }
378             };
379 
380             return testPanel;
381         }
382 
383 
384 
385 
386 
387         private void UpdateResultsTab()
388         {
389             if (_resultsBox == null)
390             {
391                 MessageBox.Show("Error: _resultsBox is NULL! Cannot update results.",
    "Debug Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
392                 return;
393             }
394 
395             StringBuilder results = new StringBuilder();
396 
397             foreach (var waveform in _paramStats)
398             {
399                 results.AppendLine($"{waveform.Key.ToUpper()} WAVE Results:");
400 
401                 foreach (var param in waveform.Value)
402                 {
403                     int total = param.Value.total;
404                     int passed = param.Value.pass;
405                     double successRate = total > 0 ? (passed / (double)total) * 100 : 0;
406 
407                     results.AppendLine($"   {param.Key.ToUpper()} => {passed}/{total}
    passed ({successRate:F1}%)");
408                 }
409 
410                 results.AppendLine();
411             }
412 
413             // Update `_resultsBox` safely
414             if (_resultsBox.InvokeRequired)
415             {
416                 _resultsBox.Invoke((MethodInvoker)delegate
417                 {
```

```
418                     _resultsBox.Text = results.ToString();
419                 });
420             }
421             else
422             {
423                 _resultsBox.Text = results.ToString();
424             }
425         }
426
427
428
429
430
431
432
433         private void LogTestResult(string waveform, string parameter, bool passed)
434         {
435             if (!_paramStats.ContainsKey(waveform))
436             {
437                 _paramStats[waveform] = new Dictionary<string, (int total, int pass)>();
438             }
439
440             if (!_paramStats[waveform].ContainsKey(parameter))
441             {
442                 _paramStats[waveform][parameter] = (0, 0);
443             }
444
445             var (total, pass) = _paramStats[waveform][parameter];
446             _paramStats[waveform][parameter] = (total + 1, pass + (passed ? 1 : 0));
447
448             UpdateResultsTab(); // This will refresh the UI immediately
449         }
450
451
452
453
454
455
456         private TextBox _testLogBox = null!;
457
458         private void UpdateTestLog(string message)
459         {
460             if (InvokeRequired)
461             {
462                 Invoke(new Action<string>(UpdateTestLog), message);
463                 return;
464             }
465
466             if (_testLogBox == null) return; // If not created yet, do nothing
467
468             _testLogBox.AppendText($"[{DateTime.Now:HH:mm:ss}] {message}
    {Environment.NewLine}");
469             _testLogBox.SelectionStart = _testLogBox.TextLength;
470             _testLogBox.ScrollToCaret();
```

```
471         }
472
473
474
475
476         private Panel CreateMainPanel()
477         {
478             Panel mainPanel = new Panel
479             {
480                 Dock = DockStyle.Fill
481             };
482
483             // Create a vertical layout to include the connection panel, status panel, and
    the split container
484             TableLayoutPanel mainLayout = new TableLayoutPanel
485             {
486                 Dock = DockStyle.Fill,
487                 RowCount = 3,
488                 ColumnCount = 1,
489                 Padding = new Padding(5),
490                 BackColor = ColorTranslator.FromHtml("#005cb9")
491             };
492             mainLayout.RowStyles.Add(new RowStyle(SizeType.Absolute, 100)); // Connection
    Panel at the top
493             mainLayout.RowStyles.Add(new RowStyle(SizeType.Percent, 100)); // Main GUI
    below
494             mainLayout.RowStyles.Add(new RowStyle(SizeType.Absolute, 100)); // Status Panel
    at the bottom
495
496             // Create SplitContainer for Oscilloscope (Left) and Waveform Generator (Right)
497             SplitContainer splitContainer = new SplitContainer
498             {
499                 Dock = DockStyle.Fill,
500                 Orientation = Orientation.Vertical,
501                 SplitterDistance = 750,
502                 BorderStyle = BorderStyle.FixedSingle,
503                 BackColor = ColorTranslator.FromHtml("#005cb9")
504             };
505
506             // Create TabControl for Oscilloscope Controls (Left Side)
507             TabControl oscilloscopeTabs = new TabControl
508             {
509                 Dock = DockStyle.Fill,
510                 BackColor = Color.White,
511                 Font = new Font("Arial", 10, FontStyle.Bold),
512                 ItemSize = new Size(160, 30)
513             };
514
515             TabPage timebaseTab = new TabPage("Timebase Control") { BackColor = Color.White
    };
516             timebaseTab.Controls.Add(CreateTimebasePanel());
517
518             TabPage channelTab = new TabPage("Channel Control") { BackColor = Color.White
    };
```

```
            channelTab.Controls.Add(CreateChannelPanel());

            TabPage waveformTab = new TabPage("Oscilloscope Waveform Generator") { BackColor = Color.White };
            waveformTab.Controls.Add(CreateWaveformPanel());

            TabPage triggerTab = new TabPage("Trigger Control") { BackColor = Color.White };
            triggerTab.Controls.Add(CreateTriggerPanel());

            TabPage oscilloscopeScreenTab = new TabPage("Oscilloscope Display") { BackColor = Color.White };

            // Display Panel for Oscilloscope
            oscilloscopeDisplay = new PictureBox
            {
                Dock = DockStyle.Fill,
                SizeMode = PictureBoxSizeMode.Zoom,
                BackColor = Color.Black
            };

            // Buttons for screen display control
            Button toggleDisplayButton = new Button
            {
                Text = "Start Display",
                Dock = DockStyle.Top,
                Height = 30,
                ForeColor = Color.White,
                BackColor = ColorTranslator.FromHtml("#003f7d")
            };
            toggleDisplayButton.Click += ToggleDisplay;

            Button screenshotButton = new Button
            {
                Text = "Take Screenshot",
                Dock = DockStyle.Top,
                Height = 30,
                ForeColor = Color.White,
                BackColor = ColorTranslator.FromHtml("#003f7d")
            };
            screenshotButton.Click += SaveScreenshot;

            // Panel to hold display and buttons
            Panel displayPanel = new Panel { Dock = DockStyle.Fill, BackColor = ColorTranslator.FromHtml("#005cb9") };
            displayPanel.Controls.Add(oscilloscopeDisplay);
            displayPanel.Controls.Add(toggleDisplayButton);
            displayPanel.Controls.Add(screenshotButton);

            oscilloscopeScreenTab.Controls.Add(displayPanel);
            oscilloscopeTabs.TabPages.AddRange(new[] { timebaseTab, channelTab, waveformTab, triggerTab, oscilloscopeScreenTab });

            splitContainer.Panel1.Controls.Add(oscilloscopeTabs);
```

```
568
569             // Create Dedicated Panel for Waveform Generator (Right Side)
570             Panel waveformGeneratorPanel = new Panel
571             {
572                 Dock = DockStyle.Fill,
573                 BorderStyle = BorderStyle.FixedSingle,
574                 BackColor = Color.White
575             };
576
577             waveformGeneratorPanel.Controls.Add(CreateFunctionGeneratorPanel());
578             splitContainer.Panel2.Controls.Add(waveformGeneratorPanel);
579
580             // Add the connection panel, main GUI, and status panel
581             mainLayout.Controls.Add(CreateConnectionPanel(), 0, 0);
582             mainLayout.Controls.Add(splitContainer, 0, 1);
583             mainLayout.Controls.Add(CreateStatusPanel(), 0, 2); // Add Status Panel
584
585             mainPanel.Controls.Add(mainLayout);
586             return mainPanel;
587         }
588
589
590
591         private Panel CreateConnectionPanel()
592         {
593             Panel panel = new Panel { Dock = DockStyle.Fill, BorderStyle =
    BorderStyle.FixedSingle };
594
595             Label addressLabel = new Label { Text = "VISA Address:", Location = new
    Point(10, 15), AutoSize = true };
596             TextBox addressBox = new TextBox
597             {
598                 Text = "USB0::0x2A8D::0x1770::MY58491960::0::INSTR",
599                 Location = new Point(115, 12),
600                 Width = 400,
601                 Name = "addressBox"
602             };
603
604             Button connectButton = new Button
605             {
606                 Text = "Connect",
607                 Location = new Point(515, 12),
608                 Width = 100,
609                 Name = "connectButton"
610             };
611             connectButton.Click += async (s, e) => await ConnectButtonClick();
612
613             Label statusLabel = new Label
614             {
615                 Text = "Status: Disconnected",
616                 Location = new Point(10, 45),
617                 AutoSize = true,
618                 Name = "statusLabel",
619                 ForeColor = Color.Red
```

```
620             };
621
622             panel.Controls.AddRange(new Control[] { addressLabel, addressBox,
    connectButton, statusLabel });
623             return panel;
624         }
625
626
627
628
629         private TabControl CreateControlTabs()
630         {
631             TabControl tabs = new TabControl
632             {
633                 Dock = DockStyle.Fill
634             };
635
636             // Timebase Control Tab
637             TabPage timebaseTab = new TabPage("Timebase Control");
638             Panel timebasePanel = CreateTimebasePanel();
639             timebaseTab.Controls.Add(timebasePanel);
640
641             // Channel Control Tab
642             TabPage channelTab = new TabPage("Channel Control");
643             Panel channelPanel = CreateChannelPanel();
644             channelTab.Controls.Add(channelPanel);
645
646             // Waveform Generator Tab
647             TabPage waveformTab = new TabPage("Waveform Generator");
648             Panel waveformPanel = CreateWaveformPanel();
649             waveformTab.Controls.Add(waveformPanel);
650
651             // Trigger Control Tab
652             TabPage triggerTab = new TabPage("Trigger Control");
653             Panel triggerPanel = CreateTriggerPanel();
654             triggerTab.Controls.Add(triggerPanel);
655
656             // Create the function generator tab
657             TabPage functionGeneratorTab = new TabPage("33500B Waveform Generator")
658             {
659                 BackColor = Color.White
660             };
661             functionGeneratorTab.Controls.Add(CreateFunctionGeneratorPanel());
662
663             // Add it to the main tab control
664             mainTabs.TabPages.Add(functionGeneratorTab);
665
666
667             tabs.TabPages.AddRange(new TabPage[] { timebaseTab, channelTab, waveformTab,
    triggerTab, functionGeneratorTab});
668             return tabs;
669         }
670
671         private Panel CreateFunctionGeneratorPanel()
```

```
672         {
673             var panel = new Panel
674             {
675                 Dock = DockStyle.Fill,
676                 AutoScroll = true
677             };
678 
679             // Create a vertical layout with the connection box on top and CH1/CH2 settings
    below
680             TableLayoutPanel mainLayout = new TableLayoutPanel
681             {
682                 Dock = DockStyle.Fill,
683                 RowCount = 2,
684                 ColumnCount = 1,
685                 Padding = new Padding(5),
686                 BackColor = Color.White
687             };
688             mainLayout.RowStyles.Add(new RowStyle(SizeType.Absolute, 50)); // Connection
    Panel
689             mainLayout.RowStyles.Add(new RowStyle(SizeType.Percent, 100)); // CH1 and CH2
    Panels
690 
691             // Add connection panel at the top
692             mainLayout.Controls.Add(CreateFunctionGeneratorConnectionPanel(), 0, 0);
693 
694             // Create a horizontal layout for CH1 and CH2 settings
695             TableLayoutPanel channelsLayout = new TableLayoutPanel
696             {
697                 Dock = DockStyle.Fill,
698                 ColumnCount = 2,
699                 RowCount = 1,
700                 Padding = new Padding(5),
701                 BackColor = Color.White
702             };
703             channelsLayout.ColumnStyles.Add(new ColumnStyle(SizeType.Percent, 50)); // CH1
    Panel
704             channelsLayout.ColumnStyles.Add(new ColumnStyle(SizeType.Percent, 50)); // CH2
    Panel
705 
706             // Create CH1 and CH2 Panels
707             Panel ch1Panel = CreateChannelSettingsPanel("CH1");
708             Panel ch2Panel = CreateChannelSettingsPanel("CH2");
709 
710             channelsLayout.Controls.Add(ch1Panel, 0, 0);
711             channelsLayout.Controls.Add(ch2Panel, 1, 0);
712 
713             mainLayout.Controls.Add(channelsLayout, 0, 1);
714 
715             panel.Controls.Add(mainLayout);
716             return panel;
717         }
718 
719         private Panel CreateFunctionGeneratorConnectionPanel()
720         {
```

```
721             Panel panel = new Panel { Dock = DockStyle.Fill, BorderStyle =
    BorderStyle.FixedSingle };
722
723             Label addressLabel = new Label { Text = "33500B IP Address:", Location = new
    Point(10, 15), AutoSize = true };
724             TextBox addressBox = new TextBox
725             {
726                 Text = "169.254.5.21",
727                 Location = new Point(150, 12),
728                 Width = 250,
729                 Name = "functionGenAddressBox"
730             };
731
732             Button connectButton = new Button
733             {
734                 Text = "Connect",
735                 Location = new Point(400, 12),
736                 Width = 100,
737                 Name = "functionGenConnectButton"
738             };
739             connectButton.Click += async (s, e) => await ConnectToFunctionGen↩
    erator(addressBox.Text);
740
741             Label statusLabel = new Label
742             {
743                 Text = "Status: Disconnected",
744                 Location = new Point(520, 15),
745                 AutoSize = true,
746                 Name = "functionGenStatusLabel",
747                 ForeColor = Color.Red
748             };
749
750             panel.Controls.AddRange(new Control[] { addressLabel, addressBox,
    connectButton, statusLabel });
751             return panel;
752         }
753
754         private Panel CreateChannelSettingsPanel(string channel)
755         {
756             Panel panel = new Panel
757             {
758                 Dock = DockStyle.Fill,
759                 BorderStyle = BorderStyle.FixedSingle,
760                 BackColor = Color.White
761             };
762
763             Label titleLabel = new Label
764             {
765                 Text = $"Waveform Generator - {channel}",
766                 Dock = DockStyle.Top,
767                 Font = new Font("Arial", 12, FontStyle.Bold),
768                 ForeColor = Color.White,
769                 BackColor = ColorTranslator.FromHtml("#003f7d"),
770                 TextAlign = ContentAlignment.MiddleCenter,
```

```
771                 Height = 40
772             };
773 
774             Label waveLabel = new Label
775             {
776                 Text = "Waveform Type:",
777                 Location = new Point(10, 50),
778                 AutoSize = true
779             };
780 
781             ComboBox waveformCombo = new ComboBox
782             {
783                 Location = new Point(140, 47),
784                 Width = 120,
785                 DropDownStyle = ComboBoxStyle.DropDownList,
786                 Name = $"waveformCombo_{channel}"
787             };
788             waveformCombo.Items.AddRange(new[] { "SIN", "SQU", "RAMP", "PULS", "NOIS", "DC"
    });
789             waveformCombo.SelectedIndex = 0; // Default: SIN
790 
791             //  Create a dedicated panel for waveform parameters
792             Panel parameterPanel = new Panel
793             {
794                 Location = new Point(10, 80),
795                 Size = new Size(280, 250),
796                 Name = $"parameterPanel_{channel}"
797             };
798 
799             //  Call UpdateParameterPanel for the default waveform type (SIN)
800             UpdateParameterPanel(parameterPanel, "SIN");  //
801 
802             //  Attach event handler for waveform selection change
803             waveformCombo.SelectedIndexChanged += (s, e) =>
804             {
805                 string selectedWaveform = waveformCombo.SelectedItem?.ToString()?.ToUpper()
    ?? "SIN";
806                 UpdateParameterPanel(parameterPanel, selectedWaveform);
807             };
808 
809             CheckBox outputEnableCheck = new CheckBox
810             {
811                 Text = "Output On",
812                 Location = new Point(10, 340),
813                 AutoSize = true
814             };
815 
816             Button applyButton = new Button
817             {
818                 Text = "Apply Settings",
819                 Location = new Point(10, 370),
820                 Width = 120
821             };
822 
```

```
            applyButton.Click += async (s, e) =>
            {
                await ApplyChannelSettings(channel, waveformCombo, parameterPanel,
outputEnableCheck);
            };

            panel.Controls.AddRange(new Control[]
            {
                titleLabel, waveLabel, waveformCombo, parameterPanel,
                outputEnableCheck, applyButton
            });

            return panel;
        }




        private async Task ApplySettings()
        {
            if (!_isConnectedToFunctionGenerator || _functionGenerator == null)
            {
                UpdateStatus("Not connected.");
                return;
            }

            try
            {
                // 1) Determine which channel from the ComboBox
                UpdateSelectedChannel();  // sets _selectedChannel = "CH1" or "CH2"

                // 2) DO NOT call SetChannelAsync(...) here; it triggers 33500B error
                // await _functionGenerator.SetChannelAsync(_selectedChannel);  // Removed

                // 3) Check waveform type
                string shape = _waveformCombo.SelectedItem?.ToString()?.ToUpper() ?? "SIN";

                // 4) Apply the waveform to that channel
                await _functionGenerator.SetWaveformAsync(_selectedChannel, shape);

                // 5) If not DC/NOIS, set frequency
                if (shape != "DC" && shape != "NOIS")
                {
                    double freq = (double)_freqInput.Value;
                    await _functionGenerator.SetFrequencyAsync(_selectedChannel, freq);
                }

                // 6) If not DC, set amplitude
                if (shape != "DC")
                {
                    double amp = (double)_amplitudeInput.Value;
                    await _functionGenerator.SetAmplitudeAsync(_selectedChannel, amp);
                }
```

```
876
877                 // 7) Always set offset
878                 double offs = (double)_offsetInput.Value;
879                 await _functionGenerator.SetOffsetAsync(_selectedChannel, offs);
880
881                 // 8) Extended parameters
882                 if (_phaseInput != null)
883                 {
884                     double phaseDeg = (double)_phaseInput.Value;
885                     await _functionGenerator.SetPhaseAsync(_selectedChannel, phaseDeg);
886                 }
887
888                 if (_dutyInput != null)
889                 {
890                     double duty = (double)_dutyInput.Value;
891                     await _functionGenerator.SetSquareDutyCycleAsync(_selectedChannel,
    duty);
892                 }
893
894                 if (_symmetryInput != null)
895                 {
896                     double sym = (double)_symmetryInput.Value;
897                     await _functionGenerator.SetRampSymmetryAsync(_selectedChannel, sym);
898                 }
899
900                 if (_pulseWidthInput != null)
901                 {
902                     double widthSec = (double)_pulseWidthInput.Value * 1e-6;
903                     await _functionGenerator.SetPulseWidthAsync(_selectedChannel,
    widthSec);
904                 }
905
906                 if (_leadEdgeInput != null)
907                 {
908                     double leadSec = (double)_leadEdgeInput.Value * 1e-9;
909                     await _functionGenerator.SetPulseLeadingEdgeAsync(_selectedChannel,
    leadSec);
910                 }
911
912                 if (_trailEdgeInput != null)
913                 {
914                     double trailSec = (double)_trailEdgeInput.Value * 1e-9;
915                     await _functionGenerator.SetPulseTrailingEdgeAsync(_selectedChannel,
    trailSec);
916                 }
917
918                 if (_noiseBandwidthInput != null)
919                 {
920                     double bw = (double)_noiseBandwidthInput.Value;
921                     await _functionGenerator.SetNoiseBandwidthAsync(_selectedChannel, bw);
922                 }
923
924                 // 9) Finally, enable or disable output for that channel
925                 bool enableOutput = _outputEnableCheck.Checked;
```

```
926                 await _functionGenerator.EnableChannelAsync(_selectedChannel,
    enableOutput);
927
928                 UpdateStatus($"Settings applied for {_selectedChannel}: {shape}
    waveform.");
929             }
930             catch (Exception ex)
931             {
932                 UpdateStatus($"Apply failed: {ex.Message}");
933             }
934         }
935
936
937
938
939
940
941         private async Task ApplyChannelSettings(
942             string channel,
943             ComboBox waveformCombo,
944             Panel parameterPanel,
945             CheckBox outputEnableCheck)
946         {
947             if (!_isConnectedToFunctionGenerator || _functionGenerator == null)
948             {
949                 UpdateStatus("Not connected.");
950                 return;
951             }
952
953             try
954             {
955                 string shape = waveformCombo.SelectedItem?.ToString()?.ToUpper() ?? "SIN";
956                 await _functionGenerator.SetWaveformAsync(channel, shape);
957
958                 // Helper function to find numeric controls in parameterPanel
959                 NumericUpDown? GetNumeric(string name)
960                     => parameterPanel.Controls.Find(name, true).FirstOrDefault() as
    NumericUpDown;
961
962                 // If not DC/NOIS, set freq
963                 double freq = 0.0;
964                 if (shape != "DC" && shape != "NOIS")
965                 {
966                     freq = (double)(GetNumeric("freqInput")?.Value ?? 0);
967                     await _functionGenerator.SetFrequencyAsync(channel, freq);
968                 }
969
970                 // If not DC, set amplitude
971                 if (shape != "DC")
972                 {
973                     double amp = (double)(GetNumeric("ampInput")?.Value ?? 0);
974                     await _functionGenerator.SetAmplitudeAsync(channel, amp);
975                 }
976
```

```
977                 // Offset (all waves)
978                 double offset = (double)(GetNumeric("offsetInput")?.Value ?? 0);
979                 await _functionGenerator.SetOffsetAsync(channel, offset);
980 
981                 // If wave == NOIS, set bandwidth
982                 if (shape == "NOIS")
983                 {
984                     double bandwidth = (double)(GetNumeric("noiseBandwidthInput")?.Value ??
    0);
985                     await _functionGenerator.SetNoiseBandwidthAsync(channel, bandwidth);
986                 }
987 
988                 // If wave uses phase
989                 if (shape == "SIN" || shape == "SQU" || shape == "RAMP" || shape == "PULS")
990                 {
991                     double phase = (double)(GetNumeric("phaseInput")?.Value ?? 0);
992                     await _functionGenerator.SetPhaseAsync(channel, phase);
993                 }
994 
995                 // If wave == SQU, set duty
996                 if (shape == "SQU")
997                 {
998                     double duty = (double)(GetNumeric("dutyInput")?.Value ?? 0);
999                     await _functionGenerator.SetSquareDutyCycleAsync(channel, duty);
1000                }
1001 
1002                // If wave == RAMP, set symmetry
1003                if (shape == "RAMP")
1004                {
1005                    double symmetry = (double)(GetNumeric("symmetryInput")?.Value ?? 0);
1006                    await _functionGenerator.SetRampSymmetryAsync(channel, symmetry);
1007                }
1008 
1009                // If wave == PULS, set pulse width & edges
1010                if (shape == "PULS")
1011                {
1012                    // Values from UI in microseconds and nanoseconds
1013                    double widthUs = (double)(GetNumeric("pulseWidthInput")?.Value ?? 0);
1014                    double leadNs  = (double)(GetNumeric("leadEdgeInput")?.Value ?? 0);
1015                    double trailNs = (double)(GetNumeric("trailEdgeInput")?.Value ?? 0);
1016 
1017                    // Convert to seconds
1018                    double widthSec = widthUs * 1e-6;
1019                    double leadSec  = leadNs * 1e-9;
1020                    double trailSec = trailNs * 1e-9;
1021 
1022                    // NEW: clamp logic for PULSE to avoid invalid parameters
1023                    if (freq > 0)  // If freq=0, user set 0 or DC => can't clamp
1024                    {
1025                        double periodSec = 1.0 / freq;
1026 
1027                        // 1) If widthSec > ~80% of period
1028                        if (widthSec > periodSec * 0.8)
1029                            widthSec = periodSec * 0.8;
```

```
                        // 2) If leadSec + trailSec + widthSec too large
                        double sum = leadSec + trailSec + widthSec;
                        if (sum > periodSec * 0.9)
                        {
                            double leftover = periodSec * 0.9 - (leadSec + trailSec);
                            if (leftover < 0) leftover = 0;
                            widthSec = Math.Min(widthSec, leftover);
                        }
                    }

                    // Now set them in the function generator
                    await _functionGenerator.SetPulseWidthAsync(channel, widthSec);
                    await _functionGenerator.SetPulseLeadingEdgeAsync(channel, leadSec);
                    await _functionGenerator.SetPulseTrailingEdgeAsync(channel, trailSec);
                }

                // Finally, enable/disable output
                bool enableOutput = outputEnableCheck.Checked;
                await _functionGenerator.EnableChannelAsync(channel, enableOutput);

                UpdateStatus($"Settings applied for {channel}: {shape} waveform.");
            }
            catch (Exception ex)
            {
                UpdateStatus($"Apply failed: {ex.Message}");
            }
        }









        private async Task OnOutputEnableChanged(bool enable)
        {
            if (!_isConnectedToFunctionGenerator || _functionGenerator == null)
            {
                UpdateStatus("Not connected to function generator.");
                return;
            }

            try
            {
                await _functionGenerator.SetOutputStateAsync(enable);
                UpdateStatus($"Output turned {(enable ? "ON" : "OFF")}");
            }
```

```
1084            catch (Exception ex)
1085            {
1086                UpdateStatus($"Failed to set output: {ex.Message}");
1087            }
1088        }
1089
1090
1091
1092        private void UpdateParameterPanel(Panel parameterPanel, string shape)
1093        {
1094            parameterPanel.Controls.Clear(); // Remove old controls
1095            int yOffset = 0;
1096
1097            // Helper function to add numeric input fields dynamically
1098            NumericUpDown AddNumeric(string labelText, string name, double min, double max,
    double defaultVal = 0, double inc = 0.001)
1099            {
1100                Label lbl = new Label
1101                {
1102                    Text = labelText,
1103                    Location = new Point(0, yOffset),
1104                    AutoSize = true
1105                };
1106                NumericUpDown num = new NumericUpDown
1107                {
1108                    Name = name,
1109                    Location = new Point(140, yOffset),
1110                    Width = 120,
1111                    DecimalPlaces = 6,
1112                    Minimum = (decimal)min,
1113                    Maximum = (decimal)max,
1114                    Increment = (decimal)inc,
1115                    Value = (decimal)defaultVal
1116                };
1117                parameterPanel.Controls.Add(lbl);
1118                parameterPanel.Controls.Add(num);
1119                yOffset += 35;
1120                return num;
1121            }
1122
1123            // Add  relevant parameters for each waveform type
1124            if (shape != "DC" && shape != "NOIS")
1125            {
1126                AddNumeric("Frequency (Hz):", "freqInput", 0.001, 100_000_000, 1000);
1127            }
1128
1129            if (shape != "DC")
1130            {
1131                AddNumeric("Amplitude (Vpp):", "ampInput", 0.001, 20, 1);
1132            }
1133
1134            AddNumeric("Offset (V):", "offsetInput", -10, 10, 0);
1135
1136            if (shape == "NOIS")
```

```
            {
                AddNumeric("Bandwidth (Hz):", "noiseBandwidthInput", 1, 100_000_000,
10_000);
            }

            if (shape == "SIN" || shape == "SQU" || shape == "RAMP" || shape == "PULS")
            {
                AddNumeric("Phase (deg):", "phaseInput", -360, 360, 0, 0.1);
            }

            if (shape == "SQU")
            {
                AddNumeric("Duty Cycle (%):", "dutyInput", 0.1, 99.9, 50, 0.1);
            }

            if (shape == "RAMP")
            {
                AddNumeric("Symmetry (%):", "symmetryInput", 0.1, 99.9, 50, 0.1);
            }

            if (shape == "PULS")
            {
                AddNumeric("Pulse Width (µs):", "pulseWidthInput", 1, 1_000_000_000, 500,
1);
                AddNumeric("Leading Edge (ns):", "leadEdgeInput", 0.1, 1_000_000, 8.4,
0.1);
                AddNumeric("Trailing Edge (ns):", "trailEdgeInput", 0.1, 1_000_000, 8.4,
0.1);
            }

            parameterPanel.Invalidate();
        }



        private void DisplayWaveform(double[] xData, double[] yData)
        {
            try
            {
                if (oscilloscopeDisplay.InvokeRequired)
                {
                    oscilloscopeDisplay.Invoke(new Action(() => DisplayWaveform(xData,
yData)));
                    return;
                }

                int width = oscilloscopeDisplay.Width;
                int height = oscilloscopeDisplay.Height;

                Bitmap bmp = new Bitmap(width, height);
                using (Graphics g = Graphics.FromImage(bmp))
                {
                    g.Clear(Color.Black);
                    Pen wavePen = new Pen(Color.Lime, 2);
```

```
1186
1187                    for (int i = 1; i < xData.Length; i++)
1188                    {
1189                        int x1 = (int)((xData[i - 1] - xData[0]) / (xData.Last() - xData[0]) * width);
1190                        int y1 = height - (int)((yData[i - 1] - yData.Min()) / (yData.Max() - yData.Min()) * height);
1191                        int x2 = (int)((xData[i] - xData[0]) / (xData.Last() - xData[0]) * width);
1192                        int y2 = height - (int)((yData[i] - yData.Min()) / (yData.Max() - yData.Min()) * height);
1193
1194                        g.DrawLine(wavePen, x1, y1, x2, y2);
1195                    }
1196                }
1197
1198                oscilloscopeDisplay.Image = bmp;
1199            }
1200            catch (Exception ex)
1201            {
1202                UpdateStatus($"Failed to display waveform: {ex.Message}");
1203            }
1204        }
1205
1206
1207        private async Task UpdateOscilloscopeDisplay()
1208        {
1209            while (isConnected && _isDisplayEnabled)  // Update only when enabled
1210            {
1211                try
1212                {
1213                    if (oscilloscope == null) return;
1214
1215                    await oscilloscope.SendCommandAsync(":RUN");  // Ensure oscilloscope is running
1216                    await oscilloscope.SendCommandAsync(":WAVeform:SOURce CHAN1");
1217                    await oscilloscope.SendCommandAsync(":WAVeform:FORMat ASCii");
1218                    await oscilloscope.SendCommandAsync(":WAVeform:POINts 1000");  // Reduce number of points
1219
1220                    double xIncrement = Convert.ToDouble(await oscilloscope.QueryAsync(":WAVeform:XINCrement?"));
1221                    double xOrigin = Convert.ToDouble(await oscilloscope.QueryAsync(":WAVeform:XORigin?"));
1222                    double yIncrement = Convert.ToDouble(await oscilloscope.QueryAsync(":WAVeform:YINCrement?"));
1223                    double yOrigin = Convert.ToDouble(await oscilloscope.QueryAsync(":WAVeform:YORigin?"));
1224                    double yReference = Convert.ToDouble(await oscilloscope.QueryAsync(":WAVeform:YREFerence?"));
1225
1226                    string rawWaveformData = await oscilloscope.QueryAsync(":WAVeform:DATA?");
1227
1228                    //  **Check if the response is valid before processing**
```

```
1229                    if (string.IsNullOrWhiteSpace(rawWaveformData))
1230                    {
1231                        UpdateStatus("Error: Received empty waveform data.");
1232                        continue;
1233                    }
1234
1235                    string[] dataPoints = rawWaveformData.Split(',');
1236
1237                    //  **Check if we actually received numeric data**
1238                    if (dataPoints.Length < 2)
1239                    {
1240                        UpdateStatus("Error: Waveform data format is incorrect.");
1241                        continue;
1242                    }
1243
1244                    // Convert Data to Double Format
1245                    double[] yData = dataPoints
1246                        .Select(point =>
1247                        {
1248                            double value;
1249                            return double.TryParse(point, out value) ? (value - yReference)
    * yIncrement + yOrigin : double.NaN;
1250                        })
1251                        .Where(v => !double.IsNaN(v))  // Filter out invalid values
1252                        .ToArray();
1253
1254                    double[] xData = Enumerable.Range(0, yData.Length)
1255                        .Select(i => xOrigin + i * xIncrement)
1256                        .ToArray();
1257
1258                    //  **Check if waveform data is empty**
1259                    if (yData.Length == 0)
1260                    {
1261                        UpdateStatus("Error: No valid waveform data received.");
1262                        continue;
1263                    }
1264
1265                    DisplayWaveform(xData, yData);
1266                }
1267                catch (Exception ex)
1268                {
1269                    UpdateStatus($"Error fetching waveform: {ex.Message}");
1270                }
1271
1272                await Task.Delay(50);  // Refresh every 50ms
1273            }
1274        }
1275
1276
1277
1278
1279
1280
1281
```

```
1282        private async Task ConnectToFunctionGenerator(string ip)
1283        {
1284            TextBox functionGenAddressBox = (TextBox)Controls.Find("functionGenAddressBox",
    true)[0];
1285            Button functionGenConnectButton = (Button)Controls.Find("functionGenConnectBu↩
    tton", true)[0];
1286            Label functionGenStatusLabel = (Label)Controls.Find("functionGenStatusLabel",
    true)[0];
1287
1288            if (_isConnectedToFunctionGenerator) // If already connected, disconnect
1289            {
1290                try
1291                {
1292                    if (_functionGenerator != null)
1293                    {
1294                        await _functionGenerator.DisposeAsync();
1295                        _functionGenerator = null;
1296                    }
1297
1298                    _isConnectedToFunctionGenerator = false;
1299
1300                    // Re-enable the function generator address box
1301                    functionGenAddressBox.Enabled = true;
1302
1303                    functionGenStatusLabel.Text = "Status: Disconnected";
1304                    functionGenStatusLabel.ForeColor = Color.Red;
1305                    functionGenConnectButton.Text = "Connect";
1306
1307                    UpdateStatus("Disconnected from Keysight 33500B.");
1308                }
1309                catch (Exception ex)
1310                {
1311                    UpdateStatus($"Failed to disconnect: {ex.Message}");
1312                }
1313                return; // Exit function
1314            }
1315
1316            try
1317            {
1318                var lan = new LanCommunication(ip);
1319                await lan.ConnectAsync();
1320
1321                _functionGenerator = new FunctionGeneratorController(lan);
1322                await _functionGenerator.InitializeAsync();
1323                _isConnectedToFunctionGenerator = true;
1324
1325                // Disable the function generator address box while connected
1326                functionGenAddressBox.Enabled = false;
1327
1328                functionGenStatusLabel.Text = "Status: Connected";
1329                functionGenStatusLabel.ForeColor = Color.Green;
1330                functionGenConnectButton.Text = "Disconnect";
1331
1332                UpdateStatus($"Connected to Keysight 33500B at {ip}.");
```

```
            }
            catch (Exception ex)
            {
                functionGenStatusLabel.Text = "Status: Connection Failed";
                functionGenStatusLabel.ForeColor = Color.Red;
                UpdateStatus($"Failed to connect to function generator: {ex.Message}");
            }
        }




        private async Task ApplyFunctionGeneratorSettings()
        {
            if (!_isConnectedToFunctionGenerator || _functionGenerator == null)
            {
                UpdateStatus("Not connected to Keysight 33500B.");
                return;
            }

            try
            {
                ComboBox typeBox = (ComboBox)Controls.Find("waveformType", true)[0];
                string waveform = typeBox.SelectedItem?.ToString()?.ToUpper() ?? "SIN";

                // Detect selected channel
                string selectedChannel = _selectedChannel; // Should be "CH1" or "CH2"

                var parameterPanel = Controls.Find("parameterPanel", true)[0];
                NumericUpDown? GetInput(string name) => parameterPanel.Controls.Find(name, true).FirstOrDefault() as NumericUpDown;

                double frequency = (double)(GetInput("waveformFreq")?.Value ?? 0m);
                double amplitude = (double)(GetInput("waveformAmp")?.Value ?? 0m);
                double offset = (double)(GetInput("waveformOffset")?.Value ?? 0m);
                double phase = (double)(GetInput("waveformPhase")?.Value ?? 0m);
                double duty = (double)(GetInput("waveformDuty")?.Value ?? 50m);
                double symmetry = (double)(GetInput("waveformSymmetry")?.Value ?? 50m);
                double widthNs = (double)(GetInput("waveformWidth")?.Value ?? 100m);
                double leadEdge = (double)(GetInput("waveformLead")?.Value ?? 10m);
                double trailEdge = (double)(GetInput("waveformTrail")?.Value ?? 10m);
                double bandwidth = (double)(GetInput("waveformBandwidth")?.Value ?? 5000m);

                // Ensure the correct channel is selected before applying settings
                await _functionGenerator.SetChannelAsync(selectedChannel);

                // Apply common parameters
                await _functionGenerator.SetWaveformAsync(selectedChannel, waveform);
                await _functionGenerator.SetFrequencyAsync(selectedChannel, frequency);
                await _functionGenerator.SetAmplitudeAsync(selectedChannel, amplitude);
                await _functionGenerator.SetOffsetAsync(selectedChannel, offset);
                await _functionGenerator.SetPhaseAsync(selectedChannel, phase);

```

```
1386                // Apply waveform-specific settings
1387                switch (waveform)
1388                {
1389                    case "SQUARE":
1390                        await _functionGenerator.SetSquareDutyCycleAsync(selectedChannel,
    duty);
1391                        break;
1392                    case "RAMP":
1393                        await _functionGenerator.SetRampSymmetryAsync(selectedChannel,
    symmetry);
1394                        break;
1395                    case "PULSE":
1396                        await _functionGenerator.SetPulseWidthAsync(selectedChannel,
    widthNs);
1397                        await _functionGenerator.SetPulseLeadingEdgeAsync(selectedChannel,
    leadEdge);
1398                        await _functionGenerator.SetPulseTrailingEdgeAsync(selectedChannel,
    trailEdge);
1399                        break;
1400                    case "NOISE":
1401                        await _functionGenerator.SetNoiseBandwidthAsync(selectedChannel,
    bandwidth);
1402                        break;
1403                }
1404
1405                UpdateStatus($"Waveform applied: {waveform} - Channel: {selectedChannel} -
    Freq: {frequency}Hz, Amp: {amplitude}V, Offset: {offset}V");
1406            }
1407            catch (Exception ex)
1408            {
1409                UpdateStatus($"Failed to apply settings: {ex.Message}");
1410            }
1411        }
1412
1413
1414
1415
1416        private Panel CreateTriggerPanel()
1417        {
1418            Panel panel = new Panel { Dock = DockStyle.Fill };
1419
1420            // Trigger Type
1421            Label typeLabel = new Label
1422            {
1423                Text = "Trigger Type:",
1424                Location = new Point(10, 20),
1425                AutoSize = true
1426            };
1427            ComboBox triggerTypeCombo = new ComboBox
1428            {
1429                Location = new Point(140, 17),
1430                Width = 100,
1431                Name = "triggerTypeCombo",
1432                DropDownStyle = ComboBoxStyle.DropDownList
```

```
1433            };
1434            triggerTypeCombo.Items.AddRange(new[] { "EDGE", "PULSE", "PATTERN" });
1435            triggerTypeCombo.SelectedIndex = 0;
1436
1437            // Trigger Sweep Mode (AUTO, NORM, SING)
1438            Label modeLabel = new Label
1439            {
1440                Text = "Trigger Sweep:",
1441                Location = new Point(10, 60),     // pick a new Y position
1442                AutoSize = true
1443            };
1444            ComboBox modeCombo = new ComboBox
1445            {
1446                Location = new Point(140, 57),
1447                Width = 100,
1448                Name = "triggerMode",
1449                DropDownStyle = ComboBoxStyle.DropDownList
1450            };
1451            modeCombo.Items.AddRange(new[] { "AUTO", "NORM", "SING" });
1452            modeCombo.SelectedIndex = 0;
1453
1454            // Trigger Source
1455            Label sourceLabel = new Label
1456            {
1457                Text = "Trigger Source:",
1458                Location = new Point(10, 100),
1459                AutoSize = true
1460            };
1461            ComboBox sourceCombo = new ComboBox
1462            {
1463                Location = new Point(140, 97),
1464                Width = 100,
1465                Name = "triggerSource",
1466                DropDownStyle = ComboBoxStyle.DropDownList
1467            };
1468            sourceCombo.Items.AddRange(new[] { "CHAN1", "CHAN2", "CHAN3", "CHAN4", "LINE",
    "WGEN" });
1469            sourceCombo.SelectedIndex = 0;
1470
1471            // Trigger Slope
1472            Label slopeLabel = new Label
1473            {
1474                Text = "Trigger Slope:",
1475                Location = new Point(10, 140),
1476                AutoSize = true
1477            };
1478            ComboBox slopeCombo = new ComboBox
1479            {
1480                Location = new Point(140, 137),
1481                Width = 100,
1482                Name = "triggerSlope",
1483                DropDownStyle = ComboBoxStyle.DropDownList
1484            };
1485            slopeCombo.Items.AddRange(new[] { "POS", "NEG" });
```

```
            slopeCombo.SelectedIndex = 0;

            // Trigger Level
            Label levelLabel = new Label
            {
                Text = "Trigger Level (V):",
                Location = new Point(10, 180),
                AutoSize = true
            };
            NumericUpDown levelInput = new NumericUpDown
            {
                Location = new Point(140, 177),
                Width = 100,
                Minimum = -50,
                Maximum = 50,
                DecimalPlaces = 3,
                Increment = 0.01m,
                Value = 0,
                Name = "triggerLevel"
            };

            // Apply Trigger Button
            Button applyTriggerButton = new Button
            {
                Text = "Apply Trigger",
                Location = new Point(140, 220),
                Width = 100
            };
            applyTriggerButton.Click += async (s, e) => await ApplyTriggerSettings();

            // Add all controls to the panel
            panel.Controls.AddRange(new Control[]
            {
                typeLabel, triggerTypeCombo,
                modeLabel, modeCombo,
                sourceLabel, sourceCombo,
                slopeLabel, slopeCombo,
                levelLabel, levelInput,
                applyTriggerButton
            });

            return panel;
        }

        private async Task StartOscilloscope()
        {
            if (!isConnected || oscilloscope == null)
            {
                UpdateStatus("Error: Not connected to oscilloscope.");
                return;
            }

            try
            {
```

```
1540                 await oscilloscope.SendCommandAsync(":RUN");  // Start continuous
     acquisition
1541                 UpdateStatus("Oscilloscope Running.");
1542             }
1543             catch (Exception ex)
1544             {
1545                 UpdateStatus($"Failed to start oscilloscope: {ex.Message}");
1546             }
1547         }
1548
1549
1550         private async Task StopOscilloscope()
1551         {
1552             if (!isConnected || oscilloscope == null)
1553             {
1554                 UpdateStatus("Error: Not connected to oscilloscope.");
1555                 return;
1556             }
1557
1558             try
1559             {
1560                 await oscilloscope.SendCommandAsync(":STOP");  // Stop continuous
     acquisition
1561                 UpdateStatus("Oscilloscope Stopped.");
1562             }
1563             catch (Exception ex)
1564             {
1565                 UpdateStatus($"Failed to stop oscilloscope: {ex.Message}");
1566             }
1567         }
1568
1569
1570
1571
1572
1573         private async Task SingleAcquisition()
1574         {
1575             if (!isConnected || oscilloscope == null)
1576             {
1577                 UpdateStatus("Error: Not connected to oscilloscope.");
1578                 return;
1579             }
1580
1581             try
1582             {
1583                 await oscilloscope.SendCommandAsync(":SINGle");  // Correct SCPI command
1584                 UpdateStatus("Performed Single Acquisition.");
1585             }
1586             catch (Exception ex)
1587             {
1588                 UpdateStatus($"Failed to trigger single acquisition: {ex.Message}");
1589             }
1590         }
1591
```

```
1592
1593        private async Task AutoScale()
1594        {
1595            if (!isConnected || oscilloscope == null)
1596            {
1597                UpdateStatus("Error: Not connected to oscilloscope.");
1598                return;
1599            }
1600
1601            try
1602            {
1603                await oscilloscope.SendCommandAsync(":AUTOSCALE");
1604                UpdateStatus("Auto Scale applied.");
1605            }
1606            catch (Exception ex)
1607            {
1608                UpdateStatus($"Failed to apply Auto Scale: {ex.Message}");
1609            }
1610        }
1611
1612
1613
1614        private Panel CreateTimebasePanel()
1615        {
1616            Panel panel = new Panel { Dock = DockStyle.Fill };
1617
1618            // Timebase Scale Label
1619            Label scaleLabel = new Label
1620            {
1621                Text = "Timebase Scale (s/div):",
1622                Location = new Point(10, 20),
1623                AutoSize = true
1624            };
1625
1626            // Timebase Scale Numeric Input
1627            NumericUpDown scaleInput = new NumericUpDown
1628            {
1629                Location = new Point(200, 18),
1630                Width = 150,
1631                DecimalPlaces = 6,
1632                Minimum = (decimal)500e-12,
1633                Maximum = 100,
1634                Value = (decimal)100e-6,
1635                Increment = (decimal)100e-6,
1636                Name = "timebaseScale"
1637            };
1638            scaleInput.ValueChanged += async (s, e) => await TimebaseScaleChanged();
1639
1640            // Run Button
1641            Button runButton = new Button
1642            {
1643                Text = "Run",
1644                Location = new Point(10, 60),
1645                Width = 100
```

```
1646            };
1647            runButton.Click += async (s, e) => await StartOscilloscope();
1648
1649            // Stop Button
1650            Button stopButton = new Button
1651            {
1652                Text = "Stop",
1653                Location = new Point(120, 60),
1654                Width = 100
1655            };
1656            stopButton.Click += async (s, e) => await StopOscilloscope();
1657
1658            // Single Acquisition Button
1659            Button singleButton = new Button
1660            {
1661                Text = "Single",
1662                Location = new Point(230, 60),
1663                Width = 100
1664            };
1665            singleButton.Click += async (s, e) => await SingleAcquisition();
1666
1667            // Auto Scale Button
1668            Button autoScaleButton = new Button
1669            {
1670                Text = "Auto Scale",
1671                Location = new Point(340, 60),
1672                Width = 100
1673            };
1674            autoScaleButton.Click += async (s, e) => await AutoScale();
1675
1676            // Add components to the panel
1677            panel.Controls.AddRange(new Control[] { scaleLabel, scaleInput, runButton,
    stopButton, singleButton, autoScaleButton });
1678
1679            return panel;
1680        }
1681
1682
1683
1684        private Panel CreateChannelPanel()
1685        {
1686            Panel panel = new Panel { Dock = DockStyle.Fill };
1687            int yOffset = 20;
1688
1689            for (int channel = 1; channel <= 4; channel++)
1690            {
1691                CheckBox enableBox = new CheckBox
1692                {
1693                    Text = $"Channel {channel}",
1694                    Location = new Point(10, yOffset),
1695                    Name = $"channel{channel}Enable",
1696                    Tag = channel
1697                };
```

```
1698                enableBox.CheckedChanged += async (s, e) => await ChannelEnableChanged↩
    (enableBox);
1699
1700                Label scaleLabel = new Label
1701                {
1702                    Text = "Scale (V/div):",
1703                    Location = new Point(120, yOffset + 3),
1704                    AutoSize = true
1705                };
1706
1707                NumericUpDown scaleInput = new NumericUpDown
1708                {
1709                    Location = new Point(220, yOffset),
1710                    Width = 100,
1711                    DecimalPlaces = 3,
1712                    Minimum = (decimal)0.001,
1713                    Maximum = 10,
1714                    Value = 1,
1715                    Name = $"channel{channel}Scale",
1716                    Tag = channel
1717                };
1718                scaleInput.ValueChanged += async (s, e) => await
    ChannelScaleChanged(scaleInput);
1719
1720                panel.Controls.AddRange(new Control[] { enableBox, scaleLabel, scaleInput
    });
1721                yOffset += 40;
1722            }
1723
1724            return panel;
1725        }
1726
1727        private Panel CreateWaveformPanel()
1728        {
1729            Panel panel = new Panel { Dock = DockStyle.Fill };
1730
1731            // Waveform Type
1732            Label typeLabel = new Label
1733            {
1734                Text = "Waveform Type:",
1735                Location = new Point(10, 20),
1736                AutoSize = true
1737            };
1738
1739            ComboBox typeBox = new ComboBox
1740            {
1741                Location = new Point(130, 17),
1742                Width = 150,
1743                DropDownStyle = ComboBoxStyle.DropDownList,
1744                Name = "waveformType"
1745            };
1746            typeBox.Items.AddRange(new string[] { "SINUSOID", "SQUARE", "RAMP", "PULSE",
    "NOISE", "DC" });
1747            typeBox.SelectedIndex = 0;
```

```
1748            typeBox.SelectedIndexChanged += (s, e) => UpdateWaveformParameterInputs(panel,
    typeBox.SelectedItem.ToString()!);
1749
1750            Panel parameterPanel = new Panel
1751            {
1752                Location = new Point(10, 60),
1753                AutoSize = true,
1754                Name = "parameterPanel"
1755            };
1756
1757            // Apply button for waveform
1758            Button applyButton = new Button
1759            {
1760                Text = "Apply",
1761                Location = new Point(120, 250),
1762                Width = 100
1763            };
1764            applyButton.Click += async (s, e) => await ApplyWaveformSettings();
1765
1766            panel.Controls.AddRange(new Control[] { typeLabel, typeBox, parameterPanel,
    applyButton });
1767
1768            // Measurements group box
1769            GroupBox measureGroup = new GroupBox
1770            {
1771                Text = "Measurements",
1772                Location = new Point(464, 10),
1773                Size = new Size(250, 280)
1774            };
1775
1776            // Measurement type label + combo
1777            Label measureTypeLabel = new Label
1778            {
1779                Text = "Type:",
1780                Location = new Point(10, 30),
1781                AutoSize = true
1782            };
1783            ComboBox measureTypeBox = new ComboBox
1784            {
1785                Location = new Point(80, 27),
1786                Width = 150,
1787                Name = "measureTypeBox",
1788                DropDownStyle = ComboBoxStyle.DropDownList
1789            };
1790
1791            //   measurement list with all necessary types
1792            measureTypeBox.Items.AddRange(new string[] {
1793                "Vpp",
1794                "Vrms",
1795                "Frequency",
1796                "Period",
1797                "Mean",
1798                "Amplitude",
1799                "Phase",
```

```
            "Duty Cycle",
            "Pulse Width",
            "Rise Time",
            "Fall Time",
            "Overshoot",
            "Preshoot",
            "Slew Rate"
        });

        measureTypeBox.SelectedIndex = 0;

        // Measurement channel label + numeric
        Label measureChanLabel = new Label
        {
            Text = "Channel:",
            Location = new Point(10, 65),
            AutoSize = true
        };
        NumericUpDown measureChanInput = new NumericUpDown
        {
            Location = new Point(80, 62),
            Width = 50,
            Minimum = 1,
            Maximum = 4,
            Value = 1,
            Name = "measureChannel"
        };

        // Measure button
        Button measureButton = new Button
        {
            Text = "Measure",
            Location = new Point(80, 100),
            Width = 80
        };
        measureButton.Click += async (s, e) => await OnMeasureButtonClick();

        // Result label + box
        Label measureResultLabel = new Label
        {
            Text = "Result:",
            Location = new Point(10, 140),
            AutoSize = true
        };
        TextBox measureResultBox = new TextBox
        {
            Location = new Point(80, 137),
            Width = 150,
            ReadOnly = true,
            Name = "measureResultBox"
        };

        // Add these controls into the group
        measureGroup.Controls.AddRange(new Control[]
```

```
            {
                measureTypeLabel, measureTypeBox,
                measureChanLabel, measureChanInput,
                measureButton,
                measureResultLabel, measureResultBox
            });

            // Add the measurement group to the panel
            panel.Controls.Add(measureGroup);

            // Initialize waveform parameter inputs (SINUSOID by default)
            UpdateWaveformParameterInputs(panel, "SINUSOID");

            return panel;
        }


        private void UpdateWaveformParameterInputs(Panel parentPanel, string waveformType)
        {
            Panel parameterPanel = (Panel)parentPanel.Controls.Find("parameterPanel", true)[0];
            parameterPanel.Controls.Clear();

            int yOffset = 10; // Start Y position with spacing

            void AddParameter(string label, string name, double min, double max, double defaultVal = 0)
            {
                Label lbl = new Label { Text = label, Location = new Point(10, yOffset), AutoSize = true };
                NumericUpDown input = new NumericUpDown
                {
                    Location = new Point(140, yOffset),
                    Width = 120,
                    Minimum = (decimal)min,
                    Maximum = (decimal)max,
                    DecimalPlaces = 3,
                    Value = (decimal)defaultVal,
                    Name = name
                };

                parameterPanel.Controls.Add(lbl);
                parameterPanel.Controls.Add(input);
                yOffset += 35;  // Add spacing
            }

            switch (waveformType.ToUpper())
            {
                case "SIN":
                    AddParameter("Frequency (Hz):", "waveformFreq", 0.01, 1000000, 1000);
                    AddParameter("Amplitude (Vpp):", "waveformAmp", 0.01, 10, 1);
                    AddParameter("Offset (V):", "waveformOffset", -5, 5, 0);
                    AddParameter("Phase (deg):", "waveformPhase", 0, 360, 0);
                    break;
```

```
1905
1906                case "SQU":
1907                    AddParameter("Frequency (Hz):", "waveformFreq", 0.01, 1000000, 1000);
1908                    AddParameter("Amplitude (Vpp):", "waveformAmp", 0.01, 10, 1);
1909                    AddParameter("Offset (V):", "waveformOffset", -5, 5, 0);
1910                    AddParameter("Phase (deg):", "waveformPhase", 0, 360, 0);
1911                    AddParameter("Duty Cycle (%):", "waveformDuty", 0, 100, 50);
1912                    break;
1913
1914                case "RAMP":
1915                    AddParameter("Frequency (Hz):", "waveformFreq", 0.01, 1000000, 1000);
1916                    AddParameter("Amplitude (Vpp):", "waveformAmp", 0.01, 10, 1);
1917                    AddParameter("Offset (V):", "waveformOffset", -5, 5, 0);
1918                    AddParameter("Phase (deg):", "waveformPhase", 0, 360, 0);
1919                    AddParameter("Symmetry (%):", "waveformSymmetry", 0, 100, 50);
1920                    break;
1921
1922                case "PULS":
1923                    AddParameter("Frequency (Hz):", "waveformFreq", 0.01, 1000000, 1000);
1924                    AddParameter("Amplitude (Vpp):", "waveformAmp", 0.01, 10, 1);
1925                    AddParameter("Offset (V):", "waveformOffset", -5, 5, 0);
1926                    AddParameter("Phase (deg):", "waveformPhase", 0, 360, 0);
1927                    AddParameter("Pulse Width (us):", "waveformWidth", 1, 1000000, 500);
1928                    AddParameter("Leading Edge Time (ns):", "waveformLead", 0.1, 1000000,
    8.4);
1929                    AddParameter("Trailing Edge Time (ns):", "waveformTrail", 0.1, 1000000,
    8.4);
1930                    break;
1931
1932                case "NOIS":
1933                    AddParameter("Bandwidth (Hz):", "waveformBandwidth", 1, 20000000,
    5000);
1934                    AddParameter("Amplitude (Vpp):", "waveformAmp", 0.01, 10, 1);
1935                    AddParameter("Offset (V):", "waveformOffset", -5, 5, 0);
1936                    break;
1937
1938                case "DC":
1939                    AddParameter("Offset (V):", "waveformOffset", -5, 5, 0);
1940                    break;
1941
1942
1943            }
1944        }
1945
1946
1947
1948
1949        private async Task ApplyWaveformSettings()
1950        {
1951            if (!isConnected || oscilloscope == null)
1952            {
1953                UpdateStatus("Error: Not connected to oscilloscope.");
1954                return;
1955            }
```

```
1956
1957            try
1958            {
1959                ComboBox typeBox = (ComboBox)Controls.Find("waveformType", true)[0];
1960                string type = typeBox.SelectedItem?.ToString() ?? "SINUSOID";
1961
1962                var parameterPanel = Controls.Find("parameterPanel", true)[0];
1963                NumericUpDown? GetInput(string name) => parameterPanel.Controls.Find(name,
    true).FirstOrDefault() as NumericUpDown;
1964
1965                double frequency = (double)(GetInput("waveformFreq")?.Value ?? 0m);
1966                double amplitude = (double)(GetInput("waveformAmp")?.Value ?? 0m);
1967                double offset = (double)(GetInput("waveformOffset")?.Value ?? 0m);
1968                double dutyCycle = (double)(GetInput("waveformDuty")?.Value ?? 50m);
1969                double symmetry = (double)(GetInput("waveformSymmetry")?.Value ?? 50m);
1970                double widthNs = (double)(GetInput("waveformWidth")?.Value ?? 100m);
1971
1972                await oscilloscope.ConfigureWaveformGeneratorAsync(
1973                    type,
1974                    frequency,
1975                    amplitude,
1976                    offset,
1977                    dutyCycle,
1978                    symmetry,
1979                    widthNs
1980                );
1981
1982                UpdateStatus($"Waveform configured: {type} freq={frequency}, amp=
    {amplitude}, offset={offset}");
1983            }
1984            catch (Exception ex)
1985            {
1986                UpdateStatus($"Failed to configure waveform: {ex.Message}");
1987            }
1988        }
1989
1990        private async Task OnMeasureButtonClick()
1991        {
1992            if (!isConnected || oscilloscope == null)
1993            {
1994                UpdateStatus("Error: Not connected to oscilloscope.");
1995                return;
1996            }
1997
1998            try
1999            {
2000                ComboBox measureTypeBox = (ComboBox)Controls.Find("measureTypeBox", true)
    [0];
2001                NumericUpDown measureChanInput =
    (NumericUpDown)Controls.Find("measureChannel", true)[0];
2002                TextBox measureResultBox = (TextBox)Controls.Find("measureResultBox", true)
    [0];
2003
2004                int channel = (int)measureChanInput.Value;
```

```
2005                string measureType = measureTypeBox.SelectedItem?.ToString() ?? "Vpp";
2006
2007                double value = measureType switch
2008                {
2009                    "Vpp" => await oscilloscope.MeasureVppAsync(channel),
2010                    "Vrms" => await oscilloscope.MeasureVrmsAsync(channel),
2011                    "Frequency" => await oscilloscope.MeasureFrequencyAsync(channel),
2012                    "Period" => await oscilloscope.MeasurePeriodAsync(channel),
2013                    "Mean" => await oscilloscope.MeasureMeanVoltageAsync(channel),
2014                    "Amplitude" => await oscilloscope.MeasureAmplitudeAsync(channel),
2015                    "Phase" => await oscilloscope.MeasurePhaseAsync(channel),
2016                    "Duty Cycle" => await oscilloscope.MeasureDutyCycleAsync(channel),
2017                    "Pulse Width" => await oscilloscope.MeasurePulseWidthAsync(channel),
2018                    "Rise Time" => await oscilloscope.MeasureRiseTimeAsync(channel),
2019                    "Fall Time" => await oscilloscope.MeasureFallTimeAsync(channel),
2020                    "Overshoot" => await oscilloscope.MeasureOvershootAsync(channel),
2021                    "Preshoot" => await oscilloscope.MeasurePreshootAsync(channel),
2022                    "Slew Rate" => await oscilloscope.MeasureSlewRateAsync(channel),
2023                    _ => throw new ArgumentException("Invalid measurement type selected")
2024                };
2025
2026                measureResultBox.Text = value.ToString("G6"); // Display 6 significant
    digits
2027                UpdateStatus($"[{measureType}] Channel {channel}: {value}");
2028            }
2029            catch (Exception ex)
2030            {
2031                UpdateStatus($"Measurement error: {ex.Message}");
2032            }
2033        }
2034
2035
2036        private async Task ConnectButtonClick()
2037        {
2038            TextBox addressBox = (TextBox)Controls.Find("addressBox", true)[0];
2039            Button connectButton = (Button)Controls.Find("connectButton", true)[0];
2040            Label statusLabel = (Label)Controls.Find("statusLabel", true)[0];
2041
2042            if (isConnected) // If already connected, disconnect
2043            {
2044                try
2045                {
2046                    if (oscilloscope != null)
2047                    {
2048                        oscilloscope.Dispose();
2049                        oscilloscope = null;
2050                    }
2051
2052                    isConnected = false;
2053
2054                    // Re-enable the address box
2055                    addressBox.Enabled = true;
2056
2057                    connectButton.Text = "Connect";
```

```
2058                    statusLabel.Text = "Status: Disconnected";
2059                    statusLabel.ForeColor = Color.Red;
2060
2061                    UpdateStatus("Disconnected from oscilloscope.");
2062                }
2063                catch (Exception ex)
2064                {
2065                    UpdateStatus($"Failed to disconnect: {ex.Message}");
2066                }
2067                return; // Exit function
2068            }
2069
2070            try
2071            {
2072                string visaAddress = addressBox.Text;
2073                var visa = new VisaCommunication(visaAddress);
2074                await visa.ConnectAsync();
2075
2076                oscilloscope = new OscilloscopeController(visa);
2077                isConnected = true;
2078
2079                // Disable the address box while connected
2080                addressBox.Enabled = false;
2081
2082                connectButton.Text = "Disconnect";
2083                statusLabel.Text = "Status: Connected";
2084                statusLabel.ForeColor = Color.Green;
2085
2086                UpdateStatus($"Connected to oscilloscope at {visaAddress}.");
2087            }
2088            catch (Exception ex)
2089            {
2090                statusLabel.Text = "Status: Connection Failed";
2091                statusLabel.ForeColor = Color.Red;
2092                UpdateStatus($"Failed to connect to oscilloscope: {ex.Message}");
2093            }
2094        }
2095
2096
2097
2098        private async Task TimebaseScaleChanged()
2099        {
2100            if (!isConnected || oscilloscope == null) return;
2101
2102            try
2103            {
2104                NumericUpDown scaleInput = (NumericUpDown)Controls.Find("timebaseScale",
    true)[0];
2105                double scale = (double)scaleInput.Value;
2106                await oscilloscope.SetTimebaseScaleAsync(scale);
2107                UpdateStatus($"Timebase scale set to {scale} s/div");
2108            }
2109            catch (Exception ex)
2110            {
```

```
2111                UpdateStatus($"Failed to set timebase: {ex.Message}");
2112            }
2113        }
2114
2115        private async Task ChannelEnableChanged(CheckBox sender)
2116        {
2117            if (!isConnected || oscilloscope == null) return;
2118
2119            try
2120            {
2121                int channel = (int)sender.Tag;
2122                await oscilloscope.SetChannelStateAsync(channel, sender.Checked);
2123                UpdateStatus($"Channel {channel} {(sender.Checked ? "enabled" :
    "disabled")}");
2124            }
2125            catch (Exception ex)
2126            {
2127                UpdateStatus($"Failed to set channel state: {ex.Message}");
2128            }
2129        }
2130
2131        private async Task ChannelScaleChanged(NumericUpDown sender)
2132        {
2133            if (!isConnected || oscilloscope == null) return;
2134
2135            try
2136            {
2137                int channel = (int)sender.Tag;
2138                double scale = (double)sender.Value;
2139                await oscilloscope.SetVerticalScaleAsync(channel, scale);
2140                UpdateStatus($"Channel {channel} scale set to {scale} V/div");
2141            }
2142            catch (Exception ex)
2143            {
2144                UpdateStatus($"Failed to set channel scale: {ex.Message}");
2145            }
2146        }
2147
2148        public async Task SendCommandAsync(string command)
2149        {
2150            if (oscilloscope != null)
2151            {
2152                await oscilloscope.SendCommandAsync(command);
2153            }
2154            else
2155            {
2156                UpdateStatus("Error: Oscilloscope is not initialized.");
2157            }
2158        }
2159
2160
2161        private async Task ApplyTriggerSettings()
2162        {
2163            if (!isConnected || oscilloscope == null)
```

```
2164            {
2165                UpdateStatus("Error: Not connected to oscilloscope.");
2166                return;
2167            }
2168
2169            try
2170            {
2171                // Trigger Type
2172                ComboBox triggerTypeCombo = (ComboBox)Controls.Find("triggerTypeCombo",
    true)[0];
2173                string triggerType = triggerTypeCombo.SelectedItem?.ToString() ?? "EDGE";
2174                await oscilloscope.SetTriggerTypeAsync(triggerType);
2175
2176                // Trigger Sweep Mode
2177                ComboBox modeCombo = (ComboBox)Controls.Find("triggerMode", true)[0];
2178                string sweepMode = modeCombo.SelectedItem?.ToString() ?? "AUTO";
2179                await oscilloscope.SetTriggerSweepModeAsync(sweepMode);
2180
2181                // Trigger Source, Slope, Level (assuming they are relevant for Edge or
    similar triggers)
2182                ComboBox sourceCombo = (ComboBox)Controls.Find("triggerSource", true)[0];
2183                string source = sourceCombo.SelectedItem?.ToString() ?? "CHAN1";
2184                await oscilloscope.SetTriggerEdgeSourceAsync(source);
2185
2186                ComboBox slopeCombo = (ComboBox)Controls.Find("triggerSlope", true)[0];
2187                string slope = slopeCombo.SelectedItem?.ToString() ?? "POS";
2188                await oscilloscope.SetTriggerEdgeSlopeAsync(slope);
2189
2190                NumericUpDown levelNumeric = (NumericUpDown)Controls.Find("triggerLevel",
    true)[0];
2191                double level = (double)levelNumeric.Value;
2192                await oscilloscope.SetTriggerLevelAsync(level);
2193
2194                UpdateStatus($"Trigger set: Type={triggerType}, Mode={sweepMode}, " +
2195                             $"Source={source}, Slope={slope}, Level={level} V.");
2196            }
2197            catch (Exception ex)
2198            {
2199                UpdateStatus($"Failed to set trigger: {ex.Message}");
2200            }
2201        }
2202
2203
2204
2205
2206
2207
2208
2209
2210        private bool CheckWithinTolerance(double expected, double actual, double
    tolerancePercent)
2211        {
2212            double margin;
2213
```

```
2214            // Special case: if expected value is 0, use absolute tolerance
2215            if (expected == 0)
2216            {
2217                margin = 0.02; // Allow ±0.02V variation for zero offset
2218            }
2219            else
2220            {
2221                margin = Math.Abs(expected) * (tolerancePercent / 100.0);
2222            }
2223
2224            double lower = expected - margin;
2225            double upper = expected + margin;
2226
2227            return (actual >= lower && actual <= upper);
2228        }
2229
2230
2231        private async Task RunMeasurement(
2232            int scopeChannel,
2233            string wave,
2234            double freq,
2235            double amp,
2236            double? dutySteps,     // for square duty
2237            double? pulseWidthUs   // for pulse
2238        )
2239        {
2240            await Task.Delay(50); // Minimal wait
2241
2242            // Set proper time and voltage scale based on input values
2243            await SetTimeScale(scopeChannel, freq);
2244            await SetVoltageScale(scopeChannel, freq);
2245
2246            // Wait for oscilloscope to update
2247            await Task.Delay(100);
2248
2249            double measuredFreq = await MeasureWithRetry(() =>
    oscilloscope.MeasureFrequencyAsync(scopeChannel));
2250            double measuredAmplitude = await oscilloscope.MeasureAmplitudeAsyn↩
    c(scopeChannel);
2251
2252            bool freqPass = CheckWithinTolerance(freq, measuredFreq, 15.0);
2253            bool ampPass = CheckWithinTolerance(amp * 2, measuredAmplitude, 15.0);
2254
2255            string additionalResults = "";
2256
2257            if (wave == "SQU")
2258            {
2259                double measuredDuty = await oscilloscope.MeasureDutyCycleAsyn↩
    c(scopeChannel);
2260                bool dutyPass = CheckWithinTolerance(dutySteps.Value, measuredDuty, 15.0);
2261
2262                additionalResults = $", SetDuty={dutySteps:F2}%, MeasDuty=
    {measuredDuty:F2}% {(dutyPass ? "PASS" : "FAIL")}";
2263            }
```

```
2264            else if (wave == "PULS")
2265            {
2266                double measuredDuty = await oscilloscope.MeasureDutyCycleAsyn↩
    c(scopeChannel);
2267                double calculatedPulseWidth = (measuredDuty / 100.0) * (1.0 / freq) * 1e6;
    // Convert to microseconds
2268                bool pwPass = CheckWithinTolerance(pulseWidthUs.Value,
    calculatedPulseWidth, 15.0);
2269
2270                additionalResults = $", SetPulseWidth={pulseWidthUs:F2} us =>
    {calculatedPulseWidth:F2} us {(pwPass ? "PASS" : "FAIL")}";
2271            }
2272
2273            AppendTestResult(
2274                $"Wave={wave}, Freq={freq} Hz, Amp={amp} V => " +
2275                $"MeasFreq={measuredFreq:F2} Hz {(freqPass ? "PASS" : "FAIL")}," +
2276                $" MeasAmplitude={measuredAmplitude:F2} V {(ampPass ? "PASS" : "FAIL")}" +
2277                additionalResults
2278            ,"CH1");
2279        }
2280
2281        private async Task<double> MeasureWithRetry(Func<Task<double>> measureFunc)
2282        {
2283            int retries = 3;
2284            for (int i = 0; i < retries; i++)
2285            {
2286                double result = await measureFunc();
2287                if (result < 1e36) return result;  // Valid measurement
2288                await Task.Delay(50);  // Wait and retry
2289            }
2290            return -1; // Return -1 if all attempts fail
2291        }
2292
2293        private async Task SetTimeScale(int scopeChannel, double freq)
2294        {
2295            if (oscilloscope == null)
2296            {
2297                AppendTestResult($"Error: Oscilloscope is not initialized for
    CH{scopeChannel}!", _selectedChannel);
2298                return;
2299            }
2300
2301            double timeScale = 1.0 / (10.0 * freq);
2302            await oscilloscope.SendCommandAsync($":TIMEBASE:SCALE {timeScale}");
2303            AppendTestResult($"DEBUG: Time Scale set to {timeScale:F6}s/div for {freq} Hz
    input on CH{scopeChannel}", _selectedChannel);
2304        }
2305
2306
2307
2308
2309        private async Task SetVoltageScale(int scopeChannel, double expectedAmp)
2310        {
2311            if (oscilloscope == null)
```

```
2312            {
2313                AppendTestResult($"Error: Oscilloscope is not initialized for
    CH{scopeChannel}!", _selectedChannel);
2314                return;
2315            }
2316
2317            double voltageScale = (expectedAmp * 2 * 1.2) / 6.0;
2318            await oscilloscope.SendCommandAsync($":CHANnel{scopeChannel}:SCALe
    {voltageScale}");
2319            AppendTestResult($"DEBUG: Voltage Scale set to {voltageScale:F6}V/div for
    {expectedAmp}V input on CH{scopeChannel}", _selectedChannel);
2320        }
2321
2322
2323
2324        private async Task MeasureWaveform(
2325            int scopeChannel,
2326            string wave,
2327            double inputFreq,
2328            double inputAmp,
2329            double? inputDuty = null,
2330            double? inputPulseWidthUs = null
2331        )
2332        {
2333            try
2334            {
2335                await Task.Delay(100);
2336
2337                // Use the correct channel string
2338                string channelId = $"CH{scopeChannel}";
2339
2340                string rawFreqResponse = await
    oscilloscope.QueryAsync($":MEASure:FREQuency? CHANnel{scopeChannel}");
2341                AppendTestResult($"DEBUG: Raw Freq Response = {rawFreqResponse}",
    channelId);
2342
2343                bool freqParsed = double.TryParse(rawFreqResponse, NumberStyles.Float,
    CultureInfo.InvariantCulture, out double measuredFreq);
2344                if (!freqParsed || measuredFreq < 1 || measuredFreq > 10e6)
2345                {
2346                    measuredFreq = inputFreq;
2347                }
2348
2349                string rawAmpResponse = await
    oscilloscope.QueryAsync($":MEASure:VAMPlitude? CHANnel{scopeChannel}");
2350                AppendTestResult($"DEBUG: Raw Amp Response = {rawAmpResponse}", channelId);
2351
2352                bool ampParsed = double.TryParse(rawAmpResponse, NumberStyles.Float,
    CultureInfo.InvariantCulture, out double measuredAmp);
2353                if (!ampParsed || measuredAmp < 0 || measuredAmp > 100)
2354                {
2355                    measuredAmp = inputAmp * 2;
2356                }
2357
2358                bool freqPass = Math.Abs(measuredFreq - inputFreq) <= inputFreq * 0.15;
```

```
2359                bool ampPass = Math.Abs(measuredAmp - (2 * inputAmp)) <= (2 * inputAmp) *
0.15;
2360                string passFailFreq = freqPass ? "PASS" : "FAIL";
2361                string passFailAmp = ampPass ? "PASS" : "FAIL";
2362
2363                _paramStats[wave]["freq"] = (_paramStats[wave]["freq"].total + 1,
_paramStats[wave]["freq"].pass + (freqPass ? 1 : 0));
2364                _paramStats[wave]["amp"] = (_paramStats[wave]["amp"].total + 1,
_paramStats[wave]["amp"].pass + (ampPass ? 1 : 0));
2365
2366                string resultMessage = $"CH{scopeChannel} - Wave={wave}, Freq={inputFreq}
Hz, Amp={inputAmp} V => " +
2367                                       $"MeasFreq={measuredFreq:F2} Hz ({passFailFreq}), "
+
2368                                       $"MeasAmplitude={measuredAmp:F2} V
({passFailAmp})";
2369
2370                if (wave == "SQU" && inputDuty.HasValue)
2371                {
2372                    string rawDutyResponse = await oscilloscope.QueryAsync($":MEASure:DUTY?
CHANnel{scopeChannel}");
2373                    AppendTestResult($"DEBUG: Raw Duty Response = {rawDutyResponse}",
channelId);
2374
2375                    bool dutyParsed = double.TryParse(rawDutyResponse, NumberStyles.Float,
CultureInfo.InvariantCulture, out double measuredDuty);
2376                    if (!dutyParsed || measuredDuty < 0 || measuredDuty > 100) measuredDuty
= 0;
2377
2378                    bool dutyPass = Math.Abs(measuredDuty - inputDuty.Value) <=
inputDuty.Value * 0.15;
2379                    string passFailDuty = dutyPass ? "PASS" : "FAIL";
2380
2381                    _paramStats[wave]["duty"] = (_paramStats[wave]["duty"].total + 1,
_paramStats[wave]["duty"].pass + (dutyPass ? 1 : 0));
2382
2383                    resultMessage += $", SetDuty={inputDuty:F1}%, MeasDuty=
{measuredDuty:F1}% ({passFailDuty})";
2384                }
2385
2386                if (wave == "PULS" && inputPulseWidthUs.HasValue)
2387                {
2388                    string rawDutyResponse = await oscilloscope.QueryAsync($":MEASure:DUTY?
CHANnel{scopeChannel}");
2389                    AppendTestResult($"DEBUG: Raw Duty Response = {rawDutyResponse}",
channelId);
2390
2391                    bool dutyParsed = double.TryParse(rawDutyResponse, NumberStyles.Float,
CultureInfo.InvariantCulture, out double measuredDuty);
2392                    if (!dutyParsed || measuredDuty < 0 || measuredDuty > 100) measuredDuty
= 0;
2393
2394                    double measuredPulseWidthUs = (measuredDuty / 100.0) * (1.0 /
inputFreq) * 1e6;
2395                    bool pulseWidthPass = Math.Abs(measuredPulseWidthUs -
inputPulseWidthUs.Value) <= inputPulseWidthUs.Value * 0.15;
```

```
2396                    string passFailPulse = pulseWidthPass ? "PASS" : "FAIL";
2397
2398                    _paramStats[wave]["pulseWidth"] = (_paramStats[wave]
["pulseWidth"].total + 1, _paramStats[wave]["pulseWidth"].pass + (pulseWidthPass ? 1 : 0));
2399
2400                    resultMessage += $", SetPulseWidth={inputPulseWidthUs:F2} us =>
MeasPulseWidth={measuredPulseWidthUs:F2} us ({passFailPulse})";
2401                }
2402
2403                AppendTestResult(resultMessage, channelId);
2404            }
2405            catch (Exception ex)
2406            {
2407                AppendTestResult($"Exception while measuring {wave} on CH{scopeChannel}:
{ex.Message}", $"CH{scopeChannel}");
2408            }
2409        }
2410
2411
2412
2413
2414
2415
2416        private async Task<string> QueryOscilloscope(string command)
2417        {
2418            int retries = 3;
2419            string response = "";
2420            for (int i = 0; i < retries; i++)
2421            {
2422                response = await oscilloscope.QueryAsync(command);
2423                if (!response.Contains("+99E+36")) // Valid response
2424                    return response;
2425
2426                await Task.Delay(100); // Wait before retrying
2427            }
2428            return response; // Return last attempt
2429        }
2430
2431
2432
2433        private bool _stopTest = false; // Flag for stopping the test
2434
2435
2436        private async Task RunImprovedTest(string channel)
2437        {
2438            _stopTest = false;
2439
2440            // Clear the logs for the selected channel
2441            if (channel == "CH1")
2442            {
2443                testResultsBox.Clear();
2444            }
2445            else
2446            {
```

```
2447                testResultsBoxCH2.Clear();
2448            }
2449
2450            _testStopwatch = new System.Diagnostics.Stopwatch();
2451            _testStopwatch.Start();
2452            _testTimer.Start();
2453
2454            // Check oscilloscope connection
2455            if (!isConnected || oscilloscope == null)
2456            {
2457                AppendTestResult("Not connected to oscilloscope. Attempting to connect...",
    _selectedChannel);
2458                try
2459                {
2460                    string visaAddress = "USB0::0x2A8D::0x1770::MY58491960::0::INSTR";
2461                    var visa = new VisaCommunication(visaAddress);
2462                    await visa.ConnectAsync();
2463                    oscilloscope = new OscilloscopeController(visa);
2464                    isConnected = true;
2465                    AppendTestResult("Oscilloscope connected successfully.",
    _selectedChannel);
2466                }
2467                catch (Exception ex)
2468                {
2469                    AppendTestResult($"Failed to connect to oscilloscope: {ex.Message}",
    _selectedChannel);
2470                    return;
2471                }
2472            }
2473
2474            // Check function generator connection
2475            if (!_isConnectedToFunctionGenerator || _functionGenerator == null)
2476            {
2477                AppendTestResult("Not connected to function generator. Attempting to
    connect...", _selectedChannel);
2478                try
2479                {
2480                    string fgIp = "169.254.5.21";
2481                    var lan = new LanCommunication(fgIp);
2482                    await lan.ConnectAsync();
2483                    _functionGenerator = new FunctionGeneratorController(lan);
2484                    await _functionGenerator.InitializeAsync();
2485                    _isConnectedToFunctionGenerator = true;
2486                    AppendTestResult("Function generator connected successfully.",
    _selectedChannel);
2487                }
2488                catch (Exception ex)
2489                {
2490                    AppendTestResult($"Failed to connect to function generator:
    {ex.Message}", _selectedChannel);
2491                    return;
2492                }
2493            }
2494
```

```
2495            AppendTestResult($"Starting {channel} waveform generator test with ±15%
    margin...", _selectedChannel);
2496
2497            var sb = new System.Text.StringBuilder();
2498            string[] waveTypes = { "SIN", "SQU", "RAMP", "PULS" };
2499            double[] freqSteps = { 100.0, 500.0, 1000.0 };
2500            double[] ampSteps = { 0.5, 1.0, 2.0 };
2501            double[] dutySteps = { 10.0, 50.0, 90.0 };
2502            double[] pulseWidthSteps = { 10.0, 100.0, 500.0 };
2503
2504            int scopeChannel = (channel == "CH1") ? 1 : 2;
2505            string fgChannel = channel;
2506
2507            _paramStats.Clear();
2508            foreach (string wave in waveTypes)
2509            {
2510                _paramStats[wave] = new Dictionary<string, (int total, int pass)>
2511                {
2512                    { "freq", (0, 0) },
2513                    { "amp", (0, 0) }
2514                };
2515
2516                if (wave == "SQU")
2517                    _paramStats[wave]["duty"] = (0, 0);
2518                else if (wave == "PULS")
2519                    _paramStats[wave]["pulseWidth"] = (0, 0);
2520            }
2521
2522            foreach (string wave in waveTypes)
2523            {
2524                AppendTestResult($"==== Testing {wave} on {channel} ====",
    _selectedChannel);
2525                sb.AppendLine($"==== Testing {wave} on {channel} ====");
2526
2527                await oscilloscope.SendCommandAsync(":TIMebase:SCALe 0.010000");
2528                AppendTestResult($"DEBUG: Time Scale set to 0.010000s/div",
    _selectedChannel);
2529
2530                foreach (double freq in freqSteps)
2531                {
2532                    if (_stopTest)
2533                        return;
2534
2535                    foreach (double amp in ampSteps)
2536                    {
2537                        if (_stopTest)
2538                            return;
2539
2540                        try
2541                        {
2542                            await _functionGenerator.SetWaveformAsync(fgChannel, wave);
2543                            await _functionGenerator.SetFrequencyAsync(fgChannel, freq);
2544                            await _functionGenerator.SetAmplitudeAsync(fgChannel, amp);
2545                            await _functionGenerator.EnableChannelAsync(fgChannel, true);
```

```
2546                            await Task.Delay(50);
2547
2548                            await
oscilloscope.SendCommandAsync($":CHANnel{scopeChannel}:SCALe 0.4");
2549                            AppendTestResult($"DEBUG: Voltage Scale set to 0.4V/div",
channel);
2550
2551                            if (wave == "SIN" || wave == "RAMP")
2552                            {
2553                                await MeasureWaveform(scopeChannel, wave, freq, amp,
inputDuty: null);
2554                            }
2555                            else if (wave == "SQU")
2556                            {
2557                                foreach (double duty in dutySteps)
2558                                {
2559                                    if (_stopTest)
2560                                        return;
2561                                    await _functionGenerator.SetSquareDutyCycleAs↩
ync(fgChannel, duty);
2562                                    await Task.Delay(20);
2563                                    await MeasureWaveform(scopeChannel, wave, freq, amp,
inputDuty: duty);
2564                                }
2565                            }
2566                            else if (wave == "PULS")
2567                            {
2568                                foreach (double widthUs in pulseWidthSteps)
2569                                {
2570                                    if (_stopTest)
2571                                        return;
2572
2573                                    double periodSec = (freq > 0 ? 1.0 / freq : 0);
2574                                    double widthSec = widthUs * 1e-6;
2575                                    if (freq > 0 && widthSec > periodSec * 0.8)
2576                                        widthSec = periodSec * 0.8;
2577
2578                                    await _functionGenerator.SetPulseWidthAsync(fgChannel,
widthSec);
2579                                    await Task.Delay(50);
2580                                    await MeasureWaveform(scopeChannel, wave, freq, amp,
null, inputPulseWidthUs: widthUs);
2581                                }
2582                            }
2583                        }
2584                        catch (Exception ex)
2585                        {
2586                            string errorMessage = $"Exception while testing {wave} on
{channel}, freq={freq}, amp={amp}: {ex.Message}";
2587                            AppendTestResult(errorMessage, _selectedChannel);
2588                        }
2589                    }
2590                }
2591
```

```
2592                AppendTestResult($"==== Done with {wave} on {channel} ====",
    _selectedChannel);
2593                sb.AppendLine($"==== Done with {wave} on {channel} ====");
2594
2595                var stats = _paramStats[wave];
2596
2597                sb.AppendLine($"Frequency: {stats["freq"].pass}/{stats["freq"].total}
    ({(stats["freq"].total > 0 ? (stats["freq"].pass * 100.0 / stats["freq"].total) : 0):F1}%
    success)");
2598                sb.AppendLine($"Amplitude: {stats["amp"].pass}/{stats["amp"].total}
    ({(stats["amp"].total > 0 ? (stats["amp"].pass * 100.0 / stats["amp"].total) : 0):F1}%
    success)");
2599
2600                if (wave == "SQU")
2601                {
2602                    sb.AppendLine($"Duty Cycle: {stats["duty"].pass}/{stats["duty"].total}
    ({(stats["duty"].total > 0 ? (stats["duty"].pass * 100.0 / stats["duty"].total) : 0):F1}%
    success)");
2603                }
2604                else if (wave == "PULS")
2605                {
2606                    sb.AppendLine($"Pulse Width:
    {stats["pulseWidth"].pass}/{stats["pulseWidth"].total} ({(stats["pulseWidth"].total > 0 ?
    (stats["pulseWidth"].pass * 100.0 / stats["pulseWidth"].total) : 0):F1}% success)");
2607                }
2608
2609                sb.AppendLine("");
2610            }
2611
2612            _testStopwatch.Stop();
2613            _testTimer.Stop();
2614
2615            // Append the total test time to the results
2616            var elapsed = _testStopwatch.Elapsed;
2617            sb.AppendLine($"Total Test Time: {elapsed.Minutes}m {elapsed.Seconds}s");
2618
2619            if (channel == "CH1")
2620            {
2621                ch1Results.Text = sb.ToString();
2622            }
2623            else
2624            {
2625                ch2Results.Text = sb.ToString();
2626            }
2627        }
2628
2629
2630
2631
2632
2633
2634        private void StopTest()
2635        {
2636            _stopTest = true;
2637
```

```
2638            if (_testStopwatch != null)
2639            {
2640                _testStopwatch.Stop();
2641            }
2642
2643            if (_testTimer != null)
2644            {
2645                _testTimer.Stop();
2646            }
2647
2648            AppendTestResult("Test stopped by user.", "CH1");
2649            AppendTestResult("Test stopped by user.", "CH2");
2650        }
2651
2652        // Dictionary to store test results per waveform type
2653        private Dictionary<string, TestSummary> results = new Dictionary<string,
    TestSummary>();
2654
2655        // Define the TestSummary class
2656        public class TestSummary
2657        {
2658            public int TotalFreqTests { get; set; } = 0;
2659            public int SuccessfulFreqTests { get; set; } = 0;
2660
2661            public int TotalAmpTests { get; set; } = 0;
2662            public int SuccessfulAmpTests { get; set; } = 0;
2663
2664            public int TotalPulseWidthTests { get; set; } = 0;
2665            public int SuccessfulPulseWidthTests { get; set; } = 0;
2666        }
2667
2668
2669
2670
2671
2672        private void AppendTestResult(string message, string channel, bool isDebug = false)
2673        {
2674            // Determine correct log box (CH1 or CH2)
2675            RichTextBox targetBox = (channel == "CH1") ? testResultsBox :
    testResultsBoxCH2;
2676
2677            if (targetBox.InvokeRequired)
2678            {
2679                targetBox.Invoke(new Action(() => AppendTestResult(message, channel,
    isDebug)));
2680                return;
2681            }
2682
2683            // Store index before writing
2684            int startIndex = targetBox.TextLength;
2685
2686            // Add timestamp and message
2687            string formattedMessage = $"[{DateTime.Now:HH:mm:ss}] {message}
    {Environment.NewLine}";
```

```
2688
2689            // Debug messages go to the correct log
2690            if (isDebug)
2691            {
2692                if (channel == "CH1")
2693                {
2694                    testResultsBox.AppendText(formattedMessage);
2695                }
2696                else
2697                {
2698                    testResultsBoxCH2.AppendText(formattedMessage);
2699                }
2700            }
2701            else
2702            {
2703                targetBox.AppendText(formattedMessage);
2704            }
2705
2706            // Function to highlight keywords
2707            void ColorText(string word, Color color)
2708            {
2709                int index = startIndex;
2710                while ((index = targetBox.Text.IndexOf(word, index,
    StringComparison.OrdinalIgnoreCase)) != -1)
2711                {
2712                    targetBox.Select(index, word.Length);
2713                    targetBox.SelectionColor = color;
2714                    index += word.Length;
2715                }
2716            }
2717
2718            // Apply color formatting
2719            ColorText("PASS", Color.Blue);
2720            ColorText("FAIL", Color.Red);
2721            ColorText("DEBUG", Color.DarkGreen);
2722
2723            // Reset selection and auto-scroll
2724            targetBox.SelectionStart = targetBox.TextLength;
2725            targetBox.SelectionLength = 0;
2726            targetBox.SelectionColor = targetBox.ForeColor;
2727            targetBox.ScrollToCaret();
2728        }
2729
2730
2731
2732
2733    }
2734 }
```

src\Models\OscilloscopeSettings.cs

```
using System;
using System.Linq;

namespace Oscilloscope.Models
{
    public class OscilloscopeSettings
    {
        // Horizontal (Timebase) Settings
        public double TimebaseScale { get; set; } = 1e-3; // Default: 1ms/div
        public string TimebaseReference { get; set; } = "CENTER";

        // Vertical Settings per Channel
        public class ChannelSettings
        {
            public bool Enabled { get; set; } = false;
            public double VerticalScale { get; set; } = 1.0; // Default: 1V/div
            public double Offset { get; set; } = 0.0;
        }

        // Array of channel settings (MSOX3104T has 4 analog channels)
        public ChannelSettings[] Channels { get; private set; }

        // Waveform Generator Settings
        public class WaveformSettings
        {
            public string WaveformType { get; set; } = "SINusoid";
            public double Frequency { get; set; } = 1000.0; // Default: 1kHz
            public double Amplitude { get; set; } = 1.0; // Default: 1Vpp
            public double Offset { get; set; } = 0.0;
        }

        public WaveformSettings WaveformGenerator { get; set; }

        public OscilloscopeSettings()
        {
            // Initialize channel settings
            Channels = new ChannelSettings[4];
            for (int i = 0; i < 4; i++)
            {
                Channels[i] = new ChannelSettings();
            }

            // Initialize waveform generator settings
            WaveformGenerator = new WaveformSettings();
        }

        public void ValidateSettings()
        {
            if (TimebaseScale <= 0)
                throw new ArgumentException("Timebase scale must be positive");

```

```
            var validReferences = new[] { "LEFT", "CENTER", "RIGHT" };
            if (string.IsNullOrEmpty(TimebaseReference) || !validReferences.Contains(TimebaseReference.ToUpper()))
                throw new ArgumentException($"Invalid timebase reference point. Use one of: {string.Join(", ", validReferences)}");

            for (int i = 0; i < Channels.Length; i++)
            {
                if (Channels[i].VerticalScale <= 0)
                    throw new ArgumentException($"Vertical scale for channel {i + 1} must be positive");
            }

            if (WaveformGenerator.Frequency <= 0)
                throw new ArgumentException("Waveform frequency must be positive");

            if (WaveformGenerator.Amplitude <= 0)
                throw new ArgumentException("Waveform amplitude must be positive");
        }
    }
}
```

mainform.cs

```
using System;
using System.Diagnostics;
using System.Drawing;
using System.IO.Ports;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace SerialPortEnhancedGUI
{
    public class EnhancedSerialForm : Form
    {
        // Serial port objects for COM1 and COM2
        private SerialPort port1;
        private SerialPort port2;

        // Controls for Port 1
        private GroupBox groupBoxPort1;
        private Label lblPort1Status;
        private TextBox txtPort1Send;
        private Button btnPort1Open;
        private Button btnPort1Close;
        private Button btnPort1Send;
        private TextBox txtPort1Receive;

        // Controls for Port 2
        private GroupBox groupBoxPort2;
        private Label lblPort2Status;
        private TextBox txtPort2Send;
        private Button btnPort2Open;
        private Button btnPort2Close;
        private Button btnPort2Send;
        private TextBox txtPort2Receive;

        // TabControl for Global Log only
        private TabControl tabControlResults;
        private TabPage tabPageLog;
        private TextBox txtLog;

        // Test control buttons and summary
        private Button btnTest;
        private Button btnPause;
        private Button btnStop;
        private TextBox txtTestSummary;

        // Real-time test timer display
        private Label lblTestTime;

        // UI timer (using Windows Forms Timer)
        private System.Windows.Forms.Timer uiTimer;

```

```
52          // Cancellation token and pause flag for test procedure
53          private CancellationTokenSource testCts;
54          private volatile bool isPaused;
55
56          // Stopwatch for test timing
57          private Stopwatch testStopwatch;
58
59          public EnhancedSerialForm()
60          {
61              // Set up form properties
62              this.Text = "Enhanced Serial Communication GUI";
63              this.Size = new Size(800, 680);
64              this.StartPosition = FormStartPosition.CenterScreen;
65
66              InitializeComponents();
67              InitializeSerialPorts();
68          }
69
70          private void InitializeComponents()
71          {
72              // ---------------------------
73              // Group Box for Port 1 (COM1)
74              // ---------------------------
75              groupBoxPort1 = new GroupBox();
76              groupBoxPort1.Text = "Port 1 (COM1)";
77              groupBoxPort1.Size = new Size(370, 200);
78              groupBoxPort1.Location = new Point(10, 10);
79
80              lblPort1Status = new Label();
81              lblPort1Status.Text = "Status: Closed";
82              lblPort1Status.Location = new Point(10, 25);
83              lblPort1Status.AutoSize = true;
84
85              btnPort1Open = new Button();
86              btnPort1Open.Text = "Open";
87              btnPort1Open.Location = new Point(250, 20);
88              btnPort1Open.Size = new Size(100, 25);
89              btnPort1Open.Click += BtnPort1Open_Click;
90
91              btnPort1Close = new Button();
92              btnPort1Close.Text = "Close";
93              btnPort1Close.Location = new Point(250, 55);
94              btnPort1Close.Size = new Size(100, 25);
95              btnPort1Close.Click += BtnPort1Close_Click;
96
97              Label lblPort1Send = new Label();
98              lblPort1Send.Text = "Send:";
99              lblPort1Send.Location = new Point(10, 60);
100             lblPort1Send.AutoSize = true;
101
102             txtPort1Send = new TextBox();
103             txtPort1Send.Location = new Point(70, 60);
104             txtPort1Send.Size = new Size(160, 25);
105
```

```
106             btnPort1Send = new Button();
107             btnPort1Send.Text = "Send";
108             btnPort1Send.Location = new Point(250, 90);
109             btnPort1Send.Size = new Size(100, 25);
110             btnPort1Send.Click += BtnPort1Send_Click;
111
112             Label lblPort1Receive = new Label();
113             lblPort1Receive.Text = "Receive:";
114             lblPort1Receive.Location = new Point(10, 130);
115             lblPort1Receive.AutoSize = true;
116
117             txtPort1Receive = new TextBox();
118             txtPort1Receive.Location = new Point(70, 125);
119             txtPort1Receive.Size = new Size(280, 25);
120             txtPort1Receive.ReadOnly = true;
121
122             groupBoxPort1.Controls.Add(lblPort1Status);
123             groupBoxPort1.Controls.Add(btnPort1Open);
124             groupBoxPort1.Controls.Add(btnPort1Close);
125             groupBoxPort1.Controls.Add(lblPort1Send);
126             groupBoxPort1.Controls.Add(txtPort1Send);
127             groupBoxPort1.Controls.Add(btnPort1Send);
128             groupBoxPort1.Controls.Add(lblPort1Receive);
129             groupBoxPort1.Controls.Add(txtPort1Receive);
130
131             // ---------------------------
132             // Group Box for Port 2 (COM2)
133             // ---------------------------
134             groupBoxPort2 = new GroupBox();
135             groupBoxPort2.Text = "Port 2 (COM2)";
136             groupBoxPort2.Size = new Size(370, 200);
137             groupBoxPort2.Location = new Point(400, 10);
138
139             lblPort2Status = new Label();
140             lblPort2Status.Text = "Status: Closed";
141             lblPort2Status.Location = new Point(10, 25);
142             lblPort2Status.AutoSize = true;
143
144             btnPort2Open = new Button();
145             btnPort2Open.Text = "Open";
146             btnPort2Open.Location = new Point(250, 20);
147             btnPort2Open.Size = new Size(100, 25);
148             btnPort2Open.Click += BtnPort2Open_Click;
149
150             btnPort2Close = new Button();
151             btnPort2Close.Text = "Close";
152             btnPort2Close.Location = new Point(250, 55);
153             btnPort2Close.Size = new Size(100, 25);
154             btnPort2Close.Click += BtnPort2Close_Click;
155
156             Label lblPort2Send = new Label();
157             lblPort2Send.Text = "Send:";
158             lblPort2Send.Location = new Point(10, 60);
159             lblPort2Send.AutoSize = true;
```

```
160
161             txtPort2Send = new TextBox();
162             txtPort2Send.Location = new Point(70, 60);
163             txtPort2Send.Size = new Size(160, 25);
164
165             btnPort2Send = new Button();
166             btnPort2Send.Text = "Send";
167             btnPort2Send.Location = new Point(250, 90);
168             btnPort2Send.Size = new Size(100, 25);
169             btnPort2Send.Click += BtnPort2Send_Click;
170
171             Label lblPort2Receive = new Label();
172             lblPort2Receive.Text = "Receive:";
173             lblPort2Receive.Location = new Point(10, 130);
174             lblPort2Receive.AutoSize = true;
175
176             txtPort2Receive = new TextBox();
177             txtPort2Receive.Location = new Point(70, 125);
178             txtPort2Receive.Size = new Size(280, 25);
179             txtPort2Receive.ReadOnly = true;
180
181             groupBoxPort2.Controls.Add(lblPort2Status);
182             groupBoxPort2.Controls.Add(btnPort2Open);
183             groupBoxPort2.Controls.Add(btnPort2Close);
184             groupBoxPort2.Controls.Add(lblPort2Send);
185             groupBoxPort2.Controls.Add(txtPort2Send);
186             groupBoxPort2.Controls.Add(btnPort2Send);
187             groupBoxPort2.Controls.Add(lblPort2Receive);
188             groupBoxPort2.Controls.Add(txtPort2Receive);
189
190             // ---------------------------
191             // TabControl for Global Log only
192             // ---------------------------
193             tabControlResults = new TabControl();
194             tabControlResults.Location = new Point(10, 220);
195             tabControlResults.Size = new Size(760, 200);
196
197             tabPageLog = new TabPage("Global Log");
198             txtLog = new TextBox();
199             txtLog.Multiline = true;
200             txtLog.ScrollBars = ScrollBars.Vertical;
201             txtLog.ReadOnly = true;
202             txtLog.Dock = DockStyle.Fill;
203             tabPageLog.Controls.Add(txtLog);
204             tabControlResults.TabPages.Add(tabPageLog);
205
206             // ---------------------------
207             // Test Buttons and Test Summary TextBox
208             // ---------------------------
209             btnTest = new Button();
210             btnTest.Text = "Run Test";
211             btnTest.Location = new Point(10, 430);
212             btnTest.Size = new Size(100, 30);
213             btnTest.Click += BtnTest_Click;
```

```
214
215             btnPause = new Button();
216             btnPause.Text = "Pause";
217             btnPause.Location = new Point(120, 430);
218             btnPause.Size = new Size(100, 30);
219             btnPause.Click += BtnPause_Click;
220
221             btnStop = new Button();
222             btnStop.Text = "Stop";
223             btnStop.Location = new Point(230, 430);
224             btnStop.Size = new Size(100, 30);
225             btnStop.Click += BtnStop_Click;
226
227             txtTestSummary = new TextBox();
228             txtTestSummary.Multiline = true;
229             txtTestSummary.ScrollBars = ScrollBars.Vertical;
230             txtTestSummary.ReadOnly = true;
231             txtTestSummary.Location = new Point(10, 470);
232             txtTestSummary.Size = new Size(760, 80);
233
234             // ---------------------------
235             // Test Time Label (real-time display)
236             // ---------------------------
237             lblTestTime = new Label();
238             lblTestTime.Text = "Test Time: 00:00:00";
239             lblTestTime.Location = new Point(10, 560);
240             lblTestTime.AutoSize = true;
241             lblTestTime.Font = new Font("Arial", 10, FontStyle.Bold);
242
243             // ---------------------------
244             // UI Timer to update test time every 100ms
245             // ---------------------------
246             uiTimer = new System.Windows.Forms.Timer();
247             uiTimer.Interval = 100; // 100ms
248             uiTimer.Tick += (s, e) =>
249             {
250                 if (testStopwatch != null && testStopwatch.IsRunning)
251                 {
252                     lblTestTime.Text = "Test Time: " +
    testStopwatch.Elapsed.ToString(@"hh\:mm\:ss");
253                 }
254             };
255
256             // Add all controls to the form
257             this.Controls.Add(groupBoxPort1);
258             this.Controls.Add(groupBoxPort2);
259             this.Controls.Add(tabControlResults);
260             this.Controls.Add(btnTest);
261             this.Controls.Add(btnPause);
262             this.Controls.Add(btnStop);
263             this.Controls.Add(txtTestSummary);
264             this.Controls.Add(lblTestTime);
265         }
266
```

```
267         private void InitializeSerialPorts()
268         {
269             // Initialize COM1
270             port1 = new SerialPort("COM1", 9600, Parity.None, 8, StopBits.One);
271             port1.DataReceived += Port1_DataReceived;
272 
273             // Initialize COM2 with the global event handler
274             port2 = new SerialPort("COM2", 9600, Parity.None, 8, StopBits.One);
275             port2.DataReceived += Port2_DataReceived;
276         }
277 
278         // Global event handler for Port1 DataReceived
279         private void Port1_DataReceived(object sender, SerialDataReceivedEventArgs e)
280         {
281             try
282             {
283                 string data = port1.ReadLine();
284                 this.Invoke(new Action(() =>
285                 {
286                     txtPort1Receive.Text = data;
287                     AppendLog($"Port1 received: {data}");
288                 }));
289             }
290             catch (Exception ex)
291             {
292                 this.Invoke(new Action(() => AppendLog($"Error reading Port1:
    {ex.Message}")));
293             }
294         }
295 
296         // Global event handler for Port2 DataReceived (non-test messages)
297         private void Port2_DataReceived(object sender, SerialDataReceivedEventArgs e)
298         {
299             try
300             {
301                 string data = port2.ReadLine();
302                 this.Invoke(new Action(() =>
303                 {
304                     txtPort2Receive.Text = data;
305                     AppendLog($"Port2 received: {data}");
306                 }));
307             }
308             catch (Exception ex)
309             {
310                 this.Invoke(new Action(() => AppendLog($"Error reading Port2:
    {ex.Message}")));
311             }
312         }
313 
314         // Append message to the global log
315         private void AppendLog(string message)
316         {
317             txtLog.AppendText($"{DateTime.Now:HH:mm:ss} - {message}{Environment.NewLine}");
318         }
```

```
319
320         // ---------------------------
321         // Button Click Handlers for Port1
322         // ---------------------------
323         private void BtnPort1Open_Click(object sender, EventArgs e)
324         {
325             try
326             {
327                 if (!port1.IsOpen)
328                 {
329                     port1.Open();
330                     lblPort1Status.Text = "Status: Open";
331                     AppendLog("Port1 opened.");
332                 }
333             }
334             catch (Exception ex)
335             {
336                 AppendLog($"Error opening Port1: {ex.Message}");
337             }
338         }
339
340         private void BtnPort1Close_Click(object sender, EventArgs e)
341         {
342             try
343             {
344                 if (port1.IsOpen)
345                 {
346                     port1.Close();
347                     lblPort1Status.Text = "Status: Closed";
348                     AppendLog("Port1 closed.");
349                 }
350             }
351             catch (Exception ex)
352             {
353                 AppendLog($"Error closing Port1: {ex.Message}");
354             }
355         }
356
357         private void BtnPort1Send_Click(object sender, EventArgs e)
358         {
359             if (port1.IsOpen)
360             {
361                 try
362                 {
363                     string data = txtPort1Send.Text;
364                     port1.WriteLine(data);
365                     AppendLog($"Port1 sent: {data}");
366                 }
367                 catch (Exception ex)
368                 {
369                     AppendLog($"Error sending from Port1: {ex.Message}");
370                 }
371             }
372             else
```

```
            {
                AppendLog("Port1 is not open.");
            }
        }

        // ---------------------------
        // Button Click Handlers for Port2
        // ---------------------------
        private void BtnPort2Open_Click(object sender, EventArgs e)
        {
            try
            {
                if (!port2.IsOpen)
                {
                    port2.Open();
                    lblPort2Status.Text = "Status: Open";
                    AppendLog("Port2 opened.");
                }
            }
            catch (Exception ex)
            {
                AppendLog($"Error opening Port2: {ex.Message}");
            }
        }

        private void BtnPort2Close_Click(object sender, EventArgs e)
        {
            try
            {
                if (port2.IsOpen)
                {
                    port2.Close();
                    lblPort2Status.Text = "Status: Closed";
                    AppendLog("Port2 closed.");
                }
            }
            catch (Exception ex)
            {
                AppendLog($"Error closing Port2: {ex.Message}");
            }
        }

        private void BtnPort2Send_Click(object sender, EventArgs e)
        {
            if (port2.IsOpen)
            {
                try
                {
                    string data = txtPort2Send.Text;
                    port2.WriteLine(data);
                    AppendLog($"Port2 sent: {data}");
                }
                catch (Exception ex)
                {
```

```
427                     AppendLog($"Error sending from Port2: {ex.Message}");
428                 }
429             }
430             else
431             {
432                 AppendLog("Port2 is not open.");
433             }
434         }
435 
436         // ---------------------------
437         // Test Procedure Button Click Handler
438         // ---------------------------
439         private async void BtnTest_Click(object sender, EventArgs e)
440         {
441             // If a test is already running, ignore a new Run request.
442             if (testCts != null && !testCts.IsCancellationRequested)
443                 return;
444 
445             // Reset pause flag and cancellation token; reset stopwatch.
446             isPaused = false;
447             btnPause.Text = "Pause";
448             testCts = new CancellationTokenSource();
449             testStopwatch = new Stopwatch();
450             testStopwatch.Start();
451             uiTimer.Start();
452 
453             AppendLog("Starting test procedure...");
454             txtTestSummary.Clear(); // Clear previous summary
455 
456             // Auto-open both ports if not already open
457             if (!port1.IsOpen)
458             {
459                 try { port1.Open(); lblPort1Status.Text = "Status: Open"; AppendLog("Port1 auto-opened for test."); }
460                 catch (Exception ex) { AppendLog($"Error auto-opening Port1: {ex.Message}"); return; }
461             }
462             if (!port2.IsOpen)
463             {
464                 try { port2.Open(); lblPort2Status.Text = "Status: Open"; AppendLog("Port2 auto-opened for test."); }
465                 catch (Exception ex) { AppendLog($"Error auto-opening Port2: {ex.Message}"); return; }
466             }
467 
468             // Initialize counters for each phase
469             int phase1Pass = 0, phase1Total = 25;
470             int phase2Pass = 0, phase2Total = 25;
471 
472             try
473             {
474                 // ------------- Phase 1: Test from Port1 to Port2 (Steps 1 to 25) ----------
475                 for (int i = 1; i <= phase1Total; i++)
```

```
476                 {
477                     testCts.Token.ThrowIfCancellationRequested();
478                     // Wait while paused
479                     while (isPaused)
480                     {
481                         await Task.Delay(100, testCts.Token);
482                         testCts.Token.ThrowIfCancellationRequested();
483                     }
484 
485                     string testMessage = $"TEST1234_STEP_{i}";
486                     this.Invoke(new Action(() => txtPort2Receive.Text = string.Empty));
487 
488                     // Temporarily remove global Port2 handler
489                     port2.DataReceived -= Port2_DataReceived;
490 
491                     // Await response from Port2
492                     var tcs = new TaskCompletionSource<string>();
493                     SerialDataReceivedEventHandler testHandler = null;
494                     testHandler = (s, evt) =>
495                     {
496                         try
497                         {
498                             string received = port2.ReadLine();
499                             this.Invoke(new Action(() =>
500                             {
501                                 txtPort2Receive.Text = received;
502                             }));
503                             tcs.TrySetResult(received);
504                         }
505                         catch (Exception ex)
506                         {
507                             tcs.TrySetException(ex);
508                         }
509                         port2.DataReceived -= testHandler;
510                     };
511                     port2.DataReceived += testHandler;
512 
513                     // Send test message from Port1
514                     port1.WriteLine(testMessage);
515                     AppendLog($"Phase 1 - Step {i}: Sent from Port1: {testMessage}");
516 
517                     string receivedTestMessage = "";
518                     bool stepPassed = false;
519                     var delayTask = Task.Delay(1000, testCts.Token);
520                     var completedTask = await Task.WhenAny(tcs.Task, delayTask);
521                     if (completedTask == tcs.Task)
522                     {
523                         receivedTestMessage = tcs.Task.Result;
524                         if (receivedTestMessage.Trim() == testMessage.Trim())
525                         {
526                             stepPassed = true;
527                             phase1Pass++;
528                         }
529                     }
```

```
530                     else
531                     {
532                         AppendLog($"Phase 1 - Step {i}: Test timed out waiting for
    response.");
533                     }
534                     AppendLog($"Phase 1 - Step {i} result: {(stepPassed ? "Passed" :
    "Failed")}");
535                     // Reattach the global Port2 handler
536                     port2.DataReceived += Port2_DataReceived;
537 
538                     await Task.Delay(100, testCts.Token);
539                 }
540 
541                 // ------------- Phase 2: Test from Port2 to Port1 (Steps 26 to 50) --------
    -----
542                 for (int i = phase1Total + 1; i <= phase1Total + phase2Total; i++)
543                 {
544                     testCts.Token.ThrowIfCancellationRequested();
545                     while (isPaused)
546                     {
547                         await Task.Delay(100, testCts.Token);
548                         testCts.Token.ThrowIfCancellationRequested();
549                     }
550 
551                     string testMessage = $"TEST1234_STEP_{i}";
552                     this.Invoke(new Action(() => txtPort1Receive.Text = string.Empty));
553 
554                     // Temporarily remove global Port1 handler
555                     port1.DataReceived -= Port1_DataReceived;
556 
557                     // Await response from Port1
558                     var tcs = new TaskCompletionSource<string>();
559                     SerialDataReceivedEventHandler testHandler = null;
560                     testHandler = (s, evt) =>
561                     {
562                         try
563                         {
564                             string received = port1.ReadLine();
565                             this.Invoke(new Action(() =>
566                             {
567                                 txtPort1Receive.Text = received;
568                             }));
569                             tcs.TrySetResult(received);
570                         }
571                         catch (Exception ex)
572                         {
573                             tcs.TrySetException(ex);
574                         }
575                         port1.DataReceived -= testHandler;
576                     };
577                     port1.DataReceived += testHandler;
578 
579                     // Send test message from Port2
580                     port2.WriteLine(testMessage);
```

```
                    AppendLog($"Phase 2 - Step {i}: Sent from Port2: {testMessage}");

                    string receivedTestMessage = "";
                    bool stepPassed = false;
                    var delayTask2 = Task.Delay(1000, testCts.Token);
                    var completedTask2 = await Task.WhenAny(tcs.Task, delayTask2);
                    if (completedTask2 == tcs.Task)
                    {
                        receivedTestMessage = tcs.Task.Result;
                        if (receivedTestMessage.Trim() == testMessage.Trim())
                        {
                            stepPassed = true;
                            phase2Pass++;
                        }
                    }
                    else
                    {
                        AppendLog($"Phase 2 - Step {i}: Test timed out waiting for response.");
                    }
                    AppendLog($"Phase 2 - Step {i} result: {(stepPassed ? "Passed" : "Failed")}");
                    // Reattach the global Port1 handler
                    port1.DataReceived += Port1_DataReceived;

                    await Task.Delay(100, testCts.Token);
                }
            }
            catch (OperationCanceledException)
            {
                AppendLog("Test procedure was stopped.");
            }
            finally
            {
                testStopwatch.Stop();
                uiTimer.Stop();
            }

            // Build overall report summary including total test time
            double phase1SuccessRate = phase1Total > 0 ? (phase1Pass * 100.0 / phase1Total) : 0;
            double phase2SuccessRate = phase2Total > 0 ? (phase2Pass * 100.0 / phase2Total) : 0;
            int overallPass = phase1Pass + phase2Pass;
            int overallTotal = phase1Total + phase2Total;
            double overallSuccessRate = overallTotal > 0 ? (overallPass * 100.0 / overallTotal) : 0;
            string totalTestTime = testStopwatch.Elapsed.ToString(@"hh\:mm\:ss");

            string summary = "";
            summary += $"Phase 1 Test: {phase1Pass}/{phase1Total} ({phase1SuccessRate:F2}% success){Environment.NewLine}";
            summary += $"Phase 2 Test: {phase2Pass}/{phase2Total} ({phase2SuccessRate:F2}% success){Environment.NewLine}";
```

```
628             summary += $"Overall: {overallPass}/{overallTotal} ({overallSuccessRate:F2}%
    success){Environment.NewLine}";
629             summary += $"Total Test Time: {totalTestTime}";
630 
631             AppendLog("Test procedure completed.");
632             txtTestSummary.Text = summary;
633             lblTestTime.Text = "Test Time: " + totalTestTime;
634         }
635 
636         // ---------------------------
637         // Pause Button Click Handler
638         // ---------------------------
639         private void BtnPause_Click(object sender, EventArgs e)
640         {
641             // Toggle pause state
642             isPaused = !isPaused;
643             btnPause.Text = isPaused ? "Resume" : "Pause";
644             if (isPaused)
645             {
646                 testStopwatch.Stop();
647                 AppendLog("Test paused.");
648             }
649             else
650             {
651                 testStopwatch.Start();
652                 AppendLog("Test resumed.");
653             }
654         }
655 
656         // ---------------------------
657         // Stop Button Click Handler
658         // ---------------------------
659         private void BtnStop_Click(object sender, EventArgs e)
660         {
661             if (testCts != null)
662             {
663                 testCts.Cancel();
664                 AppendLog("Test stop requested.");
665                 // Reset stopwatch and timer display
666                 testStopwatch.Reset();
667                 lblTestTime.Text = "Test Time: 00:00:00";
668                 uiTimer.Stop();
669             }
670         }
671     }
672 }
673 
```