Summer Project Report: Crystal Surface Mapping System

Raj Negi

17/06/2025 - 14/09/2025

Abstract

This project report documents the design and implementation of an automated crystal surface mapping program. Central to the system is the **WinRotor Client**, running on a Raspberry Pi 5, which orchestrates commands to two unique servers: the **Kohzu Server** (controlling the physical rotors) and the **WinSpec Server** (controlling the spectrometer via the WinX32 application). This report focuses on the architectural decisions, specifically the adoption of the MVVM pattern in the client and the use of modern communication protocols (gRPC and RPC) to ensure reliable and efficient operations.

Contents

Chapter 1

Introduction

Raman Spectroscopy is a standard technique used for analysing the chemical structure and composition of crystals. However, due to inhomogeneities present in the crystals, Raman mapping is often necessary to identify specific areas of interests. This mapping process typically involves a system of rotors to position the sample and a spectrometer to collect the scattered light. Performing this map manually is a laborious and time consuming process. The goal of this project is, therefore, to automate this data collection procedure. This requires synchronized control over the rotors (Kohzu rotors, Windows 10 host) and the spectrometer (WinX32 application, Windows 7 host). The **WinRotor Client** acts as the primary control application that unites these independent systems.

1.1 System Architecture Overview

The system is divided into three components:

1. **WinRotor Client:** The main control and user interface application.
2. **Kohzu Server:** Responsible for direct control of the rotors hardware.
3. **WinSpec Server:** Responsible for interfacing with the WinX32 application to control the spectrometer.

Chapter 2

WinRotor Client: Architecture and Implementation

The WinRotor Client, developed on the Raspberry Pi 5, is the command centre for the entire mapping process and adopts a MVVM architectural pattern.

2.1 Model-View-ViewModel (MVVM) Architectural Pattern

MVVM is a structural design pattern that separates the application into three distinct parts:

- **View:** The graphical user interface (UI) elements (e.g., windows, buttons, text boxes). It contains no application logic and simply displays data and captures user input.
- **ViewModel:** Acts as a mediator between the Model and the View. It exposes data from the Model in a View-friendly format (using properties and commands) and handles the View's actions. It is responsible for all UI-specific business logic.
- **Model:** Contains the core business logic, data structures, and data access components (like the gRPC/RPC communication clients). It is independent of the View and ViewModel.

This separation enables the development of the UI (View) and the logic (ViewModel/Model) to be independently carried out, providing ease of maintenance and testing.

2.2 Communication Contracts: The `Protos` Directory

The `Protos` directory is central to reliable communication with the **Kohzu Server** via gRPC. This folder contains the `.proto` files that serve as the interface definition language (IDL), defining the precise methods and data structures (messages) exchanged between the client and server. These files act as a formal contract, ensuring type-safe and version-controlled communication.

2.2.1 Rotor Service Definitions

The protos are split into two services: `RotorDrive` for issuing commands and `RotorStatus` for querying state.

The `rotorDrive.proto` file defines the actions the client can request from the Kohzu server, such as moving the stages.

```
// Protos/rotorDrive.proto
syntax = "proto3";

option csharp_namespace = "KohzuServer";

service RotorDrive {
  // Relative Position Drive - Moves a single axis by a
   specific step size
  rpc Rps (RpsRequest) returns (RpsResponse);
  // Multi-Axis Simultaneous Drive for 2 axes (Y and Z)
  rpc Mps2 (Mps2Request) returns (Mps2Response);
  // Multi-Axis Simultaneous Drive for 3 axes (X, Y, and Z)
  rpc Mps3 (Mps3Request) returns (Mps3Response);
}

// ... message definitions ...
```

Listing 2.1: Excerpt from rotorDrive.proto (Client Commands)

The `rotorStatus.proto` file defines the methods the client uses to monitor the state of the rotor stages, essential for synchronizing the motor movements with the spectrometer acquisition. This includes checking connection health and reading physical position.

```
// Protos/rotorStatus.proto
syntax = "proto3";

option csharp_namespace = "KohzuServer";

```

```
service RotorStatus {
  // Connection Test - Used to verify server availability
  rpc TestConnection (IdnRequest) returns (IdnResponse);
  // Get Encoder Position of a single axis
  rpc GetEncoderPosition (RdeRequest) returns (RdeResponse);
  // Get Present Position (Pulse Position) of a single axis
  rpc GetPresentPosition (RdpRequest) returns (RdpResponse);
}

// ... message definitions ...
```

Listing 2.2: Excerpt from rotorStatus.proto (Client Status Queries)

2.3 Configuration and Utility: The `Common` Directory

The `Common` directory contains the `GlobalParameters.cs` class, which sets the parameter values across the WinRotor client . This ensures consistency and simplifies maintenance / corrections if values need to be changed.

2.4 Core Components

2.4.1 `Models` Directory: Data, Clients, and Logic

The `Models` layer contains the application's domain logic, persistent data structures, and the communication clients that interact with the external servers.

Client Implementations

The `Clients` subdirectory contains the specific classes responsible for protocol handling and server communication, providing an asynchronous interface for the rest of the application (ViewModels) to use.

`RotorClient.cs` **(gRPC)** This class handles all communication with the **Kohzu Server** using gRPC. It uses the services defined in the `Protos` directory to create typed client objects (`RotorDriveClient` and `RotorStatusClient`) for movement commands and status queries, respectively.

`WinSpecClient.cs` **(RPC)** This class is responsible for communicating with the **WinSpec Server** using a simple, custom Remote Procedure Call (RPC) implementation over raw TCP sockets.

Data Structures and Main Controller

The remaining components of the `Models` layer contain the high-level logic and data structures required to execute the full mapping sequence.

`Controller.cs` **(The Orchestrator)** The `Controller` class serves as the **main business logic orchestrator** for the entire WinRotor Client. It is initialized with instances of both `RotorClient` and `WinSpecClient` via dependency injection, allowing it to coordinate the movement of the rotors and the acquisition of data from the spectrometer.

`MapParameters.cs` **(The Data State)** This C# `record` defines the immutable state required to run a complete mapping sequence.

`FileItem.cs` **(The Data Container)** The `FileItem` class is a simple data container used within the application to represent saved configurations or file system paths.

2.4.2 `ViewModels` Directory: Presenting Data and Handling Actions

The `ViewModels` layer is responsible for structuring the data retrieved from the `Models` and providing **Commands** that the `Views` (UI buttons/elements) can bind to.

The Dialog ViewModels (`DialogViewModel.cs` and derivatives) To handle non-blocking, asynchronous user interaction (such as displaying errors or confirming actions), the client uses a hierarchy of dialog ViewModels derived from the base class `DialogViewModel.cs`. This base class provides the necessary asynchronous methods (`WaitAsync()`, `Show()`, `Close()`) and properties (`IsDialogOpen`) to enable the View to open and wait for a dialog response without freezing the main application thread. Specific inherited classes include `InfoDialogViewModel`, `ConfirmDialogViewModel`, `ConfirmMapDialogViewModel`, `AddFilePathDialogViewModel`, and `FilesListDialogViewModel`

Other ViewModels `ControlPanelViewModel.cs`, `MapViewModel.cs`, `HomeViewModel.cs`, `MainViewModel.cs`, `HelpViewModel.cs`, and `ErrorViewModel.cs` manage their respective page logic and interactions.

Chapter 3

Kohzu Server: Rotor Control

The Kohzu Server is the bridge between the high-level control commands from the WinRotor Client (via gRPC) and the low-level, serial commands required by the physical Kohzu rotors hardware. It is structured to efficiently receive, process, and execute movement requests while handling status queries in real-time.

3.1 Communication Contracts: The `Protos` Directory

For the Kohzu Server, the `Protos` directory defines the **endpoints** the server must implement. These files are identical to those used by the WinRotor Client, ensuring a tight, binary-compatible contract for communication.

3.1.1 RotorDrive Service Definition (`rotorDrive.proto`)

This service defines the remote procedures (RPCs) that handle motor movement commands. The server's implementation must provide concrete methods for each RPC defined here.

3.1.2 RotorStatus Service Definition (`rotorStatus.proto`)

This service defines the RPCs for querying the status of the motor controller and the current position of the axes.

3.2 Serial Communication Layer: The `SerialPort` Directory

The `SerialPort` directory contains the core logic for communicating directly with the Kohzu motor controller using the **RS-232C standard** via a physical serial connection. This component is solely responsible for translating ASCII command strings into a hardware-compatible, framed byte sequence and reading the controller's response.

3.2.1 `SerialPortCommunication.cs`: Framing and I/O

The `SerialPortCommunication.cs` class encapsulates all low-level communication logic, hiding the complexity of byte-level framing from the higher-level gRPC services.

Initialization and Configuration

The class sets up the C# `System.IO.Ports.SerialPort` with parameters specific to the Kohzu controller:

- **PortName:** Set to COM4.
- **BaudRate:** Set to `115200`, matching the controller's required communication speed.
- **Timeouts:** Both `ReadTimeout` and `WriteTimeout` are configured (`5000ms`) to prevent the gRPC services from blocking indefinitely if the hardware fails to respond.
- **Line Termination:** The `NewLine` property is explicitly set to `"\r\n"` (Carriage Return + Line Feed), which is required by the Kohzu protocol for command recognition.

The `SendCommand` Method and Framing Protocol

The core function, `SendCommand(string command)`, handles the crucial task of framing the human-readable ASCII command string into the exact binary format required by the hardware: **STX + Command + CR + LF**. By isolating the serial communication within this dedicated class, the **'Services'** layer remains clean, only needing to prepare the necessary ASCII command string (e.g., `"RPS 1,100"`) and receiving the final controller feedback string (e.g., `"C00"`).

3.3 gRPC Service Implementations: The `Services` Directory

The `Services` directory contains the concrete implementations of the gRPC service contracts defined in the `Protos` directory. These classes serve as the **application logic gateway**, translating high-level gRPC requests (like "Move 100 pulses") into the specific ASCII commands required by the physical Kohzu rotors and routing them through the `SerialPortCommunication` component.

3.3.1 `RotorDriveService.cs`: Motor Movement

This class inherits from `RotorDrive.RotorDriveBase` and provides the logic for all movement commands. It uses dependency injection to receive the `SerialPortCommunication` instance, enabling it to execute commands.

- **Command Construction:** Each gRPC method (e.g., `Rps`, `Mps2`, `Mps3`) first constructs the full ASCII command string according to the Kohzu protocol, incorporating the axis, step size/position, speed, and response method constants defined locally. For example, a relative move command (`Rps`) is formatted as: `RPS{Axis}/{Speed}/{StepSize}/{ResponseMethod`
- **Execution:** The constructed command string is passed to the asynchronous `SendCommand` method of the `SerialPortCommunication` layer.
- **Feedback:** The raw feedback string received from the motor controller (e.g., `"C00"` for success) is packaged into the corresponding gRPC response message and returned to the client.

```
public override async Task<RpsResponse> Rps(RpsRequest
    request, ServerCallContext context)
{
    RpsResponse response = new();

    // Axis, Speed (4), StepSize, ResponseMethod (0)
    var command = $"RPS{request.Axis}/{Speed}/{request.
    StepSize}/{ResponseMethod}";

    // Send Command via the SerialPort layer
    var feedback = await _serialPortCommunication.SendCommand
    (command);

    response.Feedback = feedback;
    return await Task.FromResult(response);
```

```
}
```

Listing 3.1: RotorDriveService.cs: Relative Position Drive (Rps)

3.3.2 `RotorStatusService.cs`: Status Queries and Connection Health

This class inherits from `RotorStatus.RotorStatusBase` and handles all non-movement queries.

- **Connection Test (`TestConnection`):** This critical method verifies the end-to-end communication health. It attempts to send a null or basic command through the serial port. If the operation completes without raising a `TimeoutException`, the connection is deemed healthy. This is used by the client's `HomeViewModel` for status display.
- **Position Reading (`GetEncoderPosition`, `GetPresentPosition`):** These methods send commands like `RDE{Axis}` (Read Encoder) and `RDP{Axis}` (Read Pulse Position). They receive the feedback string (which contains both the command echo and the position value) and apply simple string parsing (`feedback.Split().Last()`) to extract the final integer position before sending it back to the client.

```
public override async Task<RdeResponse> GetEncoderPosition(
    RdeRequest request, ServerCallContext context)
{
    RdeResponse response = new RdeResponse();

    // Initialize command
    var command = $"RDE{request.Axis}";

    // Send command
    var feedback = await _serialPortCommunication.SendCommand
    (command);

    // Pick out position from feedback (e.g., C\tRDE1\t-8750)
    response.Position = feedback.Split(new[] { '\t', ' ' },
    StringSplitOptions.RemoveEmptyEntries).Last();

    return await Task.FromResult(response);
}
```

Listing 3.2: RotorStatusService.cs: Read Encoder Position (Rde)

Chapter 4

WinSpec Server (Spectrometer Control)

The WinSpec Server, located on the Windows 7, acts as the bridge between the WinRotor Client and the proprietary WinX32 application controlling the spectrometer. The server is implemented in **Visual Basic .NET** (VB.NET) and uses the native **WINX32Lib** to interact with the spectrometer hardware.

4.1 Remote Procedure Call (RPC) Interface: The `Module1.vb`

The communication protocol between the WinRotor Client and the WinSpec Server is a lightweight, custom **Remote Procedure Call (RPC)** system built over raw **TCP sockets**. This simple, fire-and-forget approach was chosen for its low overhead and compatibility with the older Windows 7 environment.

4.1.1 Server Initialization and Listener

The server uses the `TcpListener` class to bind to a specific IP address and port (`5000`) on the Windows 7 host and continuously waits for an incoming connection from the WinRotor Client.

```
1 Module Module1
2
3     Sub Main()
4
5         Const port = 5000
```

```
6         Dim serverIp As IPAddress = IPAddress.Parse
  ("192.168.1.1")
7         Dim listener As New TcpListener(serverIp, port)
8
9         Try
10            listener.Start()
11            Console.WriteLine("Server started 1")
12
13            ' Outer loop to wait for new clients
14            While True
15                Console.WriteLine("Waiting for a client
  connection...")
16                Using client as TcpClient = listener.
  AcceptTcpClient()
17                    Console.WriteLine("Client connected!")
18                    Using ns As NetworkStream = client.
  GetStream() 'get network stream
19' ... inner command loop ...
```

Listing 4.1: Module1.vb: Server Setup and Listener

4.1.2 Command Parsing and Dispatch

Once a client connects, the inner loop handles multiple sequential commands. The protocol uses a simple string format where the command and any optional parameters are delimited by `"::"`.

```
1' ... inside the inner loop ...
2
3                             Dim msg As String = Encoding.
  ASCII.GetString(buffer, 0, bytesRead).Trim(Chr(0))
4
5                             Dim parts() As String = msg.Split
  ({"::"}, StringSplitOptions.RemoveEmptyEntries)
6
7                             Dim command As String = If(parts.
  Length > 0, parts(0).Trim(), "")
8                             Dim optionalParameter As String =
   If(parts.Length > 1, parts(1).Trim(), "")
9
10                            Select Case command
11                                Case "SET_ACQUISITION_TIME"
12                                    Utilities.
  SetAcquisitionTime(ns, optionalParameter)
13
14                                Case "ACQUIRE_DATA"
15                                    Utilities.AcquireData(ns,
   optionalParameter)
```

```
' ... other command cases ...
                                End Select
' ...
```

Listing 4.2: Module1.vb: Command Handling and Dispatch

The `Select Case` statement is the core dispatcher, translating the string command received over the network (e.g., `"ACQUIRE_DATA"`) into a call to the corresponding static method in the `Utilities` class, passing the network stream and the parameter along.

4.2 WinX32 Command Translation: The `Utilities.vb`

The `Utilities.vb` class contains the essential methods that interface directly with the spectrometer via the imported `WINX32Lib`. This class hides the complexity of the proprietary API from the client and handles data conversion and error reporting.

4.2.1 Acquisition and Data Collection

The data collection operation is handled by the `AcquireData` method. This method demonstrates the use of the `ExpSetup` and `DocFile` objects from the `WINX32Lib` to manage the exposure sequence.

```
Public Shared Sub AcquireData(ns As NetworkStream,
    optionalParameter As String)
     Try
         'Initialise Exp and Doc objects
         Dim objExp As New ExpSetup()
         Dim objDoc As New DocFile()

         'Set file title to name plus position
         objExp.SetParam(EXP_CMD.EXP_DATFILENAME,
    optionalParameter)

         'Start acquisition, wait for it to finish, and save
    doc
         objExp.Start(CType(objDoc, IDocFile))
         objExp.WaitForExperiment()
         objDoc.Save()

         'Close doc
         objDoc.Close()

         'Send response
         SendResponse(ns, "DATA ACQUIRED")
```

```
20
21      Catch ex As Exception
22          'Catch exception
23          SendResponse(ns, "Data acquisition failed: ", ex)
24      End Try
25  End Sub
```

Listing 4.3: Utilities.vb: AcquireData Implementation

The sequence of operations is atomic:

1. Initialize `ExpSetup` (for experiment parameters) and `DocFile` (for the data file).
2. Set the filename using the client-provided parameter, which typically includes the current rotor position.
3. Call `objExp.Start()` to begin the exposure.
4. `objExp.WaitForExperiment()` blocks the thread until the spectrometer hardware completes the data collection.
5. `objDoc.Save()` and `objDoc.Close()` ensure the acquired data is persisted to disk before the server responds to the client.

4.2.2 Spectrograph Control and Conversion

Methods like `SetGratingPos` handle the physical movement of the spectrometer's internal grating. This often requires complex unit translation, as the client operates in user-friendly units (e.g., Raman shift in cm^{-1}), while the spectrometer API requires absolute position (in nm).

```
1  Public Shared Sub SetGratingPos(ns As NetworkStream,
       optionalParameter As String)
2      Dim wavenumber As Double 'hold position of grating to
       move at
3      If Double.TryParse(optionalParameter, wavenumber) Then
4          Try
5              'Convert from [rel / cm] to [nm] (assuming a 488
       nm laser)
6              Dim pos As Double = 1 / ((1 / 488) - (wavenumber
       / 1e7))
7
8              'Initialise spectrograph objects
9              Dim objSpecs As New SpectroObjMgr
10             'Select current spectrograph
11             Dim objSpec As SpectroObj = objSpecs.Current
```

```
12
13              objSpec.SetParam(SPT_CMD.SPT_NEW_POSITION, pos)
14              objSpec.Move()
15  ' ... Send response ...
16          Catch ex As Exception
17  ' ... Error handling ...
18          End Try
19  ' ...
```

Listing 4.4: Utilities.vb: SetGratingPos Unit Conversion

The calculation, $\frac{1}{\frac{1}{488} - \frac{\text{wavenumber}}{10^7}}$, converts the relative Raman wavenumber (in cm^{-1}) into the absolute wavelength position (in nm) required by the spectrometer hardware, based on the specific excitation laser wavelength of 488 nm. This conversion logic is essential for mapping coordinates accurately.

4.2.3 File System Operations

The server also exposes basic file system utilities to the client, such as `CheckDirectory` and `CreateDirectory`. These are necessary to validate the output path for the spectrometer data files before the mapping sequence begins.

4.2.4 Response Mechanism

All utility methods utilize a helper method, `SendResponse`, to manage communication back to the client. This centralized mechanism ensures that the client receives a consistent ASCII string response (`"DATA ACQUIRED"`, `"GRATING SET"`, etc.) or an error message, which is critical for the client's RPC synchronisation logic.

```
1  Private Shared Sub SendResponse(ns As NetworkStream,
       responseMessage As String, Optional ex As Exception =
       Nothing)
2       'Convert to bytes
3       Dim responseBytes As Byte() = Encoding.ASCII.GetBytes(
       responseMessage)
4       'Send response as bytes
5       ns.Write(responseBytes, 0, responseBytes.Length)
6
7       'Print out exception if passed through
8       if ex IsNot Nothing Then
9           Console.WriteLine($"An error occured: {ex}. \r\n SENT
       : {responseMessage}")
10      Else
11          Console.WriteLine(responseMessage)
```

```
12         End If
13     End Sub
```

Listing 4.5: Utilities.vb: Centralized Response Sending

Chapter 5

System Execution

The system comprises three separate components that must be launched in any order. The following subsections detail the startup procedure for each.

5.0.1 Launching the WinRotor Client

The WinRotor Client, which serves as the main user interface and system orchestrator, is launched via the command line.

1. Open a terminal.
2. Navigate to the application's directory.
3. Execute the application using the .NET runtime:

```
dotnet WinRotor.dll
```

5.0.2 Launching the WinSpec Server

The WinSpec Server, which controls the spectrometer, is launched as a standalone Windows executable.

1. Navigate to the following directory in your file explorer:
   `WinSpecServer_v1.0.4/WinSpecServer_v1.0.4/bin`
2. Run the executable file.

5.0.3 Launching the Kohzu Server

The Kohzu Server, which controls the physical rotors, is also launched as a standalone executable.

1. Navigate to the following directory in your file explorer: `KohzuServer/KohzuServer/bin/Debug/net8.0`
2. Run the executable file.

Chapter 6

Conclusion and Future Work

6.1 Project Summary

This project successfully designed and implemented a fully automated crystal surface mapping system. The system architecture, built around the WinRotor Client orchestrating two specialized servers, has proven effective. The adoption of the MVVM pattern in the client application resulted in a clean separation of concerns, making the UI development independent from the core business logic and communication protocols.

6.2 Future Work

While the current system meets its core objectives, there is still room for improvements:

- **Implementation of Step and Glue Functionality:** The installed version of the WinX32 application has a mismatched manual, making the implementation of "Step and Glue" mode a trial-and-error process. However, as the WinX32 library exposes the necessary methods to set the relevant parameters, this advanced acquisition mode remains a feasible and valuable future enhancement for complex mapping procedures.
- **Resolution of Keyboard Input Focus:** The application on the Raspberry Pi 5 requires a keyboard character to be entered in an external interface to 'wake up' the input focus before it can accept further keyboard inputs, after a button has been clicked. This platform-specific

issue does not occur on other computers and is likely related to the window management or input handling on the Raspberry Pi. Identifying and resolving this root cause would improve the user experience.