Table of Contents

About 2
Core Concepts 3
Addressing 6
Protocols 8
Helpers 13
Watchers / Recognition 14

About

IRIS also known as Intermediate Resource Integration System is a framework intended to provide a simple and easy way to create communication between embedded systems and PC.

It's main goal is to provide extensibility and flexibility to the user, allowing them to have as much freedom and control as possible.

Core Concepts

Device

Device is a physical (or virtual) device that is connected to the computer via USB, Bluetooth or other means. It is a high-abstraction access point to communicate with the device. Device uses ICommunicationInterface to process transactions between the device and the computer.

Example device can be a microcontroller that is connected to the computer via USB. The simple implementation is provided below:

```
/// <summary>
/// Base class for RUSTIC devices
/// </summary>
 public abstract class RUSTICDeviceBase(
     SerialPortDeviceAddress deviceAddress,
     SerialInterfaceSettings settings) : SerialDeviceBase(deviceAddress,
settings)
 {
     /// <summary>
     /// Sends SET message to device and returns the response <br/>
     /// E.g. PROPERTY to desired value
     /// </summary>
     /// <remarks>
     /// Uses ToString() method to convert <see cref="value"/> to string
     /// </remarks>
     public async Task SetProperty<TValueType>(string message,
TValueType value)
     {
         // Check if value is null, if so throw exception
         if (value == null) throw new
ArgumentNullException(nameof(value));

         // Send message with embedded value
         await
RawHardwareAccess.TransmitRawData(Encoding.ASCII.GetBytes(message));
```

```
        // Send request information
        await
RawHardwareAccess.TransmitRawData(Encoding.ASCII.GetBytes($"=
{value.ToString()}"));

        // Send new line
        await
RawHardwareAccess.TransmitRawData(Encoding.ASCII.GetBytes("\r\n"));
    }

    /// <summary>
    /// Sends GET message to device and returns the response <br/>
    /// </summary>
    public async Task<string> GetProperty(string propertyName)
    {
        // Send message with embedded value
        await
RawHardwareAccess.TransmitRawData(Encoding.ASCII.GetBytes(propertyName))
;

        // Send request information
        await
RawHardwareAccess.TransmitRawData(Encoding.ASCII.GetBytes("=?"));

        // Send new line
        await
RawHardwareAccess.TransmitRawData(Encoding.ASCII.GetBytes("\r\n"));

        // Wait for response
        byte[] response = await
RawHardwareAccess.ReadRawDataUntil(0x0A, CancellationToken.None);

        // Return decoded response
        return Encoding.ASCII.GetString(response);
```

```
        }
    }
```

Devices support `HardwareAccess` and `RawHardwareAccess` properties that allow to access the communication interface and send/receive data from the device. The `HardwareAccess` property is used to send and receive data using direct methods on specified communication interface (e.g. if used for Bluetooth) and `RawHardwareAccess` is used to send and receive raw data from the device using `IRawDataCommunicationInterface` interface overloads.

> If device does not support RawHardwareAccess property then accessing it will throw **NotSupportedException**. To override this behavior you can check if HardwareAccess is not null and implements IRawDataCommunicationInterface using is operator.

> As of IRIS 2.1 the **Transaction** subsystem has been removed, and thus it was made easier to access communication interface with the device. The user-created device now has direct access to the communication interface and can use it to send and receive data.
>
> It now is way more flexible and allows for more complex communication with the device without requiring too much overhead.

Communication Interfaces

Communication Interfaces are subsystems that are responsible for low-level communication with the device - it can for example read/write data to OS Serial Port. Role of the Communication Interface is to provide low-level communication methods that take transaction and execute it.

Example of a communication interface can be a SerialPortInterface that reads/writes data to the Serial Port. For that reference you can see the implementation provided within the project (`Communication/Serial/SerialPortInterface.cs`).

Addressing

IRIS contains basic addressing framework that is used to identify devices. Some simple addresses are provided by default, but the user can create custom addresses as well.

Default Addresses

SerialPortDeviceAddress(portName)

This address is used to identify devices connected to the serial port. It is created by providing the device COM (or ACM etc.) port name as a string.

This address is not capable of identifying device that is connected to port.

USBDeviceAddress(vid, pid)

This address is used to identify devices connected to the USB port. It is created by providing the device Vendor ID and Product ID as HEX string values.

This address is mostly used to identify different devices connected to USB ports of the computer. It is capable of identifying device that is connected to port and thus is recommended to be used when creating custom Watchers (Watchers / Recognition).

IPDeviceAddress(ipAddress)

This address is used to identify devices connected to the network. It is created by providing the device IP address as a IPAddress object.

Custom Addresses

To create custom address you need to implement `IDeviceAddress` interface. Objects implementing this interface should be readonly structs. It is also recommended to override equality operators to provide proper comparison between addresses.

Example of a custom address can be seen below:

```
public readonly struct RESTProtocolAddress(string restEndpoint) :
IDeviceAddress<string>
{
```

```
    public string Address { get; } = restEndpoint;
}
```

As you can see it implements `IDeviceAddress<string>` interface which is a simple interface that extends `IDeviceAddress` interface and provides a way to get the address as a string using build-in `Address` property which is always of type that is provided to the interface.

Protocols

Protocol is a data exchange format that defines the communication rules between the device and C# API. It is used as low level abstraction layer that is performing communication with the device.

Implemented Protocols

LINE

LINE protocol is a simple ASCII based protocol that is commonly used for debugging connections as it only reads and writes lines of text from/to the device. Each line is separated by new line with optional carriage return.

This protocol is good to be used with ESP32 UART logs.

Example commands can be seen below:

- `Hello, World!` - sends `Hello, World!` to the device

RUSTIC

RUSTIC protocol is a simple ASCII based protocol that is used to communicate with usage of commands. It is used to send and receive data from the device. Each command consists of property name, equals sign and value. Commands are separated by new line with optional carriage return. Question mark value is used to get property value.

Example commands can be seen below:

- `PROPERTY=VALUE` - sets property to desired value
  - E.g. `LED=1`
  - E.g. `LED=ON`
  - E.g. `LED=BLINK`
  - Good Response: `LED=OK`
  - Good Response: `LED=1`

- Error Response: `ERR=UNKNOWN_COMMAND`
- Error Response: `ERR=INVALID_VALUE`

- `PROPERTY=?` - gets property value
  - E.g. `LED=?`
  - Good Response: `LED=1`
  - Good Response: `LED=ON`
  - Good Response: `LED=BLINK`
  - Error Response: `ERR=UNKNOWN_COMMAND`
  - Error Response: `ERR=INVALID_VALUE`

Implementing Custom Protocol

To implement custom protocol you need to create new class that implements `IProtocol` interface. This interface contains two static methods that are used to receive and send data using provided communication interface.

Example of custom protocol can be seen below:

```
using System.Text;
using IRIS.Communication.Types;

namespace IRIS.Protocols.IRIS
{
    public abstract class LINE<TInterface> : IProtocol<TInterface,
string>
        where TInterface : IRawDataCommunicationInterface
    {
        /// <summary>
        /// Send a message to the device.
        /// </summary>
        /// <param name="communicationInterface">Communication interface
to use.</param>
        /// <param name="message">Message to send.</param>
```

```
        /// <param name="cancellationToken">Cancellation token.</param>
        public static async Task SendMessage(TInterface
communicationInterface,
            string message,
            CancellationToken cancellationToken = default)
            => await SendData(communicationInterface, message,
cancellationToken);

        /// <summary>
        /// Read a message from the device.
        /// </summary>
        /// <param name="communicationInterface">Communication interface
to use.</param>
        /// <param name="cancellationToken">Cancellation token.</param>
        /// <returns>Message from the device.</returns>
        public static async Task<string> ReadMessage(TInterface
communicationInterface,
            CancellationToken cancellationToken = default)
            => await ReceiveData(communicationInterface,
cancellationToken);

        /// <summary>
        /// Exchange messages with the device.
        /// </summary>
        /// <param name="communicationInterface">Communication interface
to use.</param>
        /// <param name="message">Message to send.</param>
        /// <param name="cancellationToken">Cancellation token.</param>
        /// <returns>Response from the device.</returns>
        public static async Task<string> ExchangeMessages(TInterface
communicationInterface,
            string message,
            CancellationToken cancellationToken = default)
        {
            await SendMessage(communicationInterface, message,
cancellationToken);
            return await ReadMessage(communicationInterface,
cancellationToken);
```

```
        }

        public static async Task SendData(TInterface
communicationInterface,
            string data,
            CancellationToken cancellationToken = default)
        {
            // Check if the communication interface is not null
            if (communicationInterface == null) throw new
ArgumentNullException(nameof(communicationInterface));

            // Create new string with end of line character
            string processedData = $"{data}\r\n";

            // Convert the string to a byte array
            byte[] dataBytes = Encoding.ASCII.GetBytes(processedData);

            // Send the data to the communication interface
            await communicationInterface.TransmitRawData(dataBytes);
        }

        public static async Task<string> ReceiveData(TInterface
communicationInterface,
            CancellationToken cancellationToken = default)
        {
            // Check if the communication interface is not null
            if (communicationInterface == null) throw new
ArgumentNullException(nameof(communicationInterface));

            // Receive the data from the communication interface until
the command end byte is received
            byte[] receivedData = await
communicationInterface.ReadRawDataUntil(0x0A, cancellationToken);

            // Decode the received data into a string
            string receivedString =
Encoding.ASCII.GetString(receivedData);
```

```
            // Return the received string
            return receivedString;
        }
    }
}
```

Type parameters are: interface used for current protocol (as devices may use different hardware interfaces, but both can be supported by the same protocol) and data type that is used to send and receive data.

> Recommended data types are **byte[]** and **string** as those are easy to support on most communication interfaces.

`SendData` and `ReceiveData` methods are used to send and receive data from the device and are a low-level abstraction layer that is used to communicate with the device.

`SendMessage`, `ReadMessage` and `ExchangeMessages` methods are proxy between high-level abstraction layer and low-level abstraction layer that are used to send and receive messages from the device and are more user-friendly. They also serve as example to provide additional data processing before and after sending or receiving data which can be spotted in `ExchangeMessages` method.

This abstraction can be seen in `RUSTIC` protocol implementation. For more information see protocol implementation in the library code.

Planned Protocols

BLE - Bluetooth Low Energy

MQTT - Message Queuing Telemetry Transport

REST - Representational State Transfer

SCPI - Standard Commands for Programmable Instruments

TCP/IP Stream

UDP Stream

Helpers

RequestTimeout

Used to provide a timeout CancellationToken for a request.

```
// Create new RequestTimeout
RequestTimeout timeout = new RequestTimeout(300); // 300ms timeout
```

It can be implicitly converted to `CancellationToken` and used in asynchronous methods. It also contains property `IsTimedOut` to check if the timeout has occurred which is useful to check status after Task has been cancelled.

Watchers / Recognition

Watchers are used to detect if device was connected or disconnected. They scan for all connected devices every interval (default 500ms) and compare it with the last scan. If there is a difference, it will trigger callbacks for connected and disconnected devices.

Implemented Watchers

SerialPortDeviceWatcher

Serial Port Device Watcher is used to detect serial port devices. It uses `SerialPort.GetPortNames()` to get all available serial ports and compares it with the last scan. If there is a difference, it will trigger callbacks for connected and disconnected devices.

Warning: as this watcher cannot detect type of device connected to the serial port, it is not recommended to use it for detecting specific devices. It is recommended to use it for detecting any serial port device (e.g. when your device is a generic USB to UART converter).

```
// Create new SerialPortDeviceWatcher
SerialPortDeviceWatcher watcher = new SerialPortDeviceWatcher();
```

WindowsUSBSerialDeviceWatcher

Windows USB Serial Device Watcher is used to detect USB Serial devices on Windows. It uses Windows Registry to get all available Serial devices and analyzes if they are USB Serial devices. If yes, it compares it with the last scan. If there is a difference, it will trigger callbacks for connected and disconnected devices.

This watcher is recommended to use for detecting specific USB Serial devices.

```
// Create new WindowsUSBSerialDeviceWatcher, this type of watcher
detects ANY USB Serial device
WindowsUSBSerialDeviceWatcher watcher = new
WindowsUSBSerialDeviceWatcher();

// Create new WindowsUSBSerialDeviceWatcher with specific VID and PID
```

```
WindowsUSBSerialDeviceWatcher watcher = new
WindowsUSBSerialDeviceWatcher("1234", "5678");
```

Using Watchers

To use Watchers, you need to subscribe to `OnDeviceAdded` and `OnDeviceRemoved` events.

```
// Create new SerialPortDeviceWatcher
SerialPortDeviceWatcher watcher = new SerialPortDeviceWatcher();
watcher.OnDeviceAdded += OnDeviceAddedCallback;
watcher.OnDeviceRemoved += OnDeviceRemovedCallback;
```

Then you need to start the watcher to begin scanning for devices. Watchers create own task for scanning devices and work in the background.

```
// Start the watcher
watcher.Start();

// Code goes here

// Stop the watcher
watcher.Stop();
```

To get all devices currently detected by Watcher you can use `AllSoftwareDevices` or `AllHardwareDevices` properties. Software devices are software identifiers of the device (e.g. COM port name), hardware devices are hardware identifiers of the device (e.g. VID and PID). This allows to have two different types of identifiers for the same device to handle those rare use cases when we want to scan for all devices and know their hardware identifiers.

Adding custom Watchers

To add custom watcher you need to extend `DeviceWatcherBase<TSelf, TSoftwareAddress, THardwareAddress>` or `DeviceWatcherBase<TSelf, TDeviceAddress>` class and implement `ScanForDevicesAsync` method.

```
public class CustomDeviceWatcher :
DeviceWatcherBase<CustomDeviceWatcher, string, string>
{
    protected override Task<(List<string>, List<string>)>
ScanForDevicesAsync(
            CancellationToken cancellationToken)
    {
        // Mock-up result
        List<string> softwareDevices = new List<string> { "COM1", "COM2"
};
        List<string> hardwareDevices = new List<string> { "1234:5678",
"8765:4321" };

        return Task.FromResult((softwareDevices, hardwareDevices));
    }
}
```

⚠ Watchers are costly feature as they constantly run scans. To reduce that price you can increase scan interval via ScanInterval property