

# Table of contents

About .....	2
Core Concepts .....	3
Addressing .....	8
Data Readers .....	10
Data Encoders .....	12
Transactions .....	15
Watchers / Recognition .....	18

# 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>
/// Represents an example of a device that changes or reads the value of
/// an LED
/// using RUSTIC protocol messages.
/// </summary>
public sealed class ExampleArduinoLEDChangingDevice(
    SerialPortDeviceAddress deviceAddress,
    SerialInterfaceSettings settings) : SerialDeviceBase(deviceAddress,
settings)
{
    /// <summary>
    /// Read Device Property asynchronously.
    /// </summary>
    public async Task<object?> ReadDevicePropertyAsync(CancellationTok
cancellationTok = default) =>
        await ReadDeviceProperty<MyDevicePropertyTransaction>
(cancellationTok);

    private async Task<object?>
ReadDevicePropertyAsync<TTransactionType>(CancellationTok
cancellationTok = default)
        where TTransactionType :
            IDataExchangeTransaction<PropertyTransaction,
TransactionRequestData, TransactionResponseData>
    {
```

```

        TransactionRequestData requestData = new
TransactionRequestData(PropertyNames.BuiltInLED);

        // Execute transaction logic here eg.
        // this example uses built-in transaction exchange method to
easily exchange data via current device.
        TransactionResponseData result = await TTransactionType
            .ExchangeAsync<ExampleArduinoLEDChangingDevice,
SerialPortInterface>(this, requestData,
                        cancellationToken);

        return result.Value;
    }
}

```

## 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`).

## Transactions

Transactions are the core of the communication system. They are used to exchange data between the device and the computer. Transaction is a simple object describing how the data should be exchanged. It should implement transaction-specific interface (`IReadTransaction`, `IWriteTransaction` or `IDataExchangeTransaction`).

By default, transaction is not supported by `SerialPortInterface` as it is a raw data communication interface and such interfaces require transaction to inform it if the data should be read by length or by delimiter. To support transactions you need to implement interface that will describe how the data should be exchanged. Two basic interfaces are

provided: `ITransactionReadUntilByte`, which reads data until specified byte is found and `ITransactionReadByLength` which reads data by specified length.

An example of a transaction can be seen below:

```
public struct LineReadTransaction :
    IReadTransaction<LineReadTransaction, LineTransactionData>,
    ILineTransaction
{
    public async Task<LineTransactionData> _ReadAsync<TDevice,
    TCommunicationInterface>(
        TDevice device, CancellationToken cancellationToken = default)
        where TDevice : DeviceBase<TCommunicationInterface> where
        TCommunicationInterface : ICommunicationInterface
    {
        // Get the communication interface and receive data
        ICommunicationInterface communicationInterface =
        device.GetCommunicationInterface();
        return await communicationInterface
            .ReceiveDataAsync<LineReadTransaction, LineTransactionData>
            (this, cancellationToken);
    }
}
```

This transaction implements `ILineTransaction` interface which is a simple interface that describes all transactions related to LINE protocol.

```
public interface ILineTransaction : ITransactionReadUntilByte,
    IWithEncoder<LineDataEncoder>
{
    byte ITransactionReadUntilByte.ExpectedByte => 0x0A;
}
```

## Data Encoders

As raw data is not practical to work with, Data Encoders are used to encode/decode data at edge of the transaction. Data Encoders are used to convert raw data to a more readable format and vice versa. Transaction can have multiple encoders that are used to

encode/decode data and implement them using `IWithEncoder<TEncoderType>` interface.

Example of a data encoder can be seen below:

```
/// <summary>
/// Represents an encoder for basic text communication with a line limit
/// of 128 characters.
/// </summary>
public struct LineDataEncoder : IRawDataEncoder<LineDataEncoder>
{
    public static byte[] EncodeData<TData>(TData inputData) where TData
: struct
    {
        // Check if TData is LineData
        if (inputData is not LineTransactionData lineData)
            throw new NotSupportedException("This feature is not
supported for LINE protocol");

        // Copy data from input to output
        return Encoding.ASCII.GetBytes(lineData.message);
    }

    public static bool DecodeData<TData>(byte[]? inputData, out TData
outputData) where TData : struct
    {
        outputData = new TData();

        // Check if input data is null
        if (inputData == null) return false;

        // Check if TData is LineReadResponse
        if (outputData is not LineTransactionData)
            throw new NotSupportedException("This feature is not
supported for LINE protocol");

        // We need to find first ASCII character to trim
        // Windows shitty handshake, because SerialPort class is a piece
```

```

of
    // garbage and doesn't provide any way to disable it.
    //
    // We could cover ASCII checking on the device side, but this is
just
    // an additional check to make sure we don't have any garbage
data.
    int firstValidIndex = -1;
    for (int i = 0; i < inputData.Length; i++)
    {
        // Check if character is valid (ASCII)
        if (inputData[i] >= 128) continue;
        firstValidIndex = i;
        break;
    }

    // Copy data from input to output
    string text = Encoding.ASCII.GetString(inputData,
firstValidIndex, inputData.Length - firstValidIndex);

    // Convert types
    LineTransactionData transactionData = new(text);
    outputData = (TData) Convert.ChangeType(transactionData,
typeof(TData));

    // Check if input data length is less than 128, if not, return
false
    // to indicate issue with decoding
    return true;
}
}

```

**i** It is recommended for data readers and encoders to be read-only.

# 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.

# Data Readers

Data Readers is subsystem intended to handle reading raw data (usually) in more or less structured form that does not require to modify the communication interface itself.

## Provided Data Readers

### LengthRawDataReader


LengthRawDataReader is intended to read desired amount of bytes from the communication interface and return them as a single byte array.

### UntilByteRawDataReader

UntilByteRawDataReader is intended to read data from the communication interface until specified byte is encountered. The byte is included in the result, which is returned as a byte array.

## Creating Custom Data Readers

To create custom Data Reader, you need to implement `IDataReader<TCommunicationInterface, TOutputDataType>` interface. The interface has method `PerformRead` that should use interface to read data and return it in desired form.

 Data readers should not Decode the data. This is reserved for DataEncoder subsystem.

An example below is a data reader that reads always 4 bytes from the communication interface. This is a simple example, but it should give you an idea how to create custom data readers, however the two provided data readers should be sufficient for most cases.

```
public readonly struct AlwaysFourBytesDataReader :
    IDataReader<IRawDataCommunicationInterface, byte[]>
{
    private const BYTES_TO_READ = 4;

    public Task<byte[]> PerformRead<TTransactionType>
        (IRawDataCommunicationInterface communicationInterface,
```

```
TTransactionType transaction,  
    CancellationToken cancellationToken = default) where  
TTransactionType : ICommunicationTransaction<TTransactionType>  
    {  
        return await communicationInterface.ReadRawData(BYTES_TO_READ,  
cancellationToken);  
    }  
}
```

# Data Encoders

Encoders are used to encode data from C# structures to byte arrays and vice versa. This subsystem is intended to make it easy to convert data into raw format without having to modify the communication interface itself or create new communication interface per each data formatting type.

It also is done separately from Transactions to reduce code overhead - multiple transactions would require encode methods to be implemented in each transaction, which would be most likely heavily redundant.

## Provided Data Encoders

IRIS provides several data encoders that should be sufficient for most cases. Those are mostly some basic encoders, however it's possible to implement custom encoders if needed.

### LineDataEncoder

LINE protocol is a simple message exchange protocol which is used to exchange text messages between devices. The protocol is simple and is intended to be used for handling debugging console-like interfaces.

The protocol is simple - it sends text messages terminated by 0x0A byte. The LineDataEncoder is intended to encode/decode data for LINE protocol.

It works only with `LineTransactionData` structure, which is a simple structure that contains a single string field `message`.

```
/// <summary>
/// Represents a read transaction for basic line communication.
/// Line can have up to 128 characters, other characters will be
/// ignored.
/// </summary>
public readonly struct LineTransactionData(string msg)
{
    /// <summary>
    /// Internal data storage.
    /// </summary>
    public readonly string message = msg;
```

```

    /// <summary>
    /// Gets the text stored in this instance.
    /// </summary>
    public override string ToString() => message;
}

```

Example communication with LINE protocol can be seen below:

```

> Hello, World!\r\n
< Hello, World!\r\n
> How are you?\r\n
< How are you?\r\n

```

## RusticDataEncoder

RUSTIC is a protocol used to exchange data between devices on get/set basis. The protocol is intended to work as a simple key-value store, where the key is a string and the value is a string as well (however it can be converted to any other type).

The RusticDataEncoder is intended to encode/decode data for RUSTIC protocol. It works only with RUSTIC structures, which can be seen in the respective folder.

Example communication with RUSTIC protocol can be seen below:

```

> KEY=32\r\n
< KEY=OK\r\n
> KEY=?\r\n
< KEY=32\r\n

```

## Creating Custom Data Encoders

To create custom Data Encoder, you need to implement `IRawDataEncoder<TSelf>` interface or for low-level encoders you can use `IDataEncoder<TSelf, TData>` interface. The interface requires implementation of two methods: `_EncodeData` and `_DecodeData` which are used to encode/decode data respectively.

`TData` should be encoded data object, for example byte array which is used in `IRawDataEncoder<TSelf>`.

```
/// <summary>
/// Represents communication interface that can encode and decode data
for
/// transactions from/to binary data. <br/>
/// </summary>
public interface IRawDataEncoder<TSelf> : IDataEncoder<TSelf, byte[]>
    where TSelf : IRawDataEncoder<TSelf>
{

}
```

# Transactions

Transaction is data exchange between device and computer. In most simple terms it can be described as "command", however it's not entirely true as transaction may consist of multiple commands or may not consist of any commands at all (be just pure data read from device when device is measurement unit and returns data in frames).

## Transaction Types

There are several core transaction types that are used in IRIS. If you are creating custom transaction you should use one of those types as a base for your transaction.

### **IReadTransaction<TSelf, TResponseDataType>**

IReadTransaction is a transaction that reads data from the device. It should implement `_ReadAsync` method that will read data from the device and return it as `TResponseDataType`.

### **IWriteTransaction<TSelf, TRequestDataType>**

IWriteTransaction is a transaction that writes data to the device. It should implement `_WriteAsync` method that will write data to the device.

### **IDataExchangeTransaction<TSelf, TRequestDataType, TResponseDataType>**

IDataExchangeTransaction is a transaction that reads and writes data to the device. It should implement `_ExchangeAsync` method that will read and write data to the device.

## Creating Custom Transactions

To create custom transaction you need to implement one of the transaction interfaces. Below is an example of a simple transaction that reads data from the device.

```
public struct SimpleReadTransaction :  
    IReadTransaction<SimpleReadTransaction, byte[]>,  
    ITransactionReadUntilByte  
{  
    public byte ExpectedByte => 0x0A;
```

```

    public async Task<byte[]> _ReadAsync<TDevice,
TCommunicationInterface>(
    TDevice device, CancellationToken cancellationToken = default)
    where TDevice : DeviceBase<TCommunicationInterface> where
TCommunicationInterface : ICommunicationInterface
    {
        // Get the communication interface and receive data
        ICommunicationInterface communicationInterface =
device.GetCommunicationInterface();
        return await
communicationInterface.ReceiveDataAsync<SimpleReadTransaction, byte[]>
(this, cancellationToken);
    }
}

```

## Using Transactions

To use transactions you can create an instance of the transaction and execute it. Below is an example of how to use the transaction.

```

MyDevice myDevice = //...;

// Create new SimpleReadTransaction
SimpleReadTransaction transaction = new SimpleReadTransaction();

// Execute the transaction
byte[] data = await transaction._ReadAsync<MyDevice,
SerialPortInterface>(myDevice);

```

Alternatively if your transaction does not have any specific logic you can use built-in static methods to execute the transaction.

```

MyDevice myDevice = //...;

// Execute the transaction

```



```
byte[] data = await SimpleReadTransaction.ReadAsync<MyDevice,  
SerialPortInterface>(myDevice);
```

## Data Readers

Data Readers is subsystem intended to handle reading raw data (usually) in more or less structured form that does not require to modify the communication interface itself.

To implement data reader within a transaction you need to implement `IWithDataReader<TDataReader>` as indicator that transaction uses data reader.

Alternatively if you use built-in `ITransactionReadByLength` or `ITransactionReadUntilByte` interfaces data reader is automatically used.

## Data Encoders

Data Encoders is subsystem intended to handle encoding/decoding data for transactions. It is used by raw data interfaces to handle encoding/decoding data if user requests any structure other than byte array.

To implement data encoder within a transaction you need to implement `IWithEncoder<TEncoder>` as indicator that transaction uses data encoder.

# 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 connected to 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