IP S7 LINK SDK
SIMATIC connected Development

Development Guide

Tested? You want it?

License Model Prices Quotation Order Now

C# ▼ |VB|

The App Frame

A S7 App

1. Add reference to the **IPS7LnkNet.Advanced** Namespace:
```
using IPS7Lnk.Advanced;
```
2. Create an instance of the SimaticDevice class with the address of the controller:
```
var device = new SimaticDevice("192.168.0.80");
```
3. Build and open a connection to the controller:
```
var connection = device.CreateConnection();
connection.Open();
```
4. Your code to interact with the controller:
```
// Your code to interact with the controller.
```
5. Close the connection before closing the application:
```
connection.Close();
```
6. Using the using block this looks as follows:
```
using (var connection = device.CreateConnection()) {
    connection.Open();
    // Your code to interact with the controller.
}
```

Process Data

Addressing

The following types are used: PlcAddress, PlcRawType, PlcOperand and PlcOperandType.

Regardless of the type of access, it must be described which data area you want to access. The **PlcAddress** of the data area determines where the data to be accessed is located in the controller. It is possible to construct the **PlcAddress** as a simple string (according to know addressing in the PLC) or via the API of the SDK. The paired examples below show how a **PlcAddress** can be created to address the same data in different ways.

- Fifth data word in the tenth data block:
```
var a = PlcAddress.Parse("DB10.DBW 5");
var b = new PlcAddress(PlcOperand.DataBlock(10), PlcRawType.Word, 12);
```
- First bit in the fifth byte of the tenth data block:

```
var a = PlcAddress.Parse("DB10.DBX 5.1");
var b = new PlcAddress(PlcOperand.DataBlock(10), PlcRawType.Bit, 5, 1);
```

- Eighth byte in the input:

```
var a = PlcAddress.Parse("E8");
var b = new PlcAddress(PlcOperand.Input, PlcRawType.Byte, 8);
```

- Third bit in the ninth byte of the flag:

```
var a = PlcAddress.Parse("M9.3");
var b = new PlcAddress(PlcOperand.Flag, PlcRawType.Bit, 9, 3);
```

To define the PlcAddress using the Parse-method or one of the constructors, the addressing can also be done using the implicant cast operator of the class. Here the Siemens or IEC-specific addressing is also supported.

```
PlcAddress address = "DB3.DBB 10";
PlcAddress address = "MB 5";
PlcAddress address = "AW 2";
PlcAddress address = "QW 2";
```

The character string of the PLC data address on which a PlcAddress is based on can be called up using the ToString method of the PlcAddress class. It is possible to specify the desired operand standard (Siemens or IEC).

If no special standard is specified, the Siemens standard is always used.

```
PlcAddress address = "DB3.DBB 10";
Console.WriteLine(address.ToString());                              // output: DB3.DBB 10

PlcAddress address = "MB 5";
Console.WriteLine(address.ToString());                              // output: MB 5

PlcAddress address = "AW 2";
Console.WriteLine(address.ToString());                              // output: AW 2
Console.WriteLine(address.ToString(PlcOperandStandard.IEC));       // output: QW 2
Console.WriteLine(address.ToString(PlcOperandStandard.Siemens));   // output: AW 2

PlcAddress address = "QW 2";
Console.WriteLine(address.ToString()); // output: AW 2
Console.WriteLine(address.ToString(PlcOperandStandard.IEC));       // output: QW 2
Console.WriteLine(address.ToString(PlcOperandStandard.Siemens));   // output: AW 2
```

The procedure shown here will be used in all other code sections for the sake of simplicity. Depending on the application, the desired addressing form can be used.

Reading Values

The following types are used: SimaticDevice, PlcDeviceConnection and PlcAddress.

One of the data type-specific read methods of the

PlcDeviceConnection

1. Read a sequence of three `Int32`-values:

```
int[] values = connection.ReadInt32("DB1.DBD 1", 3);
```

Depending on the format of the PlcAddress and the PLC data type to be read, many bytes are read accordingly. The read PLC data type is then converted into the form of the desired PC data type. </panel>

Writing Values

The following types are used: SimaticDevice, PlcDeviceConnection and PlcAddress.

To write one or more values (= an array) into a specific data area, coded in the format of the corresponding PLC data type, one of the data type-specific write methods of the

PlcDeviceConnection

is used. If an array is passed instead of a single value during writing, all values in the array are written in the same order from the specified address.

- Write a single `Int32`-value:

```
connection.WriteInt32("DB1.DBD 1", 123);
```

- Write a sequence of three `Int32`-values:

```
connection.WriteInt32("DB1.DBD 1", 123, 456, 789);
```

Values as PLC variables

The following types are used: SimaticDevice, PlcDeviceConnection, PlcAddress, PlcInt32, PlcInt32Array, PlcBoolean, PlcBooleanArray and PlcString.

Often a certain data area has to be read or written several times and at different points in the program flow. If the data in which the value is stored in die PLC then changes, the entire source code must be searched for the old address and adapted accordingly to the new one. In addition, it is not always clear what value is hidden behind a PLC address of a certain data area. With the help of PlcValue objects, PLC variables can be defined one in the source code and uniquely addressed in any time.

- Define and read a single `Int32`-variable:

```
var speedVariable = new PlcInt32("DB1.DBD 1");
var speed = connection.ReadValue(speedVariable);
```

- Define and read an `Int32`-array variable:

```
var coordinatesVariable = new PlcInt32Array("DB1.DBD 1", 3);
var coordinates = connection.ReadValue(coordinatesVariable);
```

- Define and write a single `Int32`-variable:

```
var speedVariable = new PlcInt32("DB1.DBD 1", 123);
connection.WriteValue(speedVariable);

speedVariable.Value = 1200;
connection.WriteValue(speedVariable);
```

- Define and write an `Int32`-array variable:

```
var coordinatesVariable = new PlcInt32Array("DB1.DBD 1", 123, 456, 789);
connection.WriteValue(coordinatesVariable);

coordinatesVariable.Value[] = 10;
coordinatesVariable.Value[1] = 20;
coordinatesVariable.Value[2] = 30;
connection.WriteValue(coordinatesVariable);
```

Depending on the format of the PlcAddress and the PLC data type to be written, many bytes are written accordingly. The PC data type to be written is first converted into the desired PLC data type.

Since the consistency of the read or written data is particularly important, it is also possible to read or write several PLC variables at the same time. At this point, the mix of different PlcValue instances works exactly as if you were only using instances of a certain PlcValue type. Assuming, for example, the following process image, the corresponding read / write accesses look as follows.

```
var speedVariable = new PlcInt32("DB1.DBD 1");
var coordinatesVariable = new PlcInt32Array("DB2.DBD 1", 3);
var jobIsActiveVariable = new PlcBoolean("DB3.DBX 1.0");
var toolSetupVariable = new PlcBooleanArray("DB4.DBX 1.0", 5);
var operatorNameVariable = new PlcString("DB5.DBB 1", 32);
```

Reading the process image could look like this:

```
connection.ReadValues(
        speedVariable,
        coordinatesVariable,
        jobIsActiveVariable,
        toolSetupVariable,
        operatorNameVariable);

Console.WriteLine($"Speed: {speedVariable.Value}");
Console.WriteLine($"Coordinates: {string.Join(",", coordinatesVariable.Value)}");
Console.WriteLine($"Job is Active: {jobIsActiveVariable.Value}");
Console.WriteLine($"Tool Setup: {string.Join(",", toolSetupVariable.Value)}");
Console.WriteLine($"Operator Name: {operatorNameVariable.Value}");
```

The process image could be written as follows:

```
speedVariable.Value += 100;
coordinatesVariable.Value = new[] { 10, 20, 30 };
jobIsActiveVariable.Value = true;
toolSetupVariable.Value = new[] { false, true, true };
operatorNameVariable.Value = Environment.UserName;

connection.WriteValues(
        speedVariable,
        coordinatesVariable,
        jobIsActiveVariable,
        toolSetupVariable,
        operatorNameVariable);
```

The value of the value property of the PLC variable can be set using the constructor of the PlcValue class and changed using the value property. The value that is passed in the constructor can then be understood as a initial value. If the value property is changed, the new value is not automatically transferred to the controller - it must be carried out via an explicit WriteValue call.

Structured data

The following sections **assume that** there is a process image in the controller that corresponds to the **(fictitious) data type "MillJob"**. The structure of the data type is defined as follows:

```
MillJob
  .Input : int
  .Number : string
  .Output : int
  .RotationSpeed : int
  .ToolDiameter : float
```

Define Structure

The following types are used: PlcObject, PlcMemberAttribute and PlcMember.

The PlcObject class is derived to define the structure for access to structured data. In the derivation, corresponding fields and / or properties are then defined for all data areas to be addressed. The PlcMemberAttribute must then be set on each member that represents a value in the controller. The associated data area is then addressed via the attribute. If it is an array or a string value, the attribute also specifies the number of elements or the length of the string.

For the previously fictionally defined structure for example, the following implementation results as a PlcObjekt:

```
public class MillJob : PlcObject
{
    [PlcMember("DB1.DBD 20")]
    public int Input;

    [PlcMember("DB1.DBB 1", Length = 16)]
    public string Number;

    [PlcMember("DB1.DBD 25")]
    public int Output;

    [PlcMember("DB1.DBD 30")]
    public int RotationSpeed;

    [PlcMember("DB1.DBW 40")]
    public float ToolDiameter;
}
```

By combining the addressing of the process data via fields and properties, non-POCOs can be implemented:

```
public class MachineData : PlcObject
{
    [PlcMember("DB1.DBX 100.0", Length = 7)]
    private bool[] toolConfigurations;

    public MachineData()
        : base()
    {
    }

    [PlcMember("DB1.DBB 120")]
    public DateTime EstimatedFinishDate { get; set; }

    [PlcMember("DB1.DBB 1", Length = 16)]
    public string JobNumber { get; set; }

    [PlcMember("DB1.DBD 100")]
    public int Speed { get; set; }

    [PlcMember("DB1.DBW 50")]
    public float Temperature { get; set; }

    [PlcMember("DB1.DBX 100.0")]
    public bool UseCuttingTool { get; set; }

    public bool IsToolConfigured(int toolIndex)
    {
        return this.toolConfigurations[toolIndex];
    }
}
```

The information required for the definition can either be obtained from the controller manual or from the responsible PLC developer.

Read Structure

The following types are used: SimaticDevice, PlcDeviceConnection and PlcObject.

The ReadObject method of the connection is used to read structured data via a previously defined structure:

```
MillJob job = connection.ReadObject<MillJob>();

Console.WriteLine("Input: {0}", job.Input);
Console.WriteLine("Number: {0}", job.Number);
Console.WriteLine("Output: {0}", job.Output);
Console.WriteLine("Rotation Speed: {0}", job.RotationSpeed);
Console.WriteLine("Total Diameter: {0}", job.ToolDiameter);
```

The fields / properties defined in the structure are read 1:1 in the order in which they were defined.

Write Structure

The following types are used: SimaticDevice, PlcDeviceConnection and PlcObject.

The WriteObject method of the connection is used to write structured data via a previously defined structure:

```
MillJob job = new MillJob();
job.Input = 1;
job.Number = "MJ:100012";
job.RotationSpeed = 3500;
job.ToolDiameter = 12.8f;
job.Output = 3;

connection.WriteObject(job);
```

The values contained in the structure at the time of the call are written 1:1 in the order in which the fields / properties were defined.

Notifications

Connection(s) status

The following types are used: PlcDeviceConnection, PlcDeviceConnectionState, PlcStatus and PlcNotifications.

The management of the connections to the individual controllers presupposes that the status of each connection is known at all times. To monitor the status of a connection, the events Opening, Opened, Connecting, Connected, Disconnected, Closing, Closed and Faulted can be handled. In summary, the StateChanged event can also be handled for all of these events.

```
connection.StateChanged += HandleConnectionStateChanged;

private static void HandleConnectionStateChanged(
        object sender,
        PlcDeviceConnectionStateChangedEventArgs e)
{
    if (connection.State == PlcDeviceConnectionState.Connected) {
        // ...
    }
}
```

Further information about the status of the connection can be queried via the status property of the connection. Part of this information is primarily the result of the last operation performed (such as the last addressed PLC data type). Here the change in status can be also treated accordingly.

```
PlcStatus status = connection.Status;
status.Changed += HandleConnectionStatusChanged;
```

In the appropriate EventHandler, the status information can then be used for user-defined logging or extended evaluation of the connection status.

```
private static void HandleConnectionStatusChanged(object sender, EventArgs e)
{
    var status = (PlcStatus)sender;

    Console.WriteLine(status.TimeStamp);
    Console.WriteLine("- Code=[{0}]", status.Code);
    Console.WriteLine("- Text=[{0}]", status.Text);
    Console.WriteLine("- Exception=[{0}]", status.Exception?.Message ?? "<none>");
}
```

In addition to the instance-related events, all connections can also be monitored globally. Furthermore to the already mentioned events, this also includes a ConnectionCreated event which enables the dynamic addition of further event handlers.

```
PlcNotifications.ConnectionCreated += HandleNotificationsConnectionCreated;

...

private static void HandleNotificationsConnectionCreated(
        object sender,
        PlcNotifications.PlcDeviceConnectionEventArgs e)
{
    var connection = e.Connection;

    if (...) {
        // ...
    }
}
```

Access Status

The following types are used: PlcDeviceConnection, PlcNotifications and PlcStatus.

Access to a certain data area can fail for several reasons. While an area either no longer exists, the start address may also be valid, but the end of data no longer corresponds to the expected end. Other situations like that are valid data, since no initial value may have been set. The cause of problems when accessing the data areas can be quickly found out using appropriate status codes.

If PlcValue instances are used for access, the status can be checked for each PLC variable.

```
var speedVariable = new PlcInt32("DB1.DBD 1");
connection.ReadValue(speedVariable);

if (speedVariable.Status.Code == PlcStatusCode.NoError) {
    // ...
}
```

In addition, the status can be checked down to the level of the PLC data type. Therefor the GetStatus-method is used. There is also the possibility to take the evaluation of an operation into your own hands. This makes it possible to evaluate a failed operation, which can also lead to an exception, yourself and to declare it as “OK”. This is useful, for example, when developing a standard application for the controls of a system, but these do not have the same data areas, which is why a non-addressable data area can also be ignored.

For user-defined evaluation, a user-defined method must be entered in the static PlcNotifications class.

```
PlcNotifications.EvaluateStatus = EvaluateStatus;

...

private static bool EvaluateStatus(IPlcStatusProvider provider)
{
    if (provider is PlcDeviceConnection connection) {
        // ...
    }
    else if (provider is IPlcValue value) {
        // ...
    }

    // Fallback to "everything is okay".
    return true;
}
```

Device Data

The following types are used: SimaticDevice, IPlcDeviceInfo, PlcBlockInfo and PlcOperand.

Occasionally the device information of the controller is crucial to find out whether you are working with the right controller and where it is exactly. An instance of the IPlcDeviceInfo interface can be called up using the GetInfo method of the PlcDevice class to call up the device data availabe for this. The instance then contains all the device data provided.

```
var device = new SimaticDevice("192.168.0.80");
var deviceInfo = device.GetInfo();

if (deviceInfo.HasName)
    Console.WriteLine($"Name:         {deviceInfo.Name}");

if (deviceInfo.HasLocation)
    Console.WriteLine($"Location:     {deviceInfo.Location}");

if (deviceInfo.HasModuleName)
    Console.WriteLine($"Modul Name:   {deviceInfo.ModuleName}");

if (deviceInfo.HasModuleType)
    Console.WriteLine($"Modul Type:   {deviceInfo.ModuleType}");

if (deviceInfo.HasModuleSerial)
    Console.WriteLine($"Modul Serial: {deviceInfo.ModuleSerial}");

if (deviceInfo.HasPlantId)
    Console.WriteLine($"Plant ID:     {deviceInfo.PlantId}");

if (deviceInfo.HasTime)
    Console.WriteLine($"Time:         {deviceInfo.Time}");

if (deviceInfo.HasCopyright)
    Console.WriteLine($"Copyright:    {deviceInfo.Copyright}");
```

It should be noted that not every controller provides all of the device information shown here. To check whether information is available, one of the has-properties can be used.

Settings

General

The following types are used: PlcDevice and PlcDeviceConnection.

The class PlcDevice and PlcDeviceConnection are used as base classes of SimaticDevice and SimaticDeviceConnection. As a result, the SimaticDevice class inherits, for example, the EndPoint property for configuring the endpoint via which the controller is to be connected. The PlcDeviceConnection also provides various properties. Part of there are properties for controlling the timeout behavior and termination detection.

Further specific settings depend on the provider used (= PlcDevice derivative) and can be configured accordingly after a cast to the PlcDeviceConnections specific types.

Device Types

The following types are used: SimaticDevice, SimaticDeviceType and SimaticChannelType.

The framework generally tries to automatically determine the device types and the appropriate channels type for the controller. Depending on the type of control used and the structure of the network, it may be necessary to define the device types or the channel type manually.

```
var device = new SimaticDevice("192.168.0.80");
device.Type = SimaticDeviceType.S71200;
device.ChannelType = SiemensChannelType.ProgrammerDevice;
```

Endpoints

The following types are used: SimaticDevice, IPDeviceEndPoint and SimaticDeviceType.

Generally, either the DNS-name or the IP address of the controller is sufficient for access. Depending on the type of control and setup of the network, the rack and slot number of the control may also have to be specified.

```
var device = new SimaticDevice();
device.Type = SimaticDeviceType.S71500;
device.ChannelType = SiemensChannelType.OperationPanel;
device.EndPoint = new IPDeviceEndPoint("192.168.0.80", rack: , slot: 2);
```

Licensing

The IP S7 LINK SDK comes with an **evaluation license which can be used unlimited for each application run for 30 minutes**. If this restriction limits your evaluation options, you can request another evaluation license from us.

Just ask our support (via support@traeger.de) or let us consult you directly and clarify open questions with our developers!

After receiving your personalized **license key for IP S7 LINK development** it has to be committed to

the framework. Hereto insert the following code line into your application **before** accessing the **SimaticDeviceConnection class** for the first time. Replace *<insert your license code here>* with the license key you received from us.

```
IPS7LnkNet.Advanced.Licenser.LicenseKey = "<insert your license code here>";
```

Additionally you receive information about the license currently used by the framework via the *LicenseInfo* property of the **IPS7Lnk.Advanced.Licenser class**. This works as follows:

```
ILicenseInfo license = IPS7LnkNet.Advanced.Licenser.LicenseInfo;

if (license.IsExpired)
    Console.WriteLine("The IP S7 LINK SDK license is expired!");
```

In the course of development/evaluation, it is mostly irrelevant whether the test license or the license already purchased is being used. However, as soon as the application goes into productive use, it is annoying if the application stops working during execution due to an invalid license. For this reason, we recommend implementing the following code snippet in the application and at least executing it when the application is started:

```
#if DEBUG
    IPS7LnkNet.Advanced.Licenser.FailIfUnlicensed();
#else
    IPS7LnkNet.Advanced.Licenser.ThrowIfUnlicensed();
#endif
```

You can receive further information about licensing, purchase or other questions directly on our product page at: www.traeger.de.

Table of Contents