IP S7 LINK SDK
SIMATIC connected Development

# IP S7 LINK SDK für .NET

## Getestet? Du willst es?

Lizenzmodell Preise Angebot Jetzt bestellen

Book - Das gesamte Handbuch als eBook

# Development Guides

Development Guide Häufige Fragen

# Download

Das IP S7 LINK .NET SDK kommt mit einer **Testlizenz die je Anwendungsstart 30 Minuten uneingeschränkt zur Entwicklung** verwendet werden kann. Sollte diese Einschränkung ihre Evaluationsmöglichkeiten einschränken, besteht die Möglichkeit eine **alternative Testlizenz** bei uns **kostenlos** zu beantragen. Fragen Sie einfach unseren Support (via support@traeger.de) oder lassen Sie sich gleich direkt von uns beraten und offene Fragen durch unsere Entwickler klären!

**Update Informationen**

- ab v2.2 wird ein neuer Lizenzschlüssel benötigt!
- ab v2.2 wurden diverse Klassen mit dem Prefix „Siemens“ in „Simatic“ umbenannt!

**IP S7 LINK .NET SDK** – Evaluationspaket[1)]
Download ZIP Archiv von IPS7LnkNet.Advanced (Version: 2.4.0.0 – 2022-10-11)
Download NuGet Paket von IPS7LnkNet.Advanced (Version: 2.4.0.0 – 2022-10-11)

S7 Watch (Version: 2.4.0.0 – 2022-10-11)
Ein kostenloser und einfacher, aber professioneller S7 Daten Monitor für den Datenzugriff auf S7 Steuerungen.

Versionshistorie - Die Liste der Verbesserungen pro Version

# Preview Download

Derzeit sind keine Preview-Versionen verfügbar. Falls Sie an einer Funktion interessiert sind, die das SDK in der neuesten Version möglicherweise nicht erfüllt: **Zögern Sie nicht und kontaktieren Sie uns einfach über support@traeger.de!**

# IP S7 LINK

Development Guide

## Beispiel Code: Überwachung der Betriebstemperatur

- C#
- VB

```
namespace App
{
    using System;
    using System.Threading;

    using IPS7Lnk.Advanced;

    public class Program
    {
        public static void Main()
        {
            var device = new SimaticDevice("192.168.0.80");

            using (var connection = device.CreateConnection()) {
                connection.Open();

                while (true) {
                    var temperature = connection.ReadDouble("DB10.DBD 20");
                    Console.WriteLine($"Current Temperature is {0} °C", temperature);

                    Thread.Sleep(1000);
                }
            }
        }
    }
}
```

```
Imports System
Imports System.Threading

Imports IPS7Lnk.Advanced

Namespace App
    Public Class Program
        Public Shared Sub Main()
            Dim device = New SimaticDevice("192.168.0.80")

            Using connection = device.CreateConnection()
                connection.Open()

                While True
                    Dim temperature = connection.ReadDouble("DB10.DBD 20")
                    Console.WriteLine("Current Temperature is {0} °C", temperature)

                    Thread.Sleep(1000)
                End While
            End Using
        End Sub
    End Class
End Namespace
```

[1] Mit Ihrem „License Code“ wird das Paket zur produktiven Vollversion.

[2] Nicht für den produktiven Einsatz empfohlen.

IP S7 LINK SDK
SIMATIC connected Development

# Development Einführung

## Getestet? Du willst es?

Lizenzmodell Preise Angebot Jetzt bestellen

# Adressierung

## Operanden

Das Framework unterstützt die Adressierung von Eingängen (engl. Input), Peripherieeingängen (engl. Periphery Input), Ausgängen (engl. Output), Peripherieausgängen (engl. Periphery Output), Merkern (engl. Flag), Datenbausteinen (engl. Data Block), Zählern (engl. Counter) und Zeitgebern (engl. Timer). Auf welchen der eben genannten Komponenten zugegriffen werden soll wird über den Operandenanteil (engl. Operand) der SPS Datenadresse (kurz SPS Adresse) beschrieben. Die folgende Tabelle beschreibt die dafür gültigen Kürzel die als Operand bezeichnet werden:

| Name | Abkürzung (Siemens, DE) | Abkürzung (IEC) |
|---|---|---|
| Eingang | E | I |
| Peripherieeingang | PE | PI |
| Ausgang | A | Q |
| Peripherieausgang | PA | PQ |
| Merker | M | M |
| Datenbaustein | DB | DB |
| Zähler | Z | C |
| Zeitgeber | T | T |

Auf den Operanden folgt, im Falle eines Datenbausteins, die Nummer des Datenbausteins. Darauf folgt ein Punkt und wiederholt das Kürzel für den Operanden für Datenbausteine.

Die Backus-Naur-Form eines Operanden ist dabei wie folgt festgelegt:

- `<Datenbaustein-Nummer> ::= 0-65535`
- `<Siemens-Operand> ::= E | PE | A | PA | M | DB<Datenbaustein-Nummer>.DB | Z | T`
- `<IEC-Operand> ::= I | PI | Q | PQ | M | DB<Datenbaustein-Nummer>.DB | C | T`

## Offsets

Der eigentliche Adressierungsteil der SPS Adresse (engl. PLC Address) kommt nach dem Operanden und unterteilt sich in den Datentypbezeichner (im Rohdatenformat) und dem Offset für das zu adressierende Byte sowie gegebenfalls das zu adressierende Bit (getrennt durch einen Punkt). Die dabei unterstützten Datentypbezeichner und ihre gültigen Kürzel (für den Rohdatentypen) beschreibt die folgende Tabelle:

| Datentyp | Kürzel | Bits | Wertebereich | Beschreibung | Array |
|---|---|---|---|---|---|
| BOOL | X | 1 | 0 bis 1 | ein einzelnes Bit zur Darstellung von wahr (1) oder falsch (0) | ja |
| BYTE | B | 8 | 0 bis 255 | Vorzeichenlose 8-Bit Ganzzahl | ja |
| WORD | W | 16 | 0 bis 65.535 | Vorzeichenlose 16-Bit Ganzzahl (Word) | ja |
| DWORD | D | 32 | 0 bis $2^{32}$ -1 | Vorzeichenlose 32-Bit Ganzzahl (Double Word) | ja |
| CHAR | B | 8 | A+00 bis A+ff | Vorzeichenloses 8-Bit Zeichen im ASCII-Code | ja |
| INT | W | 16 | -32.768 bis 32.767 | Vorzeichenbehaftete 16-Bit Ganzzahl | ja |
| DINT | D | 32 | $-2^{31}$ bis $2^{31}$-1 | Vorzeichenbehaftete 32-Bit Ganzzahl | ja |
| REAL | D | 32 | +-1.5e-45 bis +-3.4e38 | 32-Bit Gleitkommazahl mit einfacher Genauigkeit (gemäß IEEE754) | ja |

| Datentyp | Kürzel | Bits | Wertebereich | Beschreibung | Array |
|---|---|---|---|---|---|
| S5TIME | W | 16 | 00.00:00:00.100 bis 00.02:46:30.000 | binär codierte Dezimalzahl (BCD), die eine Zeitspanne repräsentiert | |
| TIME | D | 32 | 00.00:00:00.000 bis 24.20:31:23.647 | Vorzeichenbehaftete 16-Bit Ganzzahl, die eine Zeitspanne in Millisekunden repräsentiert | |
| TIME_OF_DAY | D | 32 | 00.00:00:00.000 bis 00.23:59:59.999 | Vorzeichenlose 16-Bit Ganzzahl, die eine Zeitspanne in Millisekunden repräsentiert | |
| DATE | W | 16 | 01.01.1990 bis 31.12.2168 | Vorzeichenlose 16-Bit Ganzzahl, die ein Datum in Tagen repräsentiert | |
| DATE_AND_TIME | D | 64 | 00:00:00.000 01.01.1990 bis 23:59:59.999 31.12.2089 | binär codierte Dezimalzahl (BCD), die ein Datum mit Uhrzeit repräsentiert | |
| S7STRING | B | - | A+00 bis A+ff | eine im ASCII-Code codierte Zeichenkette aus maximal 254 Bytes | |

# Rohdatentypen

Dabei stehen die Kürzel für:

- X = Bit, für die Adressierung eines einzelnen Bits *(bei der Adressierung eines Bits ist das X optional)*
- B = Byte, für die Adressierung von 8 Bits
- W = Word (= *Wort*), für die Adressierung von 16 Bits
- D = Double Word (= *Doppelwort*, kurz DWord), für die Adressierung von 32 Bits

Werden Operandenteil und die eben genannten Datentypbezeichner zusammengesetzt, ergibt sich die folgende Backus-Naur-Form:
`<Operand mit Datentypbezeichner> ::= (<Siemens-Operand> | <IEC-Operand>) [ X ] | B | W | D`

# Beispiele

Fügt man der PLC Address die Offset-Informationen hinzu, ist die Adresse vollständig und kann wie folgt aussehen:

| Beispiel | Datentyp | PLC Address (Siemens, DE) | PLC Address (IEC) |
|---|---|---|---|
| Erstes Bit im ersten Byte des Eingangs | BOOL | E 1.0 | I 1.0 |
| Siebtes Bit im ersten Byte des Ausgangs | BOOL | A 1.7 | Q 1.7 |
| Erstes Bit im zehnten Byte des Merkers | BOOL | M 10.1 | M 10.1 |
| Nulltes Bit im ersten Byte im Datenbaustein mit der Nummer 1 | BOOL | DB1.DBX 1.0 | DB1.DBX 1.0 |
| Erstes Bytes im Eingang | BYTE | EB 1 | IB 1 |
| Zehntes Byte im Ausgang | BYTE | AB 10 | QB 10 |
| Hundertes Byte im Merker | BYTE | MB 100 | MB 100 |
| Nulltes Byte im Peripherieeingang | BYTE | PEB 0 | PIB 0 |
| Erstes Byte im Peripherieausgang | BYTE | PAB 1 | PQB 1 |
| Erstes Byte im Datenbaustein mit der Nummer 2 | BYTE | DB2.DBB 1 | DB2.DBB 1 |
| Nulltes Double Word im Peripherieeingang | DWORD | PED 0 | PID 0 |

Mit der Backus-Naur-Form für den Byte und Bit Offset ergibt sich die vollständige PLC Address wie folgt:

- `<Byte Offset> ::= 0-65535`

- `<Bit Offset> ::= 0-7`
- `<PLC Address> ::= <Operand mit Datentypbezeichner> <Byte Offset> [. <Bit Offset> ]`

# Die Verbindung zur SPS

## Open

Das passiert beim Aufruf von 'Open':

1. Es wird geprüft, ob ein **Endpunkt festgelegt** wurde (EndPoint Eigenschaft).
2. Die Verbindung ändert ihren Status (**PlcDeviceConnection.State** Eigenschaft) auf den Wert ***Opening***.
3. Die Verbindung **prüft ihre Konfiguration** auf Gültigkeit und Schlüssigkeit.
4. Anschließend bereitet sich die Verbindung auf die erste Kommunikation mit der SPS vor.
5. Die Verbindung ändert ihren Status (**PlcDeviceConnection.State** Eigenschaft) auf den Wert ***Opened***.

## Connect

Das passiert beim Aufruf von 'Connect':

1. Es wird geprüft, ob die Verbindung bereits geöffent wurde (State Eigenschaft).
2. Die Verbindung ändert ihren Status (**PlcDeviceConnection.State** Eigenschaft) auf den Wert ***Connecting***.
3. Die Verbindung stellt eine für die SPS physikalische Verbindung her und belegt ab sofort eine der maximal möglichen Verbindungen zur SPS.
4. Anschließend werden noch Vorkehrungen für die **Überwachung der Verbindung** getroffen:
    1. „KeepAlive-Tracking“ zur Erkennung von Verbindungsabbrüchen
    2. ~~„Notification-Tracking“ zum Empfangen von Benachrichtigungen~~
5. Die Verbindung ändert ihren Status (**PlcDeviceConnection.State** Eigenschaft) auf den Wert ***Connected***.

## Close

Das passiert beim Aufruf von 'Close':

1. Die Verbindung ändert ihren Status (**PlcDeviceConnection.State** Eigenschaft) auf den Wert ***Closing***.
2. Die Verbindung **gibt** alle **erworbenen Ressourcen** wieder **frei**
3. **Beendet** die **Überwachung der physikalischen Verbindung**
4. Eine gegebenfalls aufgebaute physikalische **Verbindung zur SPS wird beendet** und steht somit wieder anderen Teilnehmern zur Verfügung.
5. Der beim Connect erstellte **Socket wird geschlossen und verworfen**.
6. Die Verbindung ändert ihren Status (**PlcDeviceConnection.State** Eigenschaft) auf den Wert ***Closed***.

## BreakDetection

Die „BreakDetection“-**Abbrucherkennung** bezeichnet den Mechanismus, der für die Erkennung von Verbindungsabbrüchen zuständig ist. Hierbei kommt das KeepAlive Verfahren zum Einsatz, um so einen **Timeout der Verbindung zur SPS** zu **erkennen**. Kommt es zum Timeout, so versucht das Framework

**automatisch wieder eine Verbindung zur SPS herzustellen**. Während beim KeepAlive in regelmäßigen Abständen KeepAlive-Nachrichten zur SPS gesendet werden, um so die Verbindung „zu testen“ und „aufrecht zu erhalten“, wird bei zu langen Antwortzeiten (= Timeout erreicht?) auf eine KeepAlive-Nachricht angenommen, dass die Verbindung unterbrochen ist. Ist das der Fall, wird in immer größeren Abständen eine weitere KeepAlive-Nachricht gesendet. Bleiben auch diese unbeantwortet, wird von einer abgebrochenen Verbindung ausgegangen und der zuvor beschriebene Mechanismus zur Wiederaufnahme der Verbindung tritt in Kraft. Aktiviert wird die Abbrucherkennung, welche standardmäßig nicht aktiviert ist, über die **PlcDeviceConnection.UseBreakDetection** Eigenschaft.

## Verbindungsparameter

Damit eine Verbindung zur SPS aufgebaut werden kann, müssen die richtigen Parameter festgelegt werden. **Generell** wird die **IP Adresse der SPS** (**IPDeviceEndPoint.Adress** Eigenschaft) **benötigt**, unter der sie zu erreichen ist. Die dabei vom **SiemensDevice** erwartete Endpunkt-Instanz (engl. Endpoint) liefert der Verbindung alle primär nötigen Informationen über die SPS. Anstelle der statischen IP Adresse kann auch der DNS Name der SPS verwendet werden, so könnte anstelle von „192.168.0.80“ auch „plc_man_drill_001“ verwendet werden. Zusätzlich zur IP Adresse der SPS benötigt der Endpunkt die **Nummer des Racks** (die Position der Montageschiene) und die **Nummer des Slots** der CPU. Werden keine Rack beziehungsweise Slot Nummer beim Endpunkt konfiguriert, versucht das Framework **automatisch die passenden Nummern** festzulegen. Welche Nummer im Einzelfall manuell angegeben werden muss hängt immer vom Aufbau (engl. setup) der Anlage / des Schaltschranks ab. Die Rack Nummer (beginnend bei 0) legt die Position der Montageschiene, auf der die SPS angebracht wurde, im Aufbau fest. Die Slot Nummer (beginned bei 1) legt die Position der CPU im Setup der Module der SPS fest. Eine auf die erste Hutschiene montierte S7-300 hat somit die Rack Nummer 0. Entspricht die Anordnung der Module (auf der Montageschiene) dem Schema:

1. „Power Supply (SP)“ Modul
2. „CPU“ Modul
3. „Interface“ Modul (IM)
4. „Signal“ Modul 1 (SM)
5. „Signal“ Modul 2 (SM)
6. „Signal“ Modul 3 (SM)
7. ...

Dann gilt für diese SPS die Slot Nummer 2, weil an dieser Position die CPU positioniert wurde.

IP S7 LINK SDK
SIMATIC connected Development

# Development Guide

## Getestet? Du willst es?

Lizenzmodell Preise Angebot Jetzt bestellen

C# ▼ |VB|

# Der App Frame

## Eine S7 App

1. Verweis zum **IPS7LnkNet.Advanced** Namespace hinzufügen:
```
using IPS7Lnk.Advanced;
```
2. Eine Instanz der SimaticDevice Klasse mit der Adresse der Steuerung erzeugen:
```
var device = new SimaticDevice("192.168.0.80");
```
3. Verbindung zur Steuerung erstellen und öffnen:
```
var connection = device.CreateConnection();
connection.Open();
```
4. Ihr Code zur Interaktion mit der Steuerung:
```
// Your code to interact with the controller.
```
5. Vor dem Beenden der Anwendung die Verbindung wieder trennen:
```
connection.Close();
```
6. Unter Verwendung des using Blocks sieht das dann so aus:
```
using (var connection = device.CreateConnection()) {
    connection.Open();
    // Your code to interact with the controller.
}
```

# Prozessdaten

## Adressierung

Die folgenden Typen kommen hierbei zum Einsatz: PlcAddress, PlcRawType, PlcOperand und PlcOperandType.

Unabhängig von der Art des Zugriffs, muss beschrieben werden, auf welchen Datenbereich man zugreifen möchte. Wo sich in der Steuerung die Daten befinden, auf die zugegriffen werden soll, wird anhand der **PlcAddress** des Datenbereichs festgelegt. Es besteht die Möglichkeit die **PlcAddress** als einfache Zeichenfolge (gemäß bekannter Addressierung in der SPS) oder über die API des SDKs zu konstruieren. Die im Folgenden gezeigten paarweisen Beispiele zeigen wie eine **PlcAddress** erstellt werden kann, um auf unterschiedliche Weise den gleichen Datenbereich zu adressieren.

- Fünftes Datenwort im zehnten Datenbaustein:
```
var a = PlcAddress.Parse("DB10.DBW 5");
var b = new PlcAddress(PlcOperand.DataBlock(10), PlcRawType.Word, 12);
```

- Erstes Bit im fünften Byte des zehnten Datenbausteins:

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

- Achtes Byte im Eingang:

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

- Drittes Bit im neunten Byte des Merkers:

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

Zur Definition der PlcAddress über die Parse-Methode oder einen der Konstruktoren kann die Adressierung auch über den impliziten Cast-Operator der Klasse erfolgen. Unterstützt wird auch hier die Siemens- oder IEC-spezifische Adressierung.

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

Die einer PlcAddress zugrundeliegende Zeichenkette der SPS Datenadresse kann über die ToString-Methode der PlcAddress Klasse abgerufen werden. Dabei besteht die Möglichkeit den gewünschten Operanden-Standard (Siemens oder IEC) anzugeben. Wird kein spezieller Standard angegeben wird immer vom Siemens Standard ausgegangen.

```
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
```

Das hier gezeigte Vorgehen wird aus Gründen der Einfachheit in allen weiteren Code-Ausschnitten verwendet werden. Je nach Anwendungsfall kann die gewünschte Addressierungs-Form verwendet werden.

# Werte lesen

Die folgenden Typen kommen hierbei zum Einsatz: SimaticDevice, PlcDeviceConnection und PlcAddress.

Zum Lesen eines Wertes aus einem bestimmten Datenbereich, kodiert im Format des entsprechenden SPS-Datentypen, wird eine der Datentyp-spezifischen Read-Methoden der

PlcDeviceConnection

verwendet. Zum Lesen der Daten besteht die Möglichkeit einen einzelnen Wert oder eine Folge von Werten (= ein Array) ab der angegebenen Adresse zu lesen. Soll ein Array gelesen werden, muss nach der Adresse zusätzlich noch die Anzahl der Elemente beim Aufruf der Read-Methode übergeben werden.

- Einen einzelnen `Int32`-Wert lesen:

```
int value = connection.ReadInt32("DB1.DBD 1");
```

- Eine Folge von drei `Int32`-Werten lesen:

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

Abhängig davon, welches Format die PlcAddress aufweist und welcher SPS-Datentyp gelesen werden soll, werden dementsprechend viele Bytes gelesen. Der dabei gelesene SPS-Datentyp wird anschließend in die Form des gewünschten PC-Datentypen überführt.

# Werte schreiben

Die folgenden Typen kommen hierbei zum Einsatz: SimaticDevice, PlcDeviceConnection und PlcAddress.

Zum Schreiben eines oder mehrerer Werte (= ein Array) in einen bestimmten Datenbereich, kodiert im Format des entsprechenden SPS-Datentypen, wird eine der Datentyp-spezifischen Write-Methoden der

PlcDeviceConnection

verwendet. Wird beim Schreiben ein Array anstelle eines einzelnen Wertes übergeben, dann werden alle Werte im Array in der selben Reihenfolge ab der angegebenen Adress geschrieben.

- Einen einzelnen `Int32`-Wert schreiben:

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

- Eine Folge von drei `Int32`-Werten schreiben:

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

# Werte als SPS-Variablen

Die folgenden Typen kommen hierbei zum Einsatz: SimaticDevice, PlcDeviceConnection, PlcAddress, PlcInt32, PlcInt32Array, PlcBoolean, PlcBooleanArray und PlcString.

Häufig muss ein bestimmter Datenbereich mehrmals und an unterschiedlichen Stelle im Programmfluss gelesen oder geschrieben werden. Ändert sich dann der Datenbereich, in dem der Wert in der SPS steht, muss der gesamte Quellcode nach der alten Adresse durchsucht und entsprechend der neuen angepasst werden. Zudem ist nicht immer klar, welcher Wert sich hinter einer SPS Adresse eines bestimmten Datenbereichs verbirgt. Mit der von Hilfe PlcValue-Objekten können SPS-Variablen im Quellcode einmalig definiert und jederzeit eindeutig adressiert werden.

- Eine einzelne `Int32`-Variable definieren und lesen:

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

- Eine `Int32`-Array-Variable definieren und lesen:

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

- Eine einzelne `Int32`-Variable definieren und schreiben:

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

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

- Eine `Int32`-Array-Variable definieren und schreiben:

```
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);
```

Abhängig davon, welches Format die PlcAddress aufweist und welcher SPS-Datentyp geschrieben werden soll, werden dementsprechend viele Bytes geschrieben. Der dabei zu schreibende PC-Datentyp wird zuvor in die Form des gewünschten SPS-Datentypen überführt.

Da die Konsistenz der gelesenen oder geschriebenen Daten besonders wichtig ist, besteht die Möglichkeit auch mehrere SPS-Variablen gleichzeitig zu lesen oder zu schreiben. Die Mischung von verschiedenen PlcValue-Instanzen funktioniert an dieser Stelle genau so, als würde man nur Instanzen eines bestimmten PlcValue-Typen verwenden. Geht man zum Beispiel von folgenden Prozessabbild aus, sehen die entsprechenden Read-/Write-Zugriffe wie folgt aus.

```
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);
```

Das Lesen des Prozessabbilds könnte folgendermaßen aussehen:

```
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}");
```

Das Schreiben des Prozessabbilds könnte folgendermaßen aussehen:

```
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);
```

Der Wert der Value-Eigenschaft der SPS-Variable kann über den Konstruktor der PlcValue-Klasse festgelegt und über die Value-Eigenschaft geändert werden. Der Wert, der im Konstruktor übergeben wird, kann dann als Initialwert verstanden werden. Wird die Value-Eigenschaft geändert, wird nicht automatisch der neue Werte in die Steuerung übertragen – es muss über ein explizierter WriteValue(s)-Aufruf durchgeführt werden.

# Strukturierte Daten

Im den folgenden Abschnitten **wird davon ausgegangen**, dass in der Steuerung ein Prozessabbild existiert, welches dem **(fiktiven) Datentypen „MillJob“** entspricht.
Die Struktur des Datentypen wird dabei wie folgt definiert:

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

## Struktur definieren

Die folgenden Typen kommen hierbei zum Einsatz: PlcObject, PlcMemberAttribute und PlcMember.

Zur Definition der Struktur für den Zugriff auf strukturierte Daten wird die PlcObject Klasse abgeleitet. Bei der Ableitung werden dann für alle zu adressierenden Datenbereiche entsprechende Felder und/oder Eigenschaften definiert. Auf jedem Member, das einen Wert in der Steuerung repräsentiert muss dann das PlcMemberAttribute festgelegt werden. Über das Attribut wird dann der zugehörige Datenbereich adressiert. Handelt es sich um ein Array oder einen String-Wert, wird im Attribut zusätzlich die Anzahl der Elemente beziehungsweise die Länge des Strings angegeben.

Für die zuvor fiktiv definierte Struktur ergibt sich dann zum Beispiel folgende Implementierung als PlcObject:

```
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;
}
```

Durch die Kombination der Adressierung der Prozessdaten über Felder und Eigenschaften können nicht-POCOs implementiert werden:

```
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];
    }
}
```

Die zur Definition benötigten Informationen können entweder über das Handbuch der Steuerung oder vom verantwortlichen SPS-Entwickler bezogen werden.

## Struktur lesen

Die folgenden Typen kommen hierbei zum Einsatz: SimaticDevice, PlcDeviceConnection und PlcObject.

Zum Lesen sturkturierter Daten über eine zuvor definierte Struktur wird die ReadObject-Methode der Verbindung verwendet:

```
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);
```

Die in der Struktur definierten Felder/Eigenschaften werden 1:1 in der Reihenfolge gelesen, in der diese definiert wurden.

## Struktur schreiben

Die folgenden Typen kommen hierbei zum Einsatz: SimaticDevice, PlcDeviceConnection und PlcObject.

Zum Schreiben sturkturierter Daten über eine zuvor definierte Struktur wird die WriteObject-Methode der Verbindung verwendet:

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

connection.WriteObject(job);
```

Die zum Zeitpunkt des Aufrufs in der Struktur enthaltenen Werte werden 1:1 in der Reihenfolge, in der die Felder/Eigenschaften definiert wurden, geschrieben.

# Benachrichtigungen

## Status der Verbindung(en)

Die folgenden Typen kommen hierbei zum Einsatz: PlcDeviceConnection, PlcDeviceConnectionState, PlcStatus und PlcNotifications.

Die Verwaltung der Verbindungen zu den einzelnen Steuerungen setzt voraus, dass der Zustand jeder Verbindung zu jeden Zeitpunkt bekannt ist. Zur Überwachung des Zustandes einer Verbindung können die Ereignisse Opening, Opened, Connecting, Connected, Disconnected, Closing, Closed und Faulted behandeln. Zusammenfassend für alle diese Ereignisse kann auch das StateChanged-Ereignis behandelt werden.

```
connection.StateChanged += HandleConnectionStateChanged;

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

Weitere Informationen über den Zustand der Verbindung können über die Status-Eigenschaft der Verbindung abgefragt werden. Teil dieser Informationen ist vor allem das Ergebnis der zuletzt ausgeführten Operation (wie zum Beispiel der zuletzt addressierte SPS-Datentyp). Auch hier kann die Änderung des Status entsprechend behandelt werden.

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

Im passenden EventHandler können dann die Status-Informationen für eine benutzerdefinierte Protokollierung oder erweiterten Auswertung des Verbindungszustands verwendet werden.

```
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>");
}
```

Zusätzlich zu den Instanz-bezogenen Ereignissen können auch global alle Verbindungen überwacht werden. Hierzu gehört neben den bereit genannten Ereignissen auch ein ConnectionCreated-Ereignis welches das dynamische Hinzufügen von weiteren EventHandlern ermöglicht.

```
PlcNotifications.ConnectionCreated += HandleNotificationsConnectionCreated;

...

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

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

# Status des Zugriffs

Die folgenden Typen kommen hierbei zum Einsatz: PlcDeviceConnection, PlcNotifications und PlcStatus.

Ein Zugriff auf einen bestimmten Datenbereich kann aus mehreren Gründen fehlschlagen. Während ein Bereich entweder nicht (mehr) existiert kann es auch sein, dass die Startadresse gültig ist, jedoch das Datenende nicht länger dem erwarteten Ende entspricht. Weitere solche Situationen sind ungültige Daten, da gegebenenfalls kein Initialwert festgelegt wurde. Die Ursache für Probleme beim Zugriff auf die Datenbereiche kann schnell über entsprechende StatusCodes herausgefunden werden.

Werden für den Zugriff PlcValue-Instanzen verwendet, kann für jede SPS-Variable der Status geprüft werden.

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

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

Zusätzlich kann bis auf die Ebene des SPS-Datentypen der Status geprüft werden. Hierzu verwendet man die GetStatus-Methode der Verbindung. Weiter besteht die Möglichkeit die Auswertung einer Operation selbst in die Hände zu nehmen. Dadurch ist es möglich eine fehlgeschlagene Operation, die auch zu einer Ausnahme führen kann, selbst zu bewerten und als „in Ordnung“ zu deklarieren. Das ist zum Beispiel dann sinnvoll, wenn man eine Standard-Anwendung für die Steuerungen einer Anlage entwickelt, diese besitzen jedoch nicht die gleichen Datenbereiche, weshalb ein nicht adressierbarer Datenbereich auch ignoriert werden kann.

Zur benutzerdefinerten Auswertung muss eine benutzerdefinerte Methode in der statischen PlcNotifications Klasse eingetragen werden.

```
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;
}
```

# Gerätedaten

Die folgenden Typen kommen hierbei zum Einsatz: SimaticDevice, IPlcDeviceInfo, PlcBlockInfo und PlcOperand.

Gelegentlich sind die Geräte-Informationen der Steuerung entscheidend, um herauszufinden, ob man mit der richtigen Steuerung arbeitet und wo diese genau steht. Zum Abrufen der hierfür zur Verfügung stehenden Gerätedaten kann ein Instanz der IPlcDeviceInfo-Schnittstelle über die GetInfo-Methode der PlcDevice Klasse abgerufen werden. Die Instanz enthält dann alle bereitgestellten Gerätedaten.

```
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}");
```

Zu beachten ist, dass nicht jede Steuerung alle hier gezeigten Geräteinformationen bereitstellt. Ob eine Information zur Verfügung steht kann über eine der Has-Eigenschaften geprüft werden.

# Konfiguration

## Allgemein

Die folgenden Typen kommen hierbei zum Einsatz: PlcDevice und PlcDeviceConnection.

Die Klassen PlcDevice und PlcDeviceConnection werden als Basisklassen der SimaticDevice und SimaticDeviceConnection verwendet. Dadurch erbt die Klasse SimaticDevice zum Beispiel die EndPoint-Eigenschaft zur Konfiguration des Endpunktes über den sich mit der Steuerung verbunden werden soll. Ebenso stellt die PlcDeviceConnection diverse Eigenschaften bereit. Teil dieser sind Eigenschaften zur Steuerung des Timeout-Verhaltens und der Abbrucherkennung.

Weitere spezifische Einstellungen sind vom verwendeten Provider (= PlcDevice Derivat) abhängig und können nach einen Cast auf den PlcDeviceConnection spezifische Typen entsprechend konfiguriert werden.

## Gerätetypen

Die folgenden Typen kommen hierbei zum Einsatz: SimaticDevice, SimaticDeviceType und SimaticChannelType.

Das Framework versucht generell automatisch den Gerätetypen und die passende Kanalart für die Steuerung zu ermitteln. Je nach Art der verwendeten Steuerung und Aufbau des Netzes kann es jedoch

notwendig sein manuell den Gerätetypen beziehungsweise die Kanalart festzulegen.

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

# Endpunkt

Die folgenden Typen kommen hierbei zum Einsatz: SimaticDevice, IPDeviceEndPoint und SimaticDeviceType.

Generell genügt für den Zugriff entweder der DNS-Name oder die IP-Adresse der Steuerung. Je nach Art der Steuerung und Setup des Netzes kann es sein, dass zusätzlich noch die Rack- und die Slot-Nummer der Steuerung festgelegt werden müssen.

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

# Lizenzierung

Das IP S7 LINK SDK kommt mit einer **Testlizenz die je Anwendungsstart 30 Minuten uneingeschränkt zur Softwareentwicklung** verwendet werden kann. Sollte diese Einschränkung ihre Evaluationsmöglichkeiten einschränken, besteht die Möglichkeit eine alternative Testlizenz bei uns zu beantragen.

Fragen Sie einfach unseren Support (via support@traeger.de) oder lassen Sie sich gleich direkt von uns beraten und offene Fragen durch unsere Entwickler klären!

Nach Erhalt Ihres personalisierten **Lizenzschlüssels zur Entwicklung mit IP S7 LINK** muss dieser dem Framework mitgeteilt werden. Fügen Sie hierzu die folgende Codezeile in Ihre Anwendung ein, **bevor** Sie das erste Mal auf die **SimaticDeviceConnection Klasse** zugreifen. Ersetzen Sie hierbei *<insert your license code here>* durch den von uns erhaltenen Lizenzschlüssel.

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

Zudem erhalten Sie Informationen über die aktuell vom Framework verwendete Lizenz über die *LicenseInfo* Eigenschaft der **IPS7Lnk.Advanced.Licenser Klasse**. Das funktioniert wie folgt:

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

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

Im Laufe der Entwicklung/Evaluation ist es häufig egal, ob gerade die Testlizenz oder bereits die erworbene Lizenz verwendet wird. Sobald aber die Anwendung in den produktiven Einsatz geht, ist es ärgerlich, wenn die Anwendung während der Ausführung aufgrund einer ungültigen Lizenz nicht mehr funktioniert. Aus diesem Grund empfehlen wir den folgenden Codeausschnitt in die Anwendung zu implementieren und diesen zumindest beim Start der Anwendung auszuführen:

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

Weitere Informationen zur Lizenzierung, dem Erwerb oder anderen Fragen erhalten Sie direkt auf unserer Produktseite unter: www.traeger.de.

# Inhaltsverzeichnis