S7.Net documentation

How to download s7.Net

The official repository is on GitHub (https://github.com/killnine/s7netplus), you can also download the library directly from NuGet (https://www.nuget.org/packages/S7netplus/).

What is S7.Net

S7.Net is a plc driver that works only with Siemens PLC and only with Ethernet connection. This means that your plc must have a Profinet CPU or a profinet external card (CPxxx card).
S7.Net is written entirely in C#, so you can debug it easily without having to go through native dlls.

Supported PLC

S7.Net is compatible with S7-200, S7-300, S7-400, S7-1200, S7-1500.

Getting started with S7.Net

To get started with S7.Net you have to download and include the S7.Net.dll in your project. You can do this by downloading the NuGet package, or by downloading the sources and compile them.

Create a PLC instance, connect and disconnect

To create an instance of the driver you need to use this constructor:

```
public Plc(CpuType cpu, string ip, Int16 rack, Int16 slot)
```

- **Cpu**: this specify what CPU you are connecting to. The supported CPU are:

```
public enum CpuType {
    S7200 = 0,
    S7300 = 10,
    S7400 = 20,
    S71200 = 30,
    S71500 = 40,
}
```

- **Ip**: this contains the IP address of the CPU of external Ethernet card
- **Rack**: this contains the rack of the plc, that you can find in hardware configuration in Step7
- **Slot**: this is the slot of the CPU, that you can find in hardware configuration in Step7

Example:
This code creates a Plc object for a S7-300 plc at the IP address 127.0.0.1, that it's localhost, for a plc that it's in rack 0 and a cpu that it's in slot 2 of the hardware configuration:

```
Plc plc = new Plc(CpuType.S7300, "127.0.0.1", 0, 2);
```

Connecting to the PLC

```
public ErrorCode Open()
```

For example this line of code open the connection:

```
plc.Open();
```

Disconnecting from the PLC

```
public void Close()
```

For example this closes the connection:

```
plc.Close();
```

Error handling

The Open() method returns an `ErrorCode` to check if the operation was successful. You should always check that it returns `ErrorCode`.NoError.

These are the types of errors:

```
public enum ErrorCode
{
    NoError = 0,
    WrongCPU_Type = 1,
    ConnectionError = 2,
    IPAddressNotAvailable,
    WrongVarFormat = 10,
    WrongNumberReceivedBytes = 11,
    SendData = 20,
    ReadData = 30,
    WriteData = 50
}
```

Global error handling

Not all methods returns an error. You can check for

```
public ErrorCode LastErrorCode
```

and

```
public string LastErrorString
```

on every methods that you execute, in order to catch errors while running the driver.

Check PLC availability

To check if the plc is available you can use the property

```
public bool IsAvailable
```
When you check this property, the driver will send a ping to the plc and returns true if the plc responds to the ping, false otherwise.

Check PLC connection

Checking the plc connection is trivial, because you have to check if the PC socket is connected but also if the PLC is still connected.

The property that you have to check in this case is this:

```
public bool IsConnected
```
This property can be checked after you called the method Open(), to check if the connection is still alive.

Read bytes / Write bytes

The library offers several methods to read variables. The basic one and the most used is ReadBytes.

```
public byte[] ReadBytes(DataType dataType, int db, int startByteAdr, int count)
```

```
public ErrorCode WriteBytes(DataType dataType, int db, int startByteAdr, byte[] value)
```

This reads up to 200 bytes (actual limit of the protocol) from a memory location that you determine.

- **dataType**: you have to specify the memory location with the enum DataType
  ```
  public enum DataType
  {
      Input = 129,
      Output = 130,
      Memory = 131,
      DataBlock = 132,
      Timer = 29,
      Counter = 28
  }
  ```
- **db:** this is the address of the dataType, for example if you want to read DB1, this field is "1"; if you want to read T45, this field is 45.
- **startByteAdr:** this is the address of the first byte that you want to read, for example if you want to read DB1.DBW200, this is 200.
- **count:** this contains how many bytes you want to read. It's limited to 200 bytes and if you need more, you must use recursion.
- **Value[]:** array of bytes to be written to the plc.

Example:

This method reads the first 200 bytes of DB1:

```
var bytes = plc.ReadBytes(DataType.DataBlock, 1,0,200);
```

Example with recursion:

```
private List<byte> ReadMultipleBytes(int numBytes, int db, int startByteAdr = 0)
{
    List<byte> resultBytes = new List<byte>();
    int index = startByteAdr;
    while (numBytes > 0)
    {
        var maxToRead = (int)Math.Min(numBytes, 200);
        byte[] bytes = ReadBytes (DataType.DataBlock, db, index, (int)maxToRead);
        if (bytes == null)
            return new List<byte>();
        resultBytes.AddRange(bytes);
        numBytes -= maxToRead;
        index += maxToRead;
    }
    return resultBytes;
}
```

Read and decode / Write decoded

This method permits to read and receive an already decoded result based on the varType provided. This is useful if you read several fields of the same type (for example 20 consecutive DBW). This is also limited to maximum 200 bytes. If you specify `VarType.Byte`, it has the same functionality as ReadBytes.

```
public object Read(DataType dataType, int db, int startByteAdr, VarType varType, int va
rCount)

public ErrorCode Write(DataType dataType, int db, int startByteAdr, object value)
```

- **dataType**: you have to specify the memory location with the enum DataType
  ```
  public enum DataType
  {
      Input = 129,
      Output = 130,
      Memory = 131,
      DataBlock = 132,
      Timer = 29,
      Counter = 28
  }
  ```
- **db:** this is the address of the dataType, for example if you want to read DB1, this field is "1"; if you want to read T45, this field is 45.
- **startByteAdr:** this is the address of the first byte that you want to read, for example if you want to read DB1.DBW200, this is 200.
- **varType:** specify the data that you want to get your bytes converted.
  ```
  public enum VarType
  {
  Bit,
  Byte,
  Word,
  DWord,
  Int,
  DInt,
  Real,
  String,
  Timer,
  Counter
  }
  ```
- **count:** this contains how many variables you want to read. It's limited to 200 bytes and if you need more, you must use recursion.
- **Value:** array of values to be written to the plc. It can be a single value or an array, the important is that the type is unique, for example array of double, array of int, array of shorts, etc..

Example:

This method reads the first 20 DWords of DB1:

```
var dwords = plc.Read(DataType.DataBlock, 1,0, VarType.DWord, 20);
```

Read a single variable / Write a single variable

This method reads a single variable from the plc, by parsing the string and returning the correct result. While this is the easiest method to get started, this is very inefficient because the driver sends a TCP request for every variable.

```
public object Read(string variable)
```

```
public ErrorCode Write(string variable, object value)
```

- **`variable:`** specify the variable to read by using strings like “DB1.DBW20”, “T45”, “C21”, “DB1.DBD400”, etc.

Example:

This reads the variable DB1.DBW0. The result must be cast to ushort to get the correct 16-bit format in C#.

```
ushort result = (ushort)plc.Read("DB1.DBW0");
```

Read a struct / Write a struct

This method reads all the bytes from a specified DB needed to fill a struct in C#, and returns the struct that contains the values.

```
public object ReadStruct(Type structType, int db, int startByteAdr = 0)
```

```
public ErrorCode WriteStruct(object structValue, int db, int startByteAdr = 0)
```

- structType: Type of the struct to be read, for example: typeOf(MyStruct))
- db: index of the DB to read
- startByteAdr: specified the first address of the byte to read (the default is zero).

Example:

You define a DataBlock in the plc like:

Adresse	Name	Typ	Anfangswert
0.0		STRUCT	
+0.0	varBool0	BOOL	FALSE
+0.1	varBool1	BOOL	FALSE
+0.2	varBool2	BOOL	FALSE
+0.3	varBool3	BOOL	FALSE
+0.4	varBool4	BOOL	FALSE
+0.5	varBool5	BOOL	FALSE
+0.6	varBool6	BOOL	FALSE
+1.0	varByte0	BYTE	B#16#0
+2.0	varByte1	BYTE	B#16#0
+4.0	varWord	WORD	W#16#0
+6.0	varReal	REAL	1.230000e+000
+10.0	varBool7	BOOL	TRUE
+12.0	varReal1	REAL	8.506780e+002
+16.0	varbyte2	BYTE	B#16#55
+18.0	varDWord	DWORD	DW#16#1234567
=22.0		END_STRUCT	

Then you add a struct into your .Net application that is similiar to the DB in the plc:

```
public struct testStruct
{
    public bool varBool0;
    public bool varBool1;
    public bool varBool2;
    public bool varBool3;
    public bool varBool4;
    public bool varBool5;
    public bool varBool6;

    public byte varByte0;
    public byte varByte1;
```

```
    public ushort varWord0;

    public double varReal0;
    public bool varBool7;
    public double varReal1;

    public byte varByte2;
    public UInt32 varDWord;
}
```

then you add the code to read or write the complete struct

```
// reads a struct from DataBlock 1
testStruct test = (testStruct)plc.ReadStruct(typeof(testStruct), 1);
```

Read a class / Write a class

This method reads all the bytes from a specified DB needed to fill a class in C#. The class is passed as reference and values are assigned by using reflection.

```
public void ReadClass(object sourceClass, int db, int startByteAdr = 0)
```

```
public ErrorCode WriteClass(object classValue, int db, int startByteAdr = 0)
```

- sourceClass: instance of the class that you want to assign the values
- db: index of the DB to read
- startByteAdr: specified the first address of the byte to read (the default is zero).

Example:

You define a DataBlock in the plc like:

Adresse	Name	Typ	Anfangswert
0.0		STRUCT	
+0.0	varBool0	BOOL	FALSE
+0.1	varBool1	BOOL	FALSE
+0.2	varBool2	BOOL	FALSE
+0.3	varBool3	BOOL	FALSE
+0.4	varBool4	BOOL	FALSE
+0.5	varBool5	BOOL	FALSE
+0.6	varBool6	BOOL	FALSE
+1.0	varByte0	BYTE	B#16#0
+2.0	varByte1	BYTE	B#16#0
+4.0	varWord	WORD	W#16#0
+6.0	varReal	REAL	1.230000e+000
+10.0	varBool7	BOOL	TRUE
+12.0	varReal1	REAL	8.506780e+002
+16.0	varbyte2	BYTE	B#16#55
+18.0	varDWord	DWORD	DW#16#1234567
=22.0		END_STRUCT	

Then you add a struct into your .Net application that is similiar to the DB in the plc:

```
public class TestClass
{
    public bool varBool0 { get; set;}
    public bool varBool1 { get; set;}
    public bool varBool2 { get; set;}
    public bool varBool3 { get; set;}
    public bool varBool4 { get; set;}
    public bool varBool5 { get; set;}
    public bool varBool6 { get; set;}

    public byte varByte0 { get; set;}
    public byte varByte1 { get; set;}
```

```
    public ushort varWord0 { get; set;}

    public double varReal0 { get; set;}
    public bool varBool7 { get; set;}
    public double varReal1 { get; set;}

    public byte varByte2 { get; set;}
    public UInt32 varDWord { get; set;}
}
```

then you add the code to read or write the complete struct

```
// reads a struct from DataBlock 1
TestClass testClass = new TestClass();
plc.ReadClass(testClass, 1);
```

Value conversion between C# and S7 plc

- Read S7 Word:
```
ushort result = (ushort)plc.Read("DB1.DBW0");
```

- Write S7 Word:
```
ushort val = 40000;
plc.Write("DB1.DBW0", val);
```

- Read S7 Int / Dec, you need to use the method ConvertToShort():
```
short result = ((ushort)plc.Read("DB1.DBW0")).ConvertToShort();
```

- Write S7 Int / Dec, you need to use the method ConvertToUshort():
```
short value = -100;
plc.Write("DB1.DBW0", value.ConvertToUshort());
```

- Read S7 DWord:
```
uint result = (uint)plc.Read("DB1.DBD40");
```

- Write S7 DWord:
```
uint val = 1000;
plc.Write("DB1.DBD40", val);
```

- Read S7 Dint, you need to use ConvertToInt():
```
int result2 = ((uint)plc.Read("DB1.DBD60")).ConvertToInt();
```

- Write S7 Dint:
```
int value = -60000;
plc.Write("DB1.DBD60", value);
```
- Read S7 Real, you need to use ConvertToDouble():
```
double result = ((uint)plc.Read("DB1.DBD40")).ConvertToDouble();
```

- Write S7 Real, you need to use ConvertToInt():
```
double val = 35.687;
plc.Write("DB1.DBD40", val.ConvertToUInt());
```

- Read bool from byte
```
byte myByte = 5; // 0000 0101
myByte.SelectBit(0) // true
myByte.SelectBit(1) // false
```

This is taken directly from Snap7 documentation: http://snap7.sourceforge.net/

S7 1200/1500 Notes

An external equipment can access to S71200/1500 CPU using the S7 "base" protocol, only working as an HMI, i.e. only basic data transfer are allowed.

All other PG operations (control/directory/etc..) must follow the extended protocol, not (yet) covered by Snap7.

Particularly to access a DB in S71500 some additional setting plc-side are needed.

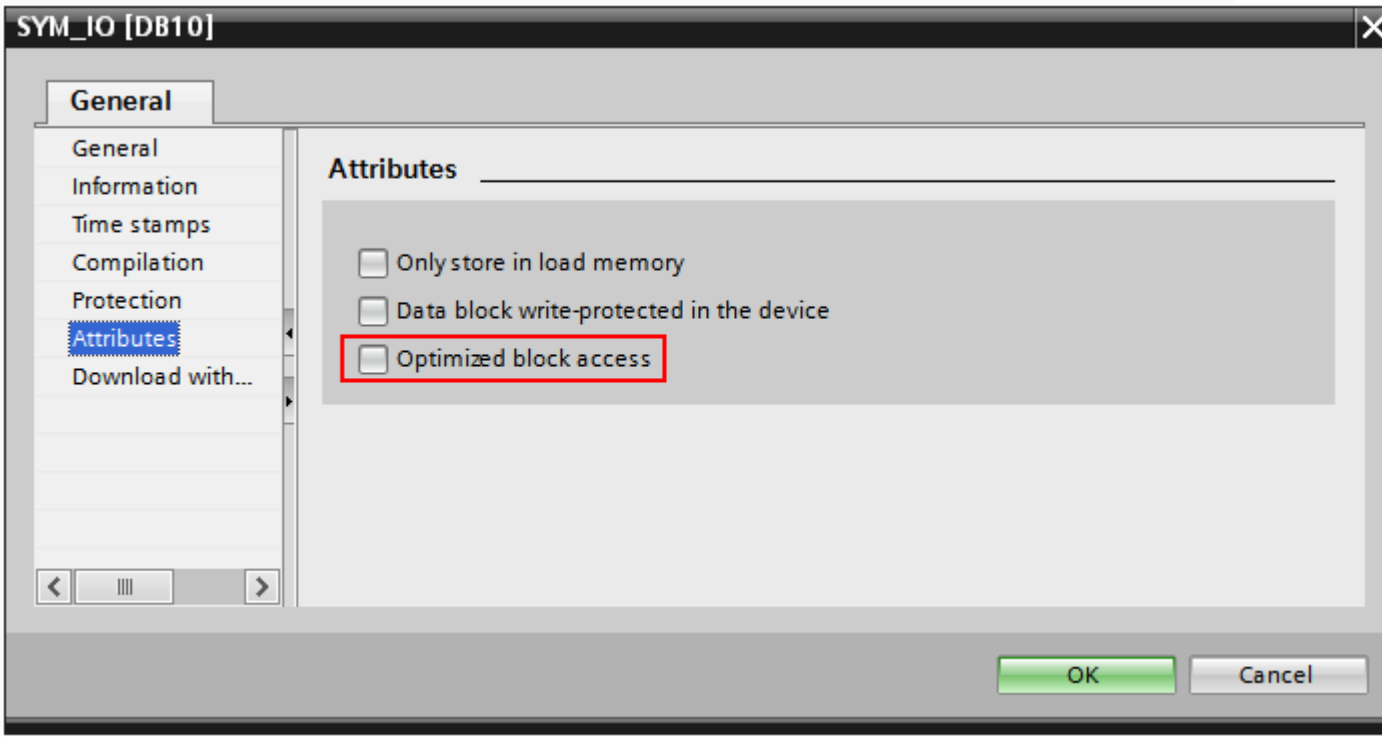
1. Only global DBs can be accessed.
2. The optimized block access must be turned off.
3. The access level must be "full" and the "connection mechanism" must allow GET/PUT.

Let's see these settings in TIA Portal V12

DB property

Select the DB in the left pane under "Program blocks" and press Alt-Enter (or in the contextual menu select "Properties…")

Uncheck Optimized block access, by default it's checked.



Protection

Select the CPU project in the left pane and press Alt-Enter (or in the contextual menu select "Properties…")

In the item Protection, select “Full access” and Check “Permit access with PUT/GET ….” as in figure.