APPLICATION NOTE YASKAWA

# Title: Communicating with the MPiec Controller using PLCi	
Product(s): All MPiec Controllers except MP2300Siec	**Doc. No. AN.MPIEC.29**

Table of Contents

Title: Communicating with the MPiec Controller using PLCi	
Product(s): All MPiec Controllers except MP2300Siec	**Doc. No. AN.MPIEC.29**

Application Overview

PLCi is a communication protocol that provides several services for a PC application:

1) Read and write variables in the MPiec controller. Elementary data types and user defined structures and arrays can be accessed.
2) Start and stop the PLC application.
3) Read & write files on the controller.

Other communication protocols supported by the MPiec such as Modbus TCP and Ethernet/IP require allocating variables at PLC hardware addresses (%I, %Q, %M), but with PLCi this is not necessary. Please note the DLLs listed in the chart below. The first two listed are the main DLLs which make reference to methods and functions in the others.

Products Used

Component	Product and Model Number
Controller	All MPiec controllers except MP2300Siec. Must be eCLR type.
Software	Optional: PLCiInterface.DLL (Use this when passing user defined types (structures or arrays) Required: PLCiDotNet.DLL (Use for all other features listed in this document) Plci.DLL PlciBridge.DLL MetalLP64_10.DLL MetalLP32_10.DLL NativeMetaAPI.DLL
Operating system	Windows 7, Windows Embedded 7, Windows 10

Application Requirements

This application note assumes the user is familiar with programming in Visual Studio and does not go into the details of creating a project.

PLCi is based on the .NET platform, therefore it is only compatible with the Windows operating system. Typical applications are programmed using Microsoft Visual Studio.

Installing PLCi

The DLLs are included in an installer ZIP file available from www.yaskawa.com by searching for document AN.MPIEC.29. They will be extracted to '[root]\Program Files (x86)\Yaskawa\PLCi for MPiec Controllers'.

Getting Help

Help for PLCi can be found in the .chm file, also available from www.yaskawa.com by searching for document AN.MPIEC.29. Often times after downloading this file and opening, it will appear to be empty. This is because the content is being blocked by Windows. To view the content, close the file, right click on it and select ‘Properties’. Under the ‘General’ tab, select ‘Unblock’.

Overview of PLCi DLLs

There are two main DLLs that can be used directly to interact with the MPiec controller:

- PLCiDotNet.dll
- PLCiInterface.dll

PLCiDotNet contains all the methods necessary to connect/disconnect to a controller, read/write files and variables, control the PLC, and view device attributes.

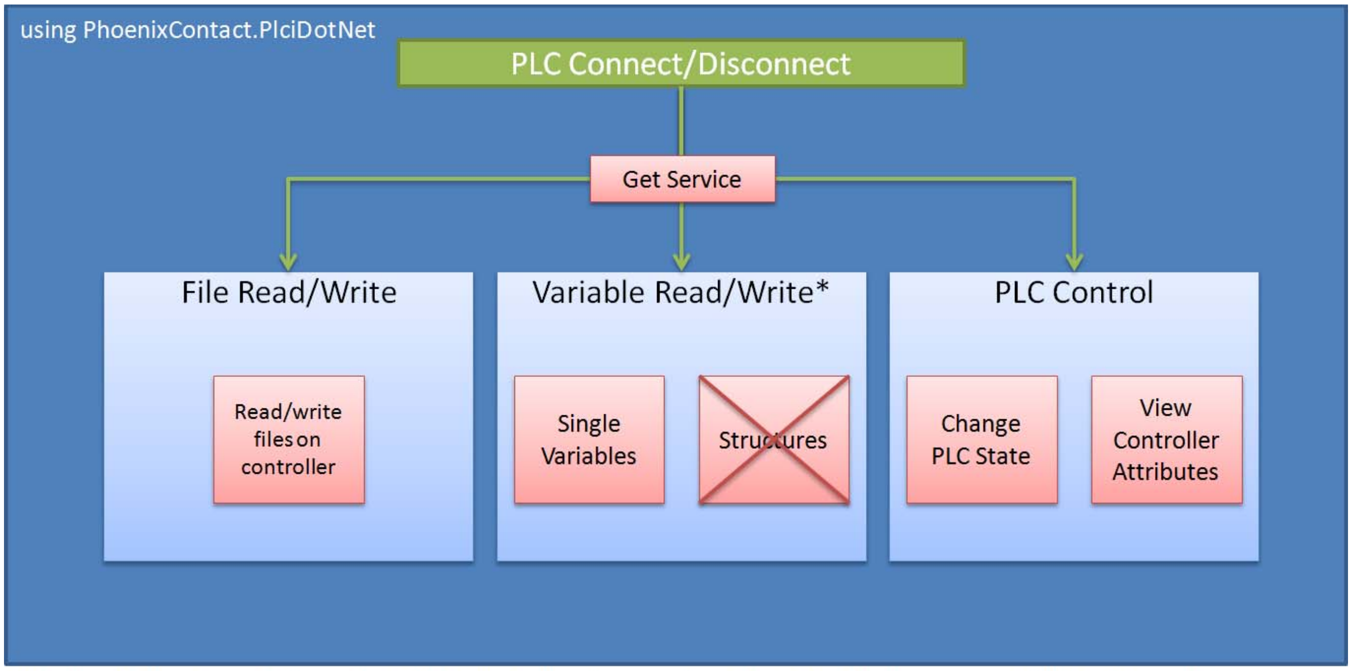
PLCiInterface is a wrapper for PLCiDotNet that makes variable/structure access simpler. PLCiInterface only contains methods to connect to the controller and read/write variables and structures.

PLCiInterface is recommended for applications that must only read/write data structures. If file reading/writing or PLC control (stopping/warm starting PLC, etc.) is required, then PLCiDotNet is recommended.

Additionally, all functions contained in these DLLs are blocking, meaning that when the function is called, processing is halted until the function returns.

PLCiDotNet DLL

An overview of the capabilities of PLCi.



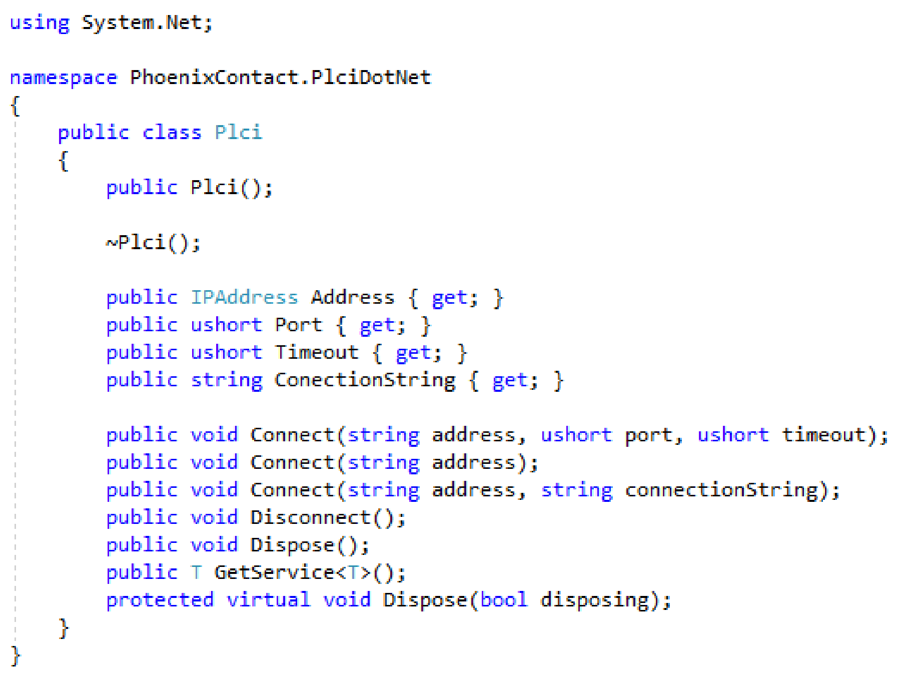
*Variables within structures are still accessible with PlciDotNet. They must be accessed as single variables—the structure as a whole will not be recognized.

PLCiDotNet: PLCi Class – PLC Connect/Disconnect and Get Service

This is the base class containing all of the services which are used for interacting with the controller.

Methods	Description
Connect	Used to connect to the controller. Accepts an IP address, port, and timeout value.
Disconnect	Disconnects from the controller socket.
Dispose	Destroys the PLCi object.
GetService	Returns a service which contains a series of functions used for interacting with the controller. Accepts a service name from the following list: IDeviceAttributeService IDataAccessService IPlcControlService IFileService

To use the PLCiDotNet DLL, add the phrase “Using PhoenixContact.PLCiDotNet;” above the namespace of the C# project.



```
using System.Net;

namespace PhoenixContact.PlciDotNet
{
    public class Plci
    {
        public Plci();

        ~Plci();

        public IPAddress Address { get; }
        public ushort Port { get; }
        public ushort Timeout { get; }
        public string ConectionString { get; }

        public void Connect(string address, ushort port, ushort timeout);
        public void Connect(string address);
        public void Connect(string address, string connectionString);
        public void Disconnect();
        public void Dispose();
        public T GetService<T>();
        protected virtual void Dispose(bool disposing);
    }
}
```

PLCiDotNet: Services

Use the following services to interact with the controller after calling the GetService method in the PLCi class using the respective service type from the options listed above.

PLCiDotNet: IFileService – File Read/Write

Manipulates files on the MPiec controller.

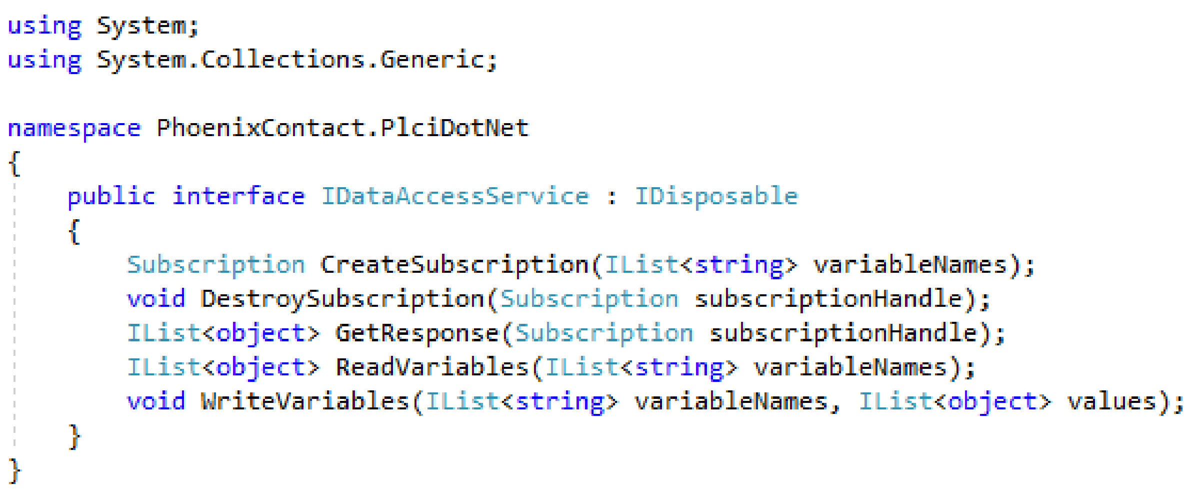
Method	Description
Close	Closes a file and deactivates it handle.
GetTransferPaketSize	Returns a size which would be acceptable to use for reading chunks of data from a file.
OpenRead	Opens a file for reading. Returns a handle which can be used to reference the file.
OpenWrite	Opens a file for writing. Returns a handle which can be used to reference the file.
Read	Reads a chunk of data from the file specified by the file handle into a target byte array. The chunk size is determined by the value returned from GetTransferPaketSize(). Returns a Boolean which reports whether there is more data to be read.
RemoveFile	Removes a file based on the file name passed to it.
Write	Writes data to a file which has been opened with OpenWrite().
Dispose	Disposes of the service. Should be used after use of the service has been completed. Disposing prevents memory leaks.

```
using System;

namespace PhoenixContact.PlciDotNet
{
    public interface IFileService : IDisposable
    {
        void Close(int fileHandle);
        uint GetTransferPaketSize();
        int OpenRead(string fileName);
        int OpenWrite(string fileName);
        bool Read(int fileHandle, out byte[] data, int lengthToRead);
        void RemoveFile(string fileName);
        void Write(int fileHandle, byte[] data);
    }
}
```

PLCiDotNet: IDataAccessService – Variable Read/Write

Reads and writes variables to / from the MPiec Controller.

Methods	Description
CreateSubscription	Subscribes to a list containing the reference names of variables within the controller. Returns a subscription object. If a variable will be read more than once, create a subscription rather than reading the variable directly. The variable name must be registered only once with the controller, allowing better performance. It is possible to create several subscription objects simultaneously.
DestroySubscription	Disposes of a subscription object.
GetResponse	Gets a list of variable values from all of the variables subscribed to in a subscription object.
ReadVariables	Reads the values of a list containing the reference names of variables within the controller.
WriteVariables	Writes a list of values to a list containing the reference names of variables within the controller.
Dispose	Disposes of the service. Should be used after use of the service has been completed. Used to prevent memory leaks.


```
using System;
using System.Collections.Generic;

namespace PhoenixContact.PlciDotNet
{
    public interface IDataAccessService : IDisposable
    {
        Subscription CreateSubscription(IList<string> variableNames);
        void DestroySubscription(Subscription subscriptionHandle);
        IList<object> GetResponse(Subscription subscriptionHandle);
        IList<object> ReadVariables(IList<string> variableNames);
        void WriteVariables(IList<string> variableNames, IList<object> values);
    }
}
```

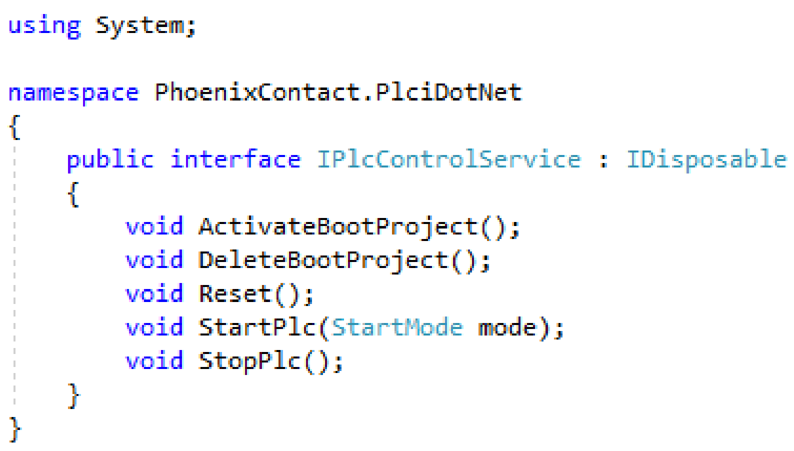
Both global and local variables can be referenced using the following syntax:

Global Variables
"@GlobalVariables.MyGlobalVariableName"
"@GlobalVariables.MyGlobalIntArray[10]"
"@GlobalVariables.MyGlobalStructArray[10].ComponentA"

Local variables:
"@InstanceVariables.MyResource.MyTask.MyProgram.MyProgramVariable"
"@InstanceVariables.MyResource.MyTask.MyProgram.MyFunctionBlock.MyFBVariable"
"@InstanceVariables.MyResource.MyTask.MyProgram.MyFunctionBlock.MyFBArray[10].ComponentB"

PLCiDotNet: IPlcControlService – Change PLC State

Method	Description
ActivateBootProject	Activates the project which was downloaded as the boot project by copying it from Flash to Ram. Usually called before StartPlc().
DeleteBootProject	Deletes the boot project that is currently on the PLC.
Reset	Resets the PLC.
StartPlc	Starts the PLC Application using one of the following modes: StartMode.ColdStart StartMode.WarmStart (normal) StartMode.HotStart
StopPlc	Stops the PLC application currently running on the MPiec. Other services on the controller continue to operate, such as I/O drivers, Mechatrolink network. Motion will be halted.
Dispose	Disposes of the service. Disposing prevents memory leaks.



```
using System;

namespace PhoenixContact.PlciDotNet
{
    public interface IPlcControlService : IDisposable
    {
        void ActivateBootProject();
        void DeleteBootProject();
        void Reset();
        void StartPlc(StartMode mode);
        void StopPlc();
    }
}
```

PLCiDotNet: IDeviceAttributeService – View Controller Attributes

Provides access to general attributes of the controller including manufacturer information and PLC state.

GetAttribute	Returns the value of a given attribute Some of the most important attributes are listed below. A complete list can be found within the help .chm under Namespaces>Namespace List>PLCi>PLCI::StandardDeviceAttributes StandardDeviceAttribute.Manufacturer StandardDeviceAttribute.ProductName StandardDeviceAttribute.ECLrBootProjName StandardDeviceAttribute.FirmwareVersion StandardDeviceAttribute.HardwareVersion StandardDeviceAttribute.ImageOnPLC StandardDeviceAttribute.SourceOnPLC StandardDeviceAttribute.PlcState
GetAttributes	Same as GetAttribute but works on a list of attributes.
SetAttribute	Allows setting an attribute value.
SetAttributes	Same as SetAttribute but applies to a list of attributes.
Dispose	Disposes of the service. Should be used after use of the service has been completed. Disposing prevents memory leaks.

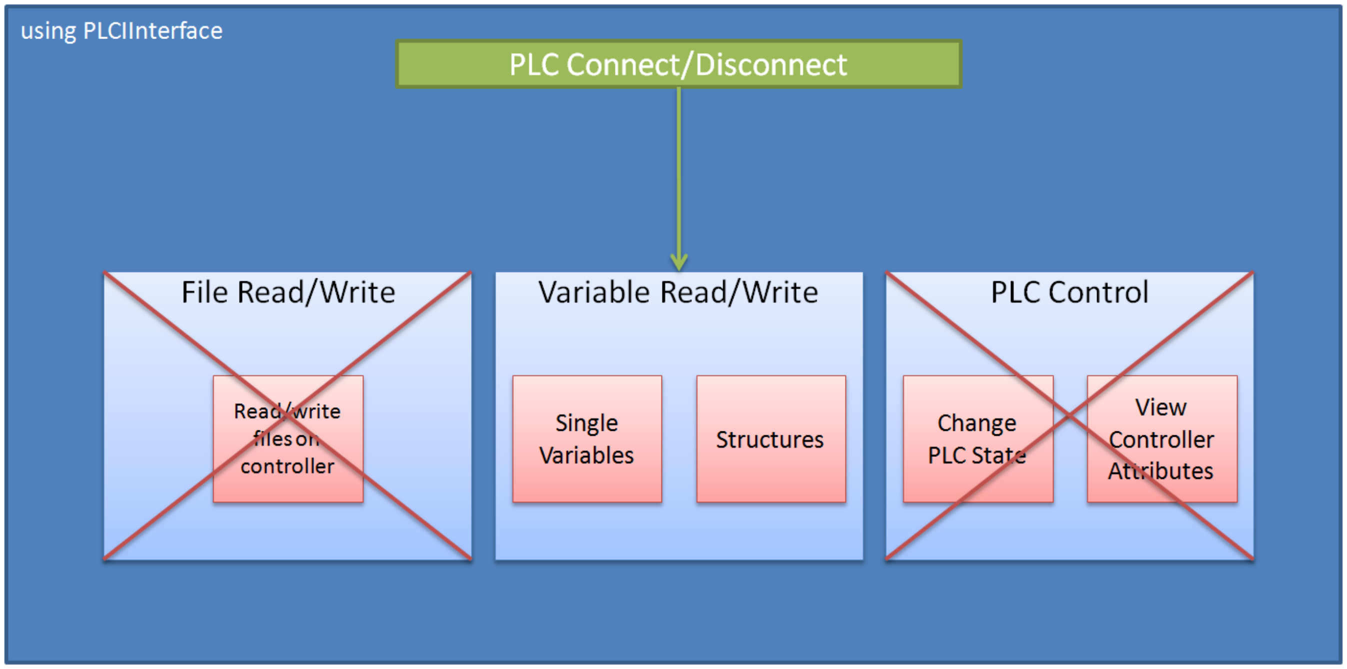
```
using System;
using System.Collections.Generic;
using PhoenixContact.PlciDotNet.Internal;

namespace PhoenixContact.PlciDotNet
{
    public interface IDeviceAttributeService : IDisposable
    {
        object GetAttribute(StandardDeviceAttribute attributeId);
        T GetAttribute<T>(StandardDeviceAttribute attributeId);
        IList<object> GetAttributes(IList<short> attributes);
        void SetAttribute<T>(StandardDeviceAttribute attributeId, T value, EncodingType encoding);
        void SetAttribute<T>(StandardDeviceAttribute attributeId, T value);
        void SetAttributes(IList<short> attributeIds, IList<object> values);
    }
}
```

Attribute	Description
BaseIndexAttribute	Indicates the base value of the array in MotionWorks IEC (0 based, 1 based, 2 based, etc.) Place above the array definition in the struct as follows: [BaseIndexAttribute(1)] Public double[] myArray = new double[6];

Title: Communicating with the MPiec Controller using PLCi	
Product(s): All MPiec Controllers except MP2300Siec	**Doc. No. AN.MPIEC.29**

PLCiInterface DLL

To use PLCiInterface, add the phrase “using Yaskawa.PLCiInterface;” above the namespace of the C# project.

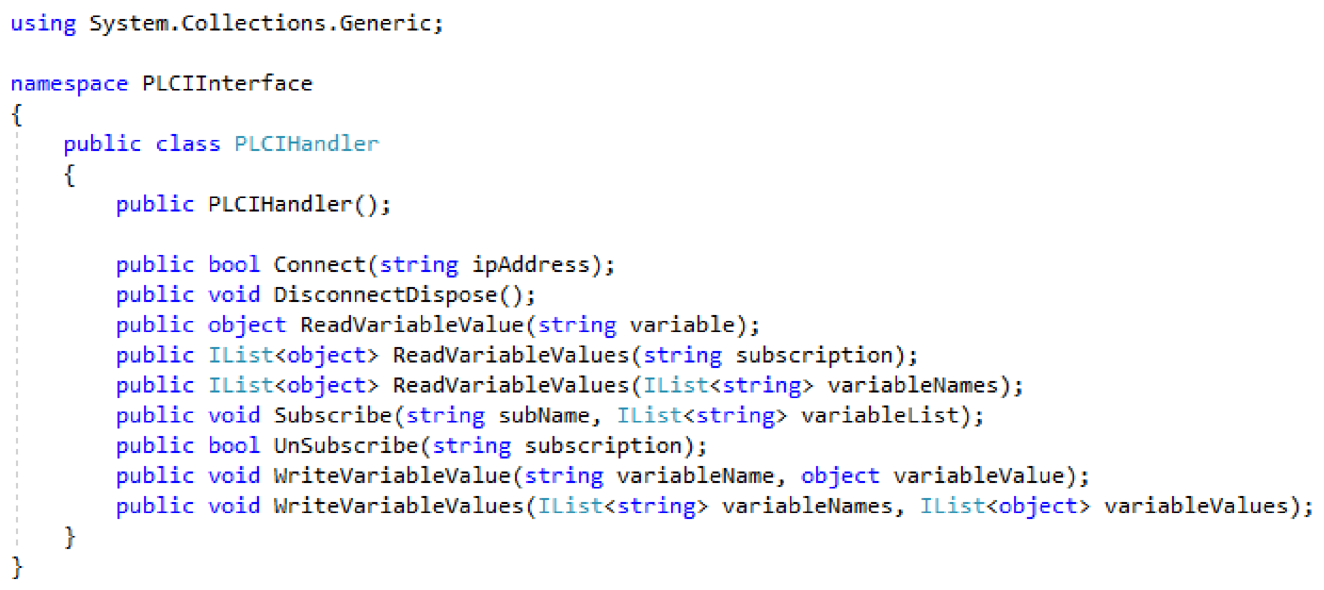


PLCInterface: PLCIHandler Class

This is the base class for PLCiInterface used to connect/disconnect from the controller and read/write variables.

Methods	Description
Connect	Connect to an MPiec Controller having the IP address passed.
DisconnectDispose	Disconnects from the MPiec controller and destroys the PLCi object.
ReadVariableValue	Reads a variable value from the MPiec controller based on the variable name passed.
ReadVariableValues	Same as ReadVariableValue except it supports multiple variables. Pass either the name of a subscription or an IList of variable names.
Subscribe	Subscribes a list of variables to an arbitrary subscription name for future reference.

UnSubscribe	Removes subscription.
WriteVariableValue	Writes to a pre-existing variable on the controller based on the variable name and value passed to it.
WriteVariableValues	Same as WriteVariableValue except it works for multiple variables. Must be passed an IList of variable names and a second IList containing corresponding variable values.



```
using System.Collections.Generic;

namespace PLCIInterface
{
    public class PLCIHandler
    {
        public PLCIHandler();

        public bool Connect(string ipAddress);
        public void DisconnectDispose();
        public object ReadVariableValue(string variable);
        public IList<object> ReadVariableValues(string subscription);
        public IList<object> ReadVariableValues(IList<string> variableNames);
        public void Subscribe(string subName, IList<string> variableList);
        public bool UnSubscribe(string subscription);
        public void WriteVariableValue(string variableName, object variableValue);
        public void WriteVariableValues(IList<string> variableNames, IList<object> variableValues);
    }
}
```

PLCiInterface : StructureConverter Class

The StructureConverter class is designed for use when more complex datatypes are required.

This class handles custom structures so that the data can be used by PLCi methods.

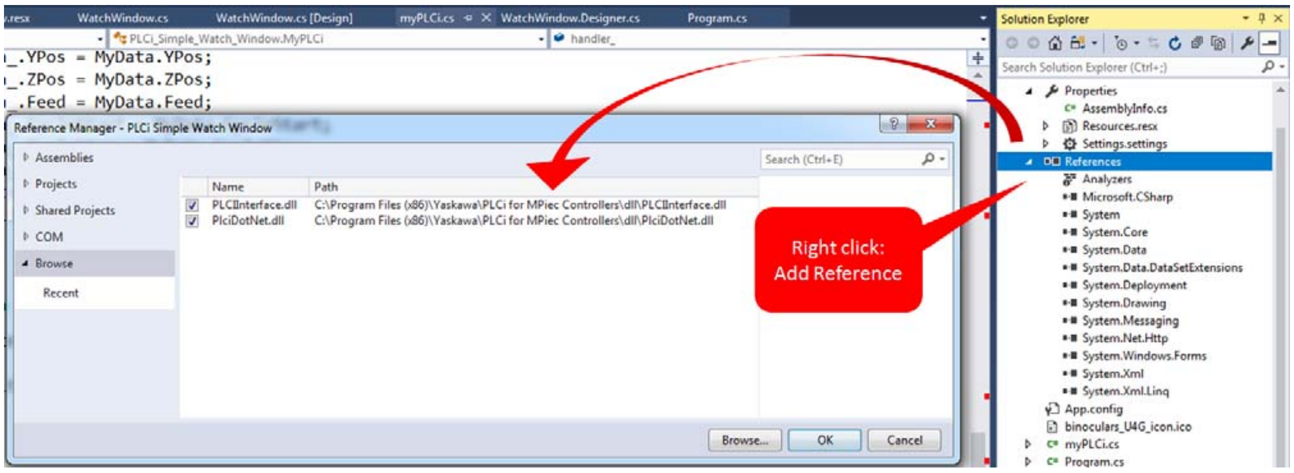
Methods	Description
GenerateVariables	Pass the structure to be broken down into a “list<string>” containing all variable names. Parent is supplied as ‘@GlobalVariables.MyMPiecStruct’ It is necessary to execute this only once per variable group.
GenerateVariableValues	Sets the values of an entire structure. Makes a list of values that correspond with the list of variable names created by GenerateVariables. This doesn’t write data to the controller, it just generates the list which can then be given to the PLCi handler.

PopulateStructValues	Copies a list of values into a structure.

```
using System.Collections.Generic;

namespace PLCIInterface
{
    public static class StructureConverter
    {
        public static List<string> GenerateVariables(object obj, string parent);
        public static IList<object> GenerateVariableValues(object obj);
        public static object PopulateStructValues(object obj, string parentName, IList<string> variableNames, IList<object> variableValues);
    }
}
```

Example – Adding PLCiInterface to a C# Project

1) **Locate the PLCiInterface DLL and add it to the project. Add PLCiDotNet.DLL if using File or PLC control services.**



```
_.YPos = MyData.YPos;
_.ZPos = MyData.ZPos;
_.Feed = MyData.Feed;
```

	Name	Path
✓	PLCIInterface.dll	C:\Program Files (x86)\Yaskawa\PLCi for MPiec Controllers\dll\PLCIInterface.dll
✓	PlciDotNet.dll	C:\Program Files (x86)\Yaskawa\PLCi for MPiec Controllers\dll\PlciDotNet.dll

2) **Add the Reference(s) above the namespace.**

```
using Yaskawa.PLCiInterface;
using PhoenixContact.PLCiDotNet; (if using File or PLC control services)
```

3) **Add the class variables inside the class.**
The example below shows demonstrates a connection to an MPiec for the variables listed in varList_ and "MachineStruct," a user defined datatype. The MachineStruct type must be defined elsewhere in the C# project. Important: The C# definition must have the same variable names (case sensitive). For information on the datatype naming differences between Visual Studio and MotionWorks IEC, see "DataType Naming Convention" on page 17.

```
private PLCiHandler handler_ = new PLCiHandler();
private List<string> varList_ = new List<string>();    // holds names of connected vars
private string subscriptionName_ = "MyPLCVariables";   // arbitrary subscription label

private const string machineSub_ = "@GlobalVariables.MyMachine";
private MachineStruct machineData_ = new MachineStruct();
private List<string> machineVarList_ = new List<string>();
```

4) **Add a method to connect to the MPiec Controller and prepare to read and write data.**

```
private void ConnectVariables()
{
       // clear out any previous connections
       handler_.DisconnectDispose();

       handler_.Connect(IPAddress);

       varList_.Add("@GlobalVariables.PLC_SYS_TICK_CNT");
       varList_.Add("@GlobalVariables.PLC_TICKS_PER_SEC");
       varList_.Add("@GlobalVariables.BoolToRead");
       varList_.Add("@GlobalVariables.ValueToRead");

       // connect variables in varList_ with matching controller variables
       handler_.Subscribe(subscriptionName_, varList_);

       // connect variables in MachineStruct to matching controller variables
       machineVarList_ =
       (MachineStruct)StructureConverter.GenerateVariables(machineData_, machineSub_);

       handler_.Subscribe(machineSub_, machineVarList_);
}
```

5) **Reading a variable.**

```
IList<object> variables = handler_.ReadVariableValues(subscriptionName_);
```

6) **Reading a User Defined Type.**

```
IList<object> structValues = handler_.ReadVariableValues(machineSub_);
machineData_ = (MachineStruct)StructureConverter.PopulateStructValues(machineData_,
       machineSub_, machineVarList_, structValues);
```

7) **Writing a variable.**

```
handler_.WriteVariableValue("@GlobalVariables.BoolToWrite", false);
handler_.WriteVariableValue("@GlobalVariables.ValueToWrite", (double)54.1);
```

8) **Writing a User Defined Type.**

```
IList<object> structValues = StructureConverter.GenerateVariableValues(machineData_);
handler_.WriteVariableValues(machineVarList_, structValues);
```

DataType Naming Convention between MotionWorks IEC and Visual Studio

The naming of data types in MotionWorks IEC varies slightly from the corresponding types in Visual Studio. For convenience, this table shows elementary datatypes in MotionWorks IEC and their corresponding datatypes in Visual Studio.

MotionWorks IEC	Visual Studio
BOOL	bool
BYTE	byte
USINT	byte
LREAL	double
REAL	float
DINT	int
SINT	sbyte
INT	short
DWORD	uint
TIME	uint
UDINT	uint
UINT	ushort
WORD	ushort

Frequently Asked Questions

Q1) Is there a license required?
A1) The DLLs are free software, however there is a license agreement which must be accepted as part of the installation process.

Q2) Can you import or export variables?
A2) Not at this time. Variables must be added manually to Visual Studio. See the VariableListRO() in the example project.

Q3) Why does the connection timeout?
A3) If the connection is inactive for more than two minutes, the connection will time out and the PLCi object must be reconnected. To avoid this, poll the controller at an interval of two minutes or less. This can be accomplished by using a timer thread and then interacting with the Mpiec through PLCi when the timer finishes (e.g. read current variables from PLC, check current PLC state, etc.). After communicating with the PLC, restart the timer so that the MPiec will be polled at a consistent interval

Q4) Does PLCi require Modbus communication?
A4) No, PLCi is a proprietary protocol native to the MPiec Controller and communicates directly with the controllers operating system.

Q5) Does PLCi require any special function blocks or programming on the MPiec Controller, such as the ones described in the YDeviceComm firmware library?
A5) No.

Q6) Is it possible to write all the files necessary to commission an MPiec controller?
A6) Yes, the configuration XML files and the PLC Image can be written, provided that the folder structure is already established. This will be an issue for creating the XML files in the 'Startup' folder, which does not exist at factory default conditions. There is another way to transfer the 'Archive.ZIP' file to the controller, which will self extract and setup a proper image. PLCi file manipulation is better suited for configuration changes, requiring modification to existing files.

Q7) When passing structure data (UDT) via PLCi, is it required that the structure definitions match exactly?

A7) No. Each sub element of a User Defined Type must exist on the MPiec, and the C# definition must match the case, but if additional elements are added to the User Defined Type in the MPiec project, that will not cause a problem for communication.

Q8) Can PLCi be used when developing plug-ins for Yaskawa Compass?

A8) Compass uses PLCi, and provides an additional layer of connectivity functions available to plug-ins developed for use with it. See document AN.MPiec.06 – Compass Configuration & Customization Guide.