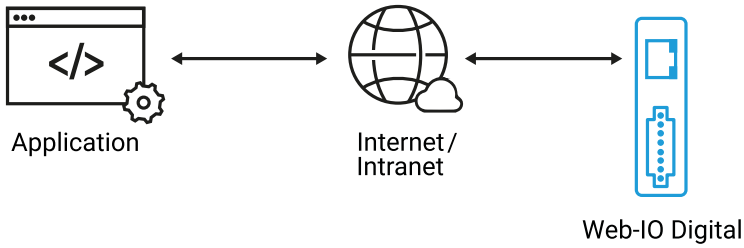
Tutorial for the Web-IO Digital:

Access the Web-IO Digital with C# using binary sockets

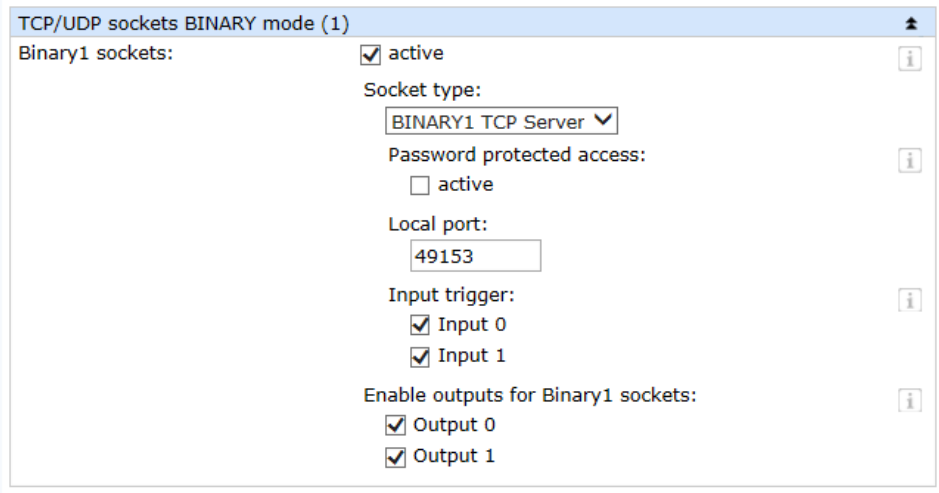
Visual C++ was until recently one of the most widely used development platforms for creating Windows applications. In the meantime, more and more programmers are working with the .Net Framework and creating their applications in C#.



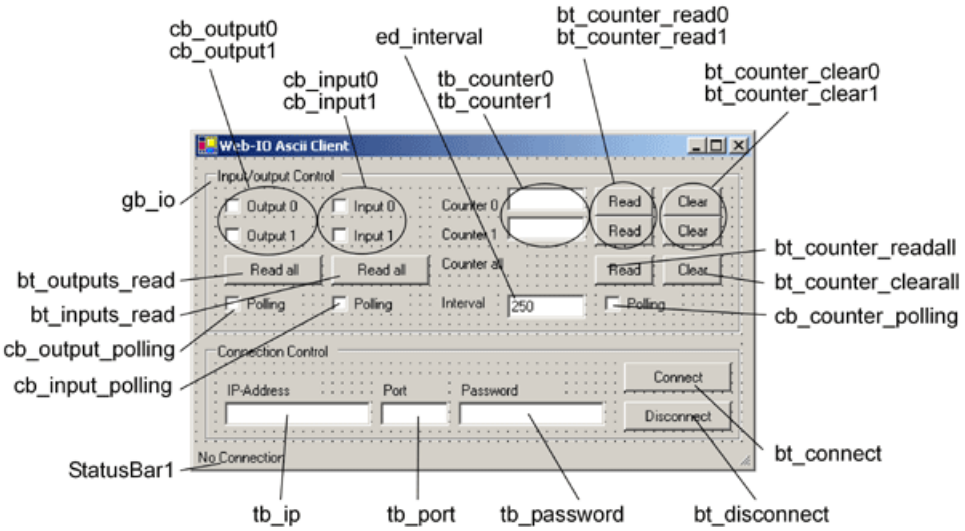
Using the following C# program example you can access your Web-IO Digital with its inputs and outputs in a Windows application using binary socket mode.

Preparations

- Provide power to the Web-IO and connect the IOs
- Connect the Web-IO to the network
- Assign IP addresses
- On the Web-IO in *Communication channels >> Socket API* activate *BINARY1 sockets*, check *Input Trigger* and enable the *outputs for switching*



Combining the various operating elements and display objects in the VB.net form



When naming the individual objects it is helpful to use logical names. In this example the first part of the name describes the type of object and the second part the function.

Importing resources and declaring member variables

Firstly, all classes required for the network connection and the GUI are imported.

```
using System;
using System.Windows.Forms;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Runtime.InteropServices;
```

Then the application components and important variables for a TCP connection are declared as member variables, thereby making them accessible to the class methods.

```
public class mainWindow : System.Windows.Forms.Form
{ // All elements of the application
  private CheckBox cb_Output0;
  private CheckBox cb_Output1;
  private CheckBox cb_Input0;
  private CheckBox cb_Input1;
  private CheckBox cb_Polling_Counter;
  private CheckBox cb_Polling_Outputs;
  private CheckBox cb_Polling_Inputs;
  private Button bt_Readall_Outputs;
  private Button bt_Readall_Inputs;
  private Button bt_Clear_Counter0;
  private Button bt_Clear_Counter1;
  private Button bt_Clearall_Counter;
  private Button bt_Read_Counter0;
  private Button bt_Read_Counter1;
  private Button bt_Readall_Counter;
  private Label lb_Counter0;
  private Label lb_Counter1;
  private Label lb_Intervall;
  private TextBox tb_Counter0;
  private TextBox tb_Counter1;
  private TextBox tb_Intervall;
  private Button bt_Connect;
  private Button bt_Disconnect;
  private TextBox tb_Password;
  private TextBox tb_Port;
  private TextBox tb_IP;
  private GroupBox gb_ioControlBox;
  private GroupBox gb_conControlBox;
  private StatusBar statusBar;
  private Label label2;
  private Label label1;
  private Label label3;
  private System.Windows.Forms.Timer counter;
  private System.Windows.Forms.Timer outputs;
  private System.Windows.Forms.Timer inputs;

  private Socket client;
  private int intervall;
  private byte[] receivebuffer = new byte[256];
  private structIOState IOState;
  delegate void delegateSub();
```

The binary structures

For binary access the necessary binary structures used for communicating with the Web-IO must be defined. A detailed description of these structures can be found in the binary brief overview or in the programming manual for the Web-IO.

The IOState structure
The IOState structure is not one of the binary structures and is not directly needed for communication between the application and Web-IO. It is used for exchanging the IO states between the receive thread and the program sections which are responsible for refreshing the display elements.

The IODriver structure
With its four 16-bit variables the *IODriver* is the basic structure which is also part of all other binary structures.

```
public struct structIOState
{
  public UInt16 InputState;
  public UInt16 OutputState;
  public UInt32 CounterValue0;
  public UInt32 CounterValue1;
}

public struct structEADriver
{
  public UInt16 Start_1;
  public UInt16 Start_2;
  public UInt16 StructType;
  public UInt16 StructLength;
}

public struct structOptions
{
  public structEADriver EADriver;
  public UInt32 Version;
  public UInt32 Options;
}

public struct structWriteRegister
{
  public structEADriver EADriver;
  public UInt16 Amount;
  public UInt16 Value;
}

public struct structSetBit
{
  public structEADriver EADriver;
  public UInt16 Mask;
  public UInt16 Value;
}

public struct structRegisterState
{
  public structEADriver EADriver;
  public UInt16 DriverID;
  public UInt16 InputValue;
  public UInt16 OutputValue;
}

public struct structReadCounter
{
  public structEADriver EADriver;
  public UInt16 CounterIndex;
}

[StructLayout(LayoutKind.Explicit)]
public struct structCounter
{
  [FieldOffset(0)]public structEADriver EADriver;
  [FieldOffset(8)] public UInt16 CounterIndex;
  [FieldOffset(10)] public UInt32 CounterValue;
}

[StructLayout(LayoutKind.Explicit)]
public struct structAllCounter
{
  [FieldOffset(0)] public structEADriver EADriver;
  [FieldOffset(8)] public UInt16 CounterNoOf;
  [FieldOffset(10)] public UInt32 CounterValue0;
  [FieldOffset(14)] public UInt32 CounterValue1;
}
```

For the structures it is important that the individual variables be stored in memory in their exact sequence. C# does not handle this automatically, especially when variables of different sizes are combined in a structure. To still ensure ordered storage in memory, [StructLayout(LayoutKind.Explicit)] is used to specify that each individual variable uses [FieldOffset(14)] to have a fixed offset to the first memory location of the structure assigned.

Starting the program

Setting up the operating elements
The group with the operating elements for the Web-IO is first blocked from operation. As soon as a connection is established, all elements are enabled which have a meaningful format.

The name of the respective operating element is derived from the element itself depending on the context. The first two characters in the name stand for the element type (cb -> Checkbox, bt -> Button, gb -> Groupbox and tb -> TextBox).

```
public mainWindow()
{
  InitializeComponent();
  gb_ioControlBox.Enabled = false;
  bt_Disconnect.Enabled = false;
  cb_Input0.Enabled = false;
  cb_Input1.Enabled = false;
  tb_Counter0.Enabled = false;
  tb_Counter1.Enabled = false;
  IOState.InputState = 0;
  IOState.OutputState = 0;
  IOState.CounterValue0 = 0;
  IOState.CounterValue1 = 0;
}
```

Connection control

Establishing the connection
The connection is opened by entering the IP address of the Web-IO in the text field *ed_ip* and clicking on the *bt_connect* button.

```
private void bt_Connect_Click(object sender, System.EventArgs e)
{
  try
  {
    if ((tb_IP.Text != "") && (tb_Port.Text != ""))
    {
      client = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
      IPEndPoint ipep = new IPEndPoint(IPAddress.Parse(tb_IP.Text), int.Parse(tb_Port.Text));
      client.BeginConnect(ipep, new AsyncCallback(connectCallback), client);
    }
    else
      statusBar.Text = "IP and Port needed!";
  }
  catch (SocketException)
  {
    statusBar.Text = "Connection not possible!";
  }
}
```

Opening the connection
For TCP/IP handling first an IPEndPoint is defined from IP address and TCP port and used to initialize the TCP_client socket. As part of the connection request a reference to a callback procedure is created.

Connection is made
As soon as the Web-IO accepts the connection, the callback procedure is executed. In addition a reference to a callback routine is created for data reception.

By sending the structure *Options* to the Web-IO, the latter is instructed to send the changed state when an output is set, using the structure *RegisterState*.

```
private void connectCallback(IAsyncResult ar)
{
  try
  {
    client = (Socket)ar.AsyncState;
    client.EndConnect(ar);
    connectupdatedisplay();
    intervall = 1000;
    client.BeginReceive(receivebuffer, 0, 255, SocketFlags.None, new AsyncCallback(receiveCallback), client);
  }
  catch (Exception)
  {
    disconnect();
  }
  if (client.Connected)
  {
    IntPtr BufPtr;
    structOptions Options;
    Options.EADriver.Start_1 = 0;
    Options.EADriver.Start_2 = 0;
    Options.EADriver.StructType = 0x1F0;
    Options.EADriver.StructLength = 16;
    Options.Version = 0;
    Options.Options = 1;
    BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(Options));
    Marshal.StructureToPtr(Options, BufPtr, true);
    sendstructure(BufPtr, (short)Options.EADriver.StructLength);
  }
}
```

In addition the connectupdatedisplay() procedure is called to enable all the needed operating elements of the application and to disable the Connect button.

```
 private void connectupdatedisplay()
{
  if (InvokeRequired)
  {
    BeginInvoke(new delegateSub(connectupdatedisplay), new object[] { });
    return;
  }
  statusBar.Text = "Connected!";
  gb_ioControlBox.Enabled = true;
  bt_Disconnect.Enabled = true;
  bt_Connect.Enabled = false;
}
```

In the current version of C# it is no longer allowed to access graphical elements of another thread (form) from a thread (here the Callback function).

The workaround here is to use delegates and invoke.

Disconnecting
The connection remains open until it is ended by the user clicking on the *Disconnect* button or by the Web-IO.

```
private void bt_Disconnect_Click(object sender, System.EventArgs e)
{
  try
  {
    client.Shutdown(SocketShutdown.Both);
    client.Close();
    disconnectupdatedisplay();
  }
  catch (Exception)
  {
    statusBar.Text = "Not able to disconnect!";
  }
}
```

In this case a corresponding procedure is invoked.

```
private void disconnect()
{
  try
  {
    client.Shutdown(SocketShutdown.Both);
    client.Close();
    disconnectupdatedisplay();
  }
  catch (Exception)
  {
    statusBar.Text = "Not able to disconnect!";
  }
}
```

To ensure thread-secure access, resetting the form elements has been moved to the `disconnectupdatedisplay()` procedure.

```
private void disconnectupdatedisplay()
{
  if (InvokeRequired)
  {
    BeginInvoke(new delegateSub(disconnectupdatedisplay), new object[] { });
    return;
  }
  statusBar.Text = "Disconnected!";
  gb_ioControlBox.Enabled = false;
  bt_Disconnect.Enabled = false;
  bt_Connect.Enabled = true;
}
```

Connection error
All actions affecting TCP/IP communication are executed within the Try instruction. If errors occur, the `disconnect();` method of the *client* object is invoked.

Operation and communication from the client side

Sending binary structures
As soon as a connection is made with the Web-IO, the user can use the corresponding program elements to send binary structures to the Web-IO.

```
private void sendstructure(IntPtr BufPtr, Int16 BufSize)
{
  byte[] senddata;
  senddata = new byte[BufSize];
  Marshal.Copy(BufPtr, senddata, 0, BufSize);
  client.BeginSend(senddata, 0, senddata.Length, 0, new AsyncCallback(sendCallback), client);
}

private void sendCallback(IAsyncResult ar)
{
  try
  {
    Socket tmp_client = (Socket) ar.AsyncState;
    int bytessend = tmp_client.EndSend(ar);
  }
  catch(Exception)
  {
    statusBar.Text = "Error by sending";
  }
}
```

C# does not provide a method for direct sending of binary structures. Binary data can only be sent as a byte array. Therefore calling the *sendstructure* function sends a pointer. This pointer references the memory location starting at which the contents of the structure to be sent can be found. Here it is apparent why the structure variables must be in sequence in memory. The second parameter sent is the length of the structure.

Using `Marshal.Copy(BufPtr, senddata, 0, Bufsize)` the structure data are then copied to a byte array. This byte array is then sent to the Web-IO.

Setting the outputs
The user sets the outputs by using the two check boxes *cb_outputx*. For this the program uses the *MouseUP* event of this object. If a *MouseUP*, i.e. releasing the output checkbox is registered, the program runs the corresponding procedure and passes the correspondingly filled in *SetBit* structure to the Web-IO depending on whether the checkbox is set or not.

```
private void cb_Output0_MouseUp(object sender, MouseEventArgs e)
{
  IntPtr BufPtr;
  structSetBit SetBit;
  SetBit.EADriver.Start_1 = 0;
  SetBit.EADriver.Start_2 = 0;
  SetBit.EADriver.StructType = 0x9;
  SetBit.EADriver.StructLength = 0xC;
  SetBit.Mask = 1;
  if (cb_Output0.Checked)
    SetBit.Value = 1;
  else
    SetBit.Value = 0;
  BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(SetBit));
  Marshal.StructureToPtr(SetBit, BufPtr, true);
  sendstructure(BufPtr, (short)SetBit.EADriver.StructLength);
}

private void cb_Output1_MouseUp(object sender, MouseEventArgs e)
{
  IntPtr BufPtr;
  structSetBit SetBit;
  SetBit.EADriver.Start_1 = 0;
  SetBit.EADriver.Start_2 = 0;
  SetBit.EADriver.StructType = 0x9;
  SetBit.EADriver.StructLength = 0xC;
  SetBit.Mask = 2;
  if (cb_Output1.Checked)
    SetBit.Value = 2;
  else
    SetBit.Value = 0;
  BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(SetBit));
  Marshal.StructureToPtr(SetBit, BufPtr, true);
  sendstructure(BufPtr, (short)SetBit.EADriver.StructLength);
}
```

Querying output/input status
The user can request the status of the outputs and inputs by clicking on the corresponding button.

```
private void bt_Readall_Outputs_Click(object sender, System.EventArgs e)
{
  IntPtr BufPtr;
  structEADriver RegisterRequest;
  RegisterRequest.Start_1 = 0;
  RegisterRequest.Start_2 = 0;
  RegisterRequest.StructType = 0x21;
  RegisterRequest.StructLength = 8;
  BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(RegisterRequest));
  Marshal.StructureToPtr(RegisterRequest, BufPtr, true);
  sendstructure(BufPtr, (short)RegisterRequest.StructLength);
}
```

By sending the structure *RegisterRequest* the switching states of inputs and outputs are requested. The Web-IO replies to this request with the structure *RegisterState*.

To query only the inputs, click on the *bt_inputs* button. Here the structure *ReadRegister* is sent, to which the Web-IO replies with the structure *WriteRegister*.

```
private void bt_Readall_Inputs_Click(object sender, System.EventArgs e)
{
  IntPtr BufPtr;
  structEADriver ReadRegister;
  ReadRegister.Start_1 = 0;
  ReadRegister.Start_2 = 0;
  ReadRegister.StructType = 1;
  ReadRegister.StructLength = 8;
  BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(ReadRegister));
  Marshal.StructureToPtr(ReadRegister, BufPtr, true);
  sendstructure(BufPtr, (short)ReadRegister.StructLength);
}
```

Read/clear counters

You can also query or clear the counter states of the input counters. Here the structure *ReadCounter* or *ReadClearCounter* is sent, whereby *CounterIndex* is used to send the number of the counter. The Web-IO replies with the structure *Counter*.

```
private void bt_Read_Counter0_Click(object sender, System.EventArgs e)
{
  IntPtr BufPtr;
  structReadCounter ReadCounter;
  ReadCounter.EADriver.Start_1 = 0;
  ReadCounter.EADriver.Start_2 = 0;
  ReadCounter.EADriver.StructType = 0xB0;
  ReadCounter.EADriver.StructLength = 0xA;
  ReadCounter.CounterIndex = 0;
  BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(ReadCounter));
  Marshal.StructureToPtr(ReadCounter, BufPtr, true);
  sendstructure(BufPtr, (short)ReadCounter.EADriver.StructLength);
}

private void bt_Read_Counter1_Click(object sender, System.EventArgs e)
{
  IntPtr BufPtr;
  structReadCounter ReadCounter;
  ReadCounter.EADriver.Start_1 = 0;
  ReadCounter.EADriver.Start_2 = 0;
  ReadCounter.EADriver.StructType = 0xB0;
  ReadCounter.EADriver.StructLength = 0xA;
  ReadCounter.CounterIndex = 1;
  BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(ReadCounter));
  Marshal.StructureToPtr(ReadCounter, BufPtr, true);
  sendstructure(BufPtr, (short)ReadCounter.EADriver.StructLength);
}

private void bt_Clear_Counter0_Click(object sender, System.EventArgs e)
{
  IntPtr BufPtr;
  structReadCounter ReadCounter;
  ReadCounter.EADriver.Start_1 = 0;
  ReadCounter.EADriver.Start_2 = 0;
  ReadCounter.EADriver.StructType = 0xC0;
  ReadCounter.EADriver.StructLength = 0xA;
  ReadCounter.CounterIndex = 0;
  BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(ReadCounter));
  Marshal.StructureToPtr(ReadCounter, BufPtr, true);
  sendstructure(BufPtr, (short)ReadCounter.EADriver.StructLength);
}

private void bt_Clear_Counter1_Click(object sender, System.EventArgs e)
{
  IntPtr BufPtr;
  structReadCounter ReadCounter;
  ReadCounter.EADriver.Start_1 = 0;
  ReadCounter.EADriver.Start_2 = 0;
  ReadCounter.EADriver.StructType = 0xC0;
  ReadCounter.EADriver.StructLength = 0xA;
  ReadCounter.CounterIndex = 1;
  BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(ReadCounter));
  Marshal.StructureToPtr(ReadCounter, BufPtr, true);
  sendstructure(BufPtr, (short)ReadCounter.EADriver.StructLength);
}
```

The structure type RegisterState *ReadAllCounter* or *ReadClearAllCounter* can be used to read or clear all the counters at the same time. The Web-IO replies with the structure *AllCounter*.

```
private void bt_Readall_Counter_Click(object sender, System.EventArgs e)
{
  IntPtr BufPtr;
  structEADriver ReadAllCounter;
  ReadAllCounter.Start_1 = 0;
  ReadAllCounter.Start_2 = 0;
  ReadAllCounter.StructType = 0xB1;
  ReadAllCounter.StructLength = 8;
  BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(ReadAllCounter));
  Marshal.StructureToPtr(ReadAllCounter, BufPtr, true);
  sendstructure(BufPtr, (short)ReadAllCounter.StructLength);
}

private void bt_Clearall_Counter_Click(object sender, System.EventArgs e)
{
  IntPtr BufPtr;
  structEADriver ReadClearAllCounter;
  ReadClearAllCounter.Start_1 = 0;
  ReadClearAllCounter.Start_2 = 0;
  ReadClearAllCounter.StructType = 0xC1;
  ReadClearAllCounter.StructLength = 8;
  BufPtr = Marshal.AllocHGlobal(Marshal.SizeOf(ReadClearAllCounter));
  Marshal.StructureToPtr(ReadClearAllCounter, BufPtr, true);
  sendstructure(BufPtr, (short)ReadClearAllCounter.StructLength);
}
```

Receiving data from the Web-IO

Process and display the received data
The Web-IO returns the appropriate structure depending on the query or triggering event. When data are received the corresponding callback procedure is invoked. Processing involves filling the first 8 bytes of the received byte array using a pointer operation first with an *IODriver* structure. The application recognizes which structure type it is using the variable *IODriver.StructType*.

- *IODriver.StructType* = 8
  structure *WriteRegister* for the status of the inputs
- *IODriver.StructType* = 31 (hex)
  structure *RegisterState* for the status of the inputs and outputs
- *IODriver.StructType* = B4 (hex)
  structure *Counter* for the value of individual counters
- *IODriver.StructType* = B5 (hex)
  structure *AllCounter* for the value of all counters

For processing the received byte array now uses a pointer operation to fill the appropriate structure.

The values thus sent are then processed and displayed. For the inputs and outputs *WriteRegisterValue*, *RegisterStae.InputValue* and *RegisterStae.outputValue* are used to send the bit pattern of all inputs and outputs.

```
private void receiveCallback(IAsyncResult ar)
{
  int BytesRead;
  string rcv = string.Empty;
  try
  {
    BytesRead = client.EndReceive(ar);
    client.BeginReceive(receivebuffer, 0, 255, SocketFlags.None, new AsyncCallback(receiveCallback), client);
  }
  catch (Exception)
  {
    BytesRead = 0;
  }
  if (BytesRead == 0)
  {
    if (client.Connected) { disconnect(); }
  }
  else
  {
    GCHandle MyGC = GCHandle.Alloc(receivebuffer, GCHandleType.Pinned);
    structEADriver EADriver = (structEADriver)Marshal.PtrToStructure(MyGC.AddrOfPinnedObject(), typeof(structEADriver))
    switch (EADriver.StructType)
    {
      case 8:
        structWriteRegister WriteRegister = (structWriteRegister)Marshal.PtrToStructure(MyGC.AddrOfPinnedObject(), typeof
        IOState.InputState = WriteRegister.Value;
      break;
      case 0x31:
        structRegisterState RegisterState = (structRegisterState)Marshal.PtrToStructure(MyGC.AddrOfPinnedObject(), typeof
        IOState.InputState = RegisterState.InputValue;
        IOState.OutputState = RegisterState.OutputValue;
      break;
      case 0xB4:
        structCounter Counter = (structCounter)Marshal.PtrToStructure(MyGC.AddrOfPinnedObject(), typeof(structCounter));
        if (Counter.CounterIndex == 0) { IOState.CounterValue0 = Counter.CounterValue; }
        if (Counter.CounterIndex == 1) { IOState.CounterValue1 = Counter.CounterValue; }
      break;
      case 0xB5:
        structAllCounter AllCounter = (structAllCounter)Marshal.PtrToStructure(MyGC.AddrOfPinnedObject(), typeof(structAllC
        IOState.CounterValue0 = AllCounter.CounterValue0;
        IOState.CounterValue1 = AllCounter.CounterValue1;
      break;
    }
    IOupdatedisplay();
    MyGC.Free();
  }
}
```

For thread-safe refreshing of the display the process iamge of the IOs is sent using the structure *IOState* to the procedure *IOupdatedisplay*. Invoke and delegate forming are used to adjust the affected display elements to the IO status.

```
private void IOupdatedisplay()
{
  if (InvokeRequired)
  {
    BeginInvoke(new delegateSub(IOupdatedisplay), new object[] { });
    return;
  }
  if ((IOState.OutputState & 1) == 1)
    cb_Output0.Checked = true;
  else
    cb_Output0.Checked = false;
  if ((IOState.OutputState & 2) == 2)
    cb_Output1.Checked = true;
  else
    cb_Output1.Checked = false;
  if ((IOState.InputState & 1) == 1)
    cb_Input0.Checked = true;
  else
    cb_Input0.Checked = false;
  if ((IOState.InputState & 2) == 2)
    cb_Input1.Checked = true;
  else
    cb_Input1.Checked = false;
    tb_Counter0.Text = IOState.CounterValue0.ToString();
    tb_Counter1.Text = IOState.CounterValue1.ToString();
}
```

Polling

Cyclical polling of particular values
In order to enable automatic refreshing of the display, a timer is used.

Depending on the check boxes for output, input and counter polling, the corresponding information is obtained from the Web-IO at a set interval.

```
private void timer_handler(object sender, System.EventArgs e)
{
  if (sender == counter) bt_Readall_Counter_Click(sender, e);
  if (sender == outputs) bt_Readall_Outputs_Click(sender, e);
  if (sender == inputs) bt_Readall_Inputs_Click(sender, e);
}

private void cb_Polling_Counter_CheckedChanged(object sender, System.EventArgs e)
{
  if(cb_Polling_Counter.Checked)
  {
    counter = new System.Windows.Forms.Timer();
    counter.Interval = intervall;
    counter.Start();
    counter.Tick += new EventHandler(timer_handler);
  }
 else
   counter.Stop();
}

private void cb_Polling_Outputs_CheckedChanged(object sender, System.EventArgs e)
{
  if(cb_Polling_Outputs.Checked)
  {
    outputs = new System.Windows.Forms.Timer();
    outputs.Interval = intervall;
    outputs.Start();
    outputs.Tick += new EventHandler(timer_handler);
  }
  else
    outputs.Stop();
}

private void cb_Polling_Inputs_CheckedChanged(object sender, System.EventArgs e)
{
  if(cb_Polling_Inputs.Checked)
  {
    inputs = new System.Windows.Forms.Timer();
    inputs.Interval = intervall;
    inputs.Start();
    inputs.Tick += new EventHandler(timer_handler);
  }
  else
    inputs.Stop();
}
```

The desired interval can be entered in the corresponding text field. When changes are made the timer interval is automatically adjusted.

```
private void tb_Intervall_TextChanged(object sender, System.EventArgs e)
{
  try
  {
    if(Convert.ToInt32(tb_Intervall.Text) > 0)
    {
      intervall = Convert.ToInt32(tb_Intervall.Text);
      statusBar.Text = "New range: " + intervall.ToString() + " ms!";
    }
    else
      statusBar.Text = "Only positiv Integer allowed!";
  }
  catch(Exception)
  {
    statusBar.Text = "Only positiv Integer allowed!";
  }
}
```

The sample program supports the common functions of the Web-IO in binary socket mode, optimized for the Web-IO 2x Digital Input, 2x Digital Output. For the other Web-IO models you may have to make changes to the program. A detailed description of the binary structures can be found in the binary brief overview or in the programming manual for the Web-IO.

Download program example

Products

Web-IO 4.0 Digital
2xIn, 2xOut

Power via PoE also when needed

Web-IO 4.0 Digital
12xIn, 12xOut

12x inputs,
12x outputs

Other Web-IOs

All W&T Web-IO Digital 24V

W&T
www.WuT.de

We are available to you in person:

Wiesemann & Theis GmbH
Porschestr. 12
42279 Wuppertal
Phone: +49 202/2680-110 (Mon.-Fri. 8 a.m. to 5 p.m.)
Fax: +49 202/2680-265
info@wut.de

Data Privacy