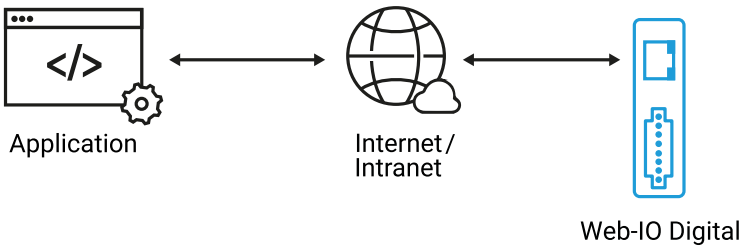
Tutorial for the Web-IO Digital:

Control and monitor Web-IO Digital with Visual C++

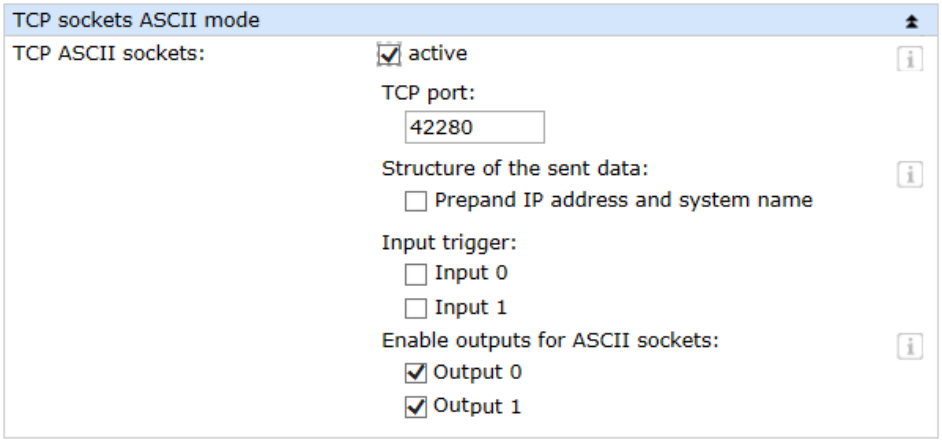
Visual C++ is one of the most used development platform for creating Windows applications.



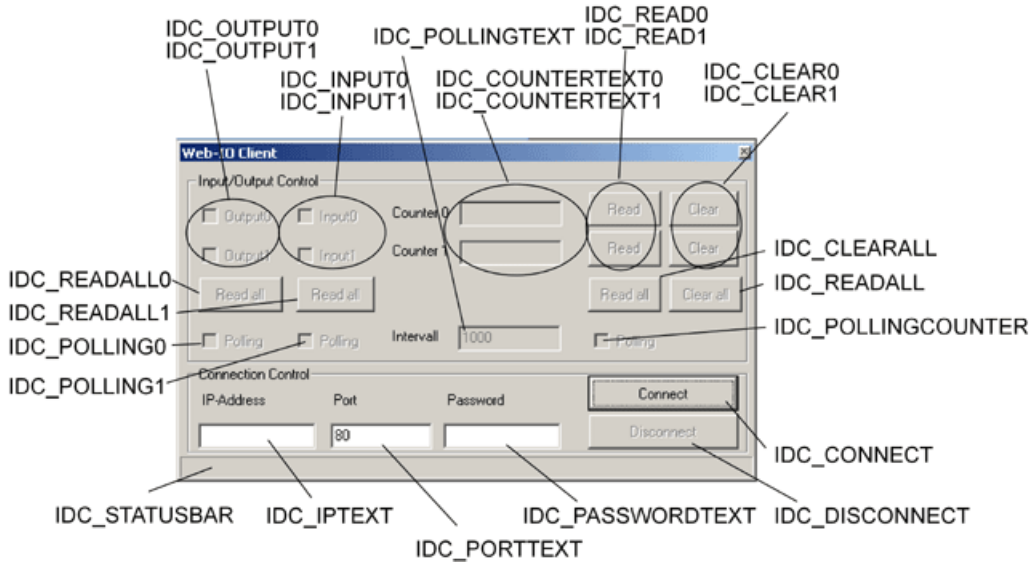
Using the following C++ program example you can represent your Web-IO Digital with its inputs and outputs in a Windows application. You can also switch the outputs on the Web-IO.

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 *TCP-ASCII sockets* and *enable outputs for switching*



Assemble the various operating elements and display objects in the Visual C++ form



Definition in the header file of MySocket and Cweb_IO_ClientDlg

To save the entries in the fields IDC_IPTEXT, IDC_PORTTEXT, IDC_PASSWORDTEXT and IDC_POLLINGTEXT in private member variables of the class CWeb_IO_ClientDlg, several variables are defined in the header file.

So as not to generate a new variable for each new sent and received message, private variables are also generated in the header. The entry in the status bar also gets a variable at this point.

```
private:
CString m_ip;
CString m_message1;
CString m_message2;
CString m_rcv;
CString m_password;
int m_port;
int m_range;
public:
CStatusBarCtrl*
m_statusBar;
```

Likewise some needed methods are pre-declared in the header.

```
public:
void changeAccess(int i);
void TryConnect();
void OnConnect();
void OnDisconnect();
void checkpass();
void OnReceive();
void OnSend();
void OnClose();
void OnTimer(UINT nIDEvent);
void readAndClearCounter(CString data)
```

To initialize a connection now, we still need a class which takes care of sending and receiving data. Here we have created a file named MySocket which inherits from the CAsyncSocket class. In this way all public methods for this class are passed on to our class and declared again in the header. To be able to access the methods for the CWeb_IO_ClientDlg class later, an object is declared in the header.

```
public:
CWeb_IO_ClientDlg*
m_client;

public:
virtual void OnReceive(int nErrorCode);
virtual void OnSend(int nErrorCode);
virtual void OnClose(int nErrorCode);
```

Definition in MySocket.cpp

Since the MySocket class has inherited from the CAsyncSocket class, some of the needed methods for a running connection are superimposed. In this way the application can later both send and receive data asynchronously and respond to scheduling of the connection to Web-IO pages.

```
void MySocket::OnReceive(int nErrorCode)
{
  m_client->OnReceive();
  CAsyncSocket::OnReceive(nErrorCode);
}

void MySocket::OnSend(int nErrorCode)
{
  m_client->OnSend();
  CAsyncSocket::OnSend(nErrorCode);
}

void MySocket::OnClose(int nErrorCode)
{
  m_client->OnClose();
  CAsyncSocket::OnClose(nErrorCode);
}

void MySocket::OnConnect(int nErrorCode);
{
  if(nErrorCode == 0)
  {
    m_client->OnConnect();
    CAsyncSocket::OnConnect(nErrorCode);
  }
  else
  {
    m_client->OnDisconnect();
    m_client->m_statusBar->SetText("Error while connecting!", 0, 0);
  }
}
```

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.

To output messages in a status bar, one is created in this method. In addition the MySocket class is transferred, so that both are linked to each other.

To link variables or methods to the application components, you can use the Visual C++ ClassWizard, and for Visual Studio 2003 and higher the corresponding functionality in the class properties.

```
BOOL CWeb_IO_ClientDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  // weitere Implementierungen...

  m_statusBar = new CStatusBarCtrl;
  m_statusBar->Create(WS_CHILD|WS_VISIBLE, CRect(0, 0, 0, 0), this, 0);
  clientSocket.m_client = this;
  changeAccess(FALSE);
  SendMessage(DM_SETDEFID,
  IDC_CONNECT);

  return TRUE;
}

void CWeb_IO_ClientDlg::changeAccess(bool b)
{
  if(b == TRUE)
  {
    GetDlgItem(IDC_CONNECT)->EnableWindow(FALSE);
    GetDlgItem(IDC_DISCONNECT)->EnableWindow(TRUE);
    GetDlgItem(IDC_OUTPUT0)->EnableWindow(TRUE);
    GetDlgItem(IDC_OUTPUT1)->EnableWindow(TRUE);
    GetDlgItem(IDC_READALL0)->EnableWindow(TRUE);
    GetDlgItem(IDC_READALL1)->EnableWindow(TRUE);
    GetDlgItem(IDC_POLLING0)->EnableWindow(TRUE);
    GetDlgItem(IDC_POLLING1)->EnableWindow(TRUE);
    GetDlgItem(IDC_POLLINGTEXT)->EnableWindow(TRUE);
    GetDlgItem(IDC_POLLINGCOUNTER)->EnableWindow(TRUE);
    GetDlgItem(IDC_READ0)->EnableWindow(TRUE);
    GetDlgItem(IDC_READ1)->EnableWindow(TRUE);
    GetDlgItem(IDC_CLEAR0)->EnableWindow(TRUE);
    GetDlgItem(IDC_CLEAR1)->EnableWindow(TRUE);
    GetDlgItem(IDC_READALL)->EnableWindow(TRUE);
    GetDlgItem(IDC_CLEARALL)->EnableWindow(TRUE);
  }
  else
  {
    GetDlgItem(IDC_CONNECT)->EnableWindow(TRUE);
    GetDlgItem(IDC_DISCONNECT)->EnableWindow(FALSE);
    GetDlgItem(IDC_OUTPUT0)->EnableWindow(FALSE);
    GetDlgItem(IDC_OUTPUT1)->EnableWindow(FALSE);
    GetDlgItem(IDC_INPUT0)->EnableWindow(FALSE);
    GetDlgItem(IDC_INPUT1)->EnableWindow(FALSE);
    GetDlgItem(IDC_READALL0)->EnableWindow(FALSE);
    GetDlgItem(IDC_READALL1)->EnableWindow(FALSE);
    GetDlgItem(IDC_POLLING0)->EnableWindow(FALSE);
    GetDlgItem(IDC_POLLING1)->EnableWindow(FALSE);
    GetDlgItem(IDC_POLLINGTEXT)->EnableWindow(FALSE);
    GetDlgItem(IDC_POLLINGCOUNTER)->EnableWindow(FALSE);
    GetDlgItem(IDC_COUNTERTEXT0)->EnableWindow(FALSE);
    GetDlgItem(IDC_COUNTERTEXT1)->EnableWindow(FALSE);
    GetDlgItem(IDC_READ0)->EnableWindow(FALSE);
    GetDlgItem(IDC_READ1)->EnableWindow(FALSE);
    GetDlgItem(IDC_CLEAR0)->EnableWindow(FALSE);
    GetDlgItem(IDC_CLEAR1)->EnableWindow(FALSE);
    GetDlgItem(IDC_READALL)->EnableWindow(FALSE);
    GetDlgItem(IDC_CLEARALL)->EnableWindow(FALSE);
  }
}
```

Connection control

Establishing the connection

By entering the IP address of the Web-IO in the text field IDC_IPTEXT and Port 80 in the text field IDC_PORTTEXT you can use the IDC_CONNECT button to establish a connection. If no IP address or port is entered, a message is displayed in the status bar.

```
void CWeb_IO_ClientDlg::OnClickConnect()
{
  GetDlgItemText(IDC_IPTEXT, m_ip);
  GetDlgItemText(IDC_PORTTEXT, m_message2);
  sscanf(m_message2, "%d", &m_port);

  if(m_ip == "")
  {
    m_statusBar->SetText("No IP entered!", 0, 0);
  }
  else if(m_port == 0)
  {
    m_statusBar->SetText("No port entered!", 0, 0);
  }
  else
  {
    TryConnect();
  }
}
```

Opening the connection

Now to be able to open a TCP connection, we use in our example a global object in the MySocket class which provides methods such as Create(), Connect() and Close().

If a negative port was entered or the connection fail to open, a corresponding message is output.

Since the connection could not be opened, the socket is closed again so that a new connection can be started with no problem.

If everything went as expected, all the components which can be used after successful connection opening are enabled and the IDC_CONNECT button is disabled to prevent opening another connection while one is already open.

```
void CWeb_IO_ClientDlg::TryConnect()
{
  if(m_port < 0)
  {
    m_statusBar->SetText("No
    negative port allowed!", 0, 0);
    return;
  }
  clientSocket.Create();
  clientSocket.Connect(m_ip, m_port);
  GetDlgItem(IDC_CONNECT)->EnableWindow(FALSE);
}
```

Connection is made

If the connection is made, the application can carry out communication with the Web-IO, using the various components.

```
void CWeb_IO_ClientDlg::OnConnect()
{
  m_statusBar->SetText("Connection made!", 0, 0);
  changeAccess(TRUE);
}
```

Disconnecting

The connection remains open until it is ended by the user clicking on the Disconnect button, or the Web-IO ends the connection. If the connection is ended, a message is output.

```
void CWeb_IO_ClientDlg::OnClickDisconnect()
{
  OnDisconnect();
}
```

If the connection is ended, all elements must be transformed back to their initial setting, since it should no longer be possible to click on the IDC_DISCONNECT button.

```
void CWeb_IO_ClientDlg::OnDisconnect()
{
  clientSocket.Close();
  m_statusBar->SetText("Disconnected!", 0, 0);
  changeAccess(FALSE);
}
```

Now the connection is closed again and the application reset to its initial state.

Operation and communication from the client side

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

Setting the outputs

The outputs on the Web-IO can be switched using the two checkboxes IDC_OUTPUT0 and IDC_OUTPUT1. The checkbox notes when it was clicked and then triggers an action. To be able to send a request to the Web-IO, a check is first made to see whether a password has been entered in the text field IDC_PASSWORDTEXT, without which the Web-IO would be unable to accept any requests. If no password has been entered, the request is sent without a password.

```
void CWeb_IO_ClientDlg::checkpass()
{
  GetDlgItemText(IDC_PASSWORDTEXT, m_password);
}
```

In the next step a check is made to see whether the checkbox is already set or not, whereby the output is either reset or set.

```
void CWeb_IO_ClientDlg::OnOutput0()
{
  checkpass();
  m_message1 = "GET /outputaccess0?PW=" + password + "&State=ON&";
  m_message2 = "GET /outputaccess0?PW=" + password + "&State=OFF&";
  if(IsDlgButtonChecked(IDC_OUTPUT0))
    clientSocket.Send(m_message1, m_message1.GetLength());
  else
    clientSocket.Send(m_message2, m_message2.GetLength());
}

void CWeb_IO_ClientDlg::OnOutput1()
{
  checkpass();
  m_message1 = "GET /outputaccess1?PW=" + password + "&State=ON&";
  m_message2 = "GET /outputaccess1?PW=" + password + "&State=OFF&";
  if(IsDlgButtonChecked(IDC_OUTPUT1))
    clientSocket.Send(m_message1, m_message1.GetLength());
  else
    clientSocket.Send(m_message2, m_message2.GetLength());
}
```

Query output/input status

Now the state of the outputs and inputs can also be checked to update the control boxes in the application. The IDC_READLL0 button reads the state of the outputs, and the IDC_READALL1 button reads the state of the inputs. The command for status query is slightly different from that for setting. You can use "GET /output?PW=&" to query the status of the outputs.

```
void CWeb_IO_ClientDlg::OnClickReadall0()
{
  checkpass();
  m_message1 = "GET /output?PW=" + password + "&";
  clientSocket.Send(m_message1, m_message1.GetLength());
}

void CWeb_IO_ClientDlg::OnClickReadall1()
{
  checkpass();
  m_message1 = "GET /input?PW=" + password + "&";
  clientSocket.Send(m_message1,
  m_message1.GetLength());
}
```

Query counters

Since arriving input events are noted only in the Web-IO itself, incrementing its inner counter, it should also be possible to query it. The following method sends a query to a particular counter and requests a reply with the current status of the counter. From the reply from the Web-IO the number of the respective counter and its count state can be obtained and displayed in the application.

```
void CWeb_IO_ClientDlg::OnClickRead0()
{
  checkpass();
  m_message1 = "GET /counter0?PW=" + password + "&";
  clientSocket.Send(m_message1, m_message1.GetLength());
}

void CWeb_IO_ClientDlg::OnClickRead1()
{
  checkpass();
  m_message1 = "GET /counter1?PW=" + password + "&";
  clientSocket.Send(m_message1, m_message1.GetLength());
}
```

Of course all the counter states can be queried using a single command.

```
void CWeb_IO_ClientDlg::OnClickReadall()
{
  checkpass();
  m_message1 = "GET /counter?PW=" + password + "&";
  clientSocket.Send(m_message1, m_message1.GetLength());
}
```

Reset counter

Since you can read the count state, it should also be possible to set the counter to 0. To this end a message is sent to the respective counter which resets it.

```
void CWeb_IO_ClientDlg::OnClickClear0()
{
  checkpass();
  m_message1 = "GET /counterclear0?PW=" + password + "&";
  clientSocket.Send(m_message1, m_message1.GetLength());
}

void CWeb_IO_ClientDlg::OnClickClear1()
{
  checkpass();
  m_message1 = "GET /counterclear1?PW=" + password + "&";
  clientSocket.Send(m_message1, m_message1.GetLength());
}
```

Of course all the counters can be reset using a single command.

```
void CWeb_IO_ClientDlg::OnClickClearall()
{
  checkpass();
  m_message1 = "GET /counterclear?PW=" + password + "&";
  clientSocket.Send(m_message1, m_message1.GetLength());
}
```

Since all counter states can be read or reset using one command, there must be a method implemented which processes the reply string from the Web-IO and assigns each counter in the application its specific state.

```
void CWeb_IO_ClientDlg::readAndClearCounter(CString data)
{
  int j = 0;
  CString counter[12];

  for(int i = 0; i < data.GetLength(); i++)
  {
    if(data[i] == ';')
      j++;
    else
      counter[j]+= data[i];
  }

  SetDlgItemText(IDC_COUNTERTEXT0, counter[0]);
  SetDlgItemText(IDC_COUNTERTEXT1, counter[1]);
}
```

Receiving data from the Web-IO

Process and display the received data

- All commands and requests to the Web-IO are acknowledged with a reply string. The replies have a specific structure depending on the type.
- A distinction is made between the reply from a single output or input and a replay from all outputs and inputs.
- For the outputs: output;<binary value of the output status in hexadecimal format>
- For a special output: outputx;<ON or OFF>
- For the inputs: input;<binary value of the input status in hexadecimal format>
- For a special input inputx;<ON or OFF>
- There is also a reply string for a counter, which looks as follows:
  counterx;<decimal counter state>
  or counter;<decimal counter state 0 >; <decimal counter state 1 >; ... if you want to read all the counters at one time.
  All reply strings are finished off with a 0 byte.
- In our application the method OnReceiver() is invoked to receive such a message. In this method the reply from the Web-IO is processed.

```
void CWeb_IO_ClientDlg::OnReceive()
{
  clientSocket.Receive(m_rcv.GetBuffer(50),50);
  m_rcv.ReleaseBuffer();

  if(m_rcv.IsEmpty())
    OnDisconnect();
  else if(m_rcv.GetLength() > 1)
  {
    if(m_rcv[0] == 'o')
    {
      int i;
      sscanf(m_rcv.Right(m_rcv.GetLength() - 7), "%x", &i);
      if((i & 1) == 1)
        CheckDlgButton(IDC_OUTPUT0, BST_CHECKED);
      else
        CheckDlgButton(IDC_OUTPUT0, BST_UNCHECKED);
      if((i & 2) == 2)
        CheckDlgButton(IDC_OUTPUT1, BST_CHECKED);
      else
        CheckDlgButton(IDC_OUTPUT0, BST_UNCHECKED);
    }
    if(m_rcv[0] == 'i')
    {
      int i;
      sscanf(m_rcv.Right(m_rcv.GetLength() - 6), "%x", &i);
      if((i & 1) == 1)
        CheckDlgButton(IDC_INPUT0, BST_CHECKED);
      else
        CheckDlgButton(IDC_INPUT0, BST_UNCHECKED);
      if((i & 2) == 2)
        CheckDlgButton(IDC_INPUT1, BST_CHECKED);
      else
        CheckDlgButton(IDC_INPUT1, BST_UNCHECKED);
    }
    if(m_rcv[0] == 'c')
    {
      if(m_rcv[7] == '0')
        SetDlgItemText(IDC_COUNTERTEXT0, m_rcv.Right(m_rcv.GetLength() - 9));
      if(m_rcv[7] == '1')
        SetDlgItemText(IDC_COUNTERTEXT1, m_rcv.Right(m_rcv.GetLength() - 9));
      if(m_rcv[7] == ';')
        readAndClearCounter(m_rcv.Right(m_rcv.GetLength() - 8));
    }
  }
}
```

Polling

Cyclical polling of particular values
Now it is desirable to have individual components update their status themselves, so that the application always indicates the latest status. For this the program uses a timer which sends cyclical queries to the Web-IO at a user-defined time interval.

First, an integer value can be entered in the field IDC_POLLINGTEXT, which defines the time in milliseconds for the cyclical query. If no value is entered, the time for the interval is set to a standard 1 second (1000 ms).

Of course this also catches cases where the user enters a nonsense value, such as a negative time value. This is immediately followed by a message and the value is not applied.

```
void CWeb_IO_ClientDlg::OnChangePollingtext()
{
  GetDlgItemText(IDC_POLLINGTEXT, m_message2);
  int check;
  sscanf(m_message2, "%d", &check);
  if(check <= 0)
  {
    m_statusBar->SetText("No negative
    value or character allowed!", 0, 0);
    return;
  }
  m_range = check;
  m_statusBar->SetText("Range
  changed!", 0, 0);
}
```

To now carry out cyclical polling of the Web-IO states, the choose from between polling the outputs, the inputs or the counters.

For each polling variant a timer is initialized. The timer is invoked with "SetTimer(timer number, interval, NULL)".

Activating the checkbox IDC_POLLING0 then means the outputs are polled. A timer is initialized if it has been set. If the checkbox is again deactivated, the timer with the corresponding number is cleared.

```
void CWeb_IO_ClientDlg::OnPolling0()
{
  if(IsDlgButtonChecked(IDC_POLLING0))
    SetTimer(1, m_range, NULL);
  else
    KillTimer(1);
}
```

Clicking the IDC_POLLING1 checkbox causes the inputs to be polled. A new timer is initialized which can be cleared again using this method.

```
void CWeb_IO_ClientDlg::OnPolling1()
{
  if(IsDlgButtonChecked(IDC_POLLING1))
    SetTimer(2, m_range, NULL);
  else
    KillTimer(2);
}
```

To query the counters as well using polling, you can use the IDC_POLLINGCOUNTER checkbox.

```
void CWeb_IO_ClientDlg::OnPollingcounter()
{
  if(IsDlgButtonChecked(IDC_POLLINGCOUNTER))
    SetTimer(3,m_range, NULL);
  else
    KillTimer(3);
}
```

Three different timers were initialized which trigger an action, at certain time intervals depending on the setting, which up until now is not captured. To capture the events a method still needs to be implemented.

In this method the respective event of the timer which is currently reporting is captured and assigned to a particular action.

Recall: In our case the states of the outputs, the inputs and the counters are cyclically polled depending on the event.

```
void CWeb_IO_ClientDlg::OnTimer(UINT nIDEvent)
{
  if(nIDEvent == 1) OnClickReadall0();
  if(nIDEvent == 2) OnClickReadall1();
  if(nIDEvent == 3) OnClickReadall();
  CDialog::OnTimer(nIDEvent);
}
```

The entire program is made available on this page, containing some files and methods which are not mentioned here. The

MFC application under Visual C++ generates an overhead which forms the graphical frame but which is not directly connected to the functionality. The methods and variables which you are shown on this page are implemented in the frames generated by Visual C++, and then work as a unit with the graphical interface.

The sample program supports all common functions of the Web-IO in command string mode, optimized for the Web-IO 2x Digital Input, 2x Digital Output PoE. For the other Web-IO models you may have to adapt the program. Additional program examples for socket programming can be found on the tool pages for the Web-IO. A detailed description for the socket interface of the Web-IO Digital models can be found in the reference manual.

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