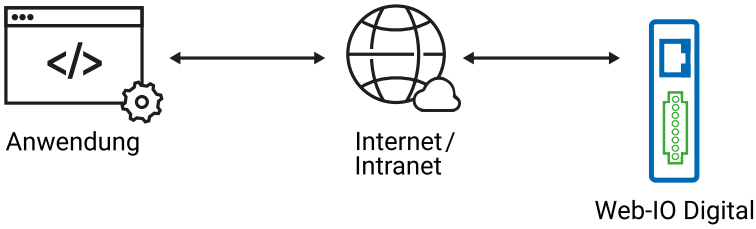
Applikation zum Web-IO Digital:

Web-IO Digital mit Java steuern und überwachen

Produktübersicht
Applikationsübersicht

Java ist eine weit verbreitete, einfach zu erlernende und plattformunabhängige Programmiersprache. Sie beinhaltet bereits in der Standardversion sämtliche Klassen und Methoden, die für die Programmierung von TCP/IP-Anwendungen benötigt werden. Das zeichnet Java als beliebtes Werkzeug aus, um Anwendungen zu erstellen, die mit dem Web-IO Digital kommunizieren.

Neben einer aktuellen Java-Umgebung werden keine weiteren Resourcen für die Ausführung benötigt.

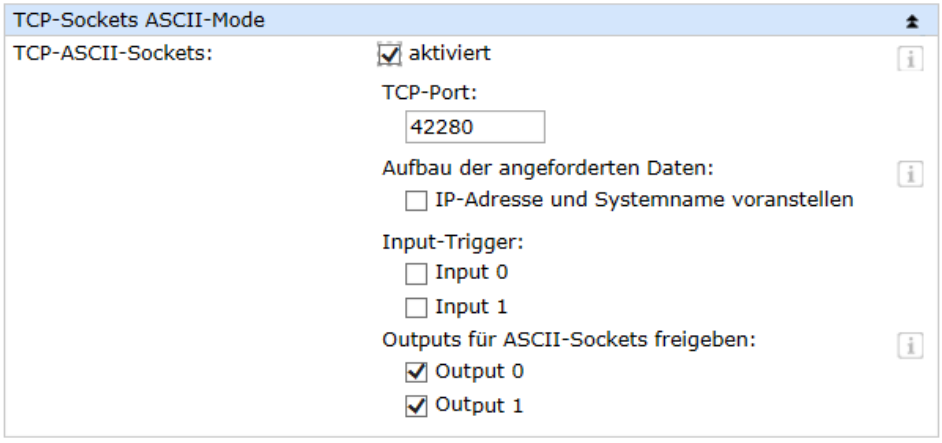


Das folgende Programmbeispiel ermöglicht das Überwachen und Steuern eines Web-IO 2x Digital und stellt dafür folgende Funktionen zur Verfügung:

- Schalten der Outputs
- Manuelles oder automatisches Lesen von Inputs, Outputs und Counters
- Lesen und Löschen einzelner oder aller Counter

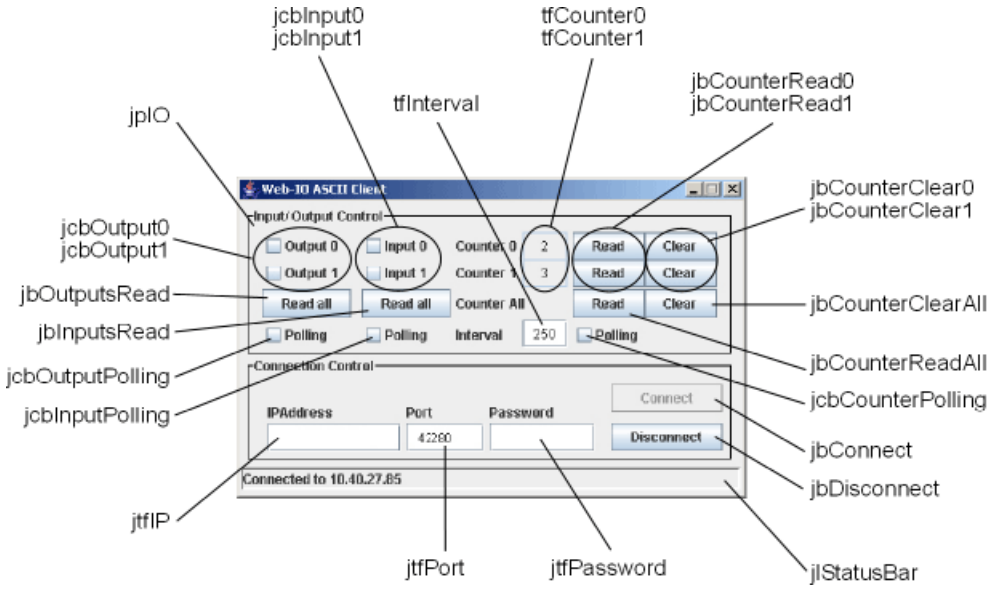
Vorbereitungen

- Web-IO mit Spannung versorgen und IOs verdrahten
- Web-IO mit dem Netzwerk verbinden
- IP-Adressen vergeben
- Beim Web-IO im Bereich *Kommunikationswege >> Socket-API* die *TCP-ASCII-Sockets* aktivieren und die *Outputs zum Schalten freigeben*



Zusammenstellung der verschiedenen Bedien- und Anzeigeelemente

In der folgenden Abbildung sind die unterschiedlichen Bedien- und Anzeigenelemente mit dem im Programmcode verwendeten Namen betitelt. Die Zuordnung soll als Referenz dienen und das Nachvollziehen des nachfolgenden Beispiels erleichtern.



Bei der Benennung der einzelnen Objekte ist es hilfreich, sinngebende Namen zu verwenden. In diesem Beipiel beschreibt der erste Teil des Namens die Art des Objektes und der zweite Teil die Funktion.

Import der benötigten Sourcen

Zu Beginn eines jeden Programms müssen alle benötigten Quellen importiert werden.

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.StringTokenizer;

import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.Timer;
import javax.swing.border.BevelBorder;
```

2. Erweiterungen der Klasse

Die Klasse "Client" erbt alle Eigenschaften der Klasse "JFrame", um das Programmfenster darzustellen, und bindet die Interfaces "ActionListener" (Erkennung von Buttonereignissen), "KeyListener" (Erkennung von Änderungen in Textfeldern) und "Runnable" (ermöglichen von Threads) durch Import deren Methoden ein.

```
public class Client extends JFrame implements ActionListener, KeyListener, Runnable {
  ...
}
```

Instanziierung der grafischen Elemente

Für die grafischen Elemente (Buttons, CheckBoxes, Label, Panel, TextFields) werden globale Instanzen erzeugt, damit sie in jeder Methode des Programms zur Verfügung stehen.

```
  private JButton jbConnect = new JButton("Connect");
  private JButton jbDisconnect = new JButton("Disconnect");
  private JButton jbOutputsRead = new JButton("Read all");
  private JButton jbInputsRead = new JButton("Read all");
  private JButton jbCounterRead0 = new JButton("Read");
  private JButton jbCounterRead1 = new JButton("Read");
  private JButton jbCounterReadAll = new JButton("Read");
  private JButton jbCounterClear0 = new JButton("Clear");
  private JButton jbCounterClear1 = new JButton("Clear");
  private JButton jbCounterClearAll = new JButton("Clear");
  private JCheckBox jcbCounterPolling = new JCheckBox("Polling");
  private JCheckBox jcbOutput0 = new JCheckBox("Output 0");
  private JCheckBox jcbOutput1 = new JCheckBox("Output 1");
  private JCheckBox jcbOutputPolling = new JCheckBox("Polling");
  private JCheckBox jcbInput0 = new JCheckBox("Input 0");
  private JCheckBox jcbInput1 = new JCheckBox("Input 1");
  private JCheckBox jcbInputPolling = new JCheckBox("Polling");
  private JLabel jlStatusBar = new JLabel("No connection");
  private JPanel jpIO = new JPanel();
  private JTextField jtfInterval = new JTextField("250");
  private JTextField jtfCounter0 = new JTextField("0");
  private JTextField jtfCounter1 = new JTextField("0");
  private JTextField jtfIP = new JTextField();
  private JTextField jtfPort = new JTextField("80");
  private JTextField jtfPassword = new JTextField();
```

Elemente für die Datenübertragung und das Zeitverhalten

Für die Datenübertragung über eine Socket-Schnittstelle werden in diesem Beispiel Instanzen der Klassen "InputStream", "OutputStream" und "Socket" verwendet.

```
  private InputStream isInStream;
  private OutputStream osOutStream;
  private Socket soClientSocket;
```

Der Timer, der das Polling steuert, ist von der Klasse "Timer" (javax.swing.Timer) abgeleitet.

```
private Timer tiPoll;
```

Die "main"-Methode

Die "main"-Methode wird automatisch beim Programmstart ausgerufen. Sie ruft den Konstruktor der Klasse auf.

```
        public static void main(String[] args) {
      new Client();
    }
```

Der Konstruktor

Im Konstruktor werden Größe, Position und Verhalten von Anzeigeelementen und dem Programmfenster definiert. Der Konstruktoraufruf erfolgt ohne die Übergabe von Parametern.

```
        public Client() {
      ...
    }
```

Im Folgenden wird das JPanel "jpIO" erstellt. Dafür werden die Anzeigeelement dimensioniert, ggf. inaktiv geschaltet, um den Ausgangszustand zu erzeugen und mit einem Listener versehen, damit auf Eingaben vom Benutzer reagiert werden kann. Abschließend werden die Elemente dem JPanel hinzugefügt.

```
jcbOutput0.setBounds(15, 25, 80, 20);
jcbOutput0.setEnabled(false);
jcbOutput0.addActionListener(this);
jcbOutput1.setBounds(15, 50, 80, 20);
jcbOutput1.setEnabled(false);
jcbOutput1.addActionListener(this);
jbOutputsRead.setBounds(15, 75, 80, 25);
jbOutputsRead.setEnabled(false);
jbOutputsRead.addActionListener(this);
jcbOutputPolling.setBounds(15, 105, 80, 20);
jcbOutputPolling.setEnabled(false);
jcbInput0.setBounds(105, 25, 80, 20);
jcbInput0.setEnabled(false);
jcbInput0.addActionListener(this);
jcbInput1.setBounds(105, 50, 80, 20);
jcbInput1.setEnabled(false);
jcbInput1.addActionListener(this);
jbInputsRead.setBounds(105, 75, 80, 25);
jbInputsRead.setEnabled(false);
jbInputsRead.addActionListener(this);
jcbInputPolling.setBounds(105, 105, 80, 20);
jcbInputPolling.setEnabled(false);
JLabel jlCounter0 = new JLabel("Counter 0");
jlCounter0.setBounds(190, 25, 55, 20);
jlCounter0.setEnabled(false);
JLabel jlCounter1 = new JLabel("Counter 1");
jlCounter1.setBounds(190, 50, 55, 20);
jlCounter1.setEnabled(false);
JLabel jlCounterAll = new JLabel("Counter All");
jlCounterAll.setBounds(190, 77, 70, 20);
jlCounterAll.setEnabled(false);
JLabel jlInterval = new JLabel("Interval");
jlInterval.setBounds(190, 105, 50, 20);
jlInterval.setEnabled(false);
tfCounter0.setBounds(250, 23, 40, 25);
jtfCounter0.setEnabled(false);
jtfCounter0.setHorizontalAlignment(JTextField.CENTER);
jtfCounter0.setEditable(false);
jtfCounter1.setBounds(250, 48, 40, 25);
jtfCounter1.setEnabled(false);
jtfCounter1.setHorizontalAlignment(JTextField.CENTER);
jtfCounter1.setEditable(false);
jtfInterval.setBounds(250, 102, 40, 25);
jtfInterval.setEnabled(false);
jtfInterval.setHorizontalAlignment(JTextField.CENTER);
jtfInterval.addKeyListener(this);
jbCounterRead0.setBounds(295, 23, 65, 25);
jbCounterRead0.setEnabled(false);
jbCounterRead0.addActionListener(this);
jbCounterRead1.setBounds(295, 48, 65, 25);
jbCounterRead1.setEnabled(false);
jbCounterRead1.addActionListener(this);
jbCounterReadAll.setBounds(295, 75, 65, 25);
jbCounterReadAll.setEnabled(false);
jbCounterReadAll.addActionListener(this);
jcbCounterPolling.setBounds(295, 105, 80, 20);
jcbCounterPolling.setEnabled(false);
jbCounterClear0.setBounds(360, 23, 65, 25);
jbCounterClear0.setEnabled(false);
jbCounterClear0.addActionListener(this);
jbCounterClear1.setBounds(360, 48, 65, 25);
jbCounterClear1.setEnabled(false);
jbCounterClear1.addActionListener(this);
jbCounterClearAll.setBounds(360, 75, 65, 25);
jbCounterClearAll.setEnabled(false);
jbCounterClearAll.addActionListener(this);
jpIO.setLayout(null);
jpIO.setBounds(5, 5, 440, 135);
jpIO.setBorder(BorderFactory.createTitledBorder(BorderFactory.createLineBorder(Color.black), "Input/ Output Control"));
jpIO.add(jcbOutput0);
jpIO.add(jcbOutput1);
jpIO.add(jbOutputsRead);
jpIO.add(jcbOutputPolling);
jpIO.add(jcbInput0);
jpIO.add(jcbInput1);
jpIO.add(jbInputsRead);
jpIO.add(jcbInputPolling);
jpIO.add(jlCounter0);
jpIO.add(jlCounter1);
jpIO.add(jlCounterAll);
jpIO.add(jlInterval);
jpIO.add(jtfCounter0);
jpIO.add(jtfCounter1);
jpIO.add(jtfInterval);
jpIO.add(jbCounterRead0);
jpIO.add(jbCounterRead1);
jpIO.add(jbCounterReadAll);
jpIO.add(jcbCounterPolling);
jpIO.add(jbCounterClear0);
jpIO.add(jbCounterClear1);
jpIO.add(jbCounterClearAll);
```

Das folgende Programmfragment erstellt die Bedien- und Anzeigeelemente, die auf der Oberfläche unter dem Begriff "Connection Control" zusammengefasst sind. Das JPanel, dem diese Elemente hinzugefügt werden, ist lokal instanziiert, da außerhalb des Konstruktors nicht mehr darauf zugegriffen wird.

```
JLabel jlIP = new JLabel("IPAddress");
jlIP.setBounds(20, 40, 120, 20);
jtfIP.setBounds(20, 60, 120, 26);
jtfIP.setHorizontalAlignment(JTextField.CENTER);
JLabel jlPort = new JLabel("Port");
jlPort.setBounds(145, 40, 70, 20);
jtfPort.setBounds(145, 60, 70, 26);
jtfPort.setHorizontalAlignment(JTextField.CENTER);
JLabel jlPassword = new JLabel("Password");
jlPassword.setBounds(220, 40, 95, 20);
jtfPassword.setBounds(220, 60, 95, 26);
jtfPassword.setHorizontalAlignment(JTextField.CENTER);
jbConnect.setBounds(330, 25, 100, 25);
jbConnect.addActionListener(this);
jbDisconnect.setBounds(330, 60, 100, 25);
jbDisconnect.setEnabled(false);
jbDisconnect.addActionListener(this);
JPanel jpConnect = new JPanel();
jpConnect.setLayout(null);
jpConnect.setBounds(5, 140, 440, 95);
jpConnect.setBorder(BorderFactory.createTitledBorder(BorderFactory.createLineBorder(Color.black), "Connection Control"));
jpConnect.add(jlIP);
jpConnect.add(jlPort);
jpConnect.add(jlPassword);
jpConnect.add(jtfIP);
jpConnect.add(jtfPort);
jpConnect.add(jtfPassword);
jpConnect.add(jbConnect);
jpConnect.add(jbDisconnect);
```

Die Statusleiste informiert während des Programmablaufs über den Status der Socket-Verbindung zu einem Web-IO. Bereits während der Instanziierung wurde ihr der Defaultwert "No Connection" zugewiesen. Im Konstruktor wird die Darstellung in Form von Größe, Position und Art des Rahmens definiert.

```
jlStatusBar.setLayout(null);
jlStatusBar.setBounds(1, 240, 450, 20);
jlStatusBar.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
```

Abschließend werden die Eigenschaften des Fensters bestimmt und die zuvor erstellten Elemente hinzugefügt. Als letzter Schritt wird das Programmfenster sichtbar gemacht. Die Initialisierungsphase ist damit abgeschlossen und das Programm ist nun betriebsbereit.

```
    setTitle("Web-IO ASCII Client");
    setLocation(400, 300);
    setLayout(null);
    setSize(460, 290);
    setResizable(false);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    add(jpIO);
    add(jpConnect);
    add(jlStatusBar);
    setVisible(true);
```

Button- und Timerereignisse verarbeiten

Ereignisverarbeitung (allgemein): Wurde bei Anzeige- und Bedienelementen (z. B. Buttons und Textfelder) ein ActionListener registriert, wird bei ausgelösten Ereignissen die Methode "actionPerformed" aufgerufen. Der übergebene Parameter enthält unter anderem die Information, welche Komponente das Ereignis ausgelöst hat.

```
public void actionPerformed(ActionEvent arg0) {
  ...
}
```

Verbindungsaufbau: Durch Betätigen der Schaltfläche "Connect" wird ein Verbindungsaufbau zu dem angegebenen Web-IO gestartet. Die Anzeige- und Bedienelemente werden dem Zustand entsprechend aktiv oder inaktiv geschaltet und, die Statusleiste gibt den Fortschritt des Verbindungsaufbaus wieder. Wurde der Socket erfolgreich geöffnet und die Streams für Dateneingabe und Datenausgabe erfolgreich generiert, startet ein Thread, der eintreffende Daten verarbeitet. Ein Timer, der entsprechend des geforderten Pollingverhaltens Daten anfordert, wird gestartet. Tritt während des Verbindungsaufbaus ein Fehler auf, erfolgt der Ausstieg über eine Exception.

```
if (arg0.getSource() == jbConnect) {
  jbConnect.setEnabled(false);
  jlStatusBar.setText("Trying to connect to " + jtfIP.getText());
  try {
    soClientSocket = new Socket(jtfIP.getText(),Integer.parseInt(jtfPort.getText()));
    isInStream = soClientSocket.getInputStream();
    osOutStream = soClientSocket.getOutputStream();
    new Thread(this).start();
    tiPoll = new Timer(Integer.parseInt(jtfInterval.getText()),this);
    tiPoll.start();
    for (int i = 0; i < jpIO.getComponentCount();i++) {
      ((JComponent) jpIO.getComponent(i)).setEnabled(true);
    }
    jbDisconnect.setEnabled(true);
    jlStatusBar.setText("Connected to " + jtfIP.getText());
    return;
  }
  catch (NumberFormatException e) {
    e.printStackTrace();
  }
  catch (UnknownHostException e) {
    e.printStackTrace();
  }
  catch (IOException e) {
    e.printStackTrace();
  }
  jbConnect.setEnabled(true);
  jlStatusBar.setText("Error - No Connection");
}
```

Gewollter Verbindungsabbau: Das Betätigen der Schaltfläche "Disconnect" ruft die Methode "disconnect" auf. Diese enthält Anweisungen, welche die Verbindung kontrolliert abbauen und das Programm für die Aufnahme einer neuen Verbindung vorbereiten. Da der Verbindungsabbau auch über eine Exception eingeleitet werden kann, erfolgt der Verbindungsabbau in einer separaten Methode und nicht innerhalb der Methode "actionPerformed".

```
else if (arg0.getSource() == jbDisconnect) {
  disconnect();
}
```

Outputs schalten: Durch Manipulation der Checkboxen "jcbOutput0" und "jcbOutput1" kann der jeweilige Ausgang geschaltet werden. Löst eine der Checkboxen ein Ereignis aus, wird in Abhängigkeit des gesetzten Status ein String über den Socket an das Web-IO gesendet, der die Selektion umsetzt.

```
else if (arg0.getSource() == jcbOutput0) {
  if (jcbOutput0.isSelected()) {
    write("GET /outputaccess0?PW=" + jtfPassword.getText() + "&State=ON&");
  }
  else {
    write("GET /outputaccess0?PW=" + jtfPassword.getText() + "&State=OFF&");
  }
}
else if (arg0.getSource() == jcbOutput1) {
  if (jcbOutput1.isSelected()) {
    write("GET /outputaccess1?PW=" + jtfPassword.getText() + "&State=ON&");
  }
  else {
    write("GET /outputaccess1?PW=" + jtfPassword.getText() + "&State=OFF&");
  }
}
```

Outputs und Inputs lesen, Counter lesen und löschen: Die Out- und Inputs können jeweils über eine Schaltfläche komplett gelesen werden. Dazu wird ein entsprechender Kommandostring an das Web-IO gesendet. Genauso verhält es sich mit den Countern. Jeder Schaltfläche ist ein Kommandostring hinterlegt, der das entsprechende Ergebnis auslöst.

```
else if (arg0.getSource() == jbOutputsRead) {
  write("GET /output?PW=" + jtfPassword.getText() + "&");
}
else if (arg0.getSource() == jbInputsRead) {
  write("GET /input?PW=" + jtfPassword.getText() + "&");
}
else if (arg0.getSource() == jbCounterRead0) {
  write("GET /counter0?PW=" + jtfPassword.getText() + "&");
}
else if (arg0.getSource() == jbCounterRead1) {
  write("GET /counter1?PW=" + jtfPassword.getText() + "&");
}
else if (arg0.getSource() == jbCounterReadAll) {
  write("GET /counter?PW=" + jtfPassword.getText() + "&");
}
else if (arg0.getSource() == jbCounterClear0) {
  write("GET /counterclear0?PW=" + jtfPassword.getText() + "&");
}
else if (arg0.getSource() == jbCounterClear1) {
  write("GET /counterclear1?PW=" + jtfPassword.getText() + "&");
}
else if (arg0.getSource() == jbCounterClearAll) {
  write("GET /counterclear?PW=" + jtfPassword.getText() + "&");
}
```

Polling: Der Timer, der mit dem Verbindungsaufbau gestartet wurde, löst ebenfalls zyklisch ein Ereignis aus. Je nach Status der drei Polling-Checkboxen werden bei jedem Zyklus die Outputs, die Inputs und die Counter angefragt.

```
else if (arg0.getSource() == tiPoll) {
  if (jcbOutputPolling.isSelected()) {
    write("GET /output?PW=" + jtfPassword.getText() + "&");
  }
  if (jcbInputPolling.isSelected()) {
    write("GET /input?PW=" + jtfPassword.getText() + "&");
  }
  if (jcbCounterPolling.isSelected()) {
    write("GET /counter?PW=" + jtfPassword.getText() + "&");
  }
 }
```

Änderungen in Textfeldern verarbeiten

Änderung des Pollingintervalls: Änderungen des Pollingintervalls werden ohne Bestätigung aktiv. Die Erkennung einer Änderung erfolgt mittels eines KeyListeners, der dem Textfeld "tfInterval" hinzufefügt wurde. Dazu muss die Klasse die Methoden des Interfaces ("keyPressed"," KeyReleased" und "keyTyped") implementieren. Bei Änderung des Inhalts des Textfelds wird in der Methode "keyPressed" der aktualisierte Wert in das int-Format konvertiert und an den Timer übergeben. Ist der Inhalt des Textfeldes nicht in eine Zahl zu konvertieren, erfolgt der Ausstieg über die Exception. Die Pollingrate wird in diesem Fall nicht geändert.

```
public void keyPressed(KeyEvent arg0) {
}
public void keyReleased(KeyEvent arg0) {
  try {
    tiPoll.setDelay(Integer.parseInt(jtfInterval.getText()));
    } catch (NumberFormatException e) {
  }
}
public void keyTyped(KeyEvent arg0) {
}
```

Datenempfang und -verarbeitung

Die Methode "run" wird bei Zustandekommen einer Socketverbindung gestartet und läuft als Thread "quasi"-parallel zum eigentlichen Programm. In der Methode wird der Socket in einer while-Schleife dauerhaft auf eintreffende Daten überprüft. Bei Verbindungsabbruch wird der Wert -1 gelesen, was das Abbruchkriterium der Schleife ist.Bei Empfang eines Wertes >0, wird dieser laut ASCII-Tabelle in ein Zeichen konvertiert und dem Empfangsstring hinzugefügt. Bei Empfang einer 0 ist der String komplett empfangen und die Verarbeitung beginnt. Der Anfang jeden Strings ("output;", "input;", "counter;" und "counter") gibt Aufschlussüber die Zuordnung der Daten. Nach der Identifikation der Informationen werden diese extrahiert und dargestellt.

```
public void run() {
  int iInput, iState;
  String sIn = "";
  StringTokenizer stToken;
  try {
    while ((iInput = isInStream.read()) != -1) {
      if (iInput > 0) {
        sIn += (char) iInput;
      }
      else {
        if (sIn.startsWith("input")) {
          iState = Integer.parseInt(sIn.replaceFirst("input;", ""));
          jcbInput0.setSelected(((iState & 1) > 0) ? true : false);
          jcbInput1.setSelected(((iState & 2) > 0) ? true : false);
        }
        else if (sIn.startsWith("output")) {
          iState = Integer.parseInt(sIn.replaceFirst("output;",""));
          jcbOutput0.setSelected(((iState & 1) > 0) ? true : false);
          jcbOutput1.setSelected(((iState & 2) > 0) ? true : false);
        }
        else if (sIn.startsWith("counter")) {
          if (sIn.startsWith("counter;")) {
            sIn = sIn.replaceFirst("counter;", "");
            stToken = new StringTokenizer(sIn, ";");
            jtfCounter0.setText(stToken.nextToken());
            jtfCounter1.setText(stToken.nextToken());
          }
          else {
            stToken = new StringTokenizer(sIn, ";");
            if (stToken.nextToken().equals("counter0")) {
              jtfCounter0.setText(stToken.nextToken());
            }
            else {
              jtfCounter1.setText(stToken.nextToken());
            }
            sIn = sIn.replaceFirst("counter", "");
          }
        }
        sIn = "";
      }
    }
    disconnect();
  }
  catch (IOException e) {
  }
}
```

Kommandostrings senden

Mit der Methode "write" werden die zu übermittelnden Kommandostrings auf den Socket geschrieben. Das Schreiben erfolgt in zwei Schritten. Die Anweisung "write" übergibt den String als Bytekette an den Socket. Die Anweisung "flush" sendet die Bytes an die Gegenstelle. Tritt bei diesem Vorgang ein Fehler auf, erfolgt der Abbruch über die Exception.

```
private void write(String sOutput) {
  try {
    osOutStream.write(sOutput.getBytes());
    osOutStream.flush();
  }
  catch (IOException e) {
  }
}
```

11. Verbindungsabbau

Die Anweisungen für einen ordentlichen Verbindungsabbau sind in einer separaten Methode ausgelagert, die zum einen über die Schaltfläche "Disconnect" und zum anderen bei einem Verbindungsabbruch aufgerufen wird. Zuerst wird die Schaltfläche "Disconnect" inaktiv geschaltet, die Mitteilung über den Verbindungsabbau in der Statuszeile ausgegeben und der Pollingtimer gestoppt. Anschließend erfolgt das Schließen des Sockets. Ist dieser Vorgang erfolgreich, werden alle Eingabe- und Anzeigeelemente in den Ausgangszustand gebracht. Schlägt das Schließen fehl, wird die Oberfläche wieder für eine bestehende Verbindung eingestellt.

```
private void disconnect() {
  jbDisconnect.setEnabled(false);
  jlStatusBar.setText("Trying to disconnect from " + jtfIP.getText());
  try {
    tiPoll.stop();
    soClientSocket.close();
    for (int i = 0; i < jpIO.getComponentCount(); i++) {
      ((JComponent)jpIO.getComponent(i)).setEnabled(false);
    }
    jbConnect.setEnabled(true);
    jlStatusBar.setText("No connection");
    return;
  }
  catch (IOException e) {
    e.printStackTrace();
  }
  jbDisconnect.setEnabled(true);
  jlStatusBar.setText("Error - Connected to " + jtfIP.getText());
}
```

Das Beispielprogramm unterstützt alle gängigen Funktionen des Web-IO im Kommando-String Modus, optimiert für das Web-IO 2x Digital Input, 2x Digital Output. Für die anderen Web-IO Modelle müssen Anpassungen am Programm vorgenommen werden. Weitere Programmbeispiele zur Socket-Programmierung finden Sie auf den Tool-Seiten zum Web-IO. Eine detaillierte Beschreibung zur Socketschnittstelle der Web-IO Digital Modelle finden Sie im Referenzhandbuch.

Programmbeispiel herunterladen

Produkte

Web-IO 4.0 Digital
2xIn, 2xOut

Bei Bedarf auch über PoE zu versorgen

Web-IO 4.0 Digital
12xIn, 12xOut

12x Eingänge,
12x Ausgänge

Weitere Web-IOs

Alle W&T Web-IO Digital 24V

W&T
www.WuT.de

Wir sind gerne persönlich für Sie da:

Wiesemann & Theis
GmbH
Porschestr. 12
42279 Wuppertal
Tel.: 0202/2680-110 (Mo-Fr. 8-17 Uhr)
Fax: 0202/2680-265
info@wut.de