Handbuch

zum Web-IO Schulungskoffer für die Programmiersprachen Delphi Visual Basic C++

1. Ausgabe zur 0-Serie

Release 1.00, November 2003
Typ 57002

W&T

Modell ab Firmware 1.36/2.01

© 09/2002 by Wiesemann und Theis GmbH
Microsoft, MS-DOS, Windows, Winsock und Visual Basic sind eingetragene Warenzeichen der Microsoft Corporation

Irrtum und Änderung vorbehalten:
Da wir Fehler machen können, darf keine unserer Aussagen ungeprüft verwendet werden. Bitte melden Sie uns alle Ihnen bekannt gewordenen Irrtümer oder Missverständlichkeiten, damit wir diese so schnell wie möglich erkennen und beseitigen können.

Führen Sie Arbeiten an bzw. mit W&T Produkten nur aus, wenn Sie hier beschrieben sind und Sie die Anleitung vollständig gelesen und verstanden haben. Eigenmächtiges Handeln kann Gefahren verursachen. Wir haften nicht für die Folgen eigenmächtigen Handelns. Fragen Sie im Zweifel lieber noch einmal bei uns bzw. Ihrem Händler nach!

Einführung

Web-IO, das steht für das Erfassen, Auswerten, Überwachen und Steuern von physikalischen Signalzuständen via TCP/IP.

Das Web-IO 12xDigital kann z.B. 12 digitale Eingänge überwachen, Zustandsänderungen zählen und 12 digitale Ausgänge setzen.

Der W&T Web-IO Schulungskoffer beinhaltet eine komplette Anwendungsumgebung um Schüler, Studenten, Auszubildende und andere interessierte Menschen an die Programmierung von Web-IO Applikationen und Netzwerkkommunikation heranzuführen.

Die Applikationsumgebung besteht aus dem Modell einer Straßenkreuzung mit Ampelsignalanlage für Fahrzeuge und Fußgänger. Die einzelnen Ampelsignale können über das Web-IO angesteuert und geschaltet werden. Ferner stehen zwei Piezopieper zur akustischen Signalisierung zur Verfügung. Die beiliegenden Autos sind mit Magneten ausgestattet und geben bei Einfahrt in die Kreuzung über Hallsensoren Signal an die Inputs des Web-IO. Außerdem sind weitere Inputs mit Anforderungstastern für die Fußgängerampel verbunden.

Da Funktion und Ablauf einer Ampel jedem hinreichend bekannt sein sollten, muss man sich nicht lange mit Erklärungen zur Applikation aufhalten, sondern kann sofort mit der programmtechnischen Umsetzung beginnen.

In den folgenden Kapiteln wird dem Leser die Programmierung von TCP/IP-Anwendungen Schritt für Schritt vermittelt; vom simplen Schalten einer LED bis zur kompletten Ampelsteuerung.

Inhalt

Einführung 3

1 Lieferumfang und Inbetriebnahme 7
1.1 Lieferumfang 8
1.2 Erstinbetriebname und Vergabe der IP-Adresse 10
1.3 Funktionsübersicht des Web-IO 12xDigital 17
1.3 Technische Beschreibung des Schulungsboards 18

2 Programmierung des Web-IO unter Delphi 20
2.1 Ein einführendes Beispiel 21
2.1.1 Die Aufgabenstellung 21
2.1.2 Die Lösung 21
2.2 Geregelter Programmabbruch 29
2.2.1 Die 2. Aufgabe 29
2.2.2 Noch ein Tipp 29
2.2.3 Erster Schritt zur Lösung 30
2.2.4 Eine mögliche Lösung 32
2.3 Zeitgesteuerte Ausgaben 33
2.3.1 Die 3. Aufgabe 33
2.3.2 Noch ein Tipp 33
2.3.3 Die Lösung 34
2.4 Daten vom Netzwerk empfangen 39
2.4.1 Die Aufgabe 39
2.4.2 Noch ein paar Tipps 39
2.4.3 Die Lösung 41
2.5 Eine komplett zeitgesteuerte Anwendung 44
2.5.1 Die Aufgabe 44
2.5.2 Noch ein paar Tipps 45
2.5.3 Die Lösung 47
2.6 Eine ereignis- und zeitgesteuerte Anwendung 56
2.6.1 Die Aufgabe 56
2.6.2 Noch ein paar Tipps 56
2.6.3 Die Lösung 65

3 Programmierung des Web-IO unter Visual Basic 79
3.1 Ein einführendes Beispiel (1) 80
3.1.1 Die Aufgabenstellung 80
3.1.2 Die Lösung 80
3.2 Geregelter Programmabbruch 87
3.2.1 Die 2. Aufgabe 87
3.2.2 Noch ein Tipp 87
3.2.3 Erster Schritt zur Lösung 87
3.2.4 Eine mögliche Lösung 89
3.3 Zeitgesteuerte Ausgaben 91
3.3.1 Die Aufgabe 91
3.3.2 Noch ein Tipp 91
3.3.3 Die Lösung 92
3.4 Daten vom Netzwerk empfangen 97
3.4.1 Die Aufgabe 97
3.4.2 Noch ein paar Tipps 97
3.4.3 Die Lösung 99
3.5 Eine komplett zeitgesteuerte Anwendung 102
3.5.1 Die Aufgabe 102
3.5.2 Noch ein paar Tipps 103
3.5.3 Die Lösung 105
3.6 Eine ereignis- und zeitgesteuerte Anwendung 113
3.6.1 Die Aufgabe 113
3.6.2 Noch ein paar Tipps 113
3.6.3 Die Lösung 118

4 Programmierung des Web-IO unter C++ 129
4.0 Ein vorbereitetes Programm-Layout 130
4.1 Ein einführendes Beispiel 133
4.1.1 Die Aufgabenstellung 133
4.1.2 Die Lösung 133
4.2 Zeitgesteuerte Ausgaben 138
4.2.1 Die Aufgabenstellung 138
4.2.2 Die Lösung 138
4.3 Auf Eingaben reagieren 142
4.3.1 Die Aufgabe 142
4.3.2 Die Lösung 142
4.4 Eingehende Daten Auswerten 148
4.4.1 Die Aufgabe 148
4.4.2 Noch ein Tipp 148
4.4.3 Die Lösung 148

4.5 Eine komplett Zeitgesteuerte Anwendung: 153
4.5.1 Die Aufgabe 153
4.5.2 Die Lösung 153
4.6 Eine zeit- und ereignisgesteuerte Anwendung 159
4.6.1 Die Aufgabe 159
4,6,2 Die Lösung 159

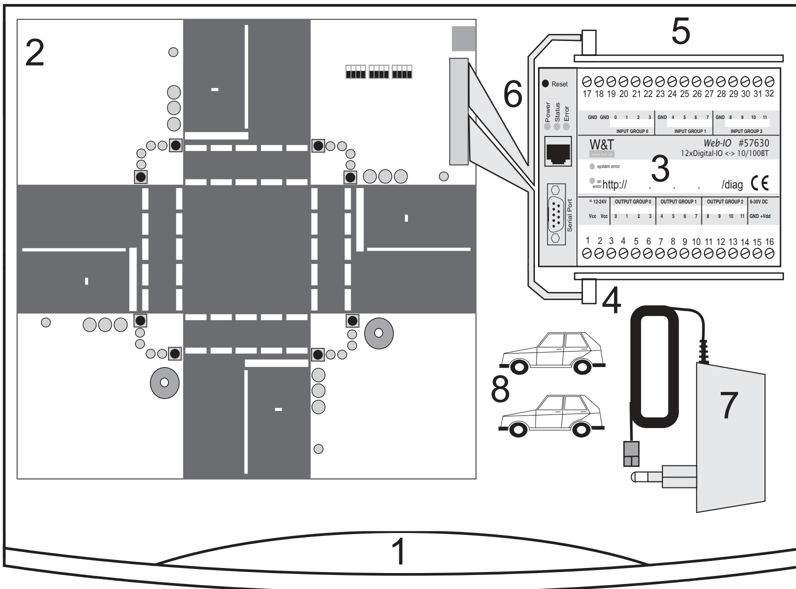
5 Anhang 165
5.1 Socketprogrammierung mit Kommandosstrings 166
5.1.1 TCP Kommunikation 167

1 Lieferumfang und Inbetriebnahme

- Der Web-IO Experimentierkoffer
- Anschluss und Inbetriebnahme
- Funktionsübersicht des Web-IO 12xDigital
- Technische Beschreibung des Schulungsboards

1.1 Lieferumfang

Wir gratulieren zum Kauf des Web-IO-Experimentier- und Schulungskoffers. Mit dem Schulungskoffer verfügen Sie über eine komplette Schulungsumgebung für Web-IO 12xDigital.



Überprüfen Sie bitte zunächst den Inhalt des Koffers auf Vollständigkeit. Der Koffer sollte folgende Positionen beinhalten:

1. Kunststoffkoffer mit Schaumstoffeinlage
2. Schulungsboard mit Aufdruck und Beschriftung
3. Web-IO 12xDigital Typ 57630
4. Outputadapterplatine (mit Web-IO verschraubt)
5. Outputadapterplatine (mit Web-IO verschraubt)
6. Flachbandkabel (Verbindung zwischen Web-IO und Schulungsboard
7. Steckernetzteil 230V/50Hz - 18V AC/270mA.
8. 2 x Modellauto mit eingebautem Magnet

ferner finden Sie in einem Fach unter dem Schulungsboard:

9. Anleitungs- und Aufgabenheft zum Schulungskoffer
10. Handbuch TCP/IP Ethernet und Web-IO
11. CD-ROM mit Tools und Programmierbeispielen

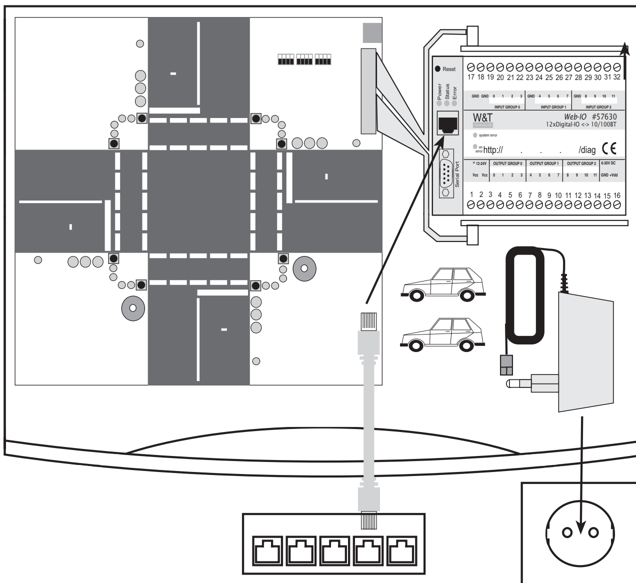
Sollte eine der aufgeführten Positionen fehlen, wenden Sie sich bitte sofort an Ihren Fachhändler.

1.2 Erstinbetriebname und Vergabe der IP-Adresse

1.2.1 Anschluss und Inbetriebnahme

Nach Öffnen des Koffers, sind die Komponenten sofort einsatzbereit. Weder das Web-IO noch das Schulungsboard müssen entnommen werden.

Um mit dem Schulungskoffer arbeiten zu können, verbinden Sie das Web-IO mit einem Patchkabel an einen freien Netzwerkanschluss (HUB oder Switch / 10 oder 100MBit). Alternativ kann das Web-IO mit einem Crosslink-Kabel direkt an den PC angeschlossen werden. Das Steckernetzteil wird mit der Niederspannungsseite in die grüne Buchse rechts oben am Schulungsboard eingesteckt. Das Netzteil selbst stecken Sie in eine freie Steckdose.



1.2.2 Vergabe der IP-Adresse unter Windows

Die Voraussetzung für die hier beschriebene Methode der IP-Adressvergabe ist, dass das Web-IO noch keine IP-Adresse hat (Werkseinstellung), der Eintrag also 0.0.0.0 lautet und der PC sich im gleichen Netzwerksegment wie das Web-IO befindet.

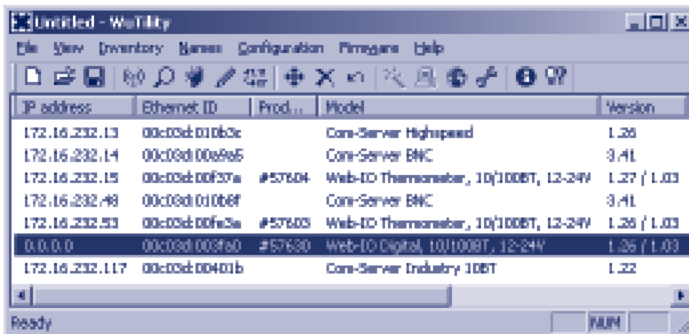
Sollte das Web-IO bereits eine IP-Adresse haben, verwenden Sie zum Ändern eine der in der Anleitung zum Web-IO beschriebenen Methoden oder den Menüpunkt Config/Device/Network im Web-based Management des Web-IO.

Zunächst starten Sie WuTility.exe aus dem Stammverzeichnis der beiliegenden CD-ROM.

Sollte die CD-ROM nicht vorliegen, ist der Download unter http://www.wut.de möglich. Navigieren Sie von dort aus am einfachsten mit Hilfe des auf der linken Seite befindlichen Menübaums. Über den Pfad Produkte & Downloads >> Web-IO *gelangen Sie in den Menüzweig der direkte Links auf Applikationen, FAQs, Tools, usw. enthält.*

Jede IP-Adresse muss immer netzwerkweit eindeutig sein.

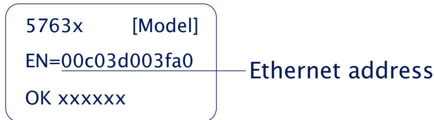
Durch Klick auf das Icon starten Sie den Netzwerk-Scan



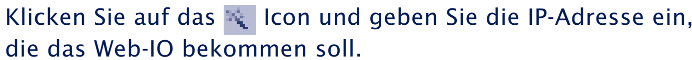
Die Liste der gefundenen W&T Netzwerkkomponenten sollte nun einen Eintrag für das Web-IO enthalten. Die IP-Adresse sollte mit 0.0.0.0 angezeigt werden, wenn es sich um eine Erstinbetriebnahme handelt. Stellen Sie anhand der ange-

zeigten Ethernet-Adresse sicher, dass der Eintrag Ihrem Web-IO entspricht.

Lesen Sie die Ethernet-Adresse des Web-IO von dem Aufkleber an der Gehäuseseite ab:



Markieren Sie den Eintrag für Ihr Web-IO.

Klicken Sie auf das  Icon und geben Sie die IP-Adresse ein, die das Web-IO bekommen soll.

Klicken auf das Icon, um die Liste zu aktualisieren.

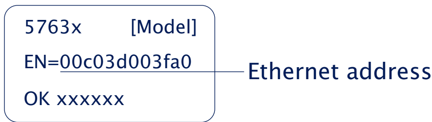
Damit hat Ihr Web-IO eine IP-Adresse und ist im lokalen Netzwerk erreichbar. Zum Test öffnen Sie Ihren Internet-Browser und geben Sie die IP-Adresse im URL-Feld ein. Nach wenigen Sekunden erscheint die Startseite des Web-IO.

1.2.3 Vergabe der IP-Adresse bei anderen Betriebssys.

Die Voraussetzungen für die hier beschriebene Methode der IP-Adressvergabe ist, dass das Web-IO noch keine IP-Adresse hat (Werkseinstellung), der Eintrag also 0.0.0.0 lautet und der PC sich im gleichen Netzwerksegment wie das Web-IO befindet.

Sollte das Web-IO bereits eine IP-Adresse haben, verwenden Sie zum Ändern eine der in der Anleitung zum Web-IO beschriebenen Methoden oder den Menüpunkt Config/Device/Network im Web-based Management des Web-IO.

Lesen Sie die Ethernet-Adresse des Web-IO von dem Aufkleber an der Gehäuseseite ab:



Fügen Sie in der Konsole jetzt mit der folgenden Befehlszeile der ARP-Tabelle des Rechners einen statischen Eintrag hinzu:

```
arp -s [IP-Adresse] [MAC-Adresse]
```

Beispiel unter SCO UNIX:

```
arp -s 172.16.231.17 00:C0:3D:00:3F:A0
```

Starten Sie den Internet-Browser und geben Sie folgende URL ein:

http://[IP-Adresse] [***Return***]

Beispiel:

Das Web-IO übernimmt die IP-Adresse des ersten an seine Ethernet-Adresse gesendeten Netzwerkpaketes als seine eigene und speichert diese nichtflüchtig ab. Die HTTP-Verbin-

dung wird daraufhin aufgebaut und die Homepage des Web-IO erscheint im Browser. Alle weiteren Einstellungen können nun bequem per Web-Based Management vorgenommen werden.

1.2.4 Konfiguration der Netzwerkparameter

In Netzwerken, bei denen mehrere Sub-Netze über Router miteinander verbunden sind, müssen beim Web-IO noch Subnet-Mask und Gateway konfiguriert werden.

Wählen Sie im Navigationsraum *Config*.

Sie werden nun aufgefordert das Administrator-Passwort einzugeben. Im Auslieferungszustand ist kein Passwort vorgegeben und es reicht, den *Login*-Button anzuklicken.

Wurde das richtige Passwort eingegeben, bestätigt das Web-IO das erfolgreiche Login. Nach Mouse-Klick auf OK baut sich die Startseite neu auf.

Wählen Sie nun im Navigationsmenü den Punkt *Config >> Device >> Basic Settings >> Network.*

Es erscheint die folgende Eingabemaske:

Config >> Device >> Basic Settings >> Network

IP Addr :	172.16.232.17
Subnet Mask :	255.255.255.0
Gateway :	172.16.232.252
BOOTP Client :	☑ BOOTP enable
	IP-Adresse des DNS Servers im Format xxx.xxx.xxx.xxx
DnsServer1 :	172.16.232.252
	IP-Adresse des DNS Servers im Format xxx.xxx.xxx.xxx
DnsServer2 :	

Zwischenspeichern | Rücksetzen | Logout

Tragen Sie hier die benötigten Netzwerkparameter ein.

IP Addr.

Die IP-Adresse kann an dieser Stelle geändert werden.

Subnet Mask / Gateway

Für Subnet-übergreifenden Datenaustausch müssen die passende Subnet-Mask und die IP-Adresse des Gateway eingetragen werden.

BOOTP Client

Soll das Web-IO nicht an der zentralen IP-Adressvergabe via BootP teilnehmen, deaktivieren Sie die Markierung bei *BootP enable*.

DNS Server

Bei einigen Netzteilnehmern, wie z.B. Mail- und Time-Servern ist es sinnvoll, diese nicht über Ihre IP-Adresse, sondern über einen Namen zu adressieren. Dazu muss ein DNS-Server angegeben werden.

Sollten Ihnen die benötigten Informationen nicht vorliegen, fragen Sie Ihren Netztwerkadministrator.

Wurden alle Eingaben gemacht, klicken Sie auf den *Logout*-Button.

Nach einem Mouse-Klick auf den *Speichern*-Button wird das Web-IO nun mit den aktuellen Parametern neu gestartet. Im Normalfall baut sich die Startseite des Web-IO nach ca.. 10 Sekunden erneut auf. Sollte der Neuaufbau ausbleiben, klicken Sie auf den *hier*-Link

Das Web-IO ist nun soweit eingerichtet, dass es auch subnetzübergreifend angesprochen werden kann.

Alle netzwerkspezifischen Ausdrücke und Zusammenhänge sind im beiliegenden Buch TCP/IP-Ethernet und Web-IO umfassend erklärt.

1.3 Funktionsübersicht des Web-IO 12xDigital

Das W&T Web-IO bietet die Möglichkeit, 12 digitale Inputs und 12 digitale Outputs über Ethernet-TCP/IP zu lesen, bzw. zu setzen.

Der Ethernet-Anschluss ist in 10/100Mbit Autosensing Technologie aufgebaut.

Zum Lesen und Setzen der Ein- und Ausgänge stehen folgende TCP-IP Protokolle zu Verfügung:

HTTP Einfache Bedienoberfläche im Browserfenster
TCP Direkter Socket-Zugriff aus eigenen Applikationen
UDP Direkter Socket-Zugriff aus eigenen Applikationen
SMTP Alarmierung per E-Mail bei konfigurierbaren Eingangsbedingungen
SNMP Einbindung in Managementsysteme und Alarmierung per SNMP-Trap
Box-to-Box
paarweises Zusammenschalten von WEB-IO

Die Konfiguration des Web-IO kann via Web-Based Management von Hand im Browserfenster erfolgen und per XML-File eingespielt werden.

In der Schulungskofferanwendung wird das Web-IO über das Schulungsboard und die Anschlussadapter mit Spannung versorgt.

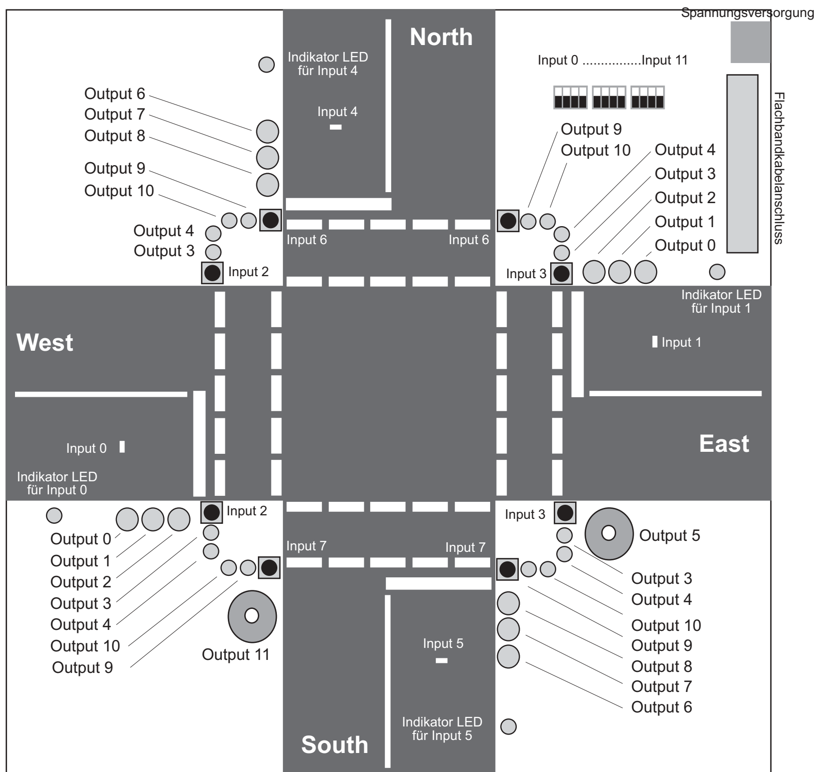
Die Versorgungspannung Vcc des Web-IO beträgt dabei 18V AC. Für die Outputs stellt das Schulungsboard dem Web-IO eine Schaltsspannung Vdd von 9V zur Verfügung.

Weitere technische Details finden Sie in der Anleitung zum Web-IO, die auf der beiliegenden CD-ROM als PDF-File zu finden ist.

1.4 Technische Beschreibung des Schulungsboards

Die Inputs und Outputs des Web-IO sind über den Flachbandadapter fest mit den einzelnen Ampellichtern (LEDs), Beepern, Tastern, Schaltern und Hallsensoren verbunden.

Welches Element wo angeschlossen ist, entnehmen Sie bitte dem folgenden Plan, bzw. der Beschriftung des Schulungsboards:

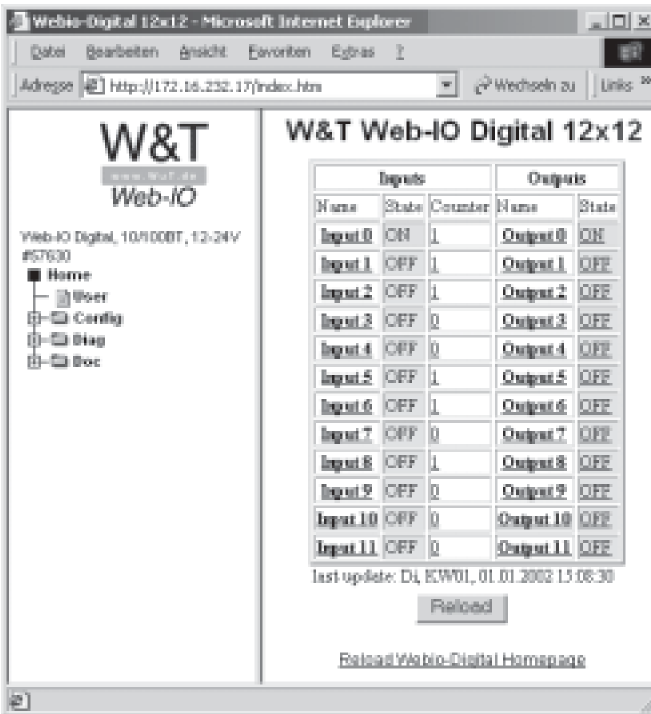


Ein erster Funktionstest lässt sich leicht über die Webseite des Web-IO 12xDigital durchführen.

Rufen Sie die Webseite des Web-IO einfach durch Eingabe der IP-Adresse ins URL-Feld des Browsers auf. Wählen Sie im Navigationsbaum *Config* und loggen Sie sich als Administra-

tor ein. Im Auslieferungszustand ist dazu kein Passwort nötig.

Nach erfolgreichem Login erhalten Sie im Browser eine Tabelle mit den Zuständen der Inputs und Outputs. Die Zustände der Outputs (ON/OFF) sind mit einem Hyperlink hinterlegt (am Unterstrich zu erkennen). Durch Mouse-Klick auf diesen Link wechselt der entsprechende Output seinen Zustand.



Inputs			Outputs	
Name	State	Counter	Name	State
Input 0	ON	1	Output 0	ON
Input 1	OFF	1	Output 1	OFF
Input 2	OFF	1	Output 2	OFF
Input 3	OFF	0	Output 3	OFF
Input 4	OFF	0	Output 4	OFF
Input 5	OFF	1	Output 5	OFF
Input 6	OFF	1	Output 6	OFF
Input 7	OFF	0	Output 7	OFF
Input 8	OFF	1	Output 8	OFF
Input 9	OFF	0	Output 9	OFF
Input 10	OFF	0	Output 10	OFF
Input 11	OFF	0	Output 11	OFF

Setzen Sie z.B. eines der Autos auf die westliche Zufahrt der Kreuzung und klicken auf *Reload* wechselt Input 0 auf ON. Durch Mouse-Klick auf das OFF Feld von Output 0 wechselt dieser von OFF auf ON und die grünen Ampellichter der Hauptstraße werden eingeschaltet.

So können alle Inputs und Outputs Stück für Stück getestet werden.

2 Programmierung des Web-IO unter Delphi

Bei allen hier gezeigten Programmbeispielen und Erklärungen wird der grundsätzliche Umgang mit der Programmierumgebung von Delphi vorausgesetzt.

2.1 Ein einführendes Beispiel

2.1.1 Die Aufgabenstellung

Es soll ein einfaches Programm erstellt werden, mit dem über das Web-IO Digital das grüne Ampellicht der Straße West/Ost ein- und ausgeschaltet werden kann.

Dazu soll eine TCP-Verbindung genutzt werden.

Folgende Aufgaben müssen vom Programm abgearbeitet werden:

1. Entgegennehmen der Verbindungsparameter (IP-Adresse und Remote-Port des Web-IO)
2. Aufbau der TCP-Verbindung
3. Freigeben der Buttons für das Schalten des Ampellichts
4. Ein- und Ausschalten des grünen Ampellichts über zwei Buttons
5. Schließen der TCP-Verbindung und der Buttons für das Schalten des Ampellichts

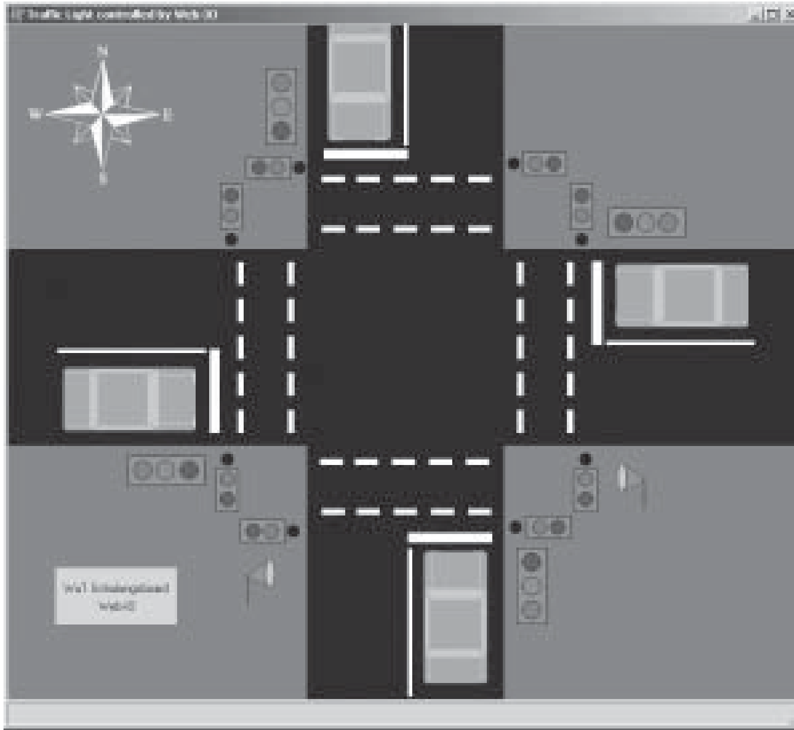
Im Programmfenster soll ein komplettes Abbild der Straßenkreuzung mit allen Elementen dargestellt werden, in dem die Abläufe auf dem Schulungsboard 1:1 wiedergegeben werden.

Vorarbeit zum eigentlichen Programm

Um einen schnellen Einstieg in die eigentliche Programmierarbeit zu ermöglichen, haben wir ein komplettes Programm-Formular vorbereitet.

Es bietet dem Programmierer ein programmtechnisches Abbild der Straßenkreuzung, bei dem bereits Steuerelemente für die einzelnen Ampeln, Anforderungsschalter und ankommenden Autos vorhanden sind.

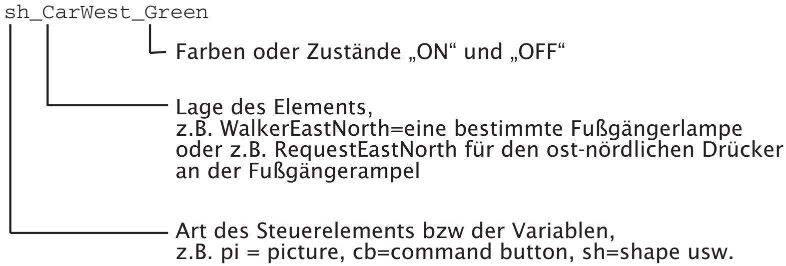
Dieses Formular wird die Grundlage für alle weiteren, hier gezeigten Programmbeispiele sein und ist auf der beiliegenden CD unter delphi\basiclayout zu finden.



Im ersten Beispiel haben wir den Quelltext bereits gemäß den Anforderungen ergänzt, so dass, Sie unter delphi\example1 den kompletten Quelltext für eine funktionierende Applikation finden.

Grundlegende Festlegungen

Die in diesem Text und den Programmen auftretenden Bezeichnungen sind von uns willkürlich gewählt, folgen aber alle einer gewissen Logik. Dabei haben wir folgendes Schema gewählt:



Bei den Fußgängerampeln und Anforderungsdrückern steht die Himmelsrichtung der zugehörigen Straßenrichtung an erster Stelle.

Beispiel: Fusgängerampel für den nördlichen Abgang linksseitig = WalkerNorthWest

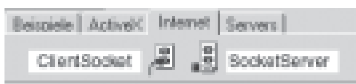
Variablennamen beginnen immer mit einem v, gefolgt von einer Zweibuchstabenkombination, die für den Variablentyp steht (z.B. bo für bool).

Beispiel: vbo_CloseConnection

Kommunication über TCP/IP

Für eine einfache Integration von Datenaustausch über TCP/IP in eigene Anwendungen, stellt Delphi das *ClientSocket*-Steuerelement zur Verfügung.

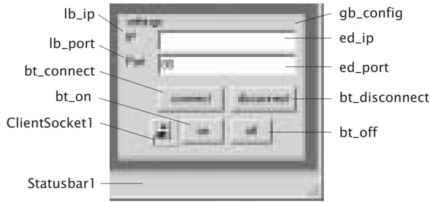
Das *ClientSocket*-Steuerelement finden Sie in der Komponentenleiste im Register Internet.



2.1.2 Die Lösung

Die benötigten Elemente

Folgende Steuerelemente haben wir in das vorgegebene Formular zusätzlich eingefügt, um das Programm bedienbar zu machen.



Die Funktion der Buttons und der Eingabefelder sollte aus der Namensgebung und Beschriftung hervorgehen.

Der Quellcode

Für die einzelnen Ampelfarben haben wir leicht verständliche Konstanten definiert; dadurch wird der Quelltext deutlich überschaubarer. *RED_OFF* lässt sich beim Lesen sicher besser zuordnen als *clMaroon*.

```
const
RED_ON =  clRed;
RED_OFF = clMaroon;
YELLOW_ON = clYellow;
YELLOW_OFF = clOlive;
GREEN_ON = clLime;
GREEN_OFF = clGreen;
BLACK_ON = clSilver;
BLACK_OFF = clBlack;
```

Selbstdefinierte Variablen werden in diesem Beispiel noch nicht verwendet, so dass an dieser Stelle nur das Formular selbst deklariert wurde.

```
var
  fm_TrafficLight: Tfm_TrafficLight;

implementation

{$R *.DFM}
```

Und hier beginnt der eigentliche Programmcode:

Die Prozedur *bt_connectClick* wird aufgerufen, wenn der Anwender den *Connect*-Button anklickt.

Zunächst wird überprüft, ob IP-Adresse und Port in die entsprechenden Felder eingegeben wurden und nur dann wird die Prozedur weiter abgearbeitet.

IP-Adresse und Port-Nr. werden an die entsprechenden Eigenschaften des *ClientSocket*-Steuerelements übergeben. Die als Zeichenfolge vorliegende Port-Nr. wird hierzu in einen Integerwert gewandelt.

Nun wird die *Connect*-Methode des *ClientSocket*-Steuerelmentes aufgerufen, um den Verbindungsaufbau einzuleiten.

```
procedure Tfm_TrafficLight.bt_connectClick(Sender: TObject);
begin
  if (ed_ip.Text <> '') and (ed_port.Text <> '') then
  begin
    ClientSocket1.Host := ed_ip.Text;
    ClientSocket1.Port := strtoint(ed_Port.Text);
    ClientSocket1.Active := True;
  end;
end;
```

Die Prozedur *ClientSocket1Connect* wird vom *Clientsocket*-Steuerelement aufgerufen, wenn eine TCP-Verbindung zu Stande kommt.

Bei erfolgreichem Verbindungsaufbau wird dann der *Connect*-Button für die Bedienung gesperrt, während der Disconnect-Button und der *ON*-Button freigegeben werden.

In die Statuszeile wird „connected“ geschrieben.

```
procedure Tfm_TrafficLight.ClientSocket1Connect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  bt_connect.Enabled := False;
  bt_disconnect.Enabled := True;
```

```
  bt_on.Enabled := True;
  StatusBar1.SimpleText := 'connected';
end;
```

Durch Klicken auf den *ON*-Button wird der String „ GET / outputaccess0?PW=&State=ON&“ an die Methode *SendText* des *ClientSocket*-Steuerelements übergeben. Das *ClientSocket*-Steuerelement sendet diesen String über die bestehende TCP-Verbindung an das Web-IO.

Das Web-IO lässt sich wie hier gezeigt mit vordefinierten Kommandostrings steuern und lesen. Die möglichen Kommandos und deren Aufbau sind als Übersicht im Anhang zu finden. Eine ausführliche Beschreibung finden Sie im Referenzhandbuch zum Web-IO.

Der gesendete Kammandostring schaltet Output0 und damit die grüne LED der Fahrzeugampeln West und East ein.

Um die Darstellung auf dem Bildschirm dem Zustand auf dem Schulungsboard anzugleichen, werden die entsprechenden Shapes (Kreise) auf die Farbe *GREEN_ON* gesetzt.

Der *ON*-Button wird gesperrt und der *OFF*-Button freigeschaltet.

```
procedure Tfm_TrafficLight.bt_onClick(Sender: TObject);
begin
  ClientSocket1.Socket.SendText('GET /outputaccess0?PW=&State=ON&');
  sh_CarWest_Green.Brush.Color := GREEN_ON;
  sh_CarEast_Green.Brush.Color := GREEN_ON;
  bt_on.Enabled := False;
  bt_off.Enabled := True;
end;
```

Die Prozedur *bt_off-Click* arbeitet analog zur *bt_on* Prozedur, nur dass alle Zustände genau umgekehrt gehandhabt werden.

```
procedure Tfm_TrafficLight.bt_offClick(Sender: TObject);
begin
  ClientSocket1.Socket.SendText('GET /outputaccess0?PW=&State=OFF&');
```

```
  sh_CarWest_Green.Brush.Color := GREEN_OFF;
  sh_CarEast_Green.Brush.Color := GREEN_OFF;
  bt_on.Enabled := True;
  bt_off.Enabled := False;
end;
```

Zu guter Letzt, wird die Prozedur *bt_disconnectClick* aufgerufen, wenn der Anwender auf den *Disconnect*-Button klickt.

Die Active-Eigenschaft des *Clientsocket*-Steuerelements wird auf *False* gesetzt. Dadurch angestoßen beendet das *ClientSocket*-Steuerelemet die TCP-Verbindung zum Web-IO.

Die Buttons *Disconnect*, *ON* und *OFF* werden deaktiviert und das *Connect*-Button wird wieder freigeschaltet. In der Statuszeile wird „disconnected“ angezeigt.

```
procedure Tfm_TrafficLight.bt_disconnectClick(Sender: TObject);
begin
  ClientSocket1.Active := False;
  bt_disconnect.Enabled := False;
  bt_on.Enabled := False;
  bt_off.Enabled := False;
  StatusBar1.SimpleText := 'disconnected';
  bt_connect.Enabled := True;
end;
```

Mit ein paar Zeilen Quellcode haben wir in diesem Beispiel gezeigt, dass die Ansteuerung des Web-IO über das Netzwerk via TCP/IP unter Delphi eigentlich ganz einfach ist.

In den folgenden Kapiteln wird es Ihre Aufgabe sein, Schritt für Schritt immer komplexere Applikationen zu entwickeln.

Natürlich finden Sie am Ende jeden Kapitels als Lösung einen passenden Quelltext, der auch auf der CD zur Verfügung steht.

Nehmen Sie sich jedoch nicht die Chance selbst zu einem Ergebnis zu kommen, indem Sie sich sofort die Lösung anschauen.

Abschließend möchten wir Ihnen noch eine Übersicht über die wichtigsten Eigenschaften, Methoden und Ereignisse des *Clientsocket*-Steuerlementes geben

Methoden (Funktionen und Prozeduren)

Open	Öffnet eine TCP-Verbindung
Close	Schließt eine TCP-Verbindung
Socket.SendText	Sendet einen String
Socket.SendBuf	Sendet Binärdaten
Socket.ReceiveText	Liest einen empfangenen String
Socket.ReceiveBuf	Liest empfangene Binärdaten
Socket.ReceiveLength	Anzahl der empfangenen Zeichen

Eigenschaften

Active	Zeigt an, ob eine Verbindung besteht
State	Zeigt den detaillierten Verbindungsstatus
Address	IP-Adresse zu der verbunden werden soll
Host	Host-Name des Servers zu dem verbunden werden soll (kann alternativ zu *Address* verwendet werden)
Port	TCP-Portnummer auf der der Server zu dem verbunden werden soll seinen Dienst zur Verfügung stellt.

Ereignisse

OnError	Tritt bei Verbindungsfehlern ein
OnConnect	Tritt ein, wenn eine Verbindung zustande kommt
OnDisconnect	Tritt ein, wenn die Verbindung vom Server abgebaut wird
OnRead	Tritt bei Dateneingang auf der bestehenden Verbindung ein

Es können eigene Prozeduren geschrieben werden, die bei Eintritt eines dieser Ereignisse automatisch aufgerufen werden. So kann sofort auf bestimmte Verbindungszustände reagiert werden.

Auf etwaige Parameter und deren Format gehen wir in den einzelnen Programmierbeispielen näher ein.

2.2 Geregelter Programmabbruch

Die Ausgangssituation

Unser erstes Programmbeispiel arbeitet im Rahmen der Aufgabenstellung gut und zuverlässig. Wird das Programm allerdings geschlossen, solange die grünen LEDs eingeschaltet sind, bleiben diese auch nach Programmende an.

2.2.1 Die 2. Aufgabe

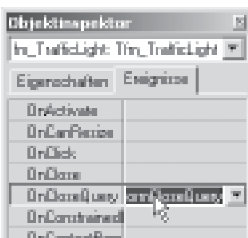
Sowohl beim Schließen der TCP-Verbindung als auch beim Beenden des Programms soll gewährleistet sein, dass die LEDs auf jeden Fall ausgeschaltet werden.

2.2.2 Noch ein Tipp

Für das Programmformular gib es das *OnCoseQuery*-Ereignis. Dieses Ereignis tritt immer ein, bevor ein Programm endgültig beendet wird und startet, wenn vorhanden, die Prozedur *FormCloseQuery*.

Durch Setzen der Variablen *CanClose* auf *False* kann das Beenden unterbrochen werden. Durch erneuten Aufruf der *Close*-Methode wird das Programm beendet.

Eine Prozedur zum Ereignis wird übrigens durch Doppelklick auf das entsprechende Ereignis im Objektinspektor generiert. Hier kann dann der eigene Quelltext eingefügt werden.



2.2.3 Erster Schritt zur Lösung

Eine mögliche Lösung könnte so aussehen:

Die Prozedur bt_disconnect wird erweitert und die Prozedur *FormCloseQuery* wird neu angelegt.

In der bt_disconnect Prozedur wird mit Hilfe der *SendText*-Methode das Kommando zum Zurücksetzen des Outputs an das Web-IO gesendet, bevor die *Close*-Methode aufgerufen wird. Ferner werden die Abbilder des Ampellichts auf *GREEN_OFF* gesetzt.

```
procedure Tfm_TrafficLight.bt_disconnectClick(Sender: TObject);
begin
  ClientSocket1.Socket.SendText('GET /outputaccess0?PW=&State=OFF&');
  sh_CarWest_Green.Brush.Color := GREEN_OFF;
  sh_CarEast_Green.Brush.Color := GREEN_OFF;
  ClientSocket1.Active := False;
  bt_connect.Enabled := True;
  bt_disconnect.Enabled := False;
  bt_on.Enabled := False;
  bt_off.Enabled := False;
  StatusBar1.SimpleText := 'connected';
end;
```

Die *FormCloseQuery*-Prozedur wird aufgerufen, sobald der Anwender versucht das Programm zu beenden.

Hier wird zunächst geprüft, ob eine Verbindung besteht. Ist das der Fall, wird durch Setzen der Variblen *CanClose = False* das beenden gestoppt.

Mit Hilfe der *SendText*-Methode des *Clientsocket*-Steuerelements wird das Kommando zum Zurücksetzen des Outputs an das Web-IO gesendet, bevor die *Close*-Methode aufgerufen wird.

Dann wird die *Close*-Methode des Programmfensters aufgerufen, die das Programm beendet.

```
procedure Tfm_TrafficLight.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if clientsocket1.active then
  begin
    CanClose := false;
    ClientSocket1.Socket.SendText('GET /outputaccess0?PW=&State=OFF&');
    ClientSocket1.Active := false;
    Close;
  end;
end;
```

In den meisten Fällen wird das erweiterte Programm ein geregelte Programmende durchführen.

Stolpersteine
Versuchen Sie nun genau Folgendes:

- Starten Sie das Programm
- Stellen Sie die Verbindung zum Web-IO her
- Setzen Sie das Ampellicht auf ON
- Geben sie in der Windows Umgebung unter Start >> Ausführen arp -d * ein
- Klicken Sie auf den Disconnect-Button des Programms

und das Ampellicht im Programmfenster ist aus, aber die LED-auf dem Schulungsboard bleibt an!

Was ist passiert?

Das Kommando arp -d * löscht alle Einträge in der ARP-Tabelle des PC. In der ARP-Tabelle ist die Zuordnung von IP-Adresse und Ethernet-Adresse der Kommunikationspartner hinterlegt. Ist der gewünschte Partner nicht eingetragen, muss der PC die zugehörige Ethernetadresse neu ermitteln.

Dieser Vorgang nimmt einige ms Zeit in Anspruch.

Nun zurück zu unserem eigentlichen Problem. Dadurch, dass das Absenden des Kommandostrings zum Web-IO und das Schließen der Verbindung in direkter Folge ausgeführt wer-

den, schafft es der PC nicht die Ethernet-Adresse zu ermitteln und den Kommandostring zu versenden bevor die Verbindung gekappt wird.

Das Programm sollte also so geändert werden, dass der PC genug Zeit hat die Daten zu senden.

2.2.4 Eine mögliche Lösung

In beide Prozeduren wird zwischen das Senden des Kommandostrings und dem Schließen der Verbindung der Prozeduraufruf *sleep(50)* eingefügt. Dadurch wartet das Programm 50ms, bevor es die nächste Programmzeile abarbeitet.

```
procedure Tfm_TrafficLight.bt_disconnectClick(Sender: TObject);
begin
  ClientSocket1.Socket.SendText('GET /outputaccess0?PW=&State=OFF&');
  sh_CarWest_Green.Brush.Color := GREEN_OFF;
  sh_CarEast_Green.Brush.Color := GREEN_OFF;
  sleep(50);
  ClientSocket1.Active := False;
  ..
  ..
end;


In
procedure Tfm_TrafficLight.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if clientsocket1.active then
  begin
    CanClose := false;
   ClientSocket1.Socket.SendText('GET /outputaccess0?PW=&State=OFF&');
    sleep(50);
    ClientSocket1.Active := false;
    Close;
  end;
end;
```

2.3 Zeitgesteuerte Ausgaben

Die Ausgangssituation

Unsere Beispielprogramme haben bis jetzt, angestoßen durch Userbedienung, einen Output des Web-IO auf *ON* oder auf *OFF* gesetzt .

2.3.1 Die 3. Aufgabe

Wir nehmen an, die Straße von West nach Ost ist die Hauptstraße. Die gelben Ampellichter der Nebenstraße (von Nord nach Süd) sollen blinken. Das Blinkintervall soll über ein Eingabefeld frei einstellbar sein.

Beim Trennen der TCP-Verbindung oder beim Beenden des Programms sollen die gelben Ampellichter zunächst ausgeschaltet werden.

2.3.2 Noch ein Tipp

Für eine einmalige Programmverzögerung haben wir im vorherigen Beispiel den sleep() Befehl verwendet. Für sich zyklisch wiederholende Programmabläufe sollte jedoch das *Timer*-Steuerelement eingesetzt werden.

Das *Timer*-Steuerelement steht in der Komponentenleiste im Register System zur Verfügung.

Die Eigenschaft Enabled legt fest, ob der Timer aktiv ist oder nicht. Unter *Interval* wird die Zeit zwischen zwei Timer-Aufrufen in ms festgelegt.

Im Objektinspektor kann unter *Ereignisse / OnTimer* festgelegt werden, welche Prozedur aufgerufen wird, wenn das Intervall abgelaufen ist.

2.3.3 Die Lösung

Programmplanung

Zunächst sollte festgelegt werden welche Aufgaben das Programm in welcher Abfolge erledigen muss:

1. Entgegennehmen der Verbindungsparameter (IP-Adresse und Remote-Port des Web-IO
2. Aufbau der TCP-Verbindung
3. Starten des Timers, der im Wechsel das gelbe Ampellicht auf dem Schulungsboard ein- und ausschaltet. Buttons bedienbar machen bsw. Bedienung sperren
4. Bei Schließen der TCP-Verbindung zunächst die Ampellichter ausschalten, dann erst Verbindung schließen und Buttons aktivieren bzw. sperren
5. Bei Beenden des Programms zunächst die Ampellichter ausschalten

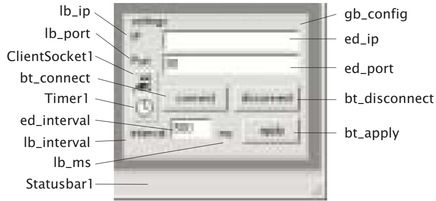
Im Programmfenster soll wie gewohnt ein komplettes Prozessabbild wiedergegeben werden.

Vorarbeiten

Auch für diese Anwendung kann das vorgefertigte Programmgerüst aus dem Verzeichnis *basiclayout* der CD als Grundlage genutzt werden. Dazu wird das komplette Projekt in ein neues Verzeichnis kopiert. Das Projekt muss dann aus dem neuen Verzeichnis geöffnet werden.

Um Zeit und Arbeit zu sparen kann man die komplette Groupbox inklusive Bedienelementen aus dem zweiten Programmbeispiel in das neue Projekt kopieren. Einfach die Groupbox gb_config durch Mouse-Klick markieren und mit <Strg + c> in die Zwischenablage kopieren und mit <Strg + v> in das neue Projekt einfügen. Die Steuerelemente, die im neuen Projekt nicht gebraucht werden einfach löschen und neue Objekte nach Bedarf hinzufügen.

Die neue Groupbox könnte dann so aussehen:



Für den Timer1 muss die *Enabled*-Eigenschaft auf *False* gesetzt werden.

Der Quellcode

Selbstdefinierte Variablen werden auch in diesem Beispiel noch nicht verwendet, so dass an dieser Stelle nur das Formular selbst deklariert wurde.

```
var
  fm_TrafficLight          : Tfm_TrafficLight;

implementation

{$R *.DFM}
```

Die Prozedur bt_connectClick wurde unverändert aus Beispiel 2 übernommen.

```
procedure Tfm_TrafficLight.bt_connectClick(Sender: TObject);
begin
  if (ed_ip.Text <> '') and (ed_port.Text <> '') then
  begin
    ClientSocket1.Host := ed_ip.Text;
    ClientSocket1.Port := strtoint(ed_Port.Text);
    ClientSocket1.Active := True;
  end;
end;
```

Bei erfolgreichem Verbindungsaufbau wird die Prozedur ClientSocket1Connect ausgeführt und dann der *Connect*-Button für die Bedienung gesperrt, während der *Disconnect*-Button freigegeben wird. In der Statuszeile wird „disconnect“ angezeigt

Ferner wird das Intervall des Timers auf den Wert gesetzt, der in *ed_interval* eingetragen wurde. Anschließend wird der Timer aktiviert und beginnt das Ein- und Ausschalten der Ampellichter zu steuern.

```
procedure Tfm_TrafficLight.ClientSocket1Connect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  bt_connect.Enabled := False;
  bt_disconnect.Enabled := True;
  StatusBar1.SimpleText := 'connected';
  Timer1.Interval := strtoint(ed_interval.Text);
  Timer1.Enabled := True;
end;
```

Die Prozedur, die nach Ablauf des Timerintervalls aufgerufen wird, besteht aus zwei Programmteilen, die über eine If-Abfrage abhängig vom Zustand des Ampellichtes ausgeführt werden.

Ist das Ampellicht ausgeschaltet, wird das Kommando zum Einschalten an das Web-IO gesendet und die Farbe der entsprechenden Shapes geändert.

Ist das Licht eingeschaltet, werden die umgekehrten Zustände geschaltet.

```
procedure Tfm_TrafficLight.Timer1Timer(Sender: TObject);
begin
  if sh_CarNorth_Yellow.Brush.Color = YELLOW_OFF then
  begin
    sh_CarNorth_Yellow.Brush.Color := YELLOW_ON;
    sh_CarSouth_Yellow.Brush.Color := YELLOW_ON;
    ClientSocket1.Socket.SendText('GET /outputaccess7?PW=&State=ON&');
  end
  else
  begin
    sh_CarNorth_Yellow.Brush.Color := YELLOW_OFF;
    sh_CarSouth_Yellow.Brush.Color := YELLOW_OFF;
    ClientSocket1.Socket.SendText('GET /outputaccess7?PW=&State=OFF&');
  end;
```

```
end;
```

Bei Klick auf den *apply*-Button wird die eingetragene Intervallzeit als Intervall des Timers gesetzt.

```
procedure Tfm_TrafficLight.bt_applyClick(Sender: TObject);
begin
  timer1.Enabled := False;
  timer1.Interval := strtoint(ed_interval.Text);
  timer1.Enabled := True;
end;
```

Die Prozedur für das Disconnect ist weitestgehend mit der aus Beispiel 2 identisch.

```
procedure Tfm_TrafficLight.bt_disconnectClick(Sender: TObject);
begin
  Timer1.Enabled := False;
  sh_CarNorth_Yellow.Brush.Color := YELLOW_OFF;
  sh_CarSouth_Yellow.Brush.Color := YELLOW_OFF;
  ClientSocket1.Socket.SendText('GET /outputaccess7?PW=&State=OFF&');
  bt_connect.Enabled := True;
  bt_disconnect.Enabled := False;
  StatusBar1.SimpleText := 'disconnected';
  sleep(50);
  ClientSocket1.Active := False;
end;
```

Wird das Programm durch den User beendet, wird zunächst das Schließen gestoppt, dann wird der Kommandostring zum Ausschalten des Ampellichts an das Web-IO gesendet und nach 50ms die Verbindung getrennt. Erst danach wird das Schließen des Programms fortgeführt.

```
procedure Tfm_TrafficLight.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if ClientSocket1.Active = True then
  begin
    CanClose := False;
    ClientSocket1.Socket.SendText('GET /outputaccess7?PW=&State=OFF&');
```

```
    sleep(50);
  end;
  ClientSocket1.Active := False;
  CanClose := True;
end;

end.
```

2.4 Daten vom Netzwerk empfangen

Die Ausgangssituation
Alle Beispielprogramme haben bis jetzt ausschließlich Daten über das Netzwerk zum Web-IO gesendet. Im folgenden Beispiel wollen wir uns mit dem Daten Empfang beschäftigen.

Beim Web-IO können die Inputs mit dem Kommandostring GET /inputx?PW=&' abgefragt werden. „x“ steht dabei für den angefragten Input. Als Antwort sendet das Web-IO einen String wie diesen zurück: 'inputx;ON'. In diesem Fall war Input x gleich ON.

Die Antwortstrings des Web-IO enden immer mit einem Nullbyte (chr(0)). Sollen die Strings z.B. über eine IF-Abfrage ausgewertet werden, muss dieses Nullbyte mit in den Vergleich einbezogen werden.

Beispiel:
if Antwort = 'input0;ON + chr(0) then ....

Eine komplette Liste der möglichen Kommandostrings finden Sie im Anhang dieser Anleitung oder im Referenzhandbuch zum Web-IO 12xDigital.

2.4.1 Die Aufgabe

Es soll überwacht werden, ob einer der Anforderungsschalter des westlichen Fußgängerüberwegs gedrückt wird. Sobald dies der Fall ist, soll der entsprechende schwarze Shape weiß angezeigt werden.

2.4.2 Noch ein paar Tipps

Das *ClientSocket*-Steuerelement löst das *OnRead* Ereignis aus, wenn Daten vom Netzwerk empfangen werden. Im Objektinspektor kann festgelegt werden, welche Prozedur aufgerufen werden soll, wenn Daten anstehen.

In dieser Prozedur können die empfangenen Daten dann weiter verarbeitet werden. Um z.B. einen String vom Netzwerk entgegenzunehmen kann die Methode *ReceiveText* des *ClientServer*-Steuerelements benutzt werden.

Beispiel:
TestString := ClientSocket1.ReceiveText;

Wird über den Kommandostringmodus gearbeitet, gibt das Web-IO den Zustand seiner Inputs nur auf Anfrage zurück. Soll die Anwendung aktuell über den Input-Zustand informiert werden, muss eine zyklische Abfrage erfolgen. Man spricht in diesem Fall von „Polling".

Für unsere konkrete Aufgabe bedeutet das, dass ein Timer eingerichtet werden muss, der das Polling ausführt. Dabei sollte das Timerintervall so bemessen sein, dass man keine Änderung verpasst.

Beispiel:
Werden die Inputs nur alle 5 Sekunden „abgepollt", und der Ampeldrücker wird zwischen zwei Inputabfragen gedrückt, so wird diese Input-Änderung vom Programm nicht wargenommen.

Für das Umfärben des Anforderungsschalters im Programmfenster wurden in der Quelltext Grundlage zwei Farbkonstanten definiert:

- BLACK_ON = clsilver //weiß
- BLACK_OFF= clblack //schwarz

Die Verwendung dieser Konstanten macht den Quelltext übersichtlicher.

2.4.3 Die Lösung

Programmplanung

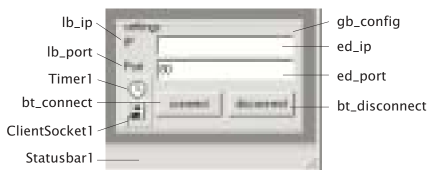
Zunächst sollte wieder festgelegt werden welche Aufgaben das Programm in welcher Abfolge erledigen muss:

1. Entgegen nehmen der Verbindungsparameter (IP-Adresse und Remote-Port des Web-IO)
2. Aufbau der TCP-Verbindung
3. Starten des Timers, der die Inputabfrage an das Web-IO sendet.
4. Entgegennehmen und Auswerten der Datensendungen des Web-IO und entsprechende Änderung des Prozessabbildes im Programmfenster
5. Beenden des Programms

Vorarbeiten

Die Quelltextgrundlage aus dem Layout-Verzeichnis kann auch für dieses Beispiel in einen neuen Ordner kopiert werden.

Auch die Groupbox gb_config aus dem ersten Beispiel kann wieder als Basis für die Bedienelemente des Programms genutzt werden. Die Anpassung an die Bedürfnisse dieser Anwendung sieht dann so aus.



Für den Timer1 muss die *Enabled*-Eigenschaft auf *False* gesetzt werden. Als Intervall wird 100ms eingestellt.

Der Quelltext

Auch in diesem Beispiel werden noch keine selbstdefinierten Variablen verwendet, so dass auch hier nur das Formular selbst deklariert wurde.

```
var
  fm_TrafficLight: Tfm_TrafficLight;

implementation

{$R *.DFM}
```

Die Prozedur *bt_connectClick* bleibt auch hier unverändert.

```
procedure Tfm_TrafficLight.bt_connectClick(Sender: TObject);
begin
  if (ed_ip.Text <> '') and (ed_port.Text <> '') then
  begin
    ClientSocket1.Host := ed_ip.Text;
    ClientSocket1.Port := strtoint(ed_Port.Text);
    ClientSocket1.Active := True;
  end;
end;
```

Bei erfolgreichem Verbindungsaufbau wird die Prozedur *ClientSocket1Connect* ausgeführt.

Die Buttons werden gesperrt bzw. freigegeben und in der Statuszeile wird „connect“ ausgegeben.

Timer1 wird durch Setzen von *Enable = True* gestartet.

```
procedure Tfm_TrafficLight.ClientSocket1Connect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  bt_connect.Enabled := False;
  bt_disconnect.Enabled := True;
  StatusBar1.SimpleText := 'connected';
  Timer1.Enabled := True;
end;
```

Mit jedem Timer-Aufruf wird der Kommandostring für die Statusabfrage von Input 2 an das Web-IO gesendet.

```
procedure Tfm_TrafficLight.Timer1Timer(Sender: TObject);
begin
```

```
  ClientSocket1.Socket.SendText('GET /input2?PW=&')
end;
```

ClientSocket1Read wird immer dann aufgerufen, wenn Daten vom Netzwerk empfangen werden. Je nach empfangenem String werden die Shapes für die Ampeldrücker im Programmfenster schwarz oder weiß eingefärbt

```
procedure Tfm_TrafficLight.ClientSocket1Read(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  if ClientSocket1.Socket.ReceiveText = 'input2;ON'+ chr(0) then
  begin
    sh_RequestWestNorth.Brush.Color := BLACK_ON;
    sh_RequestWestSouth.Brush.Color := BLACK_ON;
  end
  else
  begin
    sh_RequestWestNorth.Brush.Color := BLACK_OFF;
    sh_RequestWestSouth.Brush.Color := BLACK_OFF;
  end;
end;
```

Beim Beenden der TCP-Verbindung wird der Timer deaktiviert, Die Buttons werden gesperrt bzw. freigegeben und in der Statuszeile wird 'disconnected' angezeigt.

```
procedure Tfm_TrafficLight.bt_disconnectClick(Sender: TObject);
begin
  Timer1.Enabled := False;
  ClientSocket1.Active := False;
  bt_connect.Enabled := True;
  bt_disconnect.Enabled := False;
  StatusBar1.SimpleText := 'disconnected';
end;

end.
```

Noch ein Versuch

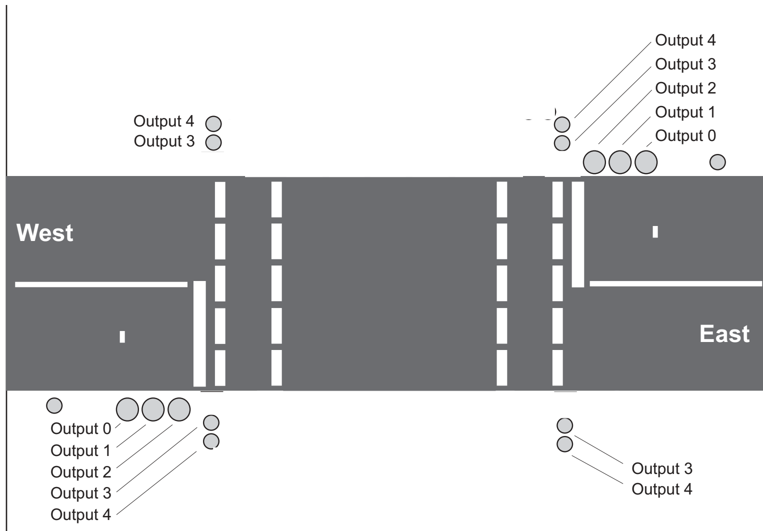
Verändern Sie einmal das Timer-Intervall und prüfen Sie, wie sich das auf die Erkennung der gedrückten Taste auswirkt!

2.5 Eine komplett zeitgesteuerte Anwendung

Nachdem die vorangegangenen Beispiele sich immer nur auf Einzelfunktionen bezogen haben, wollen wir nun eine komplette sinnvolle Anwendung programmieren

2.5.1 Die Aufgabe

Für die Straße von Westen nach Osten soll eine zeitgesteuerte Fußgängerampel eingerichtet werden. Dabei sollen sowohl beim westlichen, als auch beim östlichen Fußgängerüberweg die Ampeln funktionieren. Die Grünphase für die Autos soll genau wie die Grünphase für die Fußgänger auf 20 Sekunden voreingestellt sein. Über ein Eingabefeld soll diese Zeit aber frei einstellbar sein. Die Anforderungsschalter für die Fußgängerampeln sollen in diesem Beispiel noch ohne Funktion bleiben.



Der Plan zeigt die Elemente des Schulungsboards, die für das Beispiel benötigt werden.

2.5.2 Noch ein paar Tipps

Bis jetzt wurden Kommandostrings benutzt, mit denen man einzelne Outputs setzen bzw. Inputs lesen konnte. Um mehrere Outputs zeitgleich zu schalten, stellt das Web-IO ein Kommando zur Verfügung, das über eine Hexadezimalzahl festlegt, welche Outputs gesetzt werden sollen.

GET /outputaccess?PW=&State=0001& setzt zum Beispiel Output 0 auf ON und alle anderen Outputs auf OFF.

Letztlich stehen die Bits 0-11 der hexadezimalen Zahl für die Outputs 0-11. Um zu bestimmen welche Outputs gesetzt-werden sollen, muss also bei einer 12stelligen Dualzahl (Binärzahl) das jeweilige Bit auf 1 für einen eingeschalteten Output und auf 0 für einen ausgeschalteten Output gesetzt werden.

Diese Dualzahl muss dann in eine hexadezimale Zahl umgerechnet werden, um sie in den Kommandostring einzusetzen. Da die Syntax der Kommandostrings 4stellige hexadezimale Zahlen vorschreibt, eine 12stellige Dualzahl aber nur 3 Stellen hexadezimal ergibt, wird dem errechneten Wert einfach eine 0 vorangestellt.

Eine kleine Auffrischung der Zahlensysteme

In der Computertechnik arbeitet man mit Bits und Bytes, also eigentlich im dualen- oder auch binären Zahlensystem.

Duale Zahlen sind für den Menschen leider sehr unübersichtlich. Wer erkennt schon auf Anhieb, dass dual 110001110101 = dezimal 3189 ergibt?

Da man jeden Input bzw. Output des Web-IO als eine Stelle einer 12-stelligen Binärzahl sehen muss, ist es unausweichlich, sich noch einmal mit dieser Materie zu beschäftigen.

```
                        Dualzahl 110001110101
Bit  0  = 2^0  =    1            1 x    1 =    1
Bit  1  = 2^1  =    2            0 x    2 =    0
Bit  2  = 2^2  =    4            1 x    4 =    4
Bit  3  = 2^3  =    8            0 x    8 =    0
Bit  4  = 2^4  =   16            1 x   16 =   16
Bit  5  = 2^5  =   32            1 x   32 =   32
Bit  6  = 2^6  =   64            1 x   64 =   64
Bit  7  = 2^7  =  128            0 x  128 =    0
Bit  8  = 2^8  =  256            0 x  256 =    0
Bit  9  = 2^9  =  512            0 x  512 =    0
Bit 10  = 2^10 = 1024            1 x 1024 = 1024
Bit 11  = 2^11 = 2048            1 x 2048 = 2048
                                 Dezimalzahl 3819
```

Das Umrechnen von Dual- in Dezimal-Zahlen ist nicht schwierig. Dennoch fehlt die spontane Zuordnung zwischen gesetzten Outputs und dem Dezimalwert. Deshalb werden dort, wo der Mensch mit Bits und Bytes jonglieren muss, hexadezimale Zahlen verwendet.

Bei hexadezimalen Zahlen kann die Wertigkeit jeder Stelle durch 15 verschiedene Ziffern dargestellt werden. Da unser dezimales Zahlensystem nur Ziffern von 0 ... 9 kennt, wurde das heaxdezimale System um die Buchstaben A ... F erweitert.

A =10, B=11, C=12, D=13, E=14, F=15.

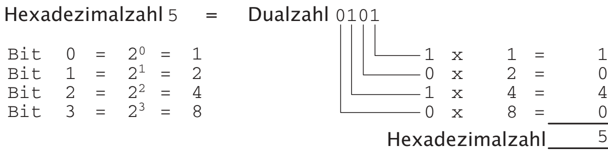
Hier noch mal etwas übersichtlicher.

```
                      Hexadezimalzahl C75
Stelle  0 = 16^0 =   1        5  x   1 =    5
Stelle  1 = 16^1 =  16        7  x  16 =  112
Stelle  2 = 16^2 = 256        12 x 256 = 3072
                              Dezimalzahl 3819
```

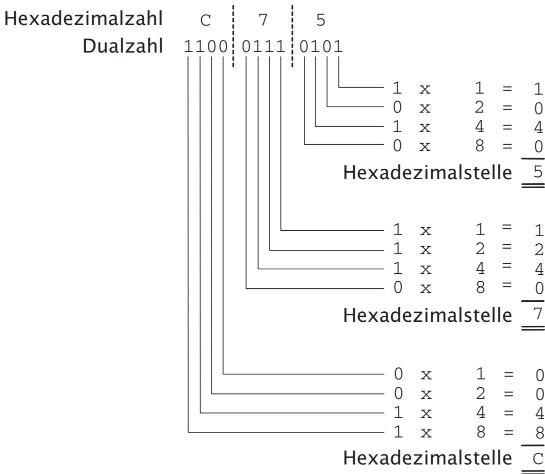
Auf den ersten Blick macht die Benutzung des hexadezimalen Zahlensystems die Darstellung der Inputs und Outputs nicht einfacher.

Aber schauen wir einmal genau hin. Jede Stelle der hexadezimalen Zahl ist eine Potenz zur Basis 16 multipliziert mit der Ziffer. 16 wiederum ist die 4. Potenz von 2, also 2^4.

Jede Stelle des hexadezimalen Zahlensystems lässt sich deshalb durch Addition der 2er Potenzen 2^0 bis 2^3 errechnen.



Zerlegt man nun eine Dualzahl mit der niedrigsten Stelle beginnend in Vierbit-Bereiche, kann man mit wenig Aufwand zwischen Dualzahlen und hexadezimalen Zahlen umrechnen.



Mit etwas Übung lässt sich das bequem im Kopf rechnen.

2.5.3 Die Lösung

Programmplanung

Zunächst sollte wieder festgelegt werden, welche Aufgaben das Programm in welcher Abfolge erledigen muss:

1. Entgegennehmen der Verbindungsparameter (IP-Adresse und Remote-Port des Web-IO
2. Aufbau der TCP-Verbindung
3. Starten des Timers, der die einzelnen Schritte der Ampelsteuerung abwickelt
   Schritt 1 Fahrzeugampeln: gelb
   Fußgängerampeln: rot

Schritt 2 Fahrzeugampeln: rot
Fußgängerampeln: rot
Schritt 3 Fahrzeugampeln: rot
Fußgängerampeln: grün
Schritt 4 Fahrzeugampeln: rot
Fußgängerampeln: rot
Schritt 5 Fahrzeugampeln: rot und gelb
Fußgängerampeln: rot
Schritt 6 Fahrzeugampeln: grün
Fußgängerampeln: rot

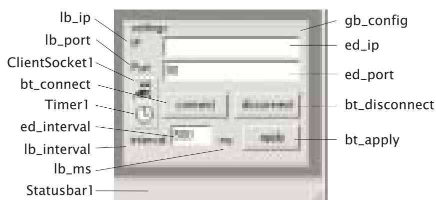
4. Bei Schließen der TCP-Verbindung zunächst die Ampelphasen so zu Ende führen, dass die Grünphase für die Fußgänger abgeschlossen und die Grünphase für die Autos erreicht wird. Danach alle Ampellichter ausschalten und dann erst die Verbindung schließen und die Buttons aktivieren bzw. sperren
5. Bei Beenden des Programms zunächst die Ampelphasen so zu Ende führen, dass die Grünphase für die Fußgänger abgeschlossen und die Grünphase für die Autos erreicht wird. Danach alle Ampellichter ausschalten und dann erst das Programm beenden

Im Programmfenster soll wie gewohnt ein komplettes Prozessabbild wiedergegeben werden.

Vorarbeiten

Als Grundgerüst für diese Anwendung kann das komplette Projekt aus Programmbeispiel 3 benutzt werden. Dazu wird das komplette Projekt in ein neues Verzeichnis kopiert. Das Projekt muss dann aus dem neuen Verzeichnis geöffnet werden.

Die Bedienelemente können so wie sie sind genutzt werden.



Für den Timer1 muss die *Enabled*-Eigenschaft auf *False* gesetzt sein.

Der Quellcode

Der Quellcode von Beispiel 3 muss nur an einigen Stellen modifiziert bzw. erweitert werden.

Die Definition der Farbkonstanten wird in diesem Beispiel das erstemal durchgehend benutzt.

```
const
RED_ON =  clRed;
RED_OFF = clMaroon;
YELLOW_ON = clYellow;
YELLOW_OFF = clOlive;
GREEN_ON = clLime;
GREEN_OFF = clGreen;
BLACK_ON = clSilver;
BLACK_OFF = clBlack;
```

Es wurden vier zusätzliche Variablen deklariert:

vin_applicationstep gibt an in welcher Ampelphase sich das Programm gerade befindet. In *vin_interval* wird das Timerinterval zwischengespeichert. *vbo_disconnectquiry* und *vbo_closequiry* sind sogenannte Flags (Merker), die gesetzt werden wenn der Benutzer die TCP-Verbindung bzw. das komplette Programm zu schließen versucht.

```
var
  fm_TrafficLight            : Tfm_TrafficLight;
  vin_applicationstep        : Integer;
  vin_interval               : Integer;
  vbo_disconnectquiry       : boolean;
  vbo_closequiry             : boolean;

implementation

{$R *.DFM}
```

Hier beginnt das eigentliche Programm.

Bei Programmstart werden zunächst die Variablen auf definierte Werte gesetzt.

```
procedure Tfm_TrafficLight.FormCreate(Sender: TObject);
begin
  vin_applicationstep := 1;
  vbo_disconnectquiry := False;
  vbo_closequiry := False;
  vin_interval := strtoint(ed_interval.text);
end;
```

Die Prozedur *bt_connectClick* wurde unverändert übernommen.

```
procedure Tfm_TrafficLight.bt_connectClick(Sender: TObject);
begin
  if (ed_ip.Text <> '') and (ed_port.Text <> '') then
  begin
    ClientSocket1.Host := ed_ip.Text;
    ClientSocket1.Port := strtoint(ed_Port.Text);
    ClientSocket1.Active := True;
  end;
end;
```

In der Prozedur *ClientSocketConnect* wird zusätzlich das Timerintervall auf ein Zehntel des eingestellten Wertes gesetzt. Ein Zehntel, weil die Zwischenphasen einer Ampel (grün nach gelb, gelb nach rot,....) deutlich kürzer sind als die normalen Grünphasen. Ferner wird der Timer aktiviert.

```
procedure Tfm_TrafficLight.ClientSocket1Connect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  bt_connect.Enabled := False;
  bt_disconnect.Enabled := True;
  StatusBar1.SimpleText := 'connected';
  Timer1.Interval := vin_interval div 10;
  Timer1.Enabled := True;
end;
```

Der Timer steuert abhängig von der Variablen *vin_applicationstep* je nach Ampelphase die LEDs auf dem Schulungsboard und die Programmoberfläche.

Zusätzlich wird über die Variablen *vbo_disconnectquiry* bzw. *vbo_closequiry* überwacht, ob der Anwender die TCP-Verbindung bzw. die Applikation schließen möchte. In diesem Fall werden die Ampelphasen weiter fortgeführt, bis die Grünphase für die Autos erreicht ist. Erst dann werden alle Ampellichter abgeschaltet und die Verbindung beendet.

```
procedure Tfm_TrafficLight.Timer1Timer(Sender: TObject);
begin
  case vin_applicationstep of
```

Phase1:   Gelbphase für Autos
Soll weder die TCP-Verbindung, noch das Programm beendet werden, werden alle Schritte ausgeführt um die Gelbphase für die Autos zu beginnen. Ist *vbo_diconnectquiry* aber gesetzt, werden alle LEDs ausgeschaltet und die TCP-Verbindung beendet. Ist *vbo_closequery* = True wird das Programm geschlossen.

```
        1: begin
             if vbo_disconnectquiry then
             begin
               Timer1.Enabled := False;
               sh_CarWest_Green.Brush.Color := GREEN_OFF;
               sh_CarEast_Green.Brush.Color := GREEN_OFF;
               sh_WalkerWestNorth_Red.Brush.Color := RED_OFF;
               sh_WalkerWestSouth_Red.Brush.Color := RED_OFF;
               sh_WalkerEastNorth_Red.Brush.Color := RED_OFF;
               sh_WalkerEastSouth_Red.Brush.Color := RED_OFF;
               ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0000&');
               sleep(50);
               ClientSocket1.Active := False;
               StatusBar1.SimpleText := 'disconnected';
               bt_connect.Enabled := True;
               if vbo_closequiry then close;
               vbo_disconnectquiry := False;
             end
```

```
    else
    begin
      sh_CarWest_Green.Brush.Color := GREEN_OFF;
      sh_CarWest_Yellow.Brush.Color := YELLOW_ON;
      sh_CarEast_Green.Brush.Color := GREEN_OFF;
      sh_CarEast_Yellow.Brush.Color := YELLOW_ON;
      sh_WalkerWestNorth_Green.Brush.Color := GREEN_OFF;
      sh_WalkerWestNorth_Red.Brush.Color := RED_ON;
      sh_WalkerWestSouth_Green.Brush.Color := GREEN_OFF;
      sh_WalkerWestSouth_Red.Brush.Color := RED_ON;
      sh_WalkerEastNorth_Green.Brush.Color := GREEN_OFF;
      sh_WalkerEastNorth_Red.Brush.Color := RED_ON;
      sh_WalkerEastSouth_Green.Brush.Color := GREEN_OFF;
      sh_WalkerEastSouth_Red.Brush.Color := RED_ON;
      ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0012&');
      Timer1.Interval := vin_interval div 10;
      inc(vin_applicationstep);
    end;
end;
```

Phase 2: Rotphase für Autos und Fußgänger

```
2: begin
    sh_CarWest_Yellow.Brush.Color := YELLOW_OFF;
    sh_CarWest_Red.Brush.Color := RED_ON;
    sh_CarEast_Yellow.Brush.Color := YELLOW_OFF;
    sh_CarEast_Red.Brush.Color := RED_ON;
    ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0014&');
    inc(vin_applicationstep);
end;
```

Phase 3: Grünphase für Fußgänger

Das Timerintervall wird hochgesetzt, damit die Grünphase ausreichend lange dauert

```
3: begin
    sh_WalkerWestNorth_Green.Brush.Color := GREEN_ON;
    sh_WalkerWestNorth_Red.Brush.Color := RED_OFF;
    sh_WalkerWestSouth_Green.Brush.Color := GREEN_ON;
    sh_WalkerWestSouth_Red.Brush.Color := RED_OFF;
```

```
        sh_WalkerEastNorth_Green.Brush.Color := GREEN_ON;
        sh_WalkerEastNorth_Red.Brush.Color := RED_OFF;
        sh_WalkerEastSouth_Green.Brush.Color := GREEN_ON;
        sh_WalkerEastSouth_Red.Brush.Color := RED_OFF;
        ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=000C&');
        Timer1.Interval := vin_interval;
        inc(vin_applicationstep);
    end;
```

Phase 4: Rot für alle
Das Timerintervall wird wieder auf ein Zehntel zurückgesetzt, damit die Zwischenphasen entsprechend kurz sind.

```
   4: begin
        sh_WalkerWestNorth_Green.Brush.Color := GREEN_OFF;
        sh_WalkerWestNorth_Red.Brush.Color := RED_ON;
        sh_WalkerWestSouth_Green.Brush.Color := GREEN_OFF;
        sh_WalkerWestSouth_Red.Brush.Color := RED_ON;
        sh_WalkerEastNorth_Green.Brush.Color := GREEN_OFF;
        sh_WalkerEastNorth_Red.Brush.Color := RED_ON;
        sh_WalkerEastSouth_Green.Brush.Color := GREEN_OFF;
        sh_WalkerEastSouth_Red.Brush.Color := RED_ON;
        ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0014&');
        Timer1.Interval := vin_interval div 10;
        inc(vin_applicationstep);
    end;
```

Phase 5: Rot-Gelb für die Autos

```
   5: begin
        sh_CarWest_Yellow.Brush.Color := YELLOW_ON;
        sh_CarEast_Yellow.Brush.Color := YELLOW_ON;
        ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0016&');
        inc(vin_applicationstep);
    end;
```

Phase 6: Grünphase für die Autos
Wenn die Variable *vbo_diconnect* gesetzt ist, bleibt das Timerintervall auf ein Zehntel, bei normaler Grünphase wird das Intervall hochgesetzt, um eine ausreichend lange Grünphase zu schalten.

```
        6: begin
             sh_CarWest_Green.Brush.Color := GREEN_ON;
             sh_CarWest_Yellow.Brush.Color := YELLOW_OFF;
             sh_CarWest_Red.Brush.Color := RED_OFF;
             sh_CarEast_Green.Brush.Color := GREEN_ON;
             sh_CarEast_Yellow.Brush.Color := YELLOW_OFF;
             sh_CarEast_Red.Brush.Color := RED_OFF;
             ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0011&');
             if vbo_disconnectquiry then
               Timer1.Interval := vin_interval div 10
             else
               Timer1.Interval := vin_interval;
             vin_applicationstep := 1;
           end;
         end;
end;
```

Durch Klick auf den *apply*-Button wird der in das Editfeld eingetragene Wert in die Variable *vin_interval* übernommen.

```
procedure Tfm_TrafficLight.bt_applyClick(Sender: TObject);
begin
  vin_interval := strtoint(ed_interval.Text);
end;
```

Wünscht der Anwender die TCP-Verbindung zu beenden, wird die Variable *vbo_disconnectquiry* auf True gesetzt. Ferner wird in der Stauszeile „waiting for disconnect“ ausgegeben.

```
procedure Tfm_TrafficLight.bt_disconnectClick(Sender: TObject);
begin
  vbo_disconnectquiry :=True;
  bt_disconnect.Enabled := False;
  StatusBar1.SimpleText := 'waiting for disconnect';
end;
```

Wünscht der Anwender bei aktiver Verbindung das Programm zu beenden, werden die Variablen *vbo_disconnectquiry* und *vbo_closequiry* auf *True* gesetzt.

Ferner wird in der Stauszeile „waiting for disconnect“ ausgegeben. Erreicht der Timer beim nächsten Durchlauf Phase 1. wird das Programm beendet.

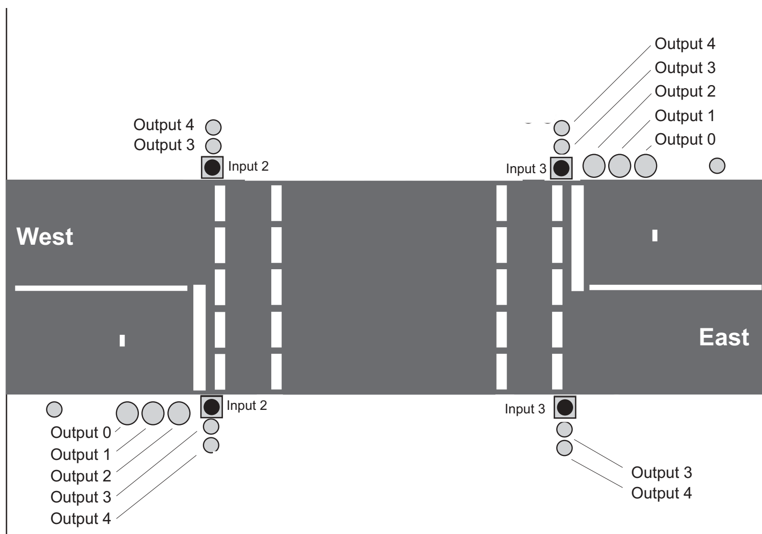
```
procedure Tfm_TrafficLight.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if ClientSocket1.Active = True then
  begin
    CanClose := False;
    vbo_closequiry := True;
    vbo_disconnectquiry :=True;
    StatusBar1.SimpleText := 'waiting for disconnect';
  end
  else CanClose := True;
end;

end.
```

2.6 Eine ereignis- und zeitgesteuerte Anwendung

2.6.1 Die Aufgabe

Das vorangegangene Beispiel soll so umprogrammiert werden, dass die Fahrzeugampeln im Normalfall grün zeigen. Nur wenn einer der Anforderungsschalter an den Fußgängerüberwegen gedrückt wird, soll eine Grünphase für die Fußgänger eingeleitet werden. Dabei sollen natürlich auch alle nötigen Zwischenphasen durchlaufen werden.



Der Plan zeigt die Elemente des Schulungsboards, die für das Beispiel benötigt werden.

2.6.2 Noch ein paar Tipps

In Beispiel 4 haben wir bereits das *OnRead* Ereignis des *Clientsocket*-Steuerlementes genutzt, um eine Prozedur zur Verarbeitung der eingehenden Daten zu starten. Dabei wurde ausschließlich der Status eines einzelnen Inputs berücksichtigt.

Soll ein komplettes Abbild der Inputs abgefragt werden, kann der Kommandostring

```
GET /input?PW=&
```

an das Web-IO gesendet werden, worauf das Web-IO mit

```
input;0001
```

gefolgt von einem Nullbyte antwortet, wenn z.B. nur Input1 gesetzt ist

Das Web-IO gibt aber auch den Zustand der Outputs zurück, wenn ein Kommando zum Setzen erfolgreich abgearbeitet wurde.

Sendet die Anwendung z.B:

```
GET /outputaccess?PW=&State=0004&,
```

antwortet das Web-IO mit

```
output;0004
```

gefolgt von einem Nullbyte.

Je komplexer eine Anwendung wird, um so komplexer muss auch die Prozedur zur Verarbeitung eingehender Daten gestaltet werden.

Wird, wie im Fall unserer bisherigen Beispiele, mit Kommandostrings gearbeitet, müssen die eingehenden Strings auf ihren Inhalt untersucht werden. Man spricht dabei auch von Parsing oder Parsen.

Bisher haben wir die Oberfläche im Programmfenster den Anforderungen entsprechend angezeigt und dabei nicht berücksichtigt, ob der Kommadostring an das Web-IO auch dazugeführt hat, die geforderten Outputs tatsächlich zu setzen.

Im Fall einer Störung oder einer Fehlkonfiguration des Web-IO durch den Einrichter, würden die Aktionen auf dem Bildschirm angezeigt, obwohl sie in der Anwendung selbst, also auf dem Schulungsbord, gar nicht gegriffen haben.

Es wäre deshalb ratsam, die Programmoberfläche basierend auf den Antworten des Web-IO anzuzeigen. Erstrebenswert wäre es natürlich, Funktionen und Prozeduren zu entwickeln, mit denen man durch Übergabe der hexadezimalen Zahl aus dem Antwortstring ein aktuelles Abbild des Schulungsboards auf der Programmoberfläche darstellt.

Die Zustände der einzelnen Outputs entsprechen den zugehörigen Bits der Hexadezimalzahl.

Um die, als String vorliegende, hexadezimale Zahl auszuwerten, muss diese also zunächst in einen Zahlenwert konvertiert werden. Dazu könnte man sich natürlich der, im vorangegangenen Beispiel erklärten, mathematischen Methode bedienen und jedes einzelne Bit ausrechnen.

Solche Operationen sind aber im Programmablauf sehr zeitaufwendig. Um die Geschwindigkeit des Programms zu erhöhen, werden wir im folgenden Beispiel mit logischen Verknüpfungen arbeiten und uns einiger Tricks bedienen die wir hier im Vorfeld schon mal beschreiben wollen. Zunächst noch ein paar Grundlagen:

Strings, Chars und Zahlenvariablen

Ein String ist eine Kette von Einzelzeichen, sogenannten Charactern oder auch Char-Variablen. Jedes Einzelzeichen ist im Speicher als ein Byte abgelegt.

Noch mal zusammengefasst:

- Char-Variablen sind darstellbare Zeichen (Buchstaben, Ziffern und Sonderzeichen).
- Ein String ist eine Kette von Char-Variablen
- Ein Byte ist ein Zahlenwert zwischen 0 und 255

	1.Char	2.Char	3.Char	4.Char
Hexadezimalzahl als String	0	C	7	5
Bytes im Speicher (dezimal)	48	67	55	53
Position im Speicher	n	n+1	n+2	n+3

Auswerten von Strings

Aus Strings können entweder einzelne Zeichen oder ein ganzer Teilbereich heraus kopiert werden. Um einen Teilbereich zu kopieren gibt es die Funktion *copy(String, Position, Zeichenanzahl)*

Beispiel:

Es soll der hexadezimale Anteil von *vst_receivestring* in *vst_hexstring* kopiert werden.

```
var
vst_receivestring : String;
vst_hexstring : String;

begin
  ..
  vst_receivestring := 'output;0001';
 vst_hexstring := copy(vst_hexstring, 8, 4);
```

Um einzelne Zeichen eines Strings auszuwerten, kann im Quelltext dem Stringnamen die Zeichenposition in eckigen Klammern angefügt werden.

Beispiel:

Das erste Zeichen von *vst_hexstring* wird in *vch_fisrtchar* kopiert

```
var     vch_firstchar : Char;
        vst_hexstring : String;

begin
  ..
```

```
  vch_firstchar := vst_hexstring[1];
```

ASCII-Werte von Char-Variablen

Den Zahlenwert, des für eine Char-Variable im Speicher abgelegten Bytes, bezeichnet man auch als ASCII-Wert.

Delphi stellt die Funktion *ord()* zur Verfügung, um den ASCII-Wert einer Char-Variablen zu ermitteln.

Beispiel:

```
var     vch_firstchar : Char;
        vby_asciivalue : Byte;

begin
  ..
  vby_asciivalue := ord(vch_firstchar);
```

Die, im hexadezimalen verwendeten, Zeichen haben folgende ASCII-Werte:

Zeichen	ASCII-Wert
0	48
1	49
2	50
3	51
4	52

Zeichen	ASCII-Wert
5	53
6	54
7	55
8	56
9	57

Zeichen	ASCII-Wert
A	65
B	66
C	67
D	68
E	69
F	70

Den dezimalen Wert eines Hexzeichens bestimmen

Nochmal zur Erinnerung: mit einer Stelle im Hexadezimalen Zahlensystem können dezimale Werte von 0 bis 15 dargestellt werden. Werte größer 9 werden mit den Buchstaben A - F dargestellt.

Die ASCII-Werte zu den Ziffern liegen zwischen 48 und 57.

Daraus kann abgeleitet werden, dass es sich bei Zeichen deren ASCII-Wert nicht größer ist als 57 um Ziffern handelt.

Um für die Ziffern den dezimalen Gegenwert zu bekommen, muss vom entsprechenden ASCII-Wert 48 abgezogen werden.

Beispiel:

Der ASCII-Wert für die Ziffer 5 entspricht 53.

53 - 48 = 5

Ist der ASCII-Wert größer als 57 handelt es sich um einen Buchstaben.

Um für einen der sechs erlaubten Buchstaben (A - F) den dezimalen Gegenwert zu bekommen, muss vom entsprechenden ASCII-Wert 55 abgezogen werden.

Beispiel:

Der ASCII-Wert für den Buchstaben C entspricht 67.

67 - 55 = 12; *hexadezimal C entspricht dezimal 12*

Eine Funktion zur Umrechnung könnte in Delphi so aussehen:

```
function hexchartobyte(vch_hexchar : Char) : Byte;
var vby_asciivalue : Byte;
begin
  vby_asciivalue := ord(vch_hexchar);
    if vby_asciiwert > 57 then
       hexchartobyte := vby_asciivalue - 55
    else
       hexchartobyte := vby_asciivalue - 48;
end;
```

Diese Funktion klappt aber nur mit einem Zeichen und bedingt, dass die Buchstaben in Großschrift übergeben werden.

Dezimaler Wert einer mehrstellingen hex. Zahl

Die Umrechnung mehrstelliger hexadezimaler Zahlen kann im Grunde genauso durchgeführt werden, wobei für jede hexadezimale Stelle ein Umrechnungsdurchlauf zu erfolgen hat.

Dazu muss zunächst die Länge des Hexadezimalen Strings ermittelt werden. Die Funktion *length(Stringname)* gibt die Länge des in Klammern stehenden Strings zurück.

Ferner sollte bei jedem Umrechnungsdurchlauf die Wertigkeit der entsprechenden Stelle berücksichtigt werden.

Erste hexidezimale Stelle = dezimaler Wert * 16^0
Zweite hexidezimale Stelle = dezimaler Wert * 16^1
Dritte hexidezimale Stelle = dezimaler Wert * 16^2
Vierte hexidezimale Stelle = dezimaler Wert * 16^3

Es muss das dezimale Ergebnis jeder hexadezimalen Stelle mit der entsprechenden Potenz zur Basis 16 multipliziert und zum Gesamtergebnis addiert werden.

Beispiel für eine zweistellige, hexadezimale Zahl:

Wert = 1.Stelle * 16^0 + 2.Stelle * 16^1

Noch ein Tipp: die Position der einzelnen Zeichen eines Strings beginnt beim ersten Zeichen von links. Die niederwertigste Stelle eines hexadezimalen Strings befindet sich dagegen rechts.

Bitschieben statt Potenzieren

Da Delphi keine Funktion zum Potenzieren von Integerwerten bereitstellt, bedienen wir uns eines kleinen Tricks.

Jede Multiplikation mit 16 entspricht einer Verschiebung um 4 Bit nach links.

Hier ein Beispiel für die hexadezimale Zahl 75:

Dezimalzahl	117	
Hexadezimalzahl	7	5
Dualzahl	0111	0101

zunächst wird die zweite Stelle separiert,

Dezimalzahl	7
Hexadezimalzahl	7
Dualzahl	0111

danach wird dieser Wert mit 16 multipliziert oder um 4 Bit nach links verschoben,

```
      Dezimalzahl  112 = 7 * 16
Hexadezimalzahl     7  |  0  |
         Dualzahl  0111| 0000|
                      <------
           Verschiebung um 4 Bit
```

addiert man nun die 5 wieder dazu

```
      Dezimalzahl     117
Hexadezimalzahl     7  |  5  |
         Dualzahl  0111| 0101|
```

bekommt man als Ergebnis wieder die ursprüngliche hexadezimale Zahl.

Es bleibt sich also gleich ob mit 16 multipliziert wird oder ob man eine Verschiebung um 4 Bit nach links vornimmt.

Der passende Befehl für das Verschieben von Bits nach links in Delphi lautet: *Wert shl Bitzahl*

4 stellige hexadezimale Zahlen nach Dezimal wandeln

Die im Folgenden beschriebene Funktion konvertiert eine maximal 4 stellige hexadezimale Zahl in eine Ganzzahlen-Variable vom Typ Word.

Der Variablentyp Word ist ein vorzeichenloser 2Byte Integerwert, es können also Werte zwischen 0 und 65535 gespeichert werden.

Um die Funktion unabhängig von Groß- und Kleinschreibung betreiben zu können, wird die Delphi-Funktion *UpperCase* benutzt, die die Buchstaben in einem String grundsätzlich in Großschreibung wandelt.

```
function HexToInt(HexString : String) : Word;
var vin_CharCount            : integer;
    vwo_HexCharValue         : Word;
    vwo_HexValue             : Word;
begin
   vwo_HexValue := 0;
   HexString := UpperCase(HexString);
```

```
   for vin_CharCount := 1 to length(HexString) do
   begin
     vwo_HexCharValue := ord(HexString[vin_CharCount]);
     if vwo_HexCharValue > 57 then
        vwo_HexCharValue := vwo_HexCharValue - 55
     else
        vwo_HexCharValue := vwo_HexCharValue - 48;
     vwo_HexCharValue :=
     vwo_HexCharValue shl ((length(HexString) - vin_CharCount) *  4);
     vwo_HexValue := vwo_HexValue or vwo_HexCharValue;
   end;
   HexToInt := vwo_HexValue;
end;
```

Einzelne Bits ausmaskieren

Mit der Funktion *HexToInt* haben wir nun die Möglichkeit, hexadezimale Strings in Zahlenwerte zu wandeln.

Zahlenwerte können automatisiert mit Hilfe logischer Operationen zur weiteren Verarbeitung ausgewertet werden.

Geht es um die Outputs, entsprechen die ersten 12 Bit den Outputs 0-11. Wird der Zustand eines einzelnes Bit benötigt, müssen zunächst alle anderen Bits ausgeblendet werden. Das kann über eine UND-Verknüpfung erreicht werden.

Beispiel:

Es soll ermittelt werden, ob Output 5, also das 5. Bit gesetzt ist oder nicht.

	Dual	Dezimal
Wert	1100 0111 0101	3819
UND	0000 0010 0000	32
=	0000 0010 0000	32

Dazu wird der vorliegende Wert mit der Zahl 32 UND-verknüpft (entspricht einer Zahl bei der nur das 5.Bit gesetzt ist). Ist der Status von Output 5 = ON, ist das Ergebnis der UND-Verknüpfung gleich 2^5=32, ist der Zustand = OFF ist das Ergebnis gleich 0.

In Delphi:

```
var vwo_HexValue;
begin
if vwo_HexValue and 32 = 32 then
begin
..          //wieteres Vorgehen bei Bit5=1
end
else
begin
..          //wieteres Vorgehen bei Bit5=0
end;
```

2.6.3 Die Lösung

Programmplanung

Zunächst sollte wieder festgelegt werden, welche Aufgaben das Programm in welcher Abfolge erledigen muss:

1. Entgegennehmen der Verbindungsparameter (IP-Adresse und Remote-Port des Web-IO
2. Aufbau der TCP-Verbindung
3. Starten des Timers, der zyklisch die Inputs des Web-IO abfragt.
   Wenn ein ON auf einem der beiden betroffenen Inputs anliegt, starten des Timers, der die einzelnen Schritte der Ampelsteuerung abwickelt
   Schritt 1 Fahrzeugampeln: gelb
   Fußgängerampeln: rot
   Schritt 2 Fahrzeugampeln: rot
   Fußgängerampeln: rot
   Schritt 3 Fahrzeugampeln: rot
   Fußgängerampeln: grün
   Schritt 4 Fahrzeugampeln: rot
   Fußgängerampeln: rot
   Schritt 5 Fahrzeugampeln: rot und gelb
   Fußgängerampeln: rot
   Schritt 6 Fahrzeugampeln: grün
   Fußgängerampeln: rot

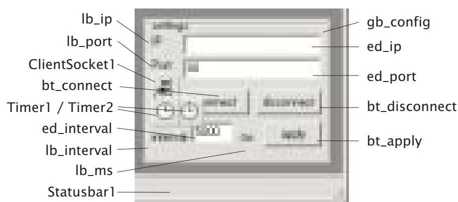
4. Bei Schließen der TCP-Verbindung zunächst die Ampelphasen so zu Ende führen, dass die Grünphase für die Fußgänger abgeschlossen und die Grünphase für die Autos erreicht wird. Danach alle Ampellichter ausschalten und dann erst die Verbindung schließen und die Buttons aktivieren bzw. sperren.
5. Bei Beenden des Programms zunächst die Ampelphasen so zu Ende führen, dass die Grünphase für die Fußgänger abgeschlossen und die Grünphase für die Autos erreicht wird. Danach alle Ampellichter ausschalten und dann erst das Programm beenden.

Im Programmfenster soll wie gewohnt ein komplettes Prozessabbild wiedergegeben werden, allerdings erst wenn das Setzen der Outputs vom Web-IO quittiert wurde.

Vorarbeiten

Als Grundgerüst für diese Anwendung kann das komplette Projekt aus Programmbeispiel 5 benutzt werden. Dazu wird das komplette Projekt in ein neues Verzeichnis kopiert. Das Projekt muss dann aus dem neuen Verzeichnis geöffnet werden.

Die Bedienelemente lassen sich, wie sie sind nutzten, es wird aber ein weiterer Timer benötigt.



Für beide Timer muss die *Enabled*-Eigenschaft auf *False* gesetzt sein.

Der Quellcode

Als Grundgerüst haben wir den Quelltext aus Beispiel 5 genommen.

Im Bereich *private* wurden drei selbstentwickelte Prozeduren deklariert, die nicht an Steuerelemente des Formulars gebunden sind. Der Quelltext und die Aufgaben dieser Prozeduren werden im weiteren Verlauf noch erklärt.

Alle Prozeduren die nicht zu einem Steuerelement gehören, müssen im Bereich Privat *deklariert werden, damit diese z.B. auf globale Variablen zugreifen können und damit die Prozeduren aus anderen Programmteilen aufrufbar sind.*

```
type
  Tfm_TrafficLight = class(TForm)
    im_junction: TImage;
    StatusBar1: TStatusBar;
    ..
    procedure ClientSocket1Read(Sender: TObject; Socket: TCustomWinSocket);
    procedure Timer2Timer(Sender: TObject);
  private
    { Private-Deklarationen }
    procedure visualiseoutputs(outputs : word);
    procedure visualiseinputs(Inputs : word);
    procedure activitiesbyinputs(Inputs : word);
  public
    { Public-Deklarationen }
  end;
```

Festlegung der benutzten Farbkonstanten.

```
const
RED_ON =  clRed;
RED_OFF = clMaroon;
YELLOW_ON = clYellow;
YELLOW_OFF = clOlive;
GREEN_ON = clLime;
GREEN_OFF = clGreen;
BLACK_ON = clSilver;
BLACK_OFF = clBlack;
```

Konstanten für die Bitwertigkeit der einzelnen Outputs.

```
OUTPUT0 = 1;
```

```
OUTPUT1 = 2;
OUTPUT2 = 4;
OUTPUT3 = 8;
OUTPUT4 = 16;
OUTPUT5 = 32;
OUTPUT6 = 64;
OUTPUT7 = 128;
OUTPUT8 = 256;
OUTPUT9 = 512;
OUTPUT10 = 1024;
OUTPUT11 = 2048;
```

Konstanten für die Bitwertigkeit der einzelnen Inputs.
(aus Platzgründen nicht alle abgebildet)

```
INPUT0 = 1;
INPUT1 = 2;
..
INPUT10 = 1024;
INPUT11 = 2048;
```

Deklaration der verwendeten Variablen:

vin_applicationstep gibt an, in welcher Ampelphase sich das Programm gerade befindet. In *vin_interval* wird das Timer-interval zwischengespeichert. *vbo_disconnectquiry* und *vbo_closequiry* sind sogenannte Flags (Merker), die gesetzt werden wenn der Benutzer die TCP-Verbindung bzw. das komplette Programm zu schließen versucht. *vbo_walkerrequest* ist ebenfalls ein Flag, das gesetzt wird, wenn ein Anforderungstaster am Fußgängerweg gedrückt wurde.

```
var
  fm_TrafficLight          : Tfm_TrafficLight;
  vin_applicationstep      : Integer;
  vin_interval             : Integer;
  vbo_disconnectquiry      : boolean;
  vbo_closequiry           : boolean;
 vbo_walkerrequest      : boolean;
```

```
implementation
{$R *.DFM}
```

Die Funktion *HexToInt* wandelt einen hexadezimalen String in eine Variable vom Typ Word

```
function HexToInt(HexString : String) : Word;
var vin_CharCount           : integer;
    vwo_HexCharValue        : Word;
    vwo_HexValue            : Word;
begin
   vwo_HexValue := 0;
   HexString := UpperCase(HexString);
   for vin_CharCount := 1 to length(HexString) do
   begin
     vwo_HexCharValue := ord(HexString[vin_CharCount]);
     if vwo_HexCharValue > 57 then
        vwo_HexCharValue := vwo_HexCharValue - 55
     else
        vwo_HexCharValue := vwo_HexCharValue - 48;
     vwo_HexCharValue := vwo_HexCharValue shl ((length(HexString) - vin_CharCount)
*  4);
     vwo_HexValue := vwo_HexValue or vwo_HexCharValue;
   end;
   HexToInt := vwo_HexValue;
end;
```

Die Prozedur *visualiseoutputs* wertet den empfangenen Outputwert durch Ausmaskieren aus und setzt die Elemente auf der Programmoberfläche analog zu den Elementen auf dem Schulungsboard.

```
procedure Tfm_TrafficLight.visualiseoutputs(Outputs : word);
begin
  if Outputs and OUTPUT0 = OUTPUT0 then
  begin
    sh_CarWest_Green.Brush.Color := GREEN_ON;
    sh_CarEast_Green.Brush.Color := GREEN_ON;
  end
  else
  begin
```

```
    sh_CarWest_Green.Brush.Color := GREEN_OFF;
    sh_CarEast_Green.Brush.Color := GREEN_OFF;
  end;
  if Outputs and OUTPUT1 = OUTPUT1 then
  begin
    sh_CarWest_Yellow.Brush.Color := YELLOW_ON;
    sh_CarEast_Yellow.Brush.Color := YELLOW_ON;
  end
  else
  begin
    sh_CarWest_Yellow.Brush.Color := YELLOW_OFF;
    sh_CarEast_Yellow.Brush.Color := YELLOW_OFF;
  end;
  if Outputs and OUTPUT2 = OUTPUT2 then
  begin
    sh_CarWest_Red.Brush.Color := RED_ON;
    sh_CarEast_Red.Brush.Color := RED_ON;
  end
  else
  begin
    sh_CarWest_Red.Brush.Color := RED_OFF;
    sh_CarEast_Red.Brush.Color := RED_OFF;
  end;
  if Outputs and OUTPUT3 = OUTPUT3 then
  begin
    sh_WalkerWestNorth_Green.Brush.Color := GREEN_ON;
    sh_WalkerWestSouth_Green.Brush.Color := GREEN_ON;
    sh_WalkerEastNorth_Green.Brush.Color := GREEN_ON;
    sh_WalkerEastSouth_Green.Brush.Color := GREEN_ON;
  end
  else
  begin
    sh_WalkerWestNorth_Green.Brush.Color := GREEN_OFF;
    sh_WalkerWestSouth_Green.Brush.Color := GREEN_OFF;
    sh_WalkerEastNorth_Green.Brush.Color := GREEN_OFF;
    sh_WalkerEastSouth_Green.Brush.Color := GREEN_OFF;
  end;
  if Outputs and OUTPUT4 = OUTPUT4 then
  begin
    sh_WalkerWestNorth_Red.Brush.Color := RED_ON;
    sh_WalkerWestSouth_Red.Brush.Color := RED_ON;
```

```
    sh_WalkerEastNorth_Red.Brush.Color := RED_ON;
    sh_WalkerEastSouth_Red.Brush.Color := RED_ON;
  end
  else
  begin
    sh_WalkerWestNorth_Red.Brush.Color := RED_OFF;
    sh_WalkerWestSouth_Red.Brush.Color := RED_OFF;
    sh_WalkerEastNorth_Red.Brush.Color := RED_OFF;
    sh_WalkerEastSouth_Red.Brush.Color := RED_OFF;
  end;
end;
```

visualiseinputs zeigt bei Übergabe des Inputwertes in der Programmoberfläche an, wenn einer der Anfordeungstaster gedrückt wurde.

```
procedure Tfm_TrafficLight.visualiseinputs(Inputs : word);
begin
  if Inputs and INPUT2 = INPUT2 then
  begin
    sh_RequestWestNorth.Brush.Color := BLACK_ON;
    sh_RequestWestSouth.Brush.Color := BLACK_ON;
  end
  else
  begin
    sh_RequestWestNorth.Brush.Color := BLACK_OFF;
    sh_RequestWestSouth.Brush.Color := BLACK_OFF;
  end;
  if Inputs and INPUT3 = INPUT3 then
  begin
    sh_RequestEastNorth.Brush.Color := BLACK_ON;
    sh_RequestEastSouth.Brush.Color := BLACK_ON;
  end
  else
  begin
    sh_RequestEastNorth.Brush.Color := BLACK_OFF;
    sh_RequestEastSouth.Brush.Color := BLACK_OFF;
  end;
end;
```

activitiesbyinputs wertet ebenfalls den Inputwert aus. Es wird *vbo_walkerrequest = True* gesetzt und *Timer1* gestartet, wenn ein gedrückter Anforderungstaster erkannt wurde.

```
procedure Tfm_TrafficLight.activitiesbyinputs(Inputs : word);
begin
  if Inputs and INPUT2 = INPUT2 then
  begin
    Timer1.Enabled := True;
    vbo_walkerrequest := True;
  end;
  if Inputs and INPUT3 = INPUT3 then
  begin
    Timer1.Enabled := True;
    vbo_walkerrequest := True;
  end;
end;
```

Bei Programmstart werden die benötigten Variablen auf einen definierten Anfangszustand gesetzt.

```
procedure Tfm_TrafficLight.FormCreate(Sender: TObject);
begin
  vin_applicationstep := 1;
  vin_interval := strtoint(ed_interval.Text);
  Timer1.Interval := vin_interval div 10;
end;
```

Die Prozedur *ConnectClick* wurde unverändert übernommen.

```
procedure Tfm_TrafficLight.bt_connectClick(Sender: TObject);
begin
  if (ed_ip.Text <> '') and (ed_port.Text <> '') then
  begin
    ClientSocket1.Host := ed_ip.Text;
    ClientSocket1.Port := strtoint(ed_Port.Text);
    ClientSocket1.Active := True;
  end;
end;
```

Bei erfolgreichem Verbindungsaufbau werden neben dem Freischalten/Sperren der Buttons beide Timer gestartet. *vbo_walkerrequest* wird auf *true* gesetzt und damit wird zunächst eine komplette Grünphase für Fußgänger durchlaufen.

```
procedure Tfm_TrafficLight.ClientSocket1Connect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  bt_connect.Enabled := False;
  bt_disconnect.Enabled := True;
  StatusBar1.SimpleText := 'connected';
  vbo_walkerrequest := True;
  Timer1.Interval := vin_interval div 10;
  Timer1.Enabled := True;
  Timer2.Enabled := True;
end;
```

Timer1 steuert den zeitlichen Ablauf der Ampelphasen und sorgt für einen geregelten Verbindungsabbau bzw. ein geregeltes Programmende.

Im Gegensatz zum voran gegangenen Beispiel, wird in der Timer Prozedur die Programmoberfläche bis auf eine Ausnahme nicht mehr beeinflusst (dies wird erst bei Empfang der Antworten vom Web-IO ausgelöst)

```
procedure Tfm_TrafficLight.Timer1Timer(Sender: TObject);
begin
 case vin_applicationstep of
```

Phase1: Gelbphase für Autos
Ist *vbo_walkerrequest = True,* werden alle Schritte ausgeführt um die Gelbphase für die Autos zu beginnen, anderenfalls wird der Timer gestoppt. Ist *vbo_diconnectquiry* gesetzt, werden alle LEDs ausgeschaltet und die TCP-Verbindung beendet. Da die Antwort vom Web-IO mit dem Outputstatus durch das Schließen der Verbindung ggf. nicht mehr gesendet wird, wird der Prozedur *visuliseoutputs* der hexadezimale Wert 0000 übergeben, um in der Programmoberfläche ein korrektes Abbild des Schulungsboards anzu-

zeigen. Ist *vbo_closequery = True* wird das Programm geschlossen.

```
1: begin
     Timer1.Interval := vin_interval div 10;
     if vbo_walkerrequest then
     begin
       ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0012&');
       inc(vin_applicationstep);
     end
     else
     begin
       Timer1.Enabled := false;
       if vbo_disconnectquiry then
       begin
         ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0000&');
         visualiseoutputs($0000);
         sleep(50);
         ClientSocket1.Active := False;
         StatusBar1.SimpleText := 'disconnected';
         bt_connect.Enabled := True;
         if vbo_closequiry then close;
         vbo_disconnectquiry := False;
       end;
     end;
 end;
```

Phase 2: Rot für alle

```
2: begin
     ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0014&');
     inc(vin_applicationstep);
 end;
```

Phase 3: Grün für die Fußgänger

Das Timer-Intervall wird auf seinen normalen Wert gesetzt.

```
3: begin
     ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=000C&');
     Timer1.Interval := vin_interval;
     inc(vin_applicationstep);
```

```
end;
```

Phase 4: Rot für alle
Das Timer-Intervall wird auf ein Zehntel seines normalen Wertes gesetzt (kurze Zwischenphase.).

```
4: begin
     ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0014&');
     Timer1.Interval := vin_interval div 10;
     inc(vin_applicationstep);
 end;
```

Phase 5: Rot-Gelb für die Fahrzeugampeln

```
5: begin
     ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0016&');
     inc(vin_applicationstep);
 end;
```

Phase 6: Grün für die Autos
Die Variable *vbo_walkerrequest* wird auf *False* gesetzt, damit der Timer beim nächsten Durchlauf gestoppt wird und die Grünphase bis zur nächsten Anforderung durch die Fußgänger bestehen bleibt.

Ist *vbo_disconnectquiry* gesetzt, wird das Timer-Intervall auf ein Zehntel seines Wertes gesetzt. Zum Abschluss wird *vin_applicationstep* auf 1 gesetzt, damit beim nächsten Durchlauf wieder mit der Gelbphase für Autos gestartet werden kann.

```
6: begin
     ClientSocket1.Socket.SendText('GET /outputaccess?PW=&State=0011&');
     vbo_walkerrequest := False;
     if vbo_disconnectquiry then
       Timer1.Interval := vin_interval div 10
     else
       Timer1.Interval := vin_interval;
     vin_applicationstep := 1;
   end;
 end;
```

```
end;
```

Timer2 wird bei erfolgreichem Verbindungsaufbau gestartet, und pollt die Eingänge des Web-IO durch zyklisches Senden des Input Kommandostrings ab.

```
procedure Tfm_TrafficLight.Timer2Timer(Sender: TObject);
begin
  ClientSocket1.Socket.SendText('GET /input?PW=&');
end;
```

Durch Klick auf den *apply*-Button wird das eingetragene Timer-Intervall in die Variable *vin_intervall* geschrieben..

```
procedure Tfm_TrafficLight.bt_applyClick(Sender: TObject);
begin
  vin_interval := strtoint(ed_interval.Text);
end;
```

Die Prozedur *ClientSocket1Read* wird gestartet, wenn Daten vom Web-IO empfangen werden.

Die empfangenen Daten werden zunächst in die Variable *vst_InBuffer* geschrieben.

Dann wird geprüft, ob das erste Zeichen des empfangenen Strings ‘o’ (output) oder ‘i’ (input) ist. Abhängig davon, wird der hexadezimale Anteil des Strings in die Variable *vst_HexString* kopiert.

Bei Empfang der Output-Zustände wird *vst_HexString* an die Prozedur *visuliseoutputs()* übergeben, die die Programmoberfläche entsprechend verändert.

Betrifft der empfangene String die Inputs wird *vst_HexString* an die Prozedur *visuliseinputs()* übergeben, die die gedrückten Taster in der Programmoberfläche anzeigt. Ferner wird die Prozedure *activitiesbyinputs()* aufgerufen, die den weiteren Programmablauf bestimmt.

```
procedure Tfm_TrafficLight.ClientSocket1Read(Sender: TObject;
  Socket: TCustomWinSocket);
```

```
var vst_InBuffer : String;
    vst_HexString : String;
begin
  vst_InBuffer := ClientSocket1.Socket.ReceiveText;
  case vst_InBuffer[1] of

      'o' :  begin
               vst_HexString := copy(vst_InBuffer, 8, 4);
               visualiseoutputs(HexToInt(vst_HexString));
             end;
      'i' :  begin
               vst_HexString := copy(vst_InBuffer, 7, 4);
               visualiseinputs(HexToInt(vst_HexString));
               activitiesbyinputs(HexToInt(vst_HexString));
             end;
  end;
end;
```

Wünscht der Anwender die TCP-Verbindung zu beenden, wird die Variable *vbo_disconnectquiry* auf *True* gesetzt und *Timer1* gestartet, der in der ersten Phase (*vin_applicationstep = 1*) die Ampellichter ausschaltet und die TCP-Verbindung beendet. Ferner wird in der Statuszeile „waiting for disconnect“ ausgegeben.

```
procedure Tfm_TrafficLight.bt_disconnectClick(Sender: TObject);
begin
  vbo_disconnectquiry :=True;
  Timer1.Enabled := True;
  bt_disconnect.Enabled := False;
  StatusBar1.SimpleText := 'waiting for disconnect';
end;
```

Wünscht der Anwender bei aktiver Verbindung das Programm zu beenden, werden die Variablen *vbo_disconnectquiry* und *vbo_closequiry* auf *True* gesetzt. *Timer1* wird gestartet, der in ersten Phase (*vin_applicationstep = 1*) die Ampellichter ausschaltet und die TCP-Verbindung und das Programm beendet. Ferner wird in der Statuszeile „waiting for disconnect“ ausgegeben.

```
procedure Tfm_TrafficLight.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if ClientSocket1.Active = True then
  begin
    CanClose := False;
    Timer1.enabled := True;
    vbo_closequiry := True;
    vbo_disconnectquiry :=True;
    StatusBar1.SimpleText := 'waiting for disconnect';
  end
  else CanClose := True;
end;

end.
```

Mit einem überschaubaren Maß an Quelltext haben wir bereits eine komplexe IO-Anwendung realisiert.

Die Prozeduren *visuliseoutputs* und *visuliseinputs* können für noch komplexere Anwendung so erweitert werden, dass alle Anzeigeobjekte (auch für die Straße von Nord nach Süd) berücksichtigt werden.

3 Programmierung des Web-IO unter Visual Basic

Bei allen hier gezeigten Programmbeispielen und Erklärungen wird der grundsätzliche Umgang mit der Programmierumgebung von Microsoft Visual Basic vorausgesetzt.

Es wurde Microsoft Visual Basic in der Version 6.0 verwendet.

3.1 Ein einführendes Beispiel (1)

3.1.1 Die Aufgabenstellung

Es soll ein einfaches Programm erstellt werden, mit dem über das Web-IO Digital die grünen Ampellichter der Straße West/Ost ein- und ausgeschaltet werden können.

Dazu soll eine TCP-Verbindung genutzt werden.

Folgende Aufgaben müssen vom Programm abgearbeitet werden:

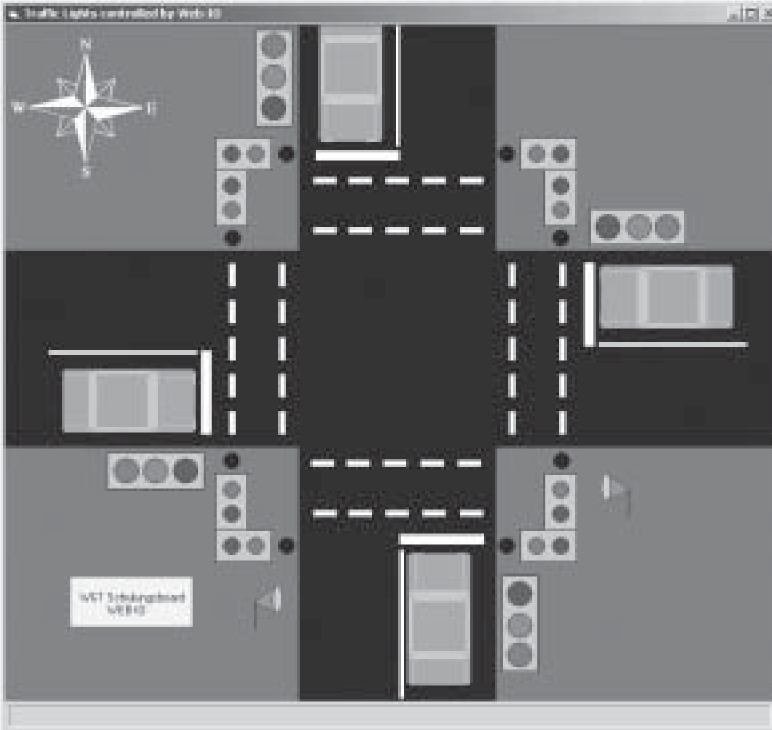
1. Entgegennehmen der Verbindungsparameter (IP-Adresse und Remote-Port des Web-IO)
2. Aufbau der TCP-Verbindung
3. Freigeben der Buttons für das Schalten der Ampellichter
4. Ein- und Ausschalten der grünen Ampellichter über zwei Buttons („on“ und „off“)
5. Schließen der TCP-Verbindung und der Buttons für das Schalten der Ampellichter

Im Programmfenster soll ein komplettes Abbild der Straßenkreuzung mit allen Elementen dargestellt werden, in dem die Abläufe auf dem Schulungsboard 1:1 wiedergegeben werden.

Vorarbeit zum eigentlichen Programm
Um einen schnellen Einstieg in die eigentliche Programmierarbeit zu ermöglichen, haben wir ein komplettes Programm-Formular vorbereitet.

Es bietet dem Programmierer ein programmtechnisches Abbild der Straßenkreuzung, bei dem bereits Steuerelemente für die einzelnen Ampeln, Anfordeungsschalter und ankommenden Autos vorhanden sind.

Dieses Formular wird die Grundlage für alle weiteren hier gezeigten Programmbeispiele sein und ist auf der beiliegenden CD unter visualbasic\basislayout zu finden.



Im ersten Beispiel haben wir den Quelltext bereits gemäß den Anforderungen ergänzt, so dass Sie unter VisualBasic/Step1 den kompletten Quelltext für eine funktionierende Applikation finden.

Grundlegende Festlegungen

Die in diesem Text und den Programmen auftretenden Bezeichnungen sind von uns willkürlich gewählt, folgen aber alle einer gewissen „Logik“. Dabei haben wir folgendes Schema gewählt:

```
sh_CarWest_Green
│    │         └─ Farben oder Zustände „ON“ und „OFF“
│    │
│    └─────────── Lage des Elements,
│                 z.B. WalkerEastNorth=eine bestimmte Fußgängerlampe
│                 oder z.B. RequestEastNorth für den ost-nördlichen Drücker
│                 an der Fußgängerampel
│
└──────────────── Art des Steuerelements bzw der Variablen,
                  z.B. pi = picture, cb=command button, sh=shape usw.
```

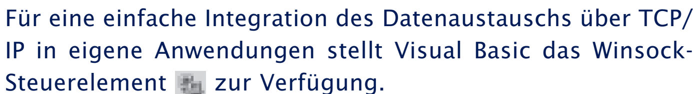
Bei den Fußgängerampeln und Anforderungsdrückern steht die Himmelsrichtung der zugehörigen Straßenrichtung an erster Stelle.

Beispiel: Fußgängerampel für den nördlichen Überweg linksseitig = WalkerNorthWest

Variablennamen beginnen immer mit einem v, gefolgt von einer Zwei-Buchstaben-Kombination, die für den Variablentyp steht (z.B. bo für bool).

Beispiel: vbo_CloseQuiry

Kommunikation über TCP/IP

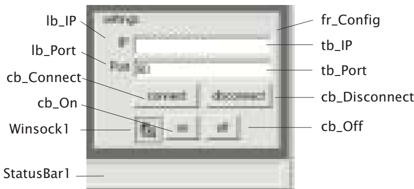
Für eine einfache Integration des Datenaustauschs über TCP/IP in eigene Anwendungen stellt Visual Basic das Winsock-Steuerelement  zur Verfügung.

Das *Winsock*-Steuerelement können Sie in Visual Basic neu in ein Form einfügen, indem Sie unter „Allgemein“ mit der rechten Maustaste klicken, „Komponenten...“ auswählen und unter dem Reiter „Steuerelemente“ die Checkbox vor *Microsoft Winsock Control 6.0* aktivieren. Danach erscheint das Symbol wie die anderen Symbole unter „Allgemein“ und man kann es dem Form hinzufügen. Es ist zur Laufzeit des Programms nicht sichtbar.

3.1.2 Die Lösung

Die benötigten Elemente

Folgende Steuerelemente haben wir in das vorgegebene Formular zusätzlich eingefügt, um das Programm bedienbar zu machen.



Die Funktion der Buttons und der Eingabefelder sollte aus der Namensgebung und Beschriftung hervorgehen.

Der Quellcode

Für die einzelnen Ampelfarben haben wir leicht verständliche Konstanten definiert. Dadurch wird der Quelltext deutlich überschaubarer. RED_OFF lässt sich beim Lesen sicher besser zuordnen als &H80&.

```
Private Const GREEN_OFF = &H8000&              'dunkelgrün
Private Const GREEN_ON = &HFF00&               'hellgrün
```

```
Private Const RED_OFF = &H80&                'dunkelrot
Private Const RED_ON = &HFF&                 'hellrot
Private Const YELLOW_OFF = &H8080&           'dunkelgelb
Private Const YELLOW_ON = &HFFFF&            'hellgelb
```

Selbst definierte Variablen werden in diesem Beispiel noch nicht verwendet.

Nun beginnt der eigentliche Programmcode:

die Prozedur *cb_Connect_Click* wird aufgerufen, wenn der Anwender den *connect*-Button anklickt.

Zunächst wird überprüft, ob IP-Adresse und Port in die entsprechenden Felder eingegeben wurden, und nur dann wird die Prozedur weiter abgearbeitet.

IP-Adresse und Port-Nr. werden an die entsprechenden Eigenschaften des *Winsock*-Steuerelements übergeben.

Nun wird die *Connect*-Methode des *Winsock*-Steuerelements aufgerufen, um den Verbindungsaufbau einzuleiten.

```
Private Sub cb_Connect_Click()
  If (tb_Port.Text <> "") And (tb_IP.Text <> "") Then
      Winsock1.RemoteHost = tb_IP.Text
      Winsock1.RemotePort = tb_Port.Text
      Winsock1.Connect
  End if
End Sub
```

Wenn eine TCP-Verbindung zustande kommt, wird die Prozedur *Winsock1_Connect()* vom *Winsock*-Steuerelement aufgerufen.

Es wird dann der *connect*-Button für die Bedienung gesperrt, während der *disconnect*-Button und der *on*-Button freigegeben werden.

In die Statuszeile wird „connected“ geschrieben.

```
Private Sub Winsock1_Connect()
  cb_Connect.Enabled = False
  cb_Disconnect.Enabled = True
  StatusBar1.SimpleText = "connected"
  cb_ON.Enabled = True
End Sub
```

Durch Klicken auf den *on*-Button wird der Kommandostring „GET /outputaccess0?PW=&State=ON&“ an die Methode *SendData* des *Winsock*-Steuerelements übergeben. Das *Winsock*-Steuerelement sendet diesen String über die bestehende TCP-Verbindung an das Web-IO.

Das Web-IO lässt sich -wie hier gezeigt- mit vordefinierten Kommandostrings steuern und lesen. Die möglichen Kommandos und deren Aufbau sind als Übersicht im Anhang zu finden. Eine ausführliche Beschreibung finden Sie im Referenzhandbuch zum Web-IO.

Der gesendete Kommandostring schaltet Output0 und damit die grüne LED der Fahrzeugampeln West und Ost ein.

Um die Darstellung auf dem Bildschirm dem Zustand auf dem Schulungsboard anzugleichen, werden die entsprechenden Shapes (Kreise) auf die Farbe G*REEN_ON* gesetzt.

Der *on*-Button wird gesperrt und der *off*-Button freigeschaltet.

```
Private Sub cb_On_Click()
  Winsock1.SendData ("GET /outputaccess0?PW=&State=ON&")
  sh_CarEast_Green.FillColor = GREEN_ON
  sh_CarWest_Green.FillColor = GREEN_ON
  cb_On.Enabled = False
  cb_Off.Enabled = True
End Sub
```

Die Prozedur *cb_Off-Click* arbeitet analog zur cb_On Prozedur, nur dass alle Zustände genau umgekehrt gehandhabt werden.

```
Private Sub cb_Off_Click()
  Winsock1.SendData ("GET /outputaccess0?PW=&State=OFF&")
  sh_CarEast_Green.FillColor = GREEN_OFF
  sh_CarWest_Green.FillColor = GREEN_OFF
  cb_Off.Enabled = False
  cb_On.Enabled = True
End Sub
```

Zu guter Letzt, wird die Prozedur *cb_DisconnectClick* aufgerufen, wenn der Anwender auf den *disconnect*-Button klickt.

Die *Close*-Eigenschaft des *Winsock*-Steuerlementes wird aufgerufen. Dadurch angestoßen, beendet das *Winsock*-Steuerelemet die TCP-Verbindung zum Web-IO.

Die Buttons disconnect, on und off werden deaktiviert und der connect-Button wird wieder freigeschaltet. In der Statuszeile wird „disconnected“ angezeigt.

```
Private Sub cb_Disconnect_Click()
  Winsock1.Close
  cb_Connect.Enabled = True
  cb_Disconnect.Enabled = False
  StatusBar1.SimpleText = "disconnected"
  cb_On.Enabled = False
  cb_Off.Enabled = False
End Sub
```

Mit ein paar Zeilen Quellcode haben wir in diesem Beispiel gezeigt, dass die Ansteuerung des Web-IO über das Netzwerk via TCP/IP unter Visual Basic eigentlich ganz einfach ist.

In den folgenden Kapiteln wird es Ihre Aufgabe sein, Schritt für Schritt immer komplexere Applikationen zu entwickeln.

Natürlich finden Sie am Ende jedes Kapitels als Lösung einen passenden Quelltext, der auch auf der CD zur Verfügung steht.

Nehmen Sie sich jedoch nicht die Chance, selbst zu einem Ergebnis zu kommen, indem Sie sich sofort die Lösung anschauen.

3.2 Geregelter Programmabbruch

Die Ausgangssituation
Unser erstes Programmbeispiel arbeitet im Rahmen der Aufgabenstellung gut und zuverlässig. Wird das Programm allerdings geschlossen, solange die grünen LEDs eingeschaltet sind, bleiben diese auch nach Programmende an.

3.2.1 Die 2. Aufgabe

Sowohl beim Schließen der TCP-Verbindung, als auch beim Beenden des Programms soll gewährleistet sein, dass die LEDs auf jeden Fall ausgeschaltet werden.

3.2.2 Noch ein Tipp

Für das Programmformular gib es das *Unload*-Ereignis. Dieses Ereignis tritt immer ein, bevor ein Programm endgültig beendet wird und startet, wenn vorhanden, die Prozedur *Form_Unload*.

Das Beenden wird solange unterbrochen, bis die Prozedur *Form_Unload* abgearbeitet ist.

Eine Prozedur zum Ereignis wird übrigens im Code-Fenster angelegt, wenn man das betroffene Steuerelement auswählt und danach auf das Ereignis im rechten Listfeld anklickt.

3.2.3 Erster Schritt zur Lösung

Eine mögliche Lösung könnte so aussehen:

Die Prozedur *bt_disconnect* wird erweitert und die Prozedur *FormCloseQuery* wird neu angelegt.

In der *cb_Disconnect_Click* Prozedur wird mit Hilfe der *SendData*-Methode das Kommando zum Zurücksetzen des Outputs an das Web-IO gesendet, bevor die *Close*-Methode aufgerufen wird. Ferner werden die Abbilder des Ampellichts auf *GREEN_OFF* gesetzt.

```
Private Sub cb_Disconnect_Click()                              'Trennen der Verbindung
  Winsock1.SendData ("GET /outputaccess0?PW=&State=OFF&")
  sh_CarEast_Green.FillColor = GREEN_OFF
  sh_CarWest_Green.FillColor = GREEN_OFF
  Winsock1.Close
  cb_Connect.Enabled = True
  cb_Disconnect.Enabled = False
  StatusBar1.SimpleText = "disconnected"
  cb_On.Enabled = False
  cb_Off.Enabled = False
End Sub
```

Die *Form_Unload*-Prozedur wird aufgerufen, sobald der Anwender versucht das Programm zu beenden.

Aus der Form_Unload-Prozedur wird die *cb_Disconnect_Click* Prozedur aufgerufen, wenn eine TCP-Verbindung besteht. Erst danach das Programm beendet.

```
Private Sub Form_Unload(Cancel As Integer)
  If Winsock1.State = sckConnected Then cb_Disconnect_Click
End Sub
```

In den meisten Fällen wird das erweiterte Programm ein geregeltes Programmende durchführen.

Es kann aber sein, dass die grünen LEDs auf der Platine anbleiben, obwohl der entsprechende Kommandostring dem Winsock-Steuerelement zum Senden übergeben wurde.

Stolpersteine

Wenn die grünen LEDs auf dem Schulungsboard bei Programmende ordnungsgemäß ausgehen, versuchen Sie nun genau Folgendes:

- Starten Sie das Programm
- Stellen Sie die Verbindung zum Web-IO her
- Setzen Sie das Ampellicht auf ON
- Geben sie in der Windows Umgebung unter Start >> Ausführen arp -d * ein
- Klicken Sie auf den Disconnect-Button des Programms

und das Ampellicht im Programmfenster ist aus, aber die LED-auf dem Schulungsboard bleibt an!

Was ist passiert?

Das Kommando arp -d * löscht alle Einträge in der ARP-Tabelle des PC. In der ARP-Tabelle ist die Zuordnung von IP-Adresse und Ethernet-Adresse der Kommunikationspartner hinterlegt. Ist der gewünschte Partner nicht eingetragen, muss der PC die zugehörige Ethernetadresse neu ermitteln.

Dieser Vorgang nimmt einige ms Zeit in Anspruch.

Nun zurück zu unserem eigentlichen Problem. Dadurch, dass das Absenden des Kommandostrings zum Web-IO und das Schließen der Verbindung in direkter Folge ausgeführt werden, schafft es der PC nicht die Ethernet-Adresse zu ermitteln und den Kommandostring zu versenden bevor die Verbindung gekappt wird.

Das Programm sollte also so geändert werden, dass der PC genug Zeit hat die Daten zu senden.

3.2.4 Eine mögliche Lösung

In die *cb_Disconnect_Click*-Prozedur wird zwischen das Senden des Kommandostrings und dem Schließen der Verbindung der Prozeduraufruf *DoEvents* eingefügt. Damit veran-

lasst das Programm das Betriebssystem, zunächst anstehende Aufgaben abzuarbeiten, bevor das Programm fortgesetzt wird.

Dadurch wartet das Programm mit dem Schließen der TCP-Verbindung bis der Kommandostring wirklich gesendet wurde.

```
Private Sub cb_Disconnect_Click()                         'Trennen der Verbindung
  Winsock1.SendData ("GET /outputaccess0?PW=&State=OFF&")
  sh_CarEast_Green.FillColor = GREEN_OFF
  sh_CarWest_Green.FillColor = GREEN_OFF
  DoEvents
  Winsock1.Close
  cb_Connect.Enabled = True
  cb_Disconnect.Enabled = False
  StatusBar1.SimpleText = "disconnected"
  cb_On.Enabled = False
  cb_Off.Enabled = False
End Sub
```

3.3 Zeitgesteuerte Ausgaben

Die Ausgangssituation

Unsere Beispielprogramme haben bis jetzt, angestoßen durch Userbedienung, einen Output des Web-IO auf ON oder auf OFF gesetzt .

3.3.1 Die Aufgabe

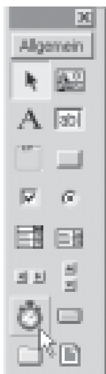
Wir nehmen an, die Straße von West nach Ost ist die Hauptstraße. Die gelben Ampellichter der Nebenstraße (von Nord nach Süd) sollen blinken. Das Blinkintervall soll über ein Eingabefeld frei einstellbar sein.

Beim Trennen der TCP-Verbindung oder beim Beenden des Programms sollen die gelben Ampellichter zunächst ausgeschaltet werden.

3.3.2 Noch ein Tipp

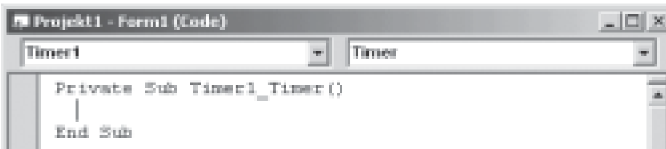
Für sich zyklisch wiederholende Programmabläufe sollte das Timer-Steuerelement eingesetzt werden.

Das Timer-Steuerelement finden Sie in der Komponentenleiste.



Die Eigenschaft *Enabled* legt fest, ob der Timer aktiv ist oder nicht. Unter *Interval* wird die Zeit zwischen zwei Timer-Aufrufen in ms festgelegt.

Die Prozedur zum Timer wird im Code-Fenster angelegt, in dem man im Linken Listfeld *Timer1* auswählt und danach im rechten Listfeld *Timer* anklickt. Alternativ reicht ein Doppelklick auf das *Timer1*-Icon im Form-Fenster.



3.3.3 Die Lösung

Programmplanung

Zunächst sollte festgelegt werden welche Aufgaben das Programm in welcher Abfolge erledigen muss:

1. Entgegennehmen der Verbindungsparameter (IP-Adresse und Remote-Port des Web-IO
2. Aufbau der TCP-Verbindung
3. Starten des Timers, der im Wechsel das gelbe Ampellicht auf dem Schulungsboard ein- und ausschaltet. Buttons bedienbar machen bsw. Bedienung sperren
4. Bei Schließen der TCP-Verbindungs zunächst die Ampellichter ausschalten, dann erst Verbindung schließen und Buttons aktivieren bzw. sperren
5. Bei Beenden des Programms zunächst die Ampellichter ausschalten

Im Programmfenster soll wie gewohnt ein komplettes Prozessabbild wiedergegeben werden.

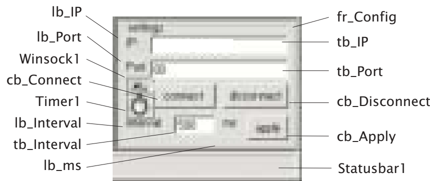
Vorarbeiten

Auch für diese Anwendung kann das vorgefertigte Programmgerüst aus dem Verzeichnis *Basiclayout* der CD als Grundlage genutzt werden. Dazu wird das komplette Projekt

in ein neues Verzeichnis kopiert. Das Projekt muss dann aus dem neuen Verzeichnis geöffnet werden.

Um Zeit und Arbeit zu sparen kann man die komplette Groupbox inklusive Bedienelementen aus dem zweiten Programmbeispiel in das neue Projekt kopieren. Einfach die Groupbox *gb_config* durch Mouse-Klick markieren und mit <Strg + c> in die Zwischenablage kopieren und mit <Strg + v> in das neue Projekt einfügen. Die Steuerelemente, die im neuen Projekt nicht gebraucht werden einfach löschen und neue Objekte nach Bedarf hinzufügen.

Die neue Groupbox könnte dann so aussehen:



Für den Timer1 muss die *Enabled*-Eigenschaft auf *False* gesetzt werden.

Der Quellcode

Auch in diesem Beispiel wurden für die Ampelfarben entsprechende Konstanten definiert.

```
Private Const GREEN_OFF = &H8000&              'dunkelgrün
Private Const GREEN_ON = &HFF00&               'hellgrün
Private Const RED_OFF = &H80&                  'dunkelrot
Private Const RED_ON = &HFF&                   'hellrot
Private Const YELLOW_OFF = &H8080&             'dunkelgelb
Private Const YELLOW_ON = &HFFFF&              'hellgelb
```

Selbst definierte Variablen werden auch in diesem Beispiel noch nicht verwendet.

Nun beginnt der eigentliche Programmcode:

Die Prozedur *cb_Connect_Click* wurde unverändert aus Beispiel 2 übernommen.

```
Private Sub cb_Connect_Click()
  If (tb_Port.Text <> "") And (tb_IP.Text <> "") Then
    Winsock1.RemoteHost = tb_IP.Text
    Winsock1.RemotePort = tb_Port.Text
    Winsock1.Connect
  End If
End Sub
```

Bei erfolgreichem Verbindungsaufbau wird die Prozedur *Winsock1_Connect* ausgeführt und dann der *Connect*-Button für die Bedienung gesperrt, während der *Disconnect*-Button freigegeben wird. In der Statuszeile wird „disconnect“ angezeigt

Ferner wird das Interval des Timers auf den Wert gesetzt, der in *tb_interval* eingetragen wurde. Anschließend wird der Timer aktiviert und beginnt das Ein- und Ausschalten der Ampellichter zu steuern.

```
Private Sub Winsock1_Connect()
  cb_Connect.Enabled = False
  cb_Disconnect.Enabled = True
  StatusBar1.SimpleText = "connected"
  Timer1.Interval = Val(tb_Interval.Text)
  Timer1.Enabled = True
End Sub
```

Die Prozedur, die nach Ablauf des Timerintervalls aufgerufen wird, besteht aus zwei Programmteilen, die über eine If-Abfrage abhängig vom Zustand des Ampellichtes ausgeführt werden.

Ist das Ampellicht ausgeschaltet, wird das Kommando zum Einschalten an das Web-IO gesendet und die Farbe der entsprechenden Shapes geändert.

Ist das Licht eingeschaltet, werden die umgekehrten Zustände geschaltet.

```
Private Sub Timer1_Timer()
  If sh_CarEast_Yellow.FillColor = YELLOW_OFF Then
    sh_CarEast_Yellow.FillColor = YELLOW_ON
    sh_CarWest_Yellow.FillColor = YELLOW_ON
    Winsock1.SendData ("GET /outputaccess1?PW=&State=ON&")
  Else
    sh_CarEast_Yellow.FillColor = YELLOW_OFF
    sh_CarWest_Yellow.FillColor = YELLOW_OFF
    Winsock1.SendData ("GET /outputaccess1?PW=&State=OFF&")
  End If
End Sub
```

Bei Klick auf den *apply*-Button wird die eingetragene Intervallzeit als Intervall des Timers gesetzt.

```
Private Sub cb_Apply_Click()
  Timer1.Enabled = False
  Timer1.Interval = Val(tb_Interval.Text)
  Timer1.Enabled = True
End Sub
```

Die Prozedur für das Disconnect ist weitestgehend mit der aus Beispiel 2 identisch und wurde nur um das Deaktivieren des Timers erweitert.

```
Private Sub cb_Disconnect_Click()
  Timer1.Enabled = False
  Winsock1.SendData ("GET /outputaccess1?PW=&State=OFF&")
  sh_CarEast_Yellow.FillColor = YELLOW_OFF
  sh_CarWest_Yellow.FillColor = YELLOW_OFF
  DoEvents
  Winsock1.Close
  cb_Connect.Enabled = True
  cb_Disconnect.Enabled = False
  StatusBar1.SimpleText = "disconnected"
End Sub
```

Die *Form_Unload*-Prozedur wird aufgerufen, sobald der Anwender versucht das Programm zu beenden.

Wenn eine TCP-Verbindung besteht wird aus der *Form_Unload*-Prozedur die *cb_Disconnect_Click* Prozedur aufgerufen. Erst erst wenn *cb_Disconnect_Click* ausgeführt wurde, wird das Programm beendet.

```
Private Sub Form_Unload(Cancel As Integer)
  If Winsock1.State = sckConnected Then cb_Disconnect_Click
End Sub
```

3.4 Daten vom Netzwerk empfangen

Die Ausgangssituation
Alle Beispielprogramme haben bis jetzt ausschließlich Daten über das Netzwerk zum Web-IO gesendet. Im folgenden Beispiel wollen wir uns mit dem Datenempfang beschäftigen.

Beim Web-IO können die Inputs mit dem Kommandostring GET /inputx?PW=&' abgefragt werden. „x" steht dabei für den angefragten Input. Als Antwort sendet das Web-IO einen String wie diesen zurück: 'inputx;ON'. In diesem Fall war Input x gleich ON.

Die Antwortstrings des Web-IO enden immer mit einem Nullbyte (*chr(0)*). Sollen die Strings z.B. über eine IF-Abfrage ausgewertet werden, muss dieses Nullbyte mit in den Vergleich einbezogen werden.

Beispiel:
if Antwort = 'input0;ON + chr(0) then ....

Eine komplette Liste der möglichen Kommandostrings finden Sie im Anhang dieser Anleitung oder im Referenzhandbuch zum Web-IO 12xDigital.

3.4.1 Die Aufgabe

Es soll überwacht werden, ob einer der Anforderungsschalter des westlichen Fußgängerüberwegs gedrückt wird. Sobald dies der Fall ist, soll der entsprechende schwarze Shape weiß angezeigt werden.

3.4.2 Noch ein paar Tipps

Das *Winsock*-Steuerelement löst das *DataArrival* Ereignis aus, wenn Daten vom Netzwerk empfangen werden. Die Prozedur zum *DataArrival* Ereignis wird im Code-Fenster angelegt, in dem man im Linken Listfeld *Winsock1* auswählt und danach im rechten Listfeld *DataArrival* anklickt.

In dieser Prozedur können die empfangenen Daten dann weiter verarbeitet werden. Um z.B. einen String vom Netzwerk entgegenzunehmen kann die Methode *GetData* des *Winsock*-Steuerelements benutzt werden.

Beispiel:
Winsock1.GetData TestString

Wird über den Kommandostringmodus gearbeitet, gibt das Web-IO den Zustand seiner Inputs nur auf Anfrage zurück. Soll die Anwendung aktuell über den Input-Zustand informiert werden, muss eine zyklische Abfrage erfolgen. Man spricht in diesem Fall von „Polling“.

Für unsere konkrete Aufgabe bedeutet das, dass ein Timer eingerichtet werden muss, der das Polling ausführt. Dabei sollte das Timerintervall so bemessen sein, dass man keine Änderung verpasst.

Beispiel:
Werden die Inputs nur alle 5 Sekunden „abgepollt“, und der Ampeldrücker wird zwischen zwei Inputabfragen gedrückt, so wird diese Input-Änderung vom Programm nicht wahrgenommen.

Für das Umfärben des Anforderungsschalters im Programmfenster wurden in der Quelltext Grundlage zwei Farbkonstanten definiert:

- BLACK_OFF = &H0&   ‘schwarz
- BLACK_ON = &HFFFFFF   ‘weis

Die Verwendung dieser Konstanten macht den Quelltext übersichtlicher.

3.4.3 Die Lösung

Programmplanung

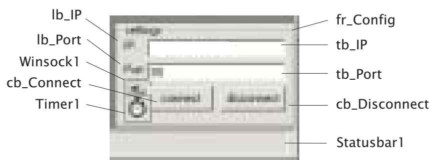
Zunächst sollte wieder festgelegt werden, welche Aufgaben das Programm in welcher Abfolge erledigen muss:

1. Entgegen nehmen der Verbindungsparameter (IP-Adresse und Remote-Port des Web-IO)
2. Aufbau der TCP-Verbindung
3. Starten des Timers, der die Inputabfrage an das Web-IO sendet.
4. Entgegennehmen und Auswerten der Datensendungen des Web-IO und entsprechende Änderung des Prozessabbildes im Programmfenster
5. Beenden des Programms

Vorarbeiten

Die Quelltextgrundlage aus dem Basiclayout-Verzeichnis kann auch für dieses Beispiel in einen neuen Ordner kopiert werden.

Auch die Groupbox *gb_config* aus dem ersten Beispiel kann wieder als Basis für die Bedienelemente des Programms genutzt werden. Die Anpassung an die Bedürfnisse dieser Anwendung sieht dann so aus.



Für den Timer1 muss die *Enabled*-Eigenschaft auf *False* gesetzt werden. Als Intervall wird 100ms eingestellt.

Der Quelltext

Wie bereits angesprochen, werden zwei zusätzliche Farbkonstanten definiert, welche die Farbe für die Anzeige der Anforderungstaster bestimmen.

```
Private Const BLACK_OFF = &H0&
```

```
Private Const BLACK_ON = &HFFFFFF
```

Die Prozedur *cb_Connect_Click* bleibt auch hier unverändert.

```
Private Sub cb_Connect_Click()
  If (tb_Port.Text <> "") And (tb_IP.Text <> "") Then
    Winsock1.RemoteHost = tb_IP.Text
    Winsock1.RemotePort = tb_Port.Text
    Winsock1.Connect
  End If
End Sub
```

Bei erfolgreichem Verbindungsaufbau wird die Prozedur *Winsock1_Connect* ausgeführt.

Die Buttons werden gesperrt bzw. freigegeben und in der Statuszeile wird „connect“ ausgegeben.

Timer1 wird durch Setzen von *Enable = True* gestartet.

```
Private Sub Winsock1_Connect()
  cb_Connect.Enabled = False
  cb_Disconnect.Enabled = True
  StatusBar1.SimpleText = "connected"
  Timer1.Enabled = True
End Sub
```

Mit jedem Timer-Aufruf wird der Kommandostring für die Statusabfrage von Input 2 an das Web-IO gesendet.

```
Private Sub Timer1_Timer()
  Winsock1.SendData ("GET /input2?PW=&")
End Sub
```

Winsock1_DataArrival wird immer dann aufgerufen, wenn Daten vom Netzwerk empfangen werden. Je nach empfangenem String werden die Shapes für die Ampeldrücker im Programmfenster schwarz oder weiß eingefärbt

```
Private Sub Winsock1_DataArrival(ByVal bytesTotal As Long)
  Dim ReceiveString As String
```

```
  Winsock1.GetData ReceiveString
  If ReceiveString = ("input2;ON" + Chr(0)) Then
    sh_RequestWestNorth.FillColor = BLACK_ON
    sh_RequestWestSouth.FillColor = BLACK_ON
  Else
    sh_RequestWestNorth.FillColor = BLACK_OFF
    sh_RequestWestSouth.FillColor = BLACK_OFF
  End If
End Sub
```

Beim Beenden der TCP-Verbindung wird der Timer deaktiviert, Die Buttons werden gesperrt bzw. freigegeben und in der Statuszeile wird „disconnected“ angezeigt.

```
Private Sub cb_Disconnect_Click()
  Timer1.Enabled = False
  Winsock1.Close
  cb_Connect.Enabled = True
  cb_Disconnect.Enabled = False
  StatusBar1.SimpleText = "disconnected"
End Sub
```

Noch ein Versuch

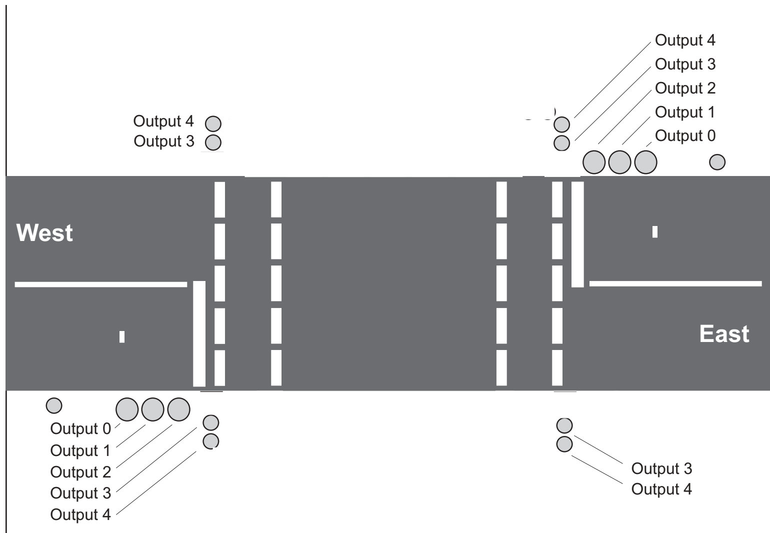
Verändern Sie einmal das Timer-Intervall und prüfen Sie, wie sich das auf die Erkennung der gedrückten Taste auswirkt!

3.5 Eine komplett zeitgesteuerte Anwendung

Nachdem die vorangegangenen Beispiele sich immer nur auf Einzelfunktionen bezogen haben, wollen wir nun eine komplette sinnvolle Anwendung programmieren

3.5.1 Die Aufgabe

Für die Straße von Westen nach Osten soll eine zeitgesteuerte Fußgängerampel eingerichtet werden. Dabei sollen sowohl beim westlichen als auch beim östlichen Fußgängerüberweg die Ampeln funktionieren. Die Grünphase für die Autos soll genau wie die Grünphase für die Fußgänger auf 20 Sekunden voreingestellt sein. Über ein Eingabefeld soll diese Zeit aber frei einstellbar sein. Die Anforderungsschalter für die Fußgängerampeln sollen in diesem Beispiel noch ohne Funktion bleiben.



Der Plan zeigt die Elemente des Schulungsboards, die für das Beispiel benötigt werden.

3.5.2 Noch ein paar Tipps

Bis jetzt wurden Kommandostrings benutzt, mit denen man einzelne Outputs setzen bzw. Inputs lesen konnte. Um mehrere Outputs zeitgleich zu schalten, stellt das Web-IO ein Kommando zur Verfügung, das über eine Hexadezimalzahl festlegt, welche Outputs gesetzt werden sollen.

GET /outputaccess?PW=&State=0001& setzt zum Beispiel Output 0 auf ON und alle anderen Outputs auf OFF.

Letztlich stehen die Bits 0-11 der hexadezimalen Zahl für die Outputs 0-11. Um zu bestimmen welche Outputs gesetzt-werden sollen, muss also bei einer 12stelligen Dualzahl (Binärzahl) das jeweilige Bit auf 1 für einen eingeschalteten Output und auf 0 für einen ausgeschalteten Output gesetzt werden.

Diese Dualzahl muss dann in eine hexadezimale Zahl umgerechnet werden, um sie in den Kommandostring einzusetzen. Da die Syntax der Kommandostrings 4stellige hexadezimale Zahlen vorschreibt, eine 12stellige Dualzahl aber nur 3 Stellen hexadezimal ergibt, wird dem errechneten Wert einfach eine 0 vorangestellt.

Eine kleine Auffrischung der Zahlensysteme
In der Computertechnik arbeitet man mit Bits und Bytes, also eigentlich im dualen- oder auch binären Zahlensystem.

Duale Zahlen sind für den Menschen leider sehr unübersichtlich. Wer erkennt schon auf Anhieb, dass dual 110001110101 = dezimal 3189 ergibt.

Da man jeden Input bzw. Output des Web-IO als eine Stelle einer 12-stelligen Binärzahl sehen muss, ist es unausweichlich, sich noch einmal mit dieser Materie zu beschäftigen.

```
                          Dualzahl  110001110101

Bit  0  =  2^0  =     1             1  x    1  =    1
Bit  1  =  2^1  =     2             0  x    2  =    0
Bit  2  =  2^2  =     4             1  x    4  =    4
Bit  3  =  2^3  =     8             0  x    8  =    0
Bit  4  =  2^4  =    16             1  x   16  =   16
Bit  5  =  2^5  =    32             1  x   32  =   32
Bit  6  =  2^6  =    64             1  x   64  =   64
Bit  7  =  2^7  =   128             0  x  128  =    0
Bit  8  =  2^8  =   256             0  x  256  =    0
Bit  9  =  2^9  =   512             0  x  512  =    0
Bit 10  =  2^10 =  1024             1  x 1024  = 1024
Bit 11  =  2^11 =  2048             1  x 2048  = 2048
                                        Dezimalzahl 3819
```

Das Umrechnen von Dual- in Dezimal-Zahlen ist nicht schwierig. Dennoch fehlt die spontane Zuordnung zwischen gesetzten Outputs und dem Dezimalwert. Deshalb werden dort, wo der Mensch mit Bits und Bytes jonglieren muss, hexadezimale Zahlen verwendet.

Bei hexadezimalen Zahlen kann die Wertigkeit jeder Stelle durch 15 verschiedene Ziffern dargestellt werden. Da unser dezimales Zahlensystem nur Ziffern von 0 ... 9 kennt, wurde das heaxdezimale System um die Buchstaben A ... F erweitert.

A =10, B=11, C=12, D=13, E=14, F=15.

Hier noch mal etwas übersichtlicher.

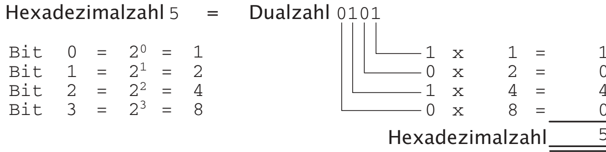
```
                          Hexadezimalzahl  C75

Stelle  0 = 16^0 =   1              5   x    1  =    5
Stelle  1 = 16^1 =  16              7   x   16  =  112
Stelle  2 = 16^2 = 256              12  x  256  = 3072
                                       Dezimalzahl 3819
```

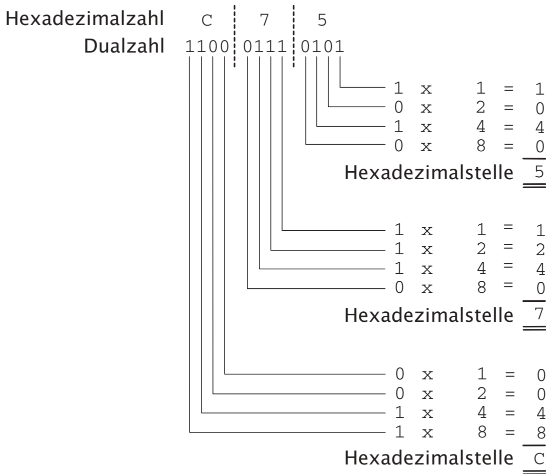
Auf den ersten Blick macht die Benutzung des hexadezimalen Zahlensystems die Darstellung der Inputs und Outputs nicht einfacher.

Aber schauen wir einmal genau hin. Jede Stelle der hexadezimalen Zahl ist eine Potenz zur Basis 16 multipliziert mit der Ziffer. 16 wiederum ist die 4. Potenz von 2, also 2^4.

Jede Stelle des hexadezimalen Zahlensystems lässt sich deshalb durch Addition der 2er Potenzen 2^0 bis 2^3 errechnen.



Zerlegt man nun eine Dualzahl mit der niedrigsten Stelle beginnend in Vierbit-Bereiche, kann man mit wenig Aufwand zwischen Dualzahlen und hexadezimalen Zahlen umrechnen.



Mit etwas Übung lässt sich das bequem im Kopf rechnen.

3.5.3 Die Lösung

Programmplanung

Zunächst sollte wieder festgelegt werden, welche Aufgaben das Programm in welcher Abfolge erledigen muss:

1. Entgegennehmen der Verbindungsparameter (IP-Adresse und Remote-Port des Web-IO
2. Aufbau der TCP-Verbindung
3. Starten des Timers, der die einzelnen Schritte der Ampelsteuerung abwickelt

   Schritt 1 Fahrzeugampeln: gelb

   Fußgängerampeln: rot

Schritt 2 Fahrzeugampeln: rot
Fußgängerampeln: rot
Schritt 3 Fahrzeugampeln: rot
Fußgängerampeln: grün
Schritt 4 Fahrzeugampeln: rot
Fußgängerampeln: rot
Schritt 5 Fahrzeugampeln: rot und gelb
Fußgängerampeln: rot
Schritt 6 Fahrzeugampeln: grün
Fußgängerampeln: rot

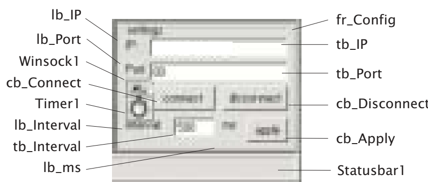
4. Bei Schließen der TCP-Verbindungs zunächst die Ampelphasen so zu Ende führen, so das die Grünphase für die Fußgänger abgeschlossen und die Grünphase für die Autos erreicht wird. Danach alle Ampellichter ausschalten und dann erst die Verbindung schließen und die Buttons aktivieren bzw. sperren
5. Bei Beenden des Programms zunächst die Ampelphasen so zu Ende führen, so das die Grünphase für die Fußgänger abgeschlossen und die Grünphase für die Autos erreicht wird. Danach alle Ampellichter ausschalten und dann erst das Programm beenden

Im Programmfenster soll wie gewohnt ein komplettes Prozessabbild wiedergegeben werden.

Vorarbeiten

Als Grundgerüst für diese Anwendung kann das komplette Projekt aus Programmbeispiel 3 benutzt werden. Dazu wird das komplette Projekt in ein neues Verzeichnis kopiert. Das Projekt muss dann aus dem neuen Verzeichnis geöffnet werden.

Die Bedienelemente können so wie sie sind genutzt werden.



Für den Timer1 muss die *Enabled*-Eigenschaft auf *False* gesetzt sein.

Der Quellcode

Der Quellcode von Beispiel 3 muss nur an einigen Stellen modifiziert bzw. erweitert werden.

Die Definition der Farbkonstanten wird in diesem Beispiel das erstemal durchgehend benutzt.

```
Private Const GREEN_OFF = &H8000&
Private Const GREEN_ON = &HFF00&
Private Const RED_OFF = &H80&
Private Const RED_ON = &HFF&
Private Const YELLOW_OFF = &H8080&
Private Const YELLOW_ON = &HFFFF&
Private Const BLACK_OFF = &H0&
Private Const BLACK_ON = &HFFFFFF
```

Es wurden vier zusätzliche Variablen deklariert:

vin_applicationstep gibt an in welcher Ampelphase sich das Programm gerade befindet. In *vin_interval* wird das Timerinterval zwischengespeichert. *vbo_disconnectquiry* und *vbo_closequiry* sind sogenannte Flags (Merker), die gesetzt werden wenn der Benutzer die TCP-Verbindung bzw. das komplette Programm zu schließen versucht.

```
Dim vin_applicationstep As Integer
Dim vin_interval As Integer
Dim vbo_disconnectquiry As Boolean
Dim vbo_closequiry As Boolean
```

Hier beginnt das eigentliche Programm.

Bei Programmstart wird zunächst die Variable *vin_applicationstep* auf 1 gesetzt.

```
Private Sub Form_Load()
  vin_applicationstep = 1
End Sub
```

Die Prozedur *cb_Connect_Click* wurde unverändert übernommen.

```
Private Sub cb_Connect_Click()
  If (tb_Port.Text <> "") And (tb_IP.Text <> "") Then
    Winsock1.RemoteHost = tb_IP.Text
    Winsock1.RemotePort = tb_Port.Text
    Winsock1.Connect
  End If
End Sub
```

In der Prozedur *Winsock1_Connect* wird zusätzlich das Timerintervall auf ein Zehntel des eingestellten Wertes gesetzt. Ein Zehntel, weil die Zwischenphasen einer Ampel (grün nach gelb, gelb nach rot,....) deutlich kürzer sind als die normalen Grünphasen. Ferner wird der Timer aktiviert.

```
Private Sub Winsock1_Connect()
  cb_Connect.Enabled = False
  cb_Disconnect.Enabled = True
  StatusBar1.SimpleText = "connected"
  vin_interval = Val(tb_Interval.Text)
  Timer1.Interval = vin_interval / 10
  Timer1.Enabled = True
End Sub
```

Der Timer steuert abhängig von der Variablen *vin_applicationstep* je nach Ampelphase die LEDs auf dem Schulungsboard und die Programmoberfläche.

Zusätzlich wird über die Variablen *vbo_disconnectquiry* bzw. *vbo_closequiry* überwacht, ob der Anwender die TCP-Verbindung bzw. die Applikation schließen möchte. In diesem Fall werden die Ampelphasen weiter fortgeführt, bis die Grünphase für die Autos erreicht ist. Erst dann werden alle Ampellichter abgeschaltet und die Verbindung beendet.

```
Private Sub Timer1_Timer()
  Select Case vin_applicationstep
```

Phase1: Gelbphase für Autos

Soll weder die TCP-Verbindung noch das Programm beendet werden, werden alle Schritte ausgeführt um die Gelbphase für die Autos zu beginnen. Ist *vbo_diconnectquiry* aber gesetzt, werden alle LEDs ausgeschaltet und die TCP-Verbindung beendet. Ist *vbo_closequery = True* wird das Programm geschlossen.

```
         Case 1
        If vbo_disconnectquiry Then
          Timer1.Enabled = False
          sh_CarWest_Green.FillColor = GREEN_OFF
          sh_CarEast_Green.FillColor = GREEN_OFF
          sh_WalkerWestNorth_Red.FillColor = RED_OFF
          sh_WalkerWestSouth_Red.FillColor = RED_OFF
          sh_WalkerEastNorth_Red.FillColor = RED_OFF
          sh_WalkerEastSouth_Red.FillColor = RED_OFF
          Winsock1.SendData ("GET /outputaccess?PW=&State=0000&")
          DoEvents
          Winsock1.Close
          StatusBar1.SimpleText = "disconnected"
          cb_Connect.Enabled = True
          If vbo_closequiry Then End
          vbo_disconnectquiry = False
        Else
          sh_CarWest_Green.FillColor = GREEN_OFF
          sh_CarWest_Yellow.FillColor = YELLOW_ON
          sh_CarEast_Green.FillColor = GREEN_OFF
          sh_CarEast_Yellow.FillColor = YELLOW_ON
          sh_WalkerWestNorth_Green.FillColor = GREEN_OFF
          sh_WalkerWestNorth_Red.FillColor = RED_ON
          sh_WalkerWestSouth_Green.FillColor = GREEN_OFF
          sh_WalkerWestSouth_Red.FillColor = RED_ON
          sh_WalkerEastNorth_Green.FillColor = GREEN_OFF
          sh_WalkerEastNorth_Red.FillColor = RED_ON
          sh_WalkerEastSouth_Green.FillColor = GREEN_OFF
          sh_WalkerEastSouth_Red.FillColor = RED_ON
          Winsock1.SendData ("GET /outputaccess?PW=&State=0012&")
          Timer1.Interval = vin_interval / 10
          vin_applicationstep = vin_applicationstep + 1
      End If
```

Phase 2: Rotphase für Autos und Fußgänger

```
    Case 2
      sh_CarWest_Yellow.FillColor = YELLOW_OFF
      sh_CarWest_Red.FillColor = RED_ON
      sh_CarEast_Yellow.FillColor = YELLOW_OFF
      sh_CarEast_Red.FillColor = RED_ON
      Winsock1.SendData ("GET /outputaccess?PW=&State=0014&")
     vin_applicationstep = vin_applicationstep + 1
```

Phase 3: Grünphase für Fußgänger

Das Timerintervall wird hochgesetzt, damit die Grünphase ausreichend lange dauert.

```
    Case 3
      sh_WalkerWestNorth_Green.FillColor = GREEN_ON
      sh_WalkerWestNorth_Red.FillColor = RED_OFF
      sh_WalkerWestSouth_Green.FillColor = GREEN_ON
      sh_WalkerWestSouth_Red.FillColor = RED_OFF
      sh_WalkerEastNorth_Green.FillColor = GREEN_ON
      sh_WalkerEastNorth_Red.FillColor = RED_OFF
      sh_WalkerEastSouth_Green.FillColor = GREEN_ON
      sh_WalkerEastSouth_Red.FillColor = RED_OFF
      Winsock1.SendData ("GET /outputaccess?PW=&State=000C&")
      Timer1.Interval = vin_interval
     vin_applicationstep = vin_applicationstep + 1
```

Phase 4: Rot für alle

Das Timerintervall wird wieder auf ein Zehntel zurückgesetzt, damit die Zwischenphasen entsprechend kurz sind.

```
    Case 4
      sh_WalkerWestNorth_Green.FillColor = GREEN_OFF
      sh_WalkerWestNorth_Red.FillColor = RED_ON
      sh_WalkerWestSouth_Green.FillColor = GREEN_OFF
      sh_WalkerWestSouth_Red.FillColor = RED_ON
      sh_WalkerEastNorth_Green.FillColor = GREEN_OFF
      sh_WalkerEastNorth_Red.FillColor = RED_ON
      sh_WalkerEastSouth_Green.FillColor = GREEN_OFF
      sh_WalkerEastSouth_Red.FillColor = RED_ON
      Winsock1.SendData ("GET /outputaccess?PW=&State=0014&")
```

```
      Timer1.Interval = vin_interval / 10
     vin_applicationstep = vin_applicationstep + 1
```

Phase 5: Rot-Gelb für die Autos

```
    Case 5
      sh_CarWest_Yellow.FillColor = YELLOW_ON
      sh_CarEast_Yellow.FillColor = YELLOW_ON
      Winsock1.SendData ("GET /outputaccess?PW=&State=0016&")
     vin_applicationstep = vin_applicationstep + 1
```

Phase 6: Grünphase für die Autos
Wenn die Variable *vbo_diconnect* gesetzt ist, bleibt das Timerintervall auf ein Zehntel, bei normaler Grünphase wird das Intervall hochgesetzt, um eine ausreichend lange Grünphase zu schalten.

```
    Case 6
      sh_CarWest_Green.FillColor = GREEN_ON
      sh_CarWest_Yellow.FillColor = YELLOW_OFF
      sh_CarWest_Red.FillColor = RED_OFF
      sh_CarEast_Green.FillColor = GREEN_ON
      sh_CarEast_Yellow.FillColor = YELLOW_OFF
      sh_CarEast_Red.FillColor = RED_OFF
      Winsock1.SendData ("GET /outputaccess?PW=&State=0011&")
      If vbo_disconnectquiry Then
        Timer1.Interval = vin_interval / 10
      Else
        Timer1.Interval = vin_interval
      End If
      vin_applicationstep = 1
    End Select
End Sub
```

Durch Klick auf den *apply*-Button wird der, in das Editfeld eingetragene Wert, in die Variable *vin_interval* übernommen.

```
Private Sub cb_Apply_Click()
  vin_interval = Val(tb_Interval.Text)
End Sub
```

Wünscht der Anwender die TCP-Verbindung zu beenden, wird die Variable *vbo_disconnectquiry* auf True gesetzt. Ferner wird in der Stauszeile „waiting for disconnect“ ausgegeben.

```
Private Sub cb_Disconnect_Click()
  vbo_disconnectquiry = True
  cb_Disconnect.Enabled = False
  StatusBar1.SimpleText = "waiting for disconnect"
End Sub
```

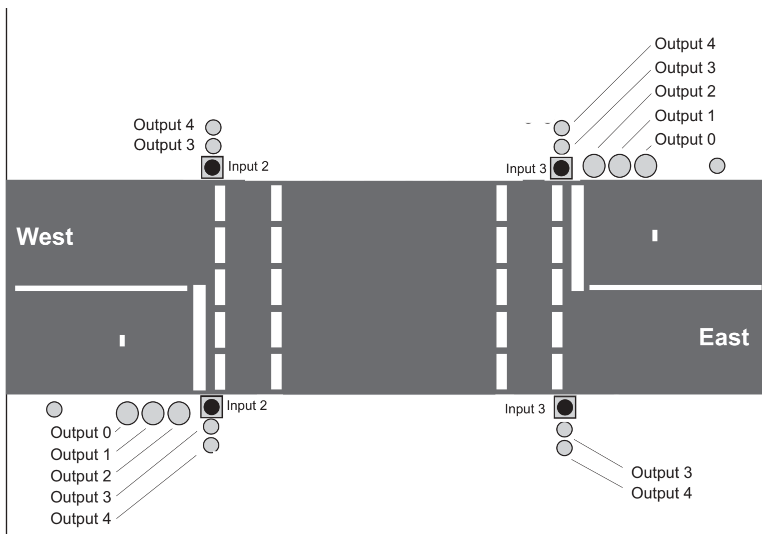
Wünscht der Anwender bei aktiver Verbindung das Programm zu beenden, werden die Variablen *vbo_disconnectquiry* und *vbo_closequiry* auf *True* gesetzt. Die Variable *Cancel* wird ebenfalls auf *True* gesetzt, wodurch das Schließen des Programmes zunächst unterbrochen wird. Ferner wird in der Stauszeile „waiting for disconnect“ ausgegeben. Erreicht der Timer beim nächsten Durchlauf Phase 1. wird das Programm beendet.

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
  If Winsock1.State = sckConnected Then
    vbo_closequiry = True
    vbo_disconnectquiry = True
    StatusBar1.SimpleText = "waiting for disconnect"
    Cancel = True
  End If
End Sub
```

3.6 Eine ereignis- und zeitgesteuerte Anwendung

3.6.1 Die Aufgabe

Das vorangegangene Beispiel soll so umprogrammiert-werden, dass die Fahrzeugampeln im Normalfall grün zeigen. Nur wenn einer der Anforderungsschalter an den Fußgängerüberwegen gedrückt wird, soll eine Grünphase für die Fußgänger eingeleitet werden. Dabei sollen natürlich auch alle nötigen Zwischenphasen durchlaufen werden.



Der Plan zeigt die Elemente des Schulungsboards, die für das Beispiel benötigt werden.

3.6.2 Noch ein paar Tipps

In Beispiel 4 haben wir bereits das *DataArrival* Ereignis des *Winsock*-Steuerlementes genutzt, um eine Prozedur zur Verarbeitung der eingehenden Daten zu starten. Dabei wurde ausschließlich der Status eines einzelnen Inputs berücksichtigt.

Soll ein komplettes Abbild der Inputs abgefragt werden, kann der Kommandostring

```
GET /input?PW=&
```

an das Web-IO gesendet werden, worauf das Web-IO mit

```
input;0001
```

gefolgt von einem Nullbyte antwortet, wenn z.B. nur Input1 gesetzt ist.

Das Web-IO gibt aber auch den Zustand der Outputs zurück, wenn ein Kommando zum Setzen erfolgreich abgearbeitet wurde.

Sendet die Anwendung z.B:

```
GET /outputaccess?PW=&State=0004&,
```

antwortet das Web-IO mit

```
output;0004
```

gefolgt von einem Nullbyte.

Je komplexer eine Anwendung wird, um so komplexer muss auch die Prozedur zur Verarbeitung eingehender Daten gestaltet werden.

Wird, wie im Fall unserer bisherigen Beispiele, mit Kommandostrings gearbeitet, müssen die eingehenden Strings auf ihren Inhalt untersucht werden. Man spricht dabei auch von Parsing oder Parsen.

Bisher haben wir die Oberfläche im Programmfenster den Anforderungen entsprechend angezeigt und dabei nicht berücksichtigt, ob der Kommadostring an das Web-IO auch dazugeführt hat, die geforderten Outputs tatsächlich zu setzen.

Im Fall einer Störung oder einer Fehlkonfiguration des Web-IO durch den Einrichter, würden die Aktionen auf dem Bildschirm angezeigt, obwohl sie in der Anwendung selbst, also auf dem Schulungsbord, gar nicht gegriffen haben.

Es wäre deshalb ratsam, die Programmoberfläche basierend auf den Antworten des Web-IO anzuzeigen. Erstrebenswert wäre es natürlich, Funktionen und Prozeduren zu entwickeln, mit denen man durch Übergabe der hexadezimalen Zahl aus dem Antwortstring ein aktuelles Abbild des Schulungsboards auf der Programmoberfläche darstellt.

Die Zustände der einzelnen Outputs entsprechen den zugehörigen Bits der Hexadezimalzahl.

Um die, als String vorliegende, hexadezimale Zahl auszuwerten, muss diese also zunächst in einen Zahlenwert konvertiert werden. Dazu könnte man sich natürlich der im vorangegangenen Beispiel, erklärten mathematischen Methode bedienen und jedes einzelne Bit ausrechnen.

Solche Operationen sind aber im Programmablauf sehr zeitaufwendig. Visual Basic bietet eine deutlich schnellere und elegantere Möglichkeit, hexadezimale Zahlenstrings in Zahlenwerte umzurechnen. Aber zunächst noch ein paar Grundlagen

Strings, Chars und Zahlenvariablen

Ein String ist eine Kette von Einzelzeichen, sogenannten Charactern oder auch Char-Variablen. Jedes Einzelzeichen ist im Speicher als ein Byte abgelegt.

Noch mal zusammengefasst:

- Char-Variablen sind darstellbare Zeichen (Buchstaben, Ziffern und Sonderzeichen).
- Ein String ist eine Kette von Char-Variablen
- Ein Byte ist ein Zahlenwert zwischen 0 und 255

	1.Char	2.Char	3.Char	4.Char
Hexadezimalzahl als String	0	C	7	5
Bytes im Speicher (dezimal)	48	67	55	53
Position im Speicher	n	n+1	n+2	n+3

Auswerten von Strings

Aus Strings können entweder einzelne Zeichen oder ein ganzer Teilbereich heraus kopiert werden. Um einen Teilbereich zu kopieren gibt es die Funktion *mid(String, Position, Zeichenanzahl)*

Beispiel:

es soll der hexadezimale Anteil von *vst_receivestring* in *vst_hexstring* kopiert werten.

```
Dim vst_receivestring as String;
Dim vst_hexstring as String;

  vst_receivestring = ‘output;0001’;
 vst_hexstring = mid(vst_hexstring, 8, 4);
```

Den dezimalen Wert eines Hexzeichens bestimmen

In den vorangegangenen Beispielen haben wir bereits die *val*-Funktion benutzt, um als String vorliegende Dezimalzahlen in einen Zahlenwert umzuwandeln.

Beispiel:

```
Winsock1.RemotePort = Val(tb_Port.Text)
```

Die in ein Eingabefeld geschriebene Portnummer wird in den Zahlenwert *RemotePort* geschrieben.

Handelt es sich bei dem als String vorliegenden Wert um eine hexadezimale Zahl, kann dem String ein "*&H*" vorangestellt werden.

Beispiel:

```
Dim vst_hexstring as String;
Dim vin_HexValue as Integer

vin_HexValue = Val("&H" + vst_HexString)
```

Einzelne Bits ausmaskieren

Zahlenwerte können automatisiert mit Hilfe logischer Operationen zur weiteren Verarbeitung ausgewertet werden.

Geht es um die Outputs, entsprechen die ersten 12 Bit den Outputs 0-11. Wird der Zustand eines einzelnes Bit benötigt, müssen zunächst alle anderen Bits ausgeblendet werden. Das kann über eine UND-Verknüpfung erreicht werden.

Beispiel:

Es soll ermittelt werden, ob Output 5, also das 5. Bit gesetzt ist oder nicht.

	Dual	Dezimal
Wert	1100 0111 0101	3819
UND	0000 0010 0000	32
=	0000 0010 0000	32

Dazu wird der vorliegende Wert mit der Zahl 32 UND-verknüpft (entspricht einer Zahl bei der nur das 5.Bit gesetzt ist). Ist der Status von Output 5 = ON, ist das Ergebnis der UND-Verknüpfung gleich 2^5=32, ist der Zustand = OFF ist das Ergebnis gleich 0.

In Visual Basic:

```
Dim vin_HexValue as Integer

If (vin_HexValue and 32) = 32 then
..         'wieteres Vorgehen bei Bit5=1
Else
..         'wieteres Vorgehen bei Bit5=0
End If
```

3.6.3 Die Lösung

Programmplanung

Zunächst sollte wieder festgelegt werden welche Aufgaben das Programm in welcher Abfolge erledigen muss:

1. Entgegennehmen der Verbindungsparameter (IP-Adresse und Remote-Port des Web-IO
2. Aufbau der TCP-Verbindung
3. Starten des Timers der zyklisch die Inputs des Web-IO abfragt.
   Wenn ein ON auf einem der beiden betroffenen Inputs anliegt, starten des Timers, der die einzelnen Schritte der Ampelsteuerung abwickelt

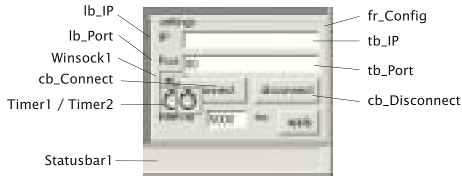
Schritt 1	Fahrzeugampeln: gelb Fußgängerampeln: rot
Schritt 2	Fahrzeugampeln: rot Fußgängerampeln: rot
Schritt 3	Fahrzeugampeln: rot Fußgängerampeln: grün
Schritt 4	Fahrzeugampeln: rot Fußgängerampeln: rot
Schritt 5	Fahrzeugampeln: rot und gelb Fußgängerampeln: rot
Schritt 6	Fahrzeugampeln: grün Fußgängerampeln: rot
4. Bei Schließen der TCP-Verbindung zunächst die Ampelphasen so zu Ende führen, dass die Grünphase für die Fußgänger abgeschlossen und die Grünphase für die Autos erreicht wird. Danach alle Ampellichter ausschalten und dann erst die Verbindung schließen und die Buttons aktivieren bzw. sperren.
5. Bei Beenden des Programms zunächst die Ampelphasen so zu Ende führen, dass die Grünphase für die Fußgänger abgeschlossen und die Grünphase für die Autos erreicht wird. Danach alle Ampellichter ausschalten und dann erst das Programm beenden.

Im Programmfenster soll wie gewohnt ein komplettes Prozessabbild wiedergegeben werden, allerdings erst wenn das Setzen der Outputs vom Web-IO quittiert wurde.

Vorarbeiten

Als Grundgerüst für diese Anwendung kann das komplette Projekt aus Programmbeispiel 5 benutzt werden. Dazu wird das komplette Projekt in ein neues Verzeichnis kopiert. Das Projekt muss dann aus dem neuen Verzeichnis geöffnet werden.

Die Bedienelemente lassen sich wie sie sind nutzten, es wird aber ein weiterer Timer benötigt.



Für beide Timer muss die *Enabled*-Eigenschaft auf *False* gesetzt sein. Für *Timer2* sollte ein Intervall von 50ms eingstellt werden.

Der Quellcode

Als Grundgerüst haben wir den Quelltext aus Beispiel 5 genommen.

Festlegung der benutzten Farbkonstanten.

```
Private Const GREEN_OFF = &H8000&
Private Const GREEN_ON = &HFF00&
Private Const RED_OFF = &H80&
Private Const RED_ON = &HFF&
Private Const YELLOW_OFF = &H8080&
Private Const YELLOW_ON = &HFFFF&
Private Const BLACK_OFF = &H0&
Private Const BLACK_ON = &HFFFFFF
```

Konstanten für die Bitwertigkeit der einzelnen Outputs.

```
Private Const OUTPUT0 = 1
Private Const OUTPUT1 = 2
Private Const OUTPUT2 = 4
Private Const OUTPUT3 = 8
```

```
Private Const OUTPUT4 = 16
Private Const OUTPUT5 = 32
Private Const OUTPUT6 = 64
Private Const OUTPUT7 = 128
Private Const OUTPUT8 = 256
Private Const OUTPUT9 = 512
Private Const OUTPUT10 = 1024
Private Const OUTPUT11 = 2048
```

Konstanten für die Bitwertigkeit der einzelnen Inputs.
(nicht alle abgebildet)

```
Private Const INPUT0 = 1
Private Const INPUT1 = 2
..
Private Const INPUT10 = 1024
Private Const INPUT11 = 2048
```

Deklaration der verwendeten Variablen:

vin_applicationstep gibt an, in welcher Ampelphase sich das Programm grade befindet. In *vin_interval* wird das Timer-interval zwischengespeichert. *vbo_disconnectquiry* und *vbo_closequiry* sind sogenannte Flags (Merker), die gesetzt werden wenn der Benutzer die TCP-Verbindung bzw. das komplette Programm zu schließen versucht. *vbo_walkerrequest* ist ebenfalls ein Flag, das gesetzt wird, wenn ein Anforderungstaster am Fußgängerweg gedrückt wurde.

```
Dim vin_applicationstep As Integer
Dim vin_interval As Integer
Dim vbo_disconnectquiry As Boolean
Dim vbo_closequiry As Boolean
Dim vbo_walkerrequest As Boolean
```

Die selbst definierte Prozedur *visualiseoutputs* ist nicht an ein Steuerelemet gebunden. Sie wertet den übergebenen Outputwert durch Ausmaskieren aus und setzt die Elemente auf der Programmoberfläche analog zu den Elementen auf dem Schulungsboard.

```
Private Sub visualiseoutputs(Outputs As Integer)
  If (Outputs And OUTPUT0) = OUTPUT0 Then
    sh_CarWest_Green.FillColor = GREEN_ON
    sh_CarEast_Green.FillColor = GREEN_ON
  Else
    sh_CarWest_Green.FillColor = GREEN_OFF
    sh_CarEast_Green.FillColor = GREEN_OFF
  End If
  If (Outputs And OUTPUT1) = OUTPUT1 Then
    sh_CarWest_Yellow.FillColor = YELLOW_ON
    sh_CarEast_Yellow.FillColor = YELLOW_ON
  Else
    sh_CarWest_Yellow.FillColor = YELLOW_OFF
    sh_CarEast_Yellow.FillColor = YELLOW_OFF
  End If
  If (Outputs And OUTPUT2) = OUTPUT2 Then
    sh_CarWest_Red.FillColor = RED_ON
    sh_CarEast_Red.FillColor = RED_ON
  Else
    sh_CarWest_Red.FillColor = RED_OFF
    sh_CarEast_Red.FillColor = RED_OFF
  End If
  If (Outputs And OUTPUT3) = OUTPUT3 Then
    sh_WalkerWestNorth_Green.FillColor = GREEN_ON
    sh_WalkerWestSouth_Green.FillColor = GREEN_ON
    sh_WalkerEastNorth_Green.FillColor = GREEN_ON
    sh_WalkerEastSouth_Green.FillColor = GREEN_ON
  Else
    sh_WalkerWestNorth_Green.FillColor = GREEN_OFF
    sh_WalkerWestSouth_Green.FillColor = GREEN_OFF
    sh_WalkerEastNorth_Green.FillColor = GREEN_OFF
    sh_WalkerEastSouth_Green.FillColor = GREEN_OFF
  End If
  If (Outputs And OUTPUT4) = OUTPUT4 Then
    sh_WalkerWestNorth_Red.FillColor = RED_ON
    sh_WalkerWestSouth_Red.FillColor = RED_ON
    sh_WalkerEastNorth_Red.FillColor = RED_ON
    sh_WalkerEastSouth_Red.FillColor = RED_ON
  Else
    sh_WalkerWestNorth_Red.FillColor = RED_OFF
    sh_WalkerWestSouth_Red.FillColor = RED_OFF
```

```
    sh_WalkerEastNorth_Red.FillColor = RED_OFF
    sh_WalkerEastSouth_Red.FillColor = RED_OFF
  End If
End Sub
```

visualiseinputs zeigt bei Übergabe des Inputwertes in der Programmoberfläche an, wenn einer der Anfordeungstaster gedrückt wurde.

```
Private Sub visualiseinputs(Inputs As Integer)
  If (Inputs And INPUT2) = INPUT2 Then
    sh_RequestWestNorth.FillColor = BLACK_ON
    sh_RequestWestSouth.FillColor = BLACK_ON
  Else
    sh_RequestWestNorth.FillColor = BLACK_OFF
    sh_RequestWestSouth.FillColor = BLACK_OFF
  End If
  If (Inputs And INPUT3) = INPUT3 Then
    sh_RequestEastNorth.FillColor = BLACK_ON
    sh_RequestEastSouth.FillColor = BLACK_ON
  Else
    sh_RequestEastNorth.FillColor = BLACK_OFF
    sh_RequestEastSouth.FillColor = BLACK_OFF
  End If
End Sub
```

activitiesbyinputs wertet ebenfalls den Inputwert aus. Es wird *vbo_walkerrequest = True* gesetzt und *Timer1* gestartet, wenn ein gedrückter Anforderungstaster erkannt wurde.

```
Private Sub activitiesbyinputs(Inputs As Integer)
  If (Inputs And INPUT2) = INPUT2 Then
    Timer1.Enabled = True
    vbo_walkerrequest = True
  End If
  If (Inputs And INPUT3) = INPUT3 Then
    Timer1.Enabled = True
    vbo_walkerrequest = True
  End If
End Sub
```

Bei Programmstart werden die benötigten Variablen auf einen definierten Anfangszustand gesetzt.

```
Private Sub Form_Load()
  vin_applicationstep = 1
End Sub
```

Die Prozedur *bt_Connect_Click* wurde unverändert übernommen

```
Private Sub cb_Connect_Click()
  If (tb_Port.Text <> "") And (tb_IP.Text <> "") Then
    Winsock1.RemoteHost = tb_IP.Text
    Winsock1.RemotePort = Val(tb_Port.Text)
    Winsock1.Connect
  End If
End Sub
```

Bei erfolgreichem Verbindungsaufbau werden neben dem Freischalten/Sperren der Buttons beide Timer gestartet. *vbo_walkerrequest* wird auf *true* gesetzt und damit wird zunächst eine komplette Grünphase für Fußgänger durchlaufen.

```
Private Sub Winsock1_Connect()
  cb_Connect.Enabled = False
  cb_Disconnect.Enabled = True
  StatusBar1.SimpleText = "connected"
  vbo_walkerrequest = True
  vin_interval = Val(tb_Interval.Text)
  Timer1.Interval = vin_interval / 10
  Timer1.Enabled = True
  Timer2.Enabled = True
End Sub
```

Timer1 steuert den zeitlichen Ablauf der Ampelphasen und sorgt für einen geregelten Verbindungsabbau bzw. ein geregeltes Programmende.

Im Gegensatz zum voran gegangenen Beispiel, wird in der Timer Prozedur die Programmoberfläche bis auf eine Aus-

nahme nicht mehr beeinflusst (dies wird erst bei Empfang der Antworten vom Web-IO ausgelöst)

```
Private Sub Timer1_Timer()
  Select Case vin_applicationstep
```

Phase1: Gelbphase für Autos

Ist *vbo_walkerrequest = True*, werden alle Schritte ausgeführt um die Gelbphase für die Autos zu beginnen, anderenfalls wird der Timer gestoppt. Ist *vbo_diconnectquiry* gesetzt, werden alle LEDs ausgeschaltet und die TCP-Verbindung beendet. Da die Antwort vom Web-IO mit dem Outputstatus durch das Schließen der Verbindung ggf. nicht mehr gesendet wird, wird der Prozedur *visuliseoutputs* der hexadezimale Wert 0000 übergeben, um in der Programmoberfläche ein korrektes Abbild des Schulungsboards anzuzeigen. Ist *vbo_closequery = True* wird das Programm geschlossen.

```
    Case 1
      Timer1.Interval = vin_interval / 10
      If vbo_walkerrequest Then
        Winsock1.SendData ("GET /outputaccess?PW=&State=0012&")
        vin_applicationstep = vin_applicationstep + 1
      Else
        Timer1.Enabled = False
        If vbo_disconnectquiry Then
          Timer2.Enabled = False
          Winsock1.SendData ("GET /outputaccess?PW=&State=0000&")
          visualiseoutputs (0)
          DoEvents
          Winsock1.Close
          StatusBar1.SimpleText = "disconnected"
          cb_Connect.Enabled = True
          If vbo_closequiry Then End
          vbo_disconnectquiry = False
        End If
     End If
```

Phase 2: Rot für alle

```
    Case 2
      Winsock1.SendData ("GET /outputaccess?PW=&State=0014&")
     vin_applicationstep = vin_applicationstep + 1
```

Phase 3: Grün für die Fußgänger
Das Timer-Intervall wird auf seinen normalen Wert gesetzt.

```
    Case 3
      Winsock1.SendData ("GET /outputaccess?PW=&State=000C&")
      Timer1.Interval = vin_interval
     vin_applicationstep = vin_applicationstep + 1
```

Phase 4: Rot für alle
Das Timer-Intervall wird auf ein Zehntel seines normalen Wertes gesetzt (kurze Zwischenphase.).

```
    Case 4
      Winsock1.SendData ("GET /outputaccess?PW=&State=0014&")
      Timer1.Interval = vin_interval / 10
     vin_applicationstep = vin_applicationstep + 1
```

Phase 5: Rot-Gelb für die Fahrzeugampeln

```
    Case 5
      Winsock1.SendData ("GET /outputaccess?PW=&State=0016&")
      vin_applicationstep = vin_applicationstep + 1
```

Phase 6: Grün für die Autos
Die Variable *vbo_walkerrequest* wird auf *False* gesetzt, damit der Timer beim nächsten Durchlauf gestoppt wird und die Grünphase bis zur nächsten Anforderung durch die Fußgänger bestehen bleibt.

Ist *vbo_disconnectquiry* gesetzt, wird das Timer-Intervall auf ein Zehntel seines Wertes gesetzt. Zum Abschluss wird *vin_applicationstep* auf 1 gesetzt, damit beim nächsten Durchlauf wieder mit der Gelbphase für Autos gestartet werden kann.

```
    Case 6
```

```
      Winsock1.SendData ("GET /outputaccess?PW=&State=0011&")
      vbo_walkerrequest = False
      If vbo_disconnectquiry Then
        Timer1.Interval = vin_interval / 10
      Else
        Timer1.Interval = vin_interval
      End If
      vin_applicationstep = 1
    End Select
End Sub
```

Timer2 wird bei erfolgreichem Verbindungsaufbau gestartet, und pollt die Eingänge des Web-IO durch zyklisches Senden des Input Kommandostrings ab.

```
Private Sub Timer2_Timer()
  Winsock1.SendData ("GET /input?PW=&")
End Sub
```

Durch Klick auf den *apply*-Button wird das eingetragene Timer-Intervall in die Variable *vin_intervall* geschrieben.

```
Private Sub cb_Apply_Click()
  vin_interval = Val(tb_Interval.Text)
End Sub
```

Die Prozedur *ClientSocket1Read* wird gestartet, wenn Daten vom Web-IO empfangen werden.

Die empfangenen Daten werden zunächst in die Variable *vst_InBuffer* geschrieben.

Dann wird geprüft, ob das erste Zeichen des empfangenen Strings ‘o’ (output) oder ‘i’ (input) ist. Abhängig davon, wird der hexadezimale Anteil des Strings in die Variable *vst_HexString* kopiert.

Bei Empfang der Output-Zustände wird *vst_HexString* an die Prozedur *visuliseoutputs()* übergeben, die die Programmoberfläche entsprechend verändert.

Betrifft der empfangene String die Inputs wird *vst_HexString* an die Prozedur *visuliseinputs()* übergeben, die die gedrückten Taster in der Programmoberfläche anzeigt. Ferner wird die Prozedure *activitiesbyinputs()* aufgerufen, die den weiteren Programmablauf bestimmt.

```
Private Sub Winsock1_DataArrival(ByVal bytesTotal As Long)
Dim vst_InBuffer As String
Dim vst_HexString As String
  Winsock1.GetData vst_InBuffer
  Select Case Left(vst_InBuffer, 1)
    Case "o"
      vst_HexString = Mid(vst_InBuffer, 8, 4)
      visualiseoutputs (Val("&H" + vst_HexString))
    Case "i"
      vst_HexString = Mid(vst_InBuffer, 7, 4)
      visualiseinputs (Val("&H" + vst_HexString))
      activitiesbyinputs (Val("&H" + vst_HexString))
    End Select
End Sub
```

Wünscht der Anwender die TCP-Verbindung zu beenden, wird die Variable *vbo_disconnectquiry* auf *True* gesetzt und *Timer1* gestartet, der in der ersten Phase (*vin_applicationstep = 1*) die Ampellichter ausschaltet und die TCP-Verbindung beendet. Ferner wird in der Stauszeile „waiting for disconnect“ ausgegeben.

```
Private Sub cb_Disconnect_Click()
  vbo_disconnectquiry = True
  Timer1.Enabled = True
  cb_Disconnect.Enabled = False
  StatusBar1.SimpleText = "waiting for disconnect"
End Sub
```

Wünscht der Anwender bei aktiver Verbindung das Programm zu beenden, werden die Variablen *vbo_disconnectquiry* und *vbo_closequiry* auf *True* gesetzt. *Timer1* wird gestartet, der der in ersten Phase (*vin_applicationstep = 1*) die Ampellichter ausschaltet und

die TCP-Verbindung und das Programm beendet. Ferner wird in der Statuszeile „waiting for disconnect“ ausgegeben.

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
  If Winsock1.State = sckConnected Then
    vbo_closequiry = True
    vbo_disconnectquiry = True
    Timer1.Enabled = True
    StatusBar1.SimpleText = "waiting for disconnect"
    Cancel = True
  End If
End Sub
```

Mit einem überschaubaren Maß an Quelltext haben wir bereits eine komplexe IO-Anwendung realisiert.

Die Prozeduren *visuliseoutputs* und *visuliseinputs* können für noch komplexere Anwendung so erweitert werden, dass alle Anzeigeobjekte (auch für die Straße von Nord nach Süd) berücksichtigt werden.

Programmierung des Web-IO unter C++

Bei allen hier gezeigten Programmbeispielen und Erklärungen wird der grundsätzliche Umgang mit der Programmierumgebung von Microsoft C++ vorausgesetzt.

4.0 Ein vorbereitetes Programm-Layout

Um einen schnellen Einstieg in die eigentliche Programmierarbeit zu ermöglichen, haben wir ein komplettes Programmgerüst vorbereitet, das sich um die graphische Ausgabe der Ampelschaltung auf den Monitor kümmert.

Das Programm wurde mit dem MFC Anwendungsassistenten erstellt (*Dialogfeldbasierend* und *Unterstützung für Sockets* aktiviert).

Um eine bessere Übersichtlichkeit zu erreichen, wurde eine globale Struktur mit Unterstrukturen angelegt, die den aktuellen Status der einzelnen Elemente der Kreuzung wiederspiegeln soll:

```
struct SStatus
{
   // 4 BOOLEANs für die Autos
   BOOLEAN bCarEast,bCarNorth,bCarWest,bCarSouth;
   // 2 BOOLEANs für die Beeper
   BOOLEAN bBeeperSouthEast,bBeeperSouthWest;

   // 4 Strukturen mit je 3 BOOLEANs für die 3 Lichter für die 4 Ampeln
   struct SCarLight
   {
      BOOLEAN bRed,bYellow,bGreen;
   }CarLightNorth,CarLightSouth,CarLightWest,CarLightEast;

   // 8 Fußgängerampeln mit je 2 BOOLEANs für die 2 Lichter und einem BOOLEAN für
den Taster
   struct SWalkerLight
   {
      BOOLEAN bRed,bGreen;
      BOOLEAN bRequest;
   }WalkerLightSouthWest,WalkerLightSouthEast,
   WalkerLightNorthEast,WalkerLightNorthWest,
    WalkerLightWestSouth,WalkerLightEastSouth,
   WalkerLightEastNorth,WalkerLightWestNorth;
}g_sStatus;
```

Die Inhalte dieser Struktur werden verwendet, um die Kreuzung darzustellen. Es muss also nur der Inhalt verändert werden und das Fenster neu gezeichnet werden, um die aktuellen Änderungen der Kreuzung darzustellen. Der bestehende Programmcode für das Zeichnen des Fensters muss nicht verändert werden.

Alle Elemente der Struktur werden in *OnInitDialog* mit

```
FillMemory(&g_sStatus,sizeof(g_sStatus),0);
```

auf

```
FALSE (=0)
```

gesetzt.

Das Hauptfenster besteht aus einem Dialog. In dessen *OnPaint*-Methode wird nun das komplette AmpelAbbild gezeichnet. Dazu werden Bitmaps geladen und abhängig vom Inhalt der globalen Struktur, die Elemente in dem Dialog angezeigt.

Beispiele:

Auto:

```
g_sStatus.bCarEast=TRUE;
```

beispielsweise setzt das Auto im Osten auf ON.

Verkehrsampel:

```
g_sStatus.CarLightSouth.bYellow=FALSE;
```

setzt das gelbe Ampellicht im Süden auf OFF.

Fußgängerampel:

```
g_sStatus.WalkerLightEastSouth.bRequest=TRUE;
```

setzt den Taster für die südliche Fußgängerampel des östlichen Srtassenabgangs auf ON. Die Richtung für die zugehörige Straße steht an erster Stelle des Bezeichners.

Beeper:

```
g_sStatus.bBeeperSouthWest=TRUE;
```

setzt den Beeper im Südwesten auf ON.

4.1 Ein einführendes Beispiel

4.1.1 Die Aufgabenstellung

Es soll ein einfaches Programm erstellt werden, mit dem über das Web-IO Digital das grüne Ampellicht der Straße West/Ost ein- und ausgeschaltet werden kann.

4.1.2 Die Lösung

Programmplanung
Folgende Aufgaben müssen vom Programm abgearbeitet werden:

1. Entgegennehmen der IP-Adresse. Als Port wird 80 für http verwendet.
2. Einrichten der Verbindung über ein Socket
3. Freigeben und Sperren der Buttons für die entsprechenden Funktionen
4. Ein- und Ausschalten des grünen Ampellichts über zwei Buttons
5. Beenden der Verbindung

Im Programmfenster soll ein komplettes Abbild der Straßenkreuzung mit allen Elementen dargestellt werden, in dem die Abläufe auf dem Schulungsboard 1:1 wiedergegeben werden.

Vorarbeiten
Folgende Steuerelemente haben wir in den Dialog eingefügt, um das Programm bedienbar zu machen:

Für die Buttons wurden im Klassenassistenten Membervariablen vom Typ CButton mit den Namen *m_Connect, m_Disconnect, m_On und m_Off* eingerichtet, für das Edit wurde eine Membervariable vom Typ CString mit dem Bezeichner *m_IP* eingerichtet.

Der Quellcode

Es wird zunächst eine globale Variable vom Typ CSocket eingerichtet, welche die Verbindung zum Web-IO repräsentiert. Dazu wird in den Kopf von *AmpelDlg.cpp* die folgende Zeile eingefügt:

```
CSocket g_cSocket;
```

Als erstes sollte ein Behandler für den *Connect*-Button implementiert werden. Dazu wird im Klassenassistenten eine Funktion für das *BN_CLICKED*-Ereignis des *IDC_CONNECT*-Buttonserstellt:

```
void CAmpelDlg::OnConnect()
{
   UpdateData();

   g_cSocket.Create();
   if (g_cSocket.Connect(m_IP,80) == FALSE)
   {
      g_cSocket.Close();
      MessageBox(„Verbindung fehlgeschlagen“);
      return;
   }

   m_Connect.EnableWindow(FALSE);
   m_Disconnect.EnableWindow(TRUE);
   m_On.EnableWindow(TRUE);
   m_Off.EnableWindow(FALSE);
}
```

Mit *UpdateData* wird automatisch der Wert des Eingabefeldes in der Membervariable *m_IP* abgespeichert.

Ein Socket wird erstellt. Anschließend findet der Verbindungsaufbau zu der angegebenen IP Adresse auf Port 80 statt. Schlägt dieser Verbindungsaufbau fehl, wird mit einer Fehlermeldung abgebrochen.

Anschließend werden die Steuerelemente aktiviert bzw. deaktiviert, je nach dem ob sie gebraucht oder nicht gebraucht werden.

Auf gleiche Weise erstellen wir eine Funktion für den *IDC_DISCONNECT*-Button:

```
void CAmpelDlg::OnDisconnect()
{
   g_cSocket.Close();

   m_Connect.EnableWindow(TRUE);
   m_Disconnect.EnableWindow(FALSE);
   m_On.EnableWindow(FALSE);
   m_Off.EnableWindow(FALSE);
}
```

Mit einem einzigen Befehl wird die Verbindung des Sockets geschlossen. Danach werden die Buttons wieder aktiviert bzw. deaktiviert.

Nun die zwei Funktionen für die *IDC_ON* und *IDC_OFF* Buttons:

```
void CAmpelDlg::OnOn()
{
   g_sStatus.CarLightWest.bGreen=TRUE;
   g_sStatus.CarLightEast.bGreen=TRUE;

   Invalidate(FALSE);

   CString szRequest="GET /outputaccess0?PW=&State=ON&";
   g_cSocket.Send(szRequest,szRequest.GetLength());

   m_On.EnableWindow(FALSE);
   m_Off.EnableWindow(TRUE);
```

```
}

void CAmpelDlg::OnOff()
{
    g_sStatus.CarLightWest.bGreen=FALSE;
    g_sStatus.CarLightEast.bGreen=FALSE;

    Invalidate(FALSE);

    CString szRequest="GET /outputaccess0?PW=&State=OFF&";
    g_cSocket.Send(szRequest,szRequest.GetLength());

    m_On.EnableWindow(TRUE);
    m_Off.EnableWindow(FALSE);
}
```

Diese beiden Funktionen sind sehr ähnlich.

Als erstes wird in der globalen Struktur das grüne Licht der westlichen und östlichen Fahrzeugampel auf grün gesetzt. Dann wird das Programm mit *Invalidate(FALSE)* angewiesen, den Inhalt des Fensters neuzuzeichnen, damit die Änderungen in der Struktur auf dem Monitor dargestellt werden.

Anschließend erstellen wir einen String, der das Kommando an das Web-IO beinhaltet.

Der Kommandostring wird als Puffer zusammen mit seiner Länge an die Funktion *Send* der CSocket-Klasse übergeben. Durch Senden dieser Zeichenkette an das Web-IO, schaltet dieses den Output0 auf ON bzw. OFF.

Danach wird der *On*-Button deaktiviert und der *Off*-Button aktiviert.

Die *OnOff* Funktion ist analog dazu.

Wichtig:

Damit das Programm sauber beendet wird, sollte das Socket ordentlich geschlossen werden, wenn das Programm ge-

schlossen wird und die Verbindung noch besteht. Dazu erstellen wir mit dem Klassenassistenten eine Methode für die WM_DESTROY *Nachricht des Fensters:*

```
void CAmpelDlg::OnDestroy()
{
   CDialog::OnDestroy();

   if (g_cSocket.m_hSocket)
      g_cSocket.Close();
}
```

Wenn das Socket noch existiert, wird es geschlossen. Danach kann das Programm problemlos beendet werden.

Mit diesen paar Zeilen Quellcode wurde gezeigt, dass man das Web-IO ganz einfach über Sockets steuern kann.

Hinweis: Wenn die Verbindung geschlossen, oder das Programm beendet wird, bleiben die Ampellichter eingeschaltet, wenn sie zu diesem Zeitpunkt an war. Um das zu verhindern, kann man die *OnDisconnect* bzw. die *OnDestroy* Methoden um den Aufruf von *OnOff* erweitern. Dies geschieht natürlich vor dem Aufruf von *g_cSocket->Close()* .

Damit verhält sich das Programm so, als ob der *Off*-Button betätigt wurde, bevor die Verbindung geschlossen wird.

4.2 Zeitgesteuerte Ausgaben

4.2.1 Die Aufgabenstellung

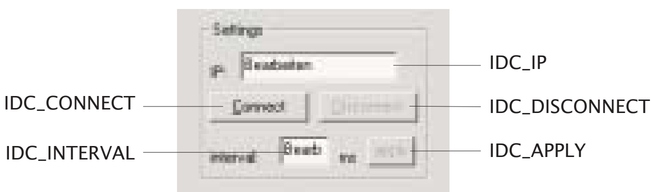
Die gelben Ampellichter der Nebenstraße (von Nord nach Süd) sollen blinken. Das Blinkintervall soll über ein Eingabefeld frei einstellbar sein.

4.2.2 Die Lösung

Programmplanung
Folgende Aufgaben müssen vom Programm abgearbeitet werden:

1. Entgegennehmen der IP-Adresse
2. Aufbau der Verbindung über ein Socket
3. Starten eines Timers, der im Wechsel das gelbe Ampellicht auf dem Schulungsboard ein- und ausschaltet.
4. Das Programm soll eine Änderung des Timerintervalls ermöglichen.

Vorarbeiten
Folgende Steuerelemente werden benötigt:



Den Steuerelementen werden wie im vorherigen Beispiel über den Klassenassistenten Membervariablen zugewiesen, für *IDC_INTERVAL* wird die Membervariable *m_IP* vom Typ int eingerichtet.

Der Quellcode
In OnInitDialog wird *m_Interval* mit dem Wert 500 initialisiert:

```
m_Interval=500;
UpdateData(FALSE);
```

Sobald die Verbindung hergestellt wurde, startet ein Timer, indem folgender Funktionsaufruf in die *OnConnect* Methode geschrieben wird.

```
SetTimer(1,m_Interval,NULL);
```

Der Timer erhält die ID 1 (das ist in diesem Beispiel unwichtig, später aber sinnvoll, um mehrere Timer unterscheiden zu können), und das eingegebene Intervall.

Wenn die Verbindung geschlossen wird, muss der Timer beendet werden. Dazu müssen die *OnDestroy*- und die *OnDisconnect*- Methode geändert werden:

```
void CAmpelDlg::OnDisconnect()
{
    KillTimer(1);

    g_cSocket.Close();

    m_Connect.EnableWindow(TRUE);
    m_Disconnect.EnableWindow(FALSE);
    m_Apply.EnableWindow(FALSE);
}

void CAmpelDlg::OnDestroy()
{
    CDialog::OnDestroy();

    if (g_cSocket.m_hSocket)
    {
        KillTimer(1);
        g_cSocket.Close();
    }
}
```

Mit *Killtimer(1)* wird der Timer gestoppt.

Innerhalb jedes Intervalls wird einmal die *WM_TIMER*-Nachricht an das Fenster gesendet. Im Klassenassistenten fügt man dazu für den Dialog eine Behandlungsroutine für die *WM_TIMER* Nachricht ein:

```
void CAmpelDlg::OnTimer(UINT nIDEvent)
{
   CString sRequest;
   if (g_sStatus.CarLightWest.bYellow)
   {
      g_sStatus.CarLightWest.bYellow=FALSE;
      g_sStatus.CarLightEast.bYellow=FALSE;

      sRequest="GET /outputaccess1?PW=&State=OFF&";
   }else{
      g_sStatus.CarLightWest.bYellow=TRUE;
      g_sStatus.CarLightEast.bYellow=TRUE;

      sRequest="GET /outputaccess1?PW=&State=ON&";
   }

   Invalidate(FALSE);

   g_cSocket.Send(sRequest,sRequest.GetLength());

   CDialog::OnTimer(nIDEvent);
}
```

Zuerst wird der Status der gelben Fahrzeugampel im Westen überprüft. Steht diese auf ON, muss sie auf OFF geschaltet werden. Dazu setzt man die entsprechenden Werte in der globalen Struktur auf FALSE. Anschließend wird der entsprechende Kommandostring an das Web-IO geschickt, um den Output auszuschalten.

Ist das gelbe Licht bei Erreichen des Timer-Ereignisses ausgeschaltet, wird es zum beschriebenen Vorgehen eingeschaltet..

Danach wird das Fenster neu gezeichnet.

Nun muss noch der Behandler für den *IDC_APPLY* Button eingefügt werden:

```
void CAmpelDlg::OnApply()
{
   KillTimer(1);
   UpdateData();
   SetTimer(1,m_Interval,NULL);
}
```

Der bestehende Timer wird entfernt, die m_Interval Variable mit dem Inhalt des entsprechenden Controls aktualisiert und der Timer danach neu mit dem neuen Intervall erstellt.

4.3 Auf Eingaben reagieren

4.3.1 Die Aufgabe

Das Programm soll überwachen, ob einer der beiden Anforderungsschalter im Süden gedrückt wird. Sobald das der Fall ist, soll der Schalter im Programmfenster seine Farbe von schwarz auf weiß ändern.

Dazu wird der String „GET /input7?PW=&" an das Web-IO geschickt. Ist der Input7 des Geräts gesetzt (d.h. hat jemand den Anforderungsschalter im Süden betätigt), schickt das Gerät den String „input7;ON" an unser Programm zurück, andernfalls schickt es „input7;OFF".

4.3.2 Die Lösung

Vorarbeiten

Es wird lediglich ein Edit Feld für die IP und zwei Buttons für Connect und Disconnect gebraucht.

Damit Daten vom Socket empfangen werden, muss zunächst eine eigene Klasse von *CSocket* abgeleitet werden.

Dazu wird der Klassenassistent mit (STRG+W) geöffnet und auf Klasse hinzufügen -> Neu geklickt. Geben Sie als Namen *kann CMySocket* ein, und als Basisklasse CSocket. Anschließend wird mit OK bestätigt.

In der Datei *AmpelDlg.cpp* wird oben ein *#include* „mysocket.h" eingefügt, um dem Dialog die neu erstellte

Klasse bekannt zu machen. Die Variable *CSocket g_cSocket* wird in *CMySocket g_cSocket* gändert.

Bisher verhält sich das Programm noch genauso wie bei den vorangegangenen Beispielen, nur dass diesmal die abgeleitete Klasse als Socket verwendet wird.

Es muss eine Behandlungsroutine hinzugefügt werden, die aufgerufen wird, wenn Nachrichten vom Socket empfangen werden. Dazu wird der Klassenassistent geöffnet, und als Klassennamen *CMySocket* gewählt werden. Nun folgt ein Doppelklick auf OnReceive in der Liste rechts. Die neu erstellte Funktion *CMySocket:OnReceive* wird immer aufgerufen, wenn Daten anstehen. OnReceive gehört zum Socket und hat keine Verbindung mit dem Dialog. Der Dialog wird also nicht über das Eintreffen von Daten informiert.

Damit der Dialog die Daten verarbeiten kann, sind einige weitere Schritte notwendig.

Zunächst muss mit der rechten Maustaste im Arbeitsbereich auf die Klasse *CMySocket*, und dann auf Membervariable hinzufügen geklickt werden. Als Typ wird „CAmpelDlg*“ eingegeben, als Namen wird „m_pAmpelDlg“.gewählt. Der Socket Klasse wird damit eine Variable zur Verfügung gestellt, die auf den Dialog zeigt. Somit kann das Socket, wenn es Daten erhält, auf den Dialog zugreifen. Über den Klassenassitenten muss die Datei „MySocket.h“ geöffnet werden. Dort wird über der Klassendefinition die Zeile

```
#include „ampeldlg.h“
```

eingefügt, damit dem Socket der Klassentyp *CAmpelDlg* des Dialogs bekannt gemacht wird.

Die Variable des Sockets ist nicht initialisiert. Die Initialisierung findet in der *OnInitDialog*-Funktion der *CAmpelDlg*-Klasse statt. Dazu fügt man ans Ende der Funktion (über das Return) die folgende Zeile ein:

```
g_cSocket.m_pAmpelDlg=this;
```

Nun wird der *CAmpelDlg*-Klasse eine *OnReceive* Methode hinzugefügt, die vom Socket aufgerufen wird, wenn Daten anstehen, die der Dialog bearbeiten soll.

Dazu klickt man mit rechts auf CAmpelDlg im Arbeitsbereich und wählt „Memberfunktion hinzufügen. Als Funktionstyp wird void eingegeben (d.h. die Funktion hat keinen Rückgabewert), Funktionsdeklaration ist „OnReceive“, Zugriffsstatus ist public. Nun schreibt man in die OnReceive Methode des Sockets (CMySocket::OnReceive in mysocket.cpp) oben die folgende Zeile:

```
m_pAmpelDlg->OnReceive();
```

Hier wird aus dem Socket, wenn Daten anstehen, die Funktion OnReceive des Dialogs aufgerufen, der die Daten verarbeiten soll.

Der Quellcode

Bei Connect wird wie in den voran gegangenen Beispielen eine Verbindung zum Web-IO hergestellt, ein Timer gestartet der alle 100 ms ein Ereignis auslöst und es werden der *Disconnect*-Button aktiviert und der Connect Button deaktiviert:

```
void CAmpelDlg::OnConnect()
{
   UpdateData();

   g_cSocket.Create();
   if (g_cSocket.Connect(m_IP,80) == FALSE)
   {
      g_cSocket.Close();
      MessageBox(„Verbindung fehlgeschlagen“);
      return;
   }

   SetTimer(1,100,NULL);

   m_Connect.EnableWindow(FALSE);
   m_Disconnect.EnableWindow(TRUE);
```

```
}
```

In der *Disconnect*-Funktion wird der Timer beendet und die Verbindung geschlossen:

```
void CAmpelDlg::OnDisconnect()
{
    KillTimer(1);

    g_cSocket.Close();

    m_Connect.EnableWindow(TRUE);
    m_Disconnect.EnableWindow(FALSE);
}
```

Beendet der Anwender das Programm, wird der Timer beendet und die Verbindung geschlossen:

```
void CAmpelDlg::OnDestroy()
{
    CDialog::OnDestroy();

    if (g_cSocket.m_hSocket != -1)
    {
        KillTimer(1);
        g_cSocket.Close();
    }
}
```

Es wird ein Behandler für das WM_TIMER Ereignis erstellt, um auf den Timer reagieren zu können:

```
void CAmpelDlg::OnTimer(UINT nIDEvent)
{
    CString sRequest;
    sRequest="GET /input7?PW=&";

    g_cSocket.Send(sRequest,sRequest.GetLength());

    CDialog::OnTimer(nIDEvent);
}
```

Ein String wird mit der Zeichenkette „GET /input7?PW=&“, gefüllt. Dieser String wird an das Web-IO Gerät senden. Das Kommando bewirkt, dass das Web-IO den Inputstatus von Input7 zurücksendet. Sobald der Inputstatus angekommen ist, wird die OnReceive Methode des Sockets aufgerufen, diese ruft die OnReceive Methode des Dialogs auf, die wie folgt aussieht:

```
void CAmpelDlg::OnReceive()
{
    CString sData;
    g_cSocket.Receive(sData.GetBuffer(100),100);
    sData.ReleaseBuffer();

    if (sData==“input7;OFF“)
    {
        if (g_sStatus.WalkerLightWestSouth.bRequest==TRUE)
        {
            g_sStatus.WalkerLightWestSouth.bRequest=FALSE;
            g_sStatus.WalkerLightEastSouth.bRequest=FALSE;
            Invalidate(FALSE);
        }
    }else{
        if (g_sStatus.WalkerLightWestSouth.bRequest==FALSE)
        {
            g_sStatus.WalkerLightWestSouth.bRequest=TRUE;
            g_sStatus.WalkerLightEastSouth.bRequest=TRUE;
            Invalidate(FALSE);
        }
    }
}
```

Es werden die ersten 100 Byte Daten, die anliegen empfangen und in einem String abgespeichert.

Wenn der empfangene String z.B.„input7;OFF“ ist, bedeutet das, dass der Input7 nicht gesetzt ist. Auf Veränderungen wird aber nur reagieret, wenn der aktuelle Status von dem alten Status abweicht.

Bei Änderung wird der Eintrag in der Struktur aktualisiert

Mit diesen doppelten Abfragen wird verhindert, dass der Inhalt des Programmfensters zu häufig neu gezeichnet wird , was zu einer flackernden Anzeige führen würde.

4.4 Eingehende Daten Auswerten

4.4.1 Die Aufgabe

Das letzte Beispiel soll so erweitert werden, dass der Status aller Inputs abgefragt wird und das Abbild des Schulungsboard für alle Elemente aktualisiert wird.

4.4.2 Noch ein Tipp

Wir könnten 8 Strings an das Web-IO schicken, damit es uns die 8 Inputsignale an das Programm zurückschickt, aber das Web-IO bietet eine elegantere Möglichkeit, gleichzeitig den Status aller Eingänge zu erhalten. Dazu senden wir den String

```
GET /input?PW=&
```

an das Web-IO. Wir erhalten als Rückgabewert Strings wie den Folgenden:

```
input;0060
```

Hierbei ist die 0060 als Hexzahl zu interpretieren. In Binärdarstellung sehen wir anhand der gesetzten Bits, welcher Eingang des Web-IO gesetzt ist.

```
0060h = 1100000b
```

In dem Beispiel sind die Bits 5 und 6 gesetzt, das heißt am Web-IO liegen die Eingänge Input5 und Input6 an.

4.4.3 Die Lösung

Die *OnReceive* Methode des Dialogs wird geändert wie folgt:. Wie im vorangegangenen Beispiel werden bis zu 100 Byte Daten empfangen und in einen String geschrieben.

Anschließend wird überprüft, ob die ersten 6 Zeichen des Strings der Zeichenfolge „input“ entsprechen. Wenn dem nicht so ist, handelt es sich bei den empfangenen Daten, nicht um die angeforderten Eingänge. Werden z.B. Lampen des Web-IOs gesetzt, sendet dieses den Status aller Lampen zurück. Im folgenden Beispiel wollen wir auf diese Daten nicht reagieren, deshalb ist diese Abfrage notwendig.

Handelt es sich bei den Empfangsdaten um eine Inputstatusmeldung, wandeln wir die 4 rechten Zeichen in einen hexadezimalen Zahlenwert.

Das geschieht mit:

```
int state;

sscanf(sData.Right(4),"%x",&state);
```

Das *%x* weist dem Befehl an, die übergebene Zeichenkette als Hexadezimalzahl zu interpretieren und in state abzuspeichern.

Mit logischen und-Verknüpfungen wird ermittelt, ob bestimmte Bits in der Zahl gesetzt sind.

Mit (state & 0x0001) erhält man 1, wenn das Bit 0 gesetzt ist, also der Input0 auf ON steht. Andernfalls ergibt (state & 0x0001) eine 0.

Für Bit 5 wäre der Wert, mit dem der Status verundet werden muss 0x0020 (=2^5).

In der neuen *OnReceive*-Methode muss für jedes Bit eine entsprechene *State*-Abfrage durchgeführt werden und über If der entsprechende Status in die Struktur geschrieben werden.

```
void CAmpelDlg::OnReceive()
{
    CString sData;
    int state;
```

```
BOOLEAN on;

g_cSocket.Receive(sData.GetBuffer(100),100);
sData.ReleaseBuffer();

if (sData.Left(6)=="input;")
{
    sscanf(sData.Right(4),"%x",&state);

    on=(state & 0x0001);
    if (on!=g_sStatus.bCarWest)
    {
        g_sStatus.bCarWest=on;
        Invalidate(FALSE);
    }

    on=(state & 0x0002);
    if (on!=g_sStatus.bCarEast)
    {
        g_sStatus.bCarEast=on;
        Invalidate(FALSE);
    }

    on=(state & 0x0010);
    if (on!=g_sStatus.bCarNorth)
    {
        g_sStatus.bCarNorth=on;
        Invalidate(FALSE);
    }

    on=(state & 0x0020);
    if (on!=g_sStatus.bCarSouth)
    {
        g_sStatus.bCarSouth=on;
        Invalidate(FALSE);
    }

    on=(state & 0x0004);
    if (on!=g_sStatus.WalkerLightNorthWest.bRequest)
    {
        g_sStatus.WalkerLightNorthWest.bRequest=on;
```

```
            g_sStatus.WalkerLightSouthWest.bRequest=on;
            Invalidate(FALSE);
        }

        on=(state & 0x0008);
        if (on!=g_sStatus.WalkerLightNorthEast.bRequest)
        {
            g_sStatus.WalkerLightNorthEast.bRequest=on;
            g_sStatus.WalkerLightSouthEast.bRequest=on;
            Invalidate(FALSE);
        }

        on=(state & 0x0040);
        if (on!=g_sStatus.WalkerLightWestNorth.bRequest)
        {
            g_sStatus.WalkerLightWestNorth.bRequest=on;
            g_sStatus.WalkerLightEastNorth.bRequest=on;
            Invalidate(FALSE);
        }

        on=(state & 0x0080);
        if (on!=g_sStatus.WalkerLightWestSouth.bRequest)
        {
            g_sStatus.WalkerLightWestSouth.bRequest=on;
            g_sStatus.WalkerLightEastSouth.bRequest=on;
            Invalidate(FALSE);
        }
    }
}
```

Das Programm aus dem vorherigen Beispiel wird um eine Funktion erweitert.

Es gibt ebenfalls ein Kommando, das man an das Web-IO schicken muss, um alle Ausgänge gleichzeitig zu setzen.

Dazu schickt man den String

```
GET /outputaccess?PW=&State=0000&
```

an das Web-IO, wobei 0000 eine Hexadezimalzahl darstellt, in der die Bits für die einzelnen Ausgänge gesetzt sind.

Aus der globalen Struktur muss ein Hexadezimalwert errechnet werden, der die Outputs des Schulungsboards gemäß unser Struktur setzt. Die folgende Funktion wird unter die Definition der Struktur geschrieben.

```
void SendStatus()
{
   CString s;
   int state=     (g_sStatus.CarLightEast.bGreen      << 0 )
          +(g_sStatus.CarLightEast.bYellow     << 1 )
          +(g_sStatus.CarLightEast.bRed        << 2 )
          +(g_sStatus.WalkerLightNorthEast.bGreen  << 3 )
          +(g_sStatus.WalkerLightNorthEast.bRed<< 4 )
          +(g_sStatus.bBeeperSouthEast         << 5 )
          +(g_sStatus.CarLightSouth.bGreen     << 6 )
          +(g_sStatus.CarLightSouth.bYellow    << 7 )
          +(g_sStatus.CarLightEast.bRed        << 8 )
          +(g_sStatus.WalkerLightSouthEast.bGreen  << 9 )
          +(g_sStatus.WalkerLightSouthEast.bRed<< 10)
          +(g_sStatus.bBeeperSouthWest         << 11);

   s.Format(„GET /outputaccess?PW=&State=%x&“,state);
   g_cSocket.Send(s,s.GetLength());
}
```

Ist z.B. der Beeper im Süd-Westen gesetzt (TRUE entspricht einer binären 1), wird zum Statuswert Bits 5 (= 2^5) dazuaddiert.

Dieser Wert wird als Hexzahl in den Kommandostring eingefügt und an das Web-IO geschickt.

4.5 Eine komplett Zeitgesteuerte Anwendung:

4.5.1 Die Aufgabe

Für die Strasse von Ost nach West soll eine zeitgesteuerte Fußgängerampel erstellt werden. Dabei soll die Grünphase der Autos, genau wie die Grünphase der Fußgänger, 20 Sekunden dauern.

4.5.2 Die Lösung

Für die Anwendung wird das Programm aus dem vorherigem Beispiel verwendet und erweitert.

Folgende Aufgaben müssen vom Programm abgearbeitet werden:

1. Entgegennehmen der IP-Adresse
2. Aufbau der Verbindung über ein Socket
3. Das Programm startet zwei Timer, einen, der die Eingänge des Geräts alle 0,1s abfragt, und einen Timer, der die einzelnen Schritte der Ampelsteuerung abwickelt. Die Eingänge des Web-IO werden abgefragt, in der Anwendung dargestellt, haben aber sonst keine Reaktion. Das kommt im nächstem Beispiel.

In diesem Beispiel werden mehrere Ausgänge gleichzeitig, gesetzt, indem ein Strings der folgenden Form an das Web-IO gesendet wird:

```
GET /outputaccess?PW=&State=0001&
```

setzt zum Beispiel Output 0 auf ON und alle anderen Outputs auf OFF.

Das Programm aus dem vorherigem Beispiel stellt bereits die Verbindung her und startet den ersten Timer.

Ein zweiter Timer muss gestartet werden. Das geschieht in der *OnConnect* Methode im Anschluss an den Start des ersten Timers:

```
SetTimer(2,500,NULL);
```

Der zweite Timer bekommt die ID 2, damit er später identifiziert werden kann. Als Intervall wird 20000 ms eingerichtet.

Die Methoden *OnDisconnect* und *OnDestroy* müssen nun auch um den Aufruf

```
KillTimer(2);
```

erweitert werden, damit der zweite Timer zusammen mit dem ersten beendet wird.

Es wird eine globale Variable benötigt, in der die aktuelle Ampelphase gespeichert wird. Sie wird zu Beginn des Programms mit 1 initialisiert. Eine zweite globale Variable gibt das Timer Intervall an, und wird mit 20000 ms initialisiert:

```
int g_iApplicationStep=1;
int g_iTimerIntervall=20000;
```

Nun laufen zwei Timer. Wenn einer der Timer ausgelöst wird, wird die Methode *OnTimer* des Fensters aufgerufen. In dieser Methode muss unterschieden werden, ob der erste, oder der zweite Timer ursächlich ist. Das wird anhand des übergebenen Parameters der Methode geprüft:

```
void CAmpelDlg::OnTimer(UINT nIDEvent)
{
    if (nIDEvent==1)
    {
        CString sRequest;
        sRequest="GET /input?PW=&";

        g_cSocket.Send(sRequest,sRequest.GetLength());
    }
```

```
if (nIDEvent==2)
{
    CString sCommand;
```

Es wird eine lokale Variable eingerichtet, in welcher der Kommandostring gespeichert wird, der an das Web-IO gesendet werden soll, um alle Ausgänge zu setzen.

Phase 1: Gelbphase für Autos:

Der erste Schritt ist die Gelbphase der Autos. Es werden die gelben Lichter der Verkehrsampel auf ON, die roten Lichter der Fußgängerampeln auf ON, und alle anderen auf OFF gesetzt.

```
    if (g_iApplicationStep==1)
    {
        g_sStatus.CarLightEast.bRed=FALSE;
        g_sStatus.CarLightEast.bYellow=TRUE;
        g_sStatus.CarLightEast.bGreen=FALSE;

        g_sStatus.CarLightWest.bRed=FALSE;
        g_sStatus.CarLightWest.bYellow=TRUE;
        g_sStatus.CarLightWest.bGreen=FALSE;

        g_sStatus.WalkerLightNorthEast.bGreen=FALSE;
        g_sStatus.WalkerLightNorthEast.bRed=TRUE;
        g_sStatus.WalkerLightSouthEast.bGreen=FALSE;
        g_sStatus.WalkerLightSouthEast.bRed=TRUE;
        g_sStatus.WalkerLightNorthWest.bGreen=FALSE;
        g_sStatus.WalkerLightNorthWest.bRed=TRUE;
        g_sStatus.WalkerLightSouthWest.bGreen=FALSE;
        g_sStatus.WalkerLightSouthWest.bRed=TRUE;

        SetTimer(2,g_iTimerIntervall / 10,NULL);
        sCommand="GET /outputaccess?PW=&State=0012&";
    }
```

Phase 2: Rotphase für Autofahrer und Fußgänger:

```
    if (g_iApplicationStep==2)
```

```
{
    g_sStatus.CarLightEast.bRed=TRUE;
    g_sStatus.CarLightEast.bYellow=FALSE;

    g_sStatus.CarLightWest.bRed=TRUE;
    g_sStatus.CarLightWest.bYellow=FALSE;
    sCommand="GET /outputaccess?PW=&State=0014&";
}
```

Phase 3: Grünphase für Fußgänger:

Das Timerintervall wird hochgesetzt, damit die Fußgängerphase ausreichend lange dauert.

```
if (g_iApplicationStep==3)
{
    g_sStatus.WalkerLightNorthEast.bGreen=TRUE;
    g_sStatus.WalkerLightNorthEast.bRed=FALSE;
    g_sStatus.WalkerLightSouthEast.bGreen=TRUE;
    g_sStatus.WalkerLightSouthEast.bRed=FALSE;
    g_sStatus.WalkerLightNorthWest.bGreen=TRUE;
    g_sStatus.WalkerLightNorthWest.bRed=FALSE;
    g_sStatus.WalkerLightSouthWest.bGreen=TRUE;
    g_sStatus.WalkerLightSouthWest.bRed=FALSE;

    SetTimer(2,g_iTimerIntervall,NULL);
    sCommand="GET /outputaccess?PW=&State=000C&";
}
```

Phase 4: Rot für alle

Das Timerintervall wird wieder runtergesetzt, damit die Zwischenphasen entsprechend kurz sind.

```
if (g_iApplicationStep==4)
{
    g_sStatus.WalkerLightNorthEast.bGreen=FALSE;
    g_sStatus.WalkerLightNorthEast.bRed=TRUE;
    g_sStatus.WalkerLightSouthEast.bGreen=FALSE;
    g_sStatus.WalkerLightSouthEast.bRed=TRUE;
    g_sStatus.WalkerLightNorthWest.bGreen=FALSE;
```

```
        g_sStatus.WalkerLightNorthWest.bRed=TRUE;
        g_sStatus.WalkerLightSouthWest.bGreen=FALSE;
        g_sStatus.WalkerLightSouthWest.bRed=TRUE;

        SetTimer(2,g_iTimerIntervall / 10,NULL);
        sCommand="GET /outputaccess?PW=&State=0014&";
    }
```

Phase 5: Rot-Gelb für Autos:

```
    if (g_iApplicationStep==5)
    {
        g_sStatus.CarLightEast.bYellow=TRUE;
        g_sStatus.CarLightWest.bYellow=TRUE;
        sCommand="GET /outputaccess?PW=&State=0016&";
    }
```

Phase 6: Grün für Autos:

Das Timerintervall wird wieder hochgesetzt, damit die Grünphase der Autos entsprechend länger dauert:

```
    if (g_iApplicationStep==6)
    {
        g_sStatus.CarLightEast.bRed=FALSE;
        g_sStatus.CarLightEast.bYellow=FALSE;
        g_sStatus.CarLightEast.bGreen=TRUE;

        g_sStatus.CarLightWest.bRed=FALSE;
        g_sStatus.CarLightWest.bYellow=FALSE;
        g_sStatus.CarLightWest.bGreen=TRUE;

        SetTimer(2,g_iTimerIntervall,NULL);
        sCommand="GET /outputaccess?PW=&State=0011&";
    }
```

Am Ende jeden Timerintervalls wird das Fenster neu gezeichnet und der vorher gesetzte Kommandostring an das Web-IO gesendet. Um den neuen Status zu setzen, wird der Zähler jeweils um 1 erhöht und wenn die Phase 6 überschritten wurde auf 1 zurückgesetzt:

```
        Invalidate(FALSE);
        g_cSocket.Send(sCommand,sCommand.GetLength());
        g_iApplicationStep++;
        if (g_iApplicationStep==7)g_iApplicationStep=1;
    }
```

Alles andere bleibt, wie im vorhergegangenen Beispiel.

4.6 Eine zeit- und ereignisgesteuerte Anwendung

4.6.1 Die Aufgabe

Das vorangegangene Beispiel soll so umprogrammiert werden, dass die Fahrzeugampeln im Normalfall grün zeigen. Nur wenn einer der Anforderungsschalter an den Fußgängerüberwegen gedrückt wird, soll eine Grünphase für die Fußgänger eingeleitet werden. Dabei sollen natürlich auch alle nötigen Zwischenphasen durchlaufen werden.

4,6,2 Die Lösung

Programmplanung
Das Programm soll folgende Aufgaben übernehmen:

1. Entgegennehmen der IP-Adresse
2. Aufbau der Verbindung über ein Socket
3. Das Programm startet zunächst einen Timer, der das Web-IO pollt und die Inputdaten in unserer globalen Struktur abspeichert. Sobald einer der Anforderungsschalter betätigt wurde, soll ein zweiter Timer gestartet werden, der die Grünphase für die Fußgänger einleiten soll. Ist die Grünphase der Fußgänger vorbei und die Grünphase der Autofahrer wieder erreicht, soll der Timer beendet werden.

In Beispiel 4 haben wir bereits die Anwendung so angepasst, dass die Eingänge des Web-IOs „gepollt“ werden, und in unserer globalen Struktur abgespeichert werden.

Bisher haben wir die Oberfläche im Programmfenster den Anforderungen entsprechend angezeigt und dabei nicht berücksichtigt, ob der Kommadostring an das Web-IO auch dazugeführt hat, die geforderten Outputs tatsächlich zu setzen.

Es wäre deshalb ratsam, die Programmoberfläche, basierend auf den Antworten des Web-IO, anzuzeigen. Erstrebenswert

wäre es natürlich, Funktionen und Prozeduren zu entwickeln, mit denen man durch Übergabe der hexadezimalen Zahl aus dem Antwortstring ein aktuelles Abbild des Schulungsboards auf der Programmoberfläche darstellt.

Es wird eine neue globale Variable hinzugefügt, die anzeigt, ob der Anforderungsschalter gedrückt wurde:

```
BOOLEAN g_bRequest=FALSE;
```

Zudem wird *g_iApplicationStep* mit 0 initialisiert, was anzeigen soll, dass die Grünphase der Autos besteht, und auf die Anforderung der Fußgänger reagiert werden kann.

```
int g_iApplicationStep=0;
```

Die OnConnect-Methode wandeln wir so ab:

```
void CAmpelDlg::OnConnect()
{
   UpdateData();

   g_cSocket.Create();
   if (g_cSocket.Connect(m_IP,80) == FALSE)
   {
      g_cSocket.Close();
      MessageBox(„Verbindung fehlgeschlagen“);
      return;
   }

   CString sCommand="GET /outputaccess?PW=&State=0011&";
   g_cSocket.Send(sCommand,sCommand.GetLength());

   SetTimer(1,100,NULL);

   m_Connect.EnableWindow(FALSE);
   m_Disconnect.EnableWindow(TRUE);
}
```

Nachdem das Socket verbunden ist, wird sofort ein String an das Web-IO gesendet, der die Fahrzeugampeln auf grün, und die Fußgängerampeln auf rot setzt. Danach startet der Timer, der die Eingänge des Geräts pollen soll (nicht aber der Timer, der für die Ampelschaltung selber zuständig ist).

Die Behandlungsroutine für den 2. Timer (der für die Ampelschaltung zu ständig ist) vereinfachen wir so:

```
if (nIDEvent==2)
{
    CString sCommand;
    if (g_iApplicationStep==1)
    {
        if (g_bRequest)
        {
            SetTimer(2,g_iTimerIntervall / 10,NULL);
            sCommand="GET /outputaccess?PW=&State=0012&";
        }else{
            KillTimer(2);
            g_iApplicationStep=0;
            return;
        }
    }
```

Das Programm befindet sich hier beim Beginn der Gelbphase für die Autos, also direkt nach der Grünphase. Wenn der Anforderungsschalter bis hierhin gedrückt wurde, schalten die Ampeln auf gelb. Wurde der Schalter nicht gedrückt, wird die Ampel in den Bereitschaftsmodus (*g_iApplicationStep=0*) gesetzt und der Timer beendet. Die Ampel bleibt auf grün.

Die weiteren Ampelphasen:

```
    if (g_iApplicationStep==2)
    {
        sCommand="GET /outputaccess?PW=&State=0014&";
    }
    if (g_iApplicationStep==3)
    {
        SetTimer(2,g_iTimerIntervall,NULL);
```

```
            sCommand="GET /outputaccess?PW=&State=000C&";
        }
        if (g_iApplicationStep==4)
        {
            SetTimer(2,g_iTimerIntervall / 10,NULL);
            sCommand="GET /outputaccess?PW=&State=0014&";
        }
        if (g_iApplicationStep==5)
        {
            sCommand="GET /outputaccess?PW=&State=0016&";
        }
        if (g_iApplicationStep==6)
        {
            SetTimer(2,g_iTimerIntervall,NULL);
            sCommand="GET /outputaccess?PW=&State=0011&";
            g_bRequest=FALSE;
        }

        g_cSocket.Send(sCommand,sCommand.GetLength());

        g_iApplicationStep++;
        if (g_iApplicationStep==7) g_iApplicationStep=1;
}
```

Wenn die letzte Ampelphase, die Grünphase für die Autos, erreicht ist, wird die Variable für den Anforderungsschalter auf 0 gesetzt. Wird die Variable hier zurückgesetzt, bedeutet das, dass ein Drücken auf den Anforderungsschalter, während der Grünphase der Fußgänger, keine Reaktion bewirkt, was sinnvoll ist. Lediglich während der Grünphase für die Autos soll auf den Anforderungsschalter reagiert werden.

In der *OnReceive* Methode wird folgender Code eingeführt, um auf die Daten vom Web-IO zu reagieren:

```
    if (sData.Left(7)=="output;")
    {
        sscanf(sData.Right(4),"%x",&state);

        g_sStatus.CarLightWest.bGreen=(state & 0x0001);
        g_sStatus.CarLightEast.bGreen=(state & 0x0001);
```

```
        g_sStatus.CarLightWest.bYellow=(state & 0x0002);
        g_sStatus.CarLightEast.bYellow=(state & 0x0002);

        g_sStatus.CarLightWest.bRed=(state & 0x0004);
        g_sStatus.CarLightEast.bRed=(state & 0x0004);

        g_sStatus.WalkerLightNorthEast.bGreen=(state & 0x0008);
        g_sStatus.WalkerLightSouthEast.bGreen=(state & 0x0008);
        g_sStatus.WalkerLightNorthWest.bGreen=(state & 0x0008);
        g_sStatus.WalkerLightSouthWest.bGreen=(state & 0x0008);

        g_sStatus.WalkerLightNorthEast.bRed=(state & 0x0010);
        g_sStatus.WalkerLightSouthEast.bRed=(state & 0x0010);
        g_sStatus.WalkerLightNorthWest.bRed=(state & 0x0010);
        g_sStatus.WalkerLightSouthWest.bRed=(state & 0x0010);

        Invalidate(FALSE);
    }
```

Wenn die ersten 7 Buchstaben das Wort „output;“ bilden, handelt es sich um Daten vom Web-IO, die den Status der Ausgänge des Boards bezeichnen. Die Zeichenkette wird in einen Zahlenwert umgewandelt, wie bei den Daten für die Eingänge im Beispiel 4.

```
    if (g_sStatus.WalkerLightNorthEast.bRequest ||
g_sStatus.WalkerLightNorthWest.bRequest)
    {
        g_bRequest=TRUE;
        if (g_iApplicationStep==0)
        {
            SetTimer(2,g_iTimerIntervall / 10,NULL);
            g_iApplicationStep=1;
        }
    }
```

Wenn nun entweder der Anforderungsschalter im Osten, oder der im Westen betätigt wurde, wird die entsprechende globale Variable auf TRUE gesetzt. Steht nun *g_iApplicationStep* auf 0, ist die Ampel im Ruhemodus, und

der Timer läuft nicht. In diesem Falle muss der Timer gestartet werden und es beginnt die Gelbphase der Autos. Wird der Anforderungsschalter während der regulären Grünphase der Autos betätigt, wird lediglich die Variable auf *TRUE* gesetzt, und nach der Grünphase wird im Timer automatisch mit der Gelbphase weitergemacht.

5 Anhang

- Kommandostrings

5.1 Socketprogrammierung mit Kommandosstrings

Um das Web-IO 12xDigital aus einfachen Anwendungs-programmen anzusprechen, ist ein direkter Zugriff über TCP oder UDP-Sockets möglich.

Der Datenaustausch zwischen PC und Web-IO geschieht dabei durch die Übergabe von einfachen Kommandostrings:

Kommandos	Parameter	Beschreibung
`GET /inputx`	`?PW=password&` An Stelle von **password,** muss das Administrator- oder Configpasswort eingesetzt werden. Wurde kein Passwort vergeben, wird "PW=&" angegeben. ***(gilt für alle Kommandos!)***	Anforderung des Inputstatus **x** kann ein Wert zwischen 0-11 sein und gibt den Input an. Die Rückmeldung des Web-IO ist ein String beginnend mit "input**x**;" gefolgt vom Input-Status: ON = Signal am Input und OFF = kein Signal am Input Wird **x** komplett weggelassen, gibt das Web-IO ein den Input-Signalen entsprechenden Bitmuster in hexadezimaler Schreibweise zurück.
`GET /counterx`	`?PW=password&`	Anforderung des Counterwertes **x** kann ein Wert zwischen 0-11 sein und gibt den Input an. Die Rückmeldung des Web-IO ist ein String beginnend mit "counter**x**;". Der Zählerstand des gewählten Counters wird in dezimaler Schreibweise angefügt.
`GET /outputx`	`?PW=password&`	Anforderung des Outputstatus **x** kann ein Wert zwischen 0-11 sein und gibt den Output an. Die Rückmeldung des Web-IO ist ein String beginnend mit "output**x**;" gefolgt vom Output-Status: ON = Signal am Output und OFF = kein Signal am Output Wird **x** komplett weggelassen, gibt das Web-IO ein den Output-Signalen entsprechendes Bitmuster in hexadezimaler Schreibweise zurück.

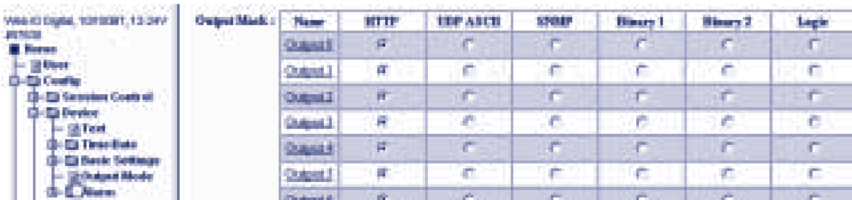
`GET /outputaccessx`	`?PW=password&`
	`State=ON/OFF/XXXX&`
	ON: Output = 1,
	OFF: Output = 0,
	XXXX: Hex-Wert zwischen 0000 und 0FFF entsprechend dem Ausgangsbitmuster.
		Setzen eines oder mehrerer Outputs
		x kann ein Wert zwischen 0-11 sein und gibt den Output an, der gesetzt werden soll.
		Die Rückmeldung des Web-IO ist ein String beginnend mit "output;" gefolgt von einem den Output-Signalen entsprechenden Bitmuster in hexadezimaler Schreibweise.

`GET /counterclearx`	`?PW=password&`	Setzt den Zählerstand eines Counters auf 0 zurück. **x** kann ein Wert zwischen 0-11 sein und gibt den Input an, dessen Counter zurückgesetzt werden soll. Die Rückmeldung des Web-IO ist ein String beginnend mit "counter**x**;". Der neue Zählerstand des gewählten Counters wird in dezimaler Schreibweise angefügt. Wird **x** nicht angegeben, werden alle 12 Counter auf 0 zurückgesetzt. Es erfolgt in diesem Fall keine Rückmeldung.
`GET /errorclear`	`?PW=password&`	Löscht den Fehler-Report im Web-IO. Siehe Kapitel *Diagnose und Test*

5.1.1 TCP Kommunikation

Vorbereitend muss festgelegt werden, welche der 12 Outputs für den Zugriff über Kommandostrings zur Verfügung stehen sollen.

Wählen Sie im Navigationsbaum *Config >> Device >> Output Mode*



Notwendige Zugriffsrechte: *Administrator*

Alle verwendeten Outputs müssen für *HTTP* aktiviert sein (Werkseinstellung)

Sollten Sie die Output Modi verändert haben, klicken Sie auf den *Zwischenspeichern-Button* und anschließend auf den Link *alle zwischengespeicherten Einstellungen speichern und aktivieren*. Durch Mouse-Klick auf den *Speichern-Button* werden die Einstellungen übernommen.

Das Web-IO arbeitet nun als TCP-Server, woraus sich unabhängig von der verwendeten Programmiersprache eine Dreiteilung des Programmablaufes ergibt.

1. Aufbau der TCP-Verbindung
Die Anwendung arbeitet in jedem Fall als Client und baut somit die TCP-Verbindung auf.

Das Socket-Interface des Web-IO ist in weiten Teilen an das HTTP-Protokoll angelehnt. Daraus ergibt sich, dass der **Listenport** fest auf **TCP-Port 80** eingestellt ist.

Zum Schutz vor blockierenden TCP-Verbindungen hat das Web-IO auf Port 80 einen Timer eingesetzt, der

die Verbindung 30 Sekunden nach Verbindungsaufbau automatisch wieder trennt, wenn keine syntaktisch korrekten Kommandos empfangen wurden. Nach erfolgreichem Verbindungsaufbau, sollte die Anwendung also immer z.B. eine Abfrage der Outputs an das Web-IO senden.

2. Kommunikation zwischen Web-IO und Anwendung

Die normale Kommunikation findet im Pollingbetrieb statt. Das bedeutet: die Client-Anwendung fordert mit Hilfe der Kommandostrings die gewünschten Werte an, bzw. setzt die Outputs.

Beispiel 1: Abfrage des Status von Input 3, an dem ein Signal von +12 anliegt. Es wurde kein Administratorpasswort für das Web-IO vergeben.

Die Anwendung sendet folgenden String zum Web-IO:

```
GET /input3?PW=&
```

Das Web-IO sendet daraufhin zurück:

```
input3;ON
```

Beispiel 2: Setzen der Outputs 3,4,7,9. Binär ergibt das 0000 0010 1001 1000, was wiederum 0298 hexadezimal ergibt.

Das Administratorpasswort für das Web-IO lautet: „blau“

Die Anwendung sendet folgenden String zum Web-IO:

```
GET /outputaccess?PW=blau&State=0298&
```

Das Web-IO sendet daraufhin zurück:

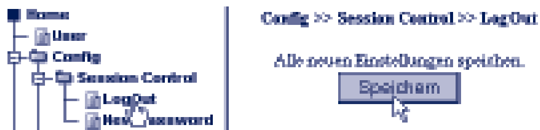
```
output;0298
```

Das Web-IO arbeitet mit Null-terminierten Strings! Das bedeutet, alles was das Web-IO zur Anwendung sendet, endet mit einem 0-Character

Enable
Aktivieren Sie im Bereich *Enable* das Feld *Send input to pending TCP connections*

Übergeben Sie die gewählten Alarmbedingungen jeweils durch Mouse-Klick auf den *Zwischenspeichern-* Button an das Web-IO.

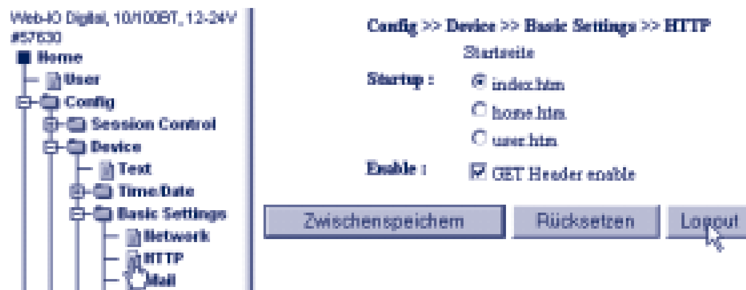
Wenn alle Alarmbedingungen festgelegt wurden, aktivieren Sie die neuen Einstellungen über *Config >> Session Control >> LogOut*



Nach einem Mouse-Klick auf das *Speichern*-Button wird das Web-IO nun mit den aktuellen Parametern neu gestartet.

Einen Header mitsenden lassen
Das Web-IO kann bei Bedarf dem gesendet String ein Header voranstellen, der die IP-Adresse und den Namen enthält.

Um den Header freizuschalten, wählen Sie im Navigationsbaum *Config >> Basic Settings >> HTTP* und markieren Sie *GET Header enable*.



Durch Klick auf *Logout* und *Speichern* wird das Versenden des Headers aktiviert.

3. Schließen der TCP-Verbindung

Hier weicht der Web-IO in seinem Verhalten vom HTTP-Protokoll ab. Im Gegensatz zum HTTP-Protokoll ist es immer die Client-Anwendung, die eine Verbindung schließt.

Ausnahme: Bei Empfang eines fehlerhaften Kommandostrings beendet das Web-IO sofort die Verbindung

Wie geht es weiter?

Diese Anleitung zum Web-IO Schulungskoffer und die darin gezeigten Programmierbeispiele bieten sicher einen guten Einstieg in den Umgang mit dem Web-IO.

Wir werden dieses Dokument ständig weiterpflegen und um neue Beispiele erweitern. Allen, die auf dem Laufenden bleiben möchten, empfehlen wir von Zeit zu Zeit auf unsere Webseiten **www.wut.de** zu schauen, wo im Bereich *Produkte und Downloads* immer die aktuelle Version dieser Anleitung zum Download bereit liegt.

Tragen Sie im Bereich *Aktuell >> Stehts aktuelle Informationen per E-Mail* Ihre Adresse ein und wir informieren Sie über alle wichtigen Neuigkeiten.

Viel Spaß mit dem Schulungskoffer wünscht

Ihr W&T Team