Hygrosens I/O-Karten in eigene Anwendungen einbinden

Gemeinsame Voraussetzungen für alle 3 Karten :

Alle drei Karten besitzen als USB-Controller des Cypress-Controller AN2131. Folgende Dateien müssen installiert werden :

EZUSB.sys	- Hardware-Treiber für USB-Controller	..\windows\system32\drivers
Borlndmm.dll	- DLL zur Verarbeitung langer Strings	..\windows

Für jede Karte muß eine spezifische Treiber-DLL installiert werden. Auf deren Funktionalität wird weiter unten genauer eingegangen. Diese DLL's können zweckmäßigerweise im Verzeichnis der jeweiligen Anwendung installiert. Im folgenden die Zugehörigkeiten der DLL's zu den einzelnen Karten :

REL8_IN8	8xIN / 8xRel OUT	USB8IO.dll
OUT32_IN32	32xIN / 32xOUT	USB32IO.dll
OUT8_IN8_AD4_DA	8xIn / 8xOut / 4-Kanal AD / 1 DA	USB8IOADD.dll

Besonderheiten : Da alle Karten ein und denselben Controller besitzen, mußte eine Möglichkeit gefunden werden, diese voneinander zu unterscheiden. Das passiert mit der Device-ID, welche in einem kleinen EEPROM auf der Karte gespeichert ist und beim ‚Anstecken' der Karte an den USB ausgelesen wird. Diese Device-ID setzt sich aus einer Konstanten für den Kartentyp und die Kartennummer zusammen. Ausgeliefert werden alle Karten mit der Nummer 0. Es besteht die Möglichkeit insgesamt 32 Karten anzuschließen. Im Lieferumfang befindet sich ein kleines Tool zum Einstellen einer anderen Kartennummer (SetCardNumber.exe) Zum Umprogrammieren darf sich nur eine Karte am USB befinden. Im folgenden die Zusammensetzung der Device-ID in Abhängigkeit vom Kartentyp :

REL8_IN8	konstant 256	+ Kartennummer
OUT32_IN32	konstant 512	+ Kartennummer
OUT8_IN8_AD4_DA	konstant 1024	+ Kartennummer

Beispiel :

Karte Nr. 0 der REL8_IN8 besitzt die Device-ID	256 + 0	= 256
Karte Nr. 3 der OUT8_IN8_AD4_DA besitzt die Device-ID	1024 + 3	= 1027

Verfahrensweise zum Ansprechen der Karten aus dem Anwendungsprogramm

Das ‚Öffnen' und ‚Schließen' eines ‚Devices' (Karte) ist bei allen Kartentypen gleich und wird im folgenden auch für alle 3 Karten gemeinsam erläutert. Eine Variable wird bei der Programmierung immer wieder eine Rolle spielen, die ‚DeviceNumber' ! Sie ist vom Typ Byte und vorzeichenlos. Beim Anschließen der Karten an den Bus, bekommen diese aufsteigend ab 0 eine Nummer zugeordnet, die DeviceNumber. Sind beim Rechnerstart jedoch schon Karten angeschlossen, ist diese Nummer nicht vorhersagbar. Die richtige DeviceNumber für eine bestimmte Karte wird beim ‚Öffnen' der Karte über die Funktion ‚OpenDevice' ermittelt und hat nach dem ‚Schließen' mit der Funktion ‚CloseDevice' ihre Gültigkeit verloren. Im folgenden ein kleines Pascal-Beispiel zum Suchen und in Betrieb nehmen der REL8_IN8 mit der Kartennummer 8 :

Die Device-ID dieser Karte muß den Wert 256 + 8 = 264 haben . In einer Schleife werden von Null beginnend alle Karten ‚Geöffnet' und deren Decice-ID's mit der geforderten verglichen. Bei Nichtübereinstimmung wird das Device sofort wieder geschlossen und der nächste Schleifendurchlauf gestartet.

Zum Öffnen des Device (der Karte) wird die Funktion **‚OpenDevice'** verwendet. Diese erwartet als Übergabeparameter die gewünschte DeviceNumber vom Typ Byte, den Treibernamen (EZUSB) vom Typ String und zwei leere Strings, die für die Verwendung der Karten bedeutungslos sind, jedoch mit übergeben werden müssen, da es sich nicht um optionale Parameter handelt. Zurück gibt die Funktion bei Erfolg die DeviceID des angesprochenen Device. Ob die Funktion erfolgreich war, wird mit der Funktion **‚GetDeviceError'** ermittelt. Diese erwartet als Übergabeparameter ebenfalls wieder die Device-Nummer und gibt den letzten aufgetretenen Fehler in Form eines 4 Byte breiten vorzeichenlosen Wertes zurück. Ist kein Fehler aufgetreten wird 0 zurück gegeben. Die möglichen Fehlerkonstanten sind zusammen mit den Funktionsdeklarationen im Anhang beschrieben.

```
Var DeviceError : DWORD;             // bekommt von GetDeviceError letzten Fehler zugewiesen
    DeviceID, CardID : WORD;         // 2 Byte vorzeichenlos
    DeviceNumber, I  : Byte;

DeviceNumber := 255;                 // ungültige DeviceNumber als Initialisierung
CardID := 256 + 8                    // Kartenkennung + Kartennummer

For I := 0 to 31 do Begin
  DeviceID := OpenDevice ( I, 'EZUSB', '', '' );
  If Err_No = GetDeviceError( I ) Then Begin
    If CardID = DeviceID Then Begin
      DeviceNumber := I;             // DeviceNumber der gesuchten Karte wird zugewiesen
      Break;
    End Else
      CloseDevice( I );              // Device wieder schließen
  End Else
    CloseDevice( I );                // Device wieder schließen
End;

// Wird die gesuchte Karte gefunden, so wird die Schleife verlassen (Break) und
// die Schleifenvariable (welche der DeviceNumber entspricht) wird der entsprechen-
// den Variablen zugewiesen. Besitzt die Variable 'DeviceNumber' immer noch den Wert
// 255 so befindet sich die gesuchte Karte nicht am Bus
```

Nachdem die gesuchte Karte gefunden wurde (angenommen wird DeviceNumber = 3) muß diese nun aktiviert werden, das heißt, es wird ein lauffähiges Programm in den, auf der Karte befindlichen, USB-Controller (AN2131) geladen. Dieses geschieht mit der Funktion **'SetDeviceEnabled'** . Diese benötigt als ersten Parameter wie fast alle Funktionen die eben ermittelte Device-Nummer. Der zweite Parameter ist der gewünschte Controller-Modus (die gewünschte Funktionalität). Für die IO-Karten ist das immer der Modus 1. Hier wieder einer paar Pascal-Zeilen zur besseren Anschaulichkeit :

```
SetDeviceEnabled ( DeviceNumber, 1);               // Firmeware für Controllermodus 1 laden
If Err_No <> GetDeviceError( DeviceNumber ) Then   // auf Fehler prüfen
  CloseDevice ( DeviceNumber );                    // gegebenfalls Device (Karte) schließen
```

War dieser Funktionsaufruf erfolgreich (GetDeviceError gibt 0 zurück) kann die Karte nun in Betrieb gehen. Die für jede Karte spezifischen Funktionen werden weiter unten beschrieben. Ich möchte im folgenden noch einmal auf mögliche Fehler und die Möglichkeiten zur deren Beseitigung eingehen.

Mögliche Fehler sind :

- der USB-Controller (AN2131) ist außer Tritt geraten
- die Betriebsspannung ist kurzzeitig ausgefallen
- die Karte wurde kurzzeitig vom Bus getrennt

Ob ein derartiger Fehler aufgetreten ist, kann wieder mit der Funktion 'GetDeviceError' ermittelt werden. Folgende Fehlerkonstanten deuten auf einen derartigen Fehler hin :

```
Err_DeviceNotEnabled  : DWord = $000008;   // dezimal =   8
Err_DeviceIOCtl       : DWord = $000010;   // dezimal = 256
```

In so einem Fall hilft es nur, das Device über **'CloseDevice' sofort** zu schliessen und anschließend die Karte, wie oben beschrieben, wieder zu suchen. Wurde der Treiber auf Grund einer falschen Bedienung (ziehen des Steckers während des Betriebes der Karten) vom System bereits entladen, so muß nach erfolgtem Schliessen des Devices das Suchen der Karte unter Umständen mehrmals wiederholt werden (bis der Treiber 'EZUSB.sys' wieder durch das System geladen wurde. Das ist durch einen Timer recht einfach zu bewerkstelligen.

Hier wieder ein paar Code-Zeilen in Pascal :

```
DeviceError := GetDeviceError( DeviceNumber );

If (DeviceError = Err_DeviceIOCtl) Or (DeviceError = Err_DeviceNotEnabled) Then
  CloseDevice( DeviceNumber);
  ErrTimer.Enabled := True;

// So könnte das Timer-Event aussehen

procedure TForm1. ErrTimerTimer(Sender: TObject);

begin

DeviceNumber := 255;                 // ungültige DeviceNumber als Initialisierung

For I := 0 to 31 do Begin
```

```
  DeviceID := OpenDevice ( I, 'EZUSB', '', '' );
  If Err_No = GetDeviceError( I ) Then Begin
    If CardID = DeviceID Then Begin
      DeviceNumber := I;             // DeviceNumber der gesuchten Karte wird zugewiesen
      Break;
    End Else
      CloseDevice( I );              // Device wieder schliessen
  End Else
    CloseDevice( I );                // Device wieder schliessen
End;

end;
```

In den meisten Fällen ist die Karte so wieder in Tritt zu bekommen. Sollte dies einmal doch nicht der Fall sein, ist die Karte vom Bus zu trennen und nach einiger Zeit wieder mit diesem zu verbinden. Genauso sollte dann das Anwenderprogramm geschlossen werden und wieder neu gestartet werden.

Soll aus einer eigenen Anwendung heraus die Kartennummer geändert werden, so ist dazu die Methode **‚SetBootEEP'** aufzurufen. Zuvor muß jedoch durch Stecken des Jumpers auf der Platine der Schreibschutz des EEPROM aufgehoben werde. Die Methode benötigt, wie üblich, die Device-Nummer, welche ja in der entsprechenden Variablen gespeichert ist und als zweiten Parameter die Device-ID. Die Device-ID setzt sich, wie weiter oben bereits beschrieben, aus einer Konstanten für den Kartentyp und der gewünschten Kartennummer zusammen. Diese Device-ID sowie weitere Konstanten werden dann in das EEPROM der Karte geschrieben. War der Schreibvorgang erfolgreich, muß die Funktion **‚GetDeviceError()'** 0 zurückgeben.
Da die Änderung erst nach einer erneuten Enumeration des Controllers (einem erneuten Anstecken an den Bus) wirksam wird muß die Karte vom Bus getrennt werden. Durch Ziehen des Jumpers wird der Schreibschutz wieder aktiviert. Nach Anstecken an den USB wird die Karte nun unter der neuen Nummer gefunden.

Die beschriebenen Vorgänge zum Identifizieren, Öffnen, Schliessen und Aktivieren einer Karte sowie zum Ändern der Kartennummer sind für alle 3 Karten identisch. Hier noch einmal die verwendeten Methoden (Funktionen, Prozeduren)

OpenDevice
SetDeviceEnabled
CloseDevice
GetDeviceError
SetBootEEP

Im folgenden wird für jede Karte gesondert beschrieben wie Ausgänge gesetzt, Eingänge gelesen und der ADC der kombinierten OUT8_IN8_AD4_DA-Karte konfiguriert und ausgelesen wird. Alle notwendigen Methoden der DLL sind genauso in der AN21XX.DLL vorhanden und haben deshalb auch die entsprechenden Namen.

Die Karte REL8_IN8

- acht Relais als Ausgänge und 8 digitale Eingänge -

Diese Karte ist am einfachsten anzusprechen und benötigt nur eine Funktion der DLL zum ändern der Ausgänge und einlesen der Eingänge. Dies ist die Funktion :

PB_ReadByte

Die Funktion benötigt als ersten Parameter wieder die Device-Nummer des Controllers - weiter oben ausführlich beschrieben – und als zweiten Wert ein Byte welches den Schaltstellungen der einzelnen Relais entspricht. Dabei entspricht das niederwertige Bit dem Relais 0 und das höchstwertige Bit dem Relais 7. Der Rückgabewert der Funktion ist ein Byte welches die Zustände der Ausgänge repräsentiert. Hier ein kurzes Beispiel in Pascal :

```
Var OUTS, INS : Byte;

// Initialisieren der Variablen OUTS
// Der Wert 129 bedeutet das Bit0 und Bit7 der Variablen gesetzt sind, das heißt :
// nach dem erfolgreichen Aufruf der Funktion ist Relais0 und Relais7 eingeschaltet

OUTS := 129;
INS := PB_ReadByte ( DeviceNumber, OUT );

// Fehlerbehandlung
If Err_No <> GetDeviceError( DeviceNumber ) Then Begin
                        .
                        .
                        .
End;

// Nach erfolgreichem Aufruf der Funktion enthält die Variable INS nun den aktuellen
```

```
// Zustand der Eingänge der Karte. Hat INS zum Beispiel den Wert 68, so bedeutet das,
// daß Eingang2 und Eingan6 High-Pegel führen und an allen anderen Eingängen Low-Pegel
// anliegt .
```

Damit ist der Betrieb der Karte bereits ausreichend erklärt. Wie oft die Anwendung nun die Methode **PB_ReadByte** aufruft muß der Anwender selbst entscheiden. Da Relais ohnehin recht große Schaltzeiten haben dürfte ein Intervall von 500 ms oftmals ausreichend sein.
Wichtig ist, in regelmäßigen Abständen durch die Funktion **GetDeviceError** eventuell aufgetretene Fehler abzufangen. Nur so ist gewährleistet, daß die richtigen Relais geschaltet haben und der zurückgegebene Eingangswert auch korrekt ist. Bei auftretenden Fehlern ist wie oben (bei den für alle Karten identischen Methoden) beschrieben zu verfahren.

Die Karte OUT32_IN32

- 32 digitale Ausgänge und 32 digitale Eingänge -

Etwas komplexer, jedoch auch wieder vollkommen unproblematisch, ist die Nutzung der OUT32_IN32-Karte. Sowohl Ausgänge, als auch Eingänge der Karte sind „gelatcht“, das heißt, sie werden in Registern nachdem schreiben oder lesen zwischen gespeichert. Hier bieten sich die Funktionen **„PB_ReadBlockInc“** und **„PB_ReadBlockInc“** der AN21XX.DLL an. Diese wurden auch in die Treiber-DLL (USB32IO.dll) der Karte übernommen. Hier nun kurz eine Erläuterung der zwei Funktionen :

Prinzipiell wird im IMode_1, welcher ja beim „Enabled“-Setzen der Karte mit der Funktion **„SetDeviceEnabled“** gewählt wird, an den Ports B und C ein bidirektionaler Datenbus mit einem 8Bit-Adressbus erzeugt. Die Funktionen schreiben bzw. lesen nun eine vom Programmierer gewählte Anzahl von Bytes und inkrementieren nach jedem geschriebenen bzw. gelesenen Byte den Adreßbus. Begonnen wird dabei mit der an die Funktion übergebenen Adresse. Die Daten müssen dabei vorher einem Byte-Array übergeben werden. Dieses Byte-Array wird dabei als Zeiger auf das ‚nullte‘ Element an die Funktionen übergeben. Das klingt recht kompliziert, ist in der Anwendung jedoch ausgesprochen einfach.

Benötigt werden zwei Byte-Arrays, eines für die Ausgänge mit 6 Elementen und eines für die Eingänge mit 4 Elementen. Im Testprogramm haben beide Arrays 6 Elemente, was aber auf die Funktion keinen Einfluß hat.

In das Array für die Ausgänge werden nun beim Element 0 beginnend die gewünschten Ausgangswerte eingegeben. Ein Element repräsentiert dabei jeweils 8 Ausgänge :

OutArray [0]	=	Ausgänge 0 ... 7
OutArray [1]	=	Ausgänge 8 ... 15
OutArray [2]	=	Ausgänge 16 ... 23
OutArray [3]	=	Ausgänge 24 ... 31

Die in den Elementen 4 und 5 enthaltenen Werte sind bedeutungslos. Es müssen jedoch 6 Schrfeibzugriffe durchgeführt werde. Die ersten vier zum Schreiben der gewünschten Ergebnisse in die Latches, der fünfte zum Weiterreichen aller Latches an die Ausgänge und der sechste Schreibzugriff zum Rücksetzen des auf der Karte befindlichen Watchdog-Timers. Ein Adreßdekoder erzeugt dabei in Verbindung mit dem /WR-Signal des Controllers die einzelnen /LE-Signale sowie das Resetsignal für den Watchdog.

Achtung !

Das an die Funktion übergebene Array muß mindestens soviel Elemente besitzen wie Daten geschrieben werden sollen ! Ist das nicht der Fall kommt es zu einem Fehler, da dann eventuell auf fremde Speicherbereiche zugegriffen wird .

Das Lesen der Eingangswerte erfolgt ähnlich wie das Schreiben der Ausgangswerte, nur daß hier vier Lesezugriffe für die vier Eingangs-Latches ausreichend sind. Die Zuordnung der Eingänge zu den Arrayelementen erfolgt analog den Ausgängen:

InArray [0]	=	Eingänge 0 ... 7
InArray [1]	=	Eingänge 8 ... 15
InArray [2]	=	Eingänge 16 ... 23
InArray [3]	=	Eingänge 24 ... 31

Diese Verfahrensweise soll nun wieder an einem Beispiel erklärt werden :

```
type TByteArray = array[0 .. 5] of Byte;

var
  InArray, OutArray : TByteArray

// Initialisieren der Variablen OutArray

OutArray[0] := 1;       // Ausgang 0 auf High-Pegel setzen – Ausgänge 1 bis 7 = Low
OutArray[1] := 64;      // Ausgang 14 auf High-Pegel setzen – Ausgänge 8 bis 13 und 15 = Low
OutArray[2] := 129;     // Ausgänge 16 und 23 auf High-Pegel setzen – Ausgänge 17 bis 22 = Low
OutArray[3] := 32;      // Ausgang 29 auf High-Pegel setzen
                        // Ausgänge 24 bis 28 und 30 + 31 = Low
```

```
// hier wird das Array zur Karte geschrieben
PB_WriteBlockInc(DevNumber, 0, OutArray[0], 6);     // Parameter in der Reihenfolge
                                                    // DevNumber wie oben beschrieben
                                                    // 0 = erstes Adreßbyte
                                                    // wird mit jedem Shreibzugriff inkrementiert
                                                    // OutArray[0] = Zeiger auf ‚nulltes' Element
                                                    // 6 Byte schreiben

// hier werden die Eingangswerte gelesen
PB_ReadBlockInc(DevNumber, 0, InArray[0], 4);       // Parameter in der Reihenfolge
                                                    // DevNumber wie oben beschrieben
                                                    // 0 = erstes Adreßbyte
                                                    // wird mit jedem Shreibzugriff inkrementiert
                                                    // InArray[0] = Zeiger auf ‚nulltes' Element
                                                    // vier Byte lesen

// Fehlerbehandlung
If Err_No <> GetDeviceError( DeviceNumber ) Then Begin
End;
```

In welchem Intervall diese Funktionen nun aufgerufen werden hängt von den jeweiligen Anforderungen ab, jedoch darf das Intervall nicht größer werden als das am Watchdog-Timer eingestellte Intervall. Ist das der Fall meint die Karte die Übertragung sei unterbrochen und geht in den Grundzustand.

Noch ein Hinweis für Visual Basic Programmierer :

Im Deklarationsmodul **USB32IO_dcl.bas** finden Sie auch die Deklaration dieser beiden Funktionen. Der Aufruf unterscheidet sich nicht von dem in Pascal.

```
Declare Sub PB_ReadBlockInc Lib "USB32IO.dll" (ByVal DevNumber As Byte, _
                                               ByVal Addr As Byte, _
                                               BArray As Byte, _
                                               ByVal ByteCnt As Long)
```

Das Array (hier Barray genannt) wird, da das Schlüsselwort ‚by Val' fehlt, by Referenz, also als Zeiger übergeben. Dabei muß beim Zugriff auf die Karte ein Zeiger auf das nullte Element übergeben werden.

Beispiel :

```
Call PB_ReadBlockInc( DevNumber, 0, InArray(0), 4 )
```

Die Karte OUT8_IN8_AD4_DA

- 8 digitale Ausgänge, 8 digitale Eingänge, ein 4 Kanal 8Bit-ADC und ein 8Bit-DAC -

Diese Karte hat neben 8 digitalen Ausgängen und 8 digitalen Eingängen einen komplexen kombinierten 8Bit ADC/DAC, den PCF8591. Dieser besitzt 4 analoge Eingänge und 1 analogen Ausgang. Über den I2C-Bus steht der Wandler mit dem Controller in Verbindung. Eine Besonderheit des ADC sind die 4 konfigurierbaren Eingänge des ADC. Folgende Konfigurationsmöglichkeiten sind vorhanden :

0 - vier nichtinvertierende Eingänge gegen GND

1 - Eingang0, Eingang1 und Eingang2 nichtinvertierende Eingänge gegen Eingang3

2 - Eingang0 und Eingang1 als nichtinvertierende Eingänge gegen GND
Eingang2 und Eingang3 bilden den dritten analogen Kanal, wobei Eingang2 der nichtinvertierende Eingang und Eingang3 der invertierende Eingang eines OPV bilden.

3 - Eingan0 und Eingang1 bilden den einen Analogkanal, Eingang2 und Eingang3 den Anderen. Dabei ist Eingang0 der nichtinvertierenden Eingang und Eingang1 der invertierende Eingang des ersten analogen Kanals. Eingang2 und Eingang3 verhalten sich für den zweiten analogen Kanal ebenso

Für die Wichtung der Eingänge ist die Referenzspannung des ADC ausschlaggebend. Beim Lesen der Eingänge ist wichtig daß der erste Wert immer verworfen wird. Das hat seine Begründung darin, daß mit dem ersten Lesezugriff des I2C-Bus die Messung erst angestoßen wird. Das zweite gelesene Byte ist dann der Wert des aktuellen Kanals. Eine weitere Konfigurationsmöglichkeit ist, das die Kanäle bei jedem Lesezugriff weiter geschaltet werden. Auch der analoge Ausgang kann zu- oder abgeschaltet werden.

All diese Konfigurationsmöglichkeiten werden in einem Byte kodiert welches vor Beginn der Messung(en) an den ADC gesandt werden muß. Ist der DAC zugeschaltet müssen 2 Bytes zum PCF8591 gesandt werden. Das erste Byte ist dabei das Konfgurations-Byte, das zweite enthält den zu setzenden Wert des analogen Ausganges. Die absolute Spannung ist auch hier von der anliegenden Referenzspannung abhängig.

Der Aufbau des Konfigurationsbytes :

```
0 X X X 0 X X X
  | |_|   | |_|_ Kanalnummer = 00 - Kanal 0  ( je nach Konfiguration der Eingänge existieren zwischen zwei
  | |     |                   01 - Kanal 1    und vier analoge Kanälen - die Konfiguration der Eingänge
  | |     |                   10 - Kanal 2    wurde weiter oben bereits beschrieben )
  | |     |                   11 - Kanal 3
  | |     |
  | |     |_____ Automatische Inkrementation der analogen Kanäle
  | |                          0 - Aus
  | |                          1 - Ein
  | |
  | |___________ Programmierung der Eingänge
  |                          00 - vier einzelne Eingänge ( gegen GND )
  |                          01 - drei Differenzeingänge ( +IN0, +IN1, +IN2 / gegen -IN3
  |                          10 - zwei einzelne Eingänge + ein Differenzeingang ( +IN2 / -IN3 )
  |                          11 - zwei Differenzeingänge ( +IN0 gegen -IN1 / +IN2 gegen -IN3 )
  |
  |_____________ Analogausgang aktiv
                             0 - inaktiv (hochohmig)
                             1 - aktiv
```

Weiter unten wird die Programmierung des PCF8591 an einem Beispiel erklärt .

Das Setzen der digitalen Aus- und Lesen der digitalen Eingänge ist identisch dem der REL8_IN8-Karte und wird über eine Funktion des Parallelbus des USB-Controller realisiert :

PB_ReadByte

Die Funktion benötigt als ersten Parameter wieder die Device-Nummer des Controllers - weiter oben ausführlich beschrieben – und als zweiten Wert ein Byte welches den Pegeln der einzelnen Ausgänge entspricht. Dabei entspricht das niederwertige Bit dem digitalen Ausgang0 und das höchstwertige Bit dem digitalen Ausgang7. Der Rückgabewert der Funktion ist ein Byte welches die Zustände der Ausgänge repräsentiert. Hier ein kurzes Beispiel in Pascal :

```
Var OUTS, INS : Byte;

// Initialisieren der Variablen OUTS
// Der Wert 129 bedeutet das Bit0 und Bit7 der Variablen gesetzt sind, das heißt :
// nach dem erfolgreichen Aufruf der Funktion ist OUT0 und OUT7 auf High-Pegel

OUTS := 129;
INS := PB_ReadByte ( DeviceNumber, OUT );

// Fehlerbehandlung
If Err_No <> GetDeviceError( DeviceNumber ) Then Begin
                        .
                        .
                        .
End;

// Nach erfolgreichem Aufruf der Funktion enthält die Variable INS nun den aktuellen
// Zustand der digitalen Eingänge der Karte. Hat INS zum Beispiel den Wert 68, so bedeutet das,
// daß Eingang2 und Eingan6 High-Pegel führen und an allen anderen Eingängen Low-Pegel
// anliegt .
```

Im Folgenden nun die Programmierung (Senden der Konfiguration, Abfragen der analogen Eingänge) ADC/DAC-PCF8591 . Zum Datentransfer vom und zum PCF8591 werden zwei Funktionen des I2C-Bus des Controllers genutzt :

I2C_WriteBlock und **I2C_ReadBlock**

Über diese Funktionen können Datenblöcke bis maximal 65535 Byte vom I2C gelesen bzw. zu diesem geschrieben werden. Für dem PCF8591 ist eine maximale Blockgröße von 5 Elementen notwendig. Diese Byte-Arrays müssen dabei als Zeiger auf das ‚nullte' Element an die Funktionen übergeben. Das klingt recht kompliziert, ist in der Anwendung jedoch ausgesprochen einfach. Zum Senden der Konfiguration muß das Array eine Größe von zwei Elementen haben (das Konfigurationsbyte und der Ausgangswert des DAC) , zum Lesen der Daten maximal 5 Elemente (eine Dummy-Byte zum Anstoßen der Wandlung und maximal 4 Byte zum Lesen der Analogkanäle) .

Hier das Einstellen der Konfiguration :

- DAC aktiv / maximale Ausgangsspannung
- ADC 4 nichtinvertierende Eingänge
- aktiver Kanal IN0
- automatische Kanalweiterschaltung beim Abfragen der Analogkanäle.

Um den Source-Code leserlich zu gestalten werden für alle Bits Konstanten deklariert :

```
const

// Konstanten für ADC/DAC-Konfigurationsregister
  IN0                : Byte =   0;            // Analogeingang 0 aktiv
  IN1                : Byte =   1;            // Analogeingang 1 aktiv
  IN2                : Byte =   2;            // Analogeingang 2 aktiv
  IN3                : Byte =   3;            // Analogeingang 3 aktiv

  AutoIncOff         : Byte =   0;            // automatische Kanalweiterschaltung – Aus (betrifft Analogkanäle)
  AutoIncOn          : Byte =   4;            // automatische Kanalweiterschaltung – Ein (betrifft Analogkanäle)

  FourSingleIN       : Byte =   0;            // vier Einzeleingänge
  ThreeDiffIN        : Byte =  16;            // 3 Differenzeingänge gegen IN3
  TwoSingleOneDiffIN : Byte =  32;            // zwei Einzeleingänge / ein Differenzeingang aus IN2 und IN3
  TwoDiffIN          : Byte =  48;            // zwei Differenzeingänge aus IN0 und IN1 bzw. IN2 und IN3

  DAC_Disabled       : Byte =   0;            // Analogausgang inaktiv (hochohmig)
  DAC_Enabled        : Byte =  64;            // Analogausgang aktiv

SlaveAddr          : Byte = 144;            // auf der Karte fest verdrahtede Slaveadresse des PCF8591

type TCfgArray = array[0 .. 1] of Byte;       // Byte-Array aus 2 Elementen
type TInArray  = array[0 .. 4] of Byte;       // Byte-Array aus 5 Elementen

Var
  DACVal : Byte;
  CfgArray : TCfgArray;
  InArray : TInArray;

// Konfigurieren des PCF8591 und anschließendes Lesen aller 4 Kanäle

CfgArray[0] := DAC_Enabled + FourSingleIN + AutIncOn + IN0;
CfgArray[1] := 255;                              // Analogausgang maximale UOUT

// Schreiben der Konfigurationsbytes
I2C_WriteBlock(DevNumber, SlaveAddr, CfgArray[0], 2); // Parameter in der Reihenfolge
                                                      // DeviceNumber wie oben beschrieben
                                                      // Slaveadresse des PCF8591 = 144
                                                      // auf der Karte fest eingestellt
                                                      // CfgArray[0] = Zeiger auf ‚nulltes' Element
                                                      // zwei Byte lesen

// Lesen der vier Analogkanäle, welche bei der gewählten Konfiguration
// den nicht invertierenden Eingängen des PCF8591 entsprechen
I2C_ReadByte(DevNumber, SlaveAddr, InArray[0], 5);    // Parameter in der Reihenfolge
                                                      // DeviceNumber wie oben beschrieben
                                                      // Slaveadresse des PCF8591 = 144
                                                      // auf der Karte fest eingestellt
                                                      // InArray[0] = Zeiger auf ‚nulltes' Element
                                                      // fünf Byte lesen

// Fehlerbehandlung
If Err_No <> GetDeviceError( DeviceNumber ) Then Begin
                      .
                      .
                      .
End;

// Nach erfolgreichem Aufruf der Funktion enthält InArray[0] das Dummybyte,
// InArray[1] ... InArray[4] enthalten in aufsteigender Reihenfolge die Eingänge
// IN0 ... IN3
```

In welchem Intervall die Eingänge des PCF8591 gelesen werden hängt von der jeweiligen Anwendung ab. Das Schreiben der Konfigurationsdaten zum PCF8591 ist beim Start der Anwendung notwendig und anschließend nur bei irgendwelchen Änderung der Konfiguration, wie zum Beispiel das Ändern des DAC-Wertes oder das Ändern der Eingangskanäle !

Noch ein Hinweis für Visual Basic Programmierer :

Im Deklarationsmodul **IO8_ADD_dcl.bas** finden Sie auch die Deklaration dieser beiden Funktionen. Der Aufruf unterscheidet sich nicht von dem in Pascal.

```
Declare Sub I2C_ReadBlock Lib " USB8IOADD.dll" (ByVal DevNumber As Byte, _
                                                ByVal SlaveAddr As Byte, _
                                                BArray As Byte, _
                                                ByVal ByteCnt As Long)
```

Das Array (hier Barray genannt) wird, da das Schlüsselwort ‚by Val' fehlt, by Referenz, also als Zeiger übergeben. Dabei muß beim Zugriff auf die Karte ein Zeiger auf das nullte Element übergeben werden.

Beispiel :

```
Call I2C_ReadBlock( DevNumber, SlaveAddr, InArray(0), 5 )
```

Achtung :

Bei der Konfiguration der Eingänge als Differenzeingänge wird ein negativer Wert im gelesenen Byte-Array durch ein gesetztes Bit7 angezeigt, der Zahlenwert wird als Zweierkomplement der Bits 0 ... 6 übergeben. Im Folgenden eine kleine Funktion welche aus dem übergebenen Byte den signierten (vorzeichenbehafteten) Eingangswert ermittelt. Zur Ermittlung des absoluten Spannungswertes müssen noch eventuelle Eingangsteiler, Vorverstärker und die Referenzspannung berücksichtigt werden. Die angegebene Funktion läßt sich leicht um einen zweiten Übergabeparameter erweitern und kann dann den absoluten Spannungswert zurückgeben .

```
// Funktion ermittelt bei Differenzeingängen den Eingangswert
// (-128 ... 127) aus dem vom IC gesandten unsignierten Byte

function U_DiffIn ( UDiff : byte ) : integer;

begin
  If (UDiff and 128) = 128 Then
    U_DiffIn := - 128 + ( $080 xor UDiff )              // negative Werte
  else
    U_DiffIn := UDiff;
end;
```

Anhang

```
// Struktur des Types _USB_DEVICE_DESCRIPTOR  ( Delphi-Deklaration ) :
type _USB_DEVICE_DESCRIPTOR = record
    bLength: Byte;                          1 Byte   vorzeichenlos
    bDescriptorType: Byte;                  1 Byte   vorzeichenlos
    bcdUSB: Word;                           2 Byte   vorzeichenlos
    bDeviceClass: Byte;                     1 Byte   vorzeichenlos
    bDeviceSubClass: Byte;                  1 Byte   vorzeichenlos
    bDeviceProtocol: Byte;                  1 Byte   vorzeichenlos
    bMaxPacketSize0: Byte;                  1 Byte   vorzeichenlos
    idVendor: Word;                         2 Byte   vorzeichenlos
    idProduct: Word;                        2 Byte   vorzeichenlos
    bcdDevice: Word;                        2 Byte   vorzeichenlos
    iManufacturer: Byte;                    1 Byte   vorzeichenlos
    iProduct: Byte;                         1 Byte   vorzeichenlos
    iSerialNumber: Byte;                    1 Byte   vorzeichenlos
    bNumConfigurations: Byte;               1 Byte   vorzeichenlos
end;

// allgemeine Prozeduren
OpenDevice (DevNumber, DriverName, ID, KeyWord)                    Rückgabewert 2 Byte vorzeichenlos
SetDeviceEnabled (DevNumber, MC_Mode)
CloseDevice (DevNumber)
CloseAllDevices()
GetDeviceError (DevNumber)
IsEnabled (DevNumber, MC_Mode)                                     Rückgabewert 4 Byte Boolean
IsOpened (DevNumber)                                               Rückgabewert 4 Byte Boolean
SetBootEEP (DevNumber, EEPType, Addr, DeviceID)
GetDeviceDescriptor (DevNumber, usbDD)


MC_Mode          1 Byte   vorzeichenlos          - bei Methode IsEnabled als Zeiger übergeben
DevNumber        1 Byte   vorzeichenlos
EEPType          1 Byte   vorzeichenlos
Addr             1 Byte   vorzeichenlos
DeviceID         2 Byte   vorzeichenlos
DriverName       langer String ( beginnt mit 4 Byte Längenangabe )
ID               langer String ( beginnt mit 4 Byte Längenangabe )
KeyWord          langer String ( beginnt mit 4 Byte Längenangabe )
UsbDD            Typ _USB_DEVICE_DESCRIPTOR


// Routine des Parallelport - Karten REL8_IN8 und OUT8_IN8_AD4_DA
PB_ReadByte(DevNumber, Addr)                                       Rückgabewert 1 Byte vorzeichenlos

// Routinen des Parallelport mit Adressbus-Inkrement - Karte OUT32_IN32
PB_ReadBlockInc(DevNumber, Addr, Barray, ByteCnt)
PB_WriteBlockInc(DevNumber, Addr, Barray, ByteCnt)

DevNumber        1 Byte   vorzeichenlos          Addr             1 Byte   vorzeichenlos
DByte            1 Byte   vorzeichenlos          ByteCnt          4 Byte   vorzeichenlos
BArray           Zeiger auf ein Element eines Bytearray's

-------------------------------------------------------------------------------------------------------------

// allgemeine I2C-Bus-Routinen - Karte OUT8_IN8_AD4_DA
I2C_ReadBlock(DevNumber, Addr, BArray, ByteCnt)
I2C_WriteBlock(DevNumber, Addr, BArray, ByteCnt)

DevNumber        1 Byte   vorzeichenlos          Addr             1 Byte   vorzeichenlos
DByte            1 Byte   vorzeichenlos          ByteCnt          4 Byte   vorzeichenlos
Barray           Zeiger auf ein Element eines Bytearray's

-------------------------------------------------------------------------------------------------------------

Es sind momentan folgende Fehlerkonstanten festgelegt :

Err_No                =     &H0      kein Fehler aufgetreten
Err_DeviceNotOpenend  =     &H1      Device noch nicht geöffnet
Err_DeviceNumber      =     &H2      unzulässige Device-Nummer (zulässig 0 .. 31)
Err_DriverName        =     &H4      Es wurde bei einem vorherigen Device anderer Name gewählt
Err_DeviceNotEnabled  =     &H8      Controller ist nicht aktiv (keine Firmeware geladen)
```

Err_DeviceIOCtl	=	&H10	Funktion DeviceI0Control hat 'False' zurückgeliefert
Err_AI2CNotAck	=	&H20	I2C-Bus NotAcknoledge nach Slaveadresse
Err_DI2CNotAck	=	&H40	I2C-Bus NotAcknoledge nach Datum
Err_I2CBus	=	&H80	allgemeiner I2C-Busfehler
Err_Array	=	&H100	ByteArray hat unerlaubte Dimension
Err_McMode	=	&H200	Methode im aktuellen Controllermodus nicht verfügbar
Err_Pointer	=	&H400	Ungültige RAM-/EEP-Adresse übergeben
Err_DownLoadFile	=	&H800	Datei existiert nicht oder kann nicht gelesen werden
Err_PinNumber	=	&H1000	unzulässige Pin-Nummer
Err_EEPType	=	&H2000	nicht implementierter EEP-Type
Err_ClockDelay	=	&H4000	unzulässige Taktverzögerung (IMode_3 + IMode_4)
Err_InterruptNumber	=	&H8000	Interrupt-Nummer existiert nicht
Err_InterruptHandling	=	&H10000	Fehler beim Abfragen der Interrupt-Flags des Controllers
Err_SetBootEEP	=	&H20000	unzulässige Device-ID (zulässig 0 - 65535) an Methode ‚SetBootEEP' übergeben bzw. aus Boot-EEP gelesenes Array entspricht nicht dem soeben geschriebenen

Bei der Programmierung der Hygrosens-IO-Karten können nicht alle hier wiedergegebenen Fehler auftreten. Für umfassendere Informationen zum verwendeten USB-Controller und der AN21XX.DLL bitte deren Dokumentation zu Rate ziehen.