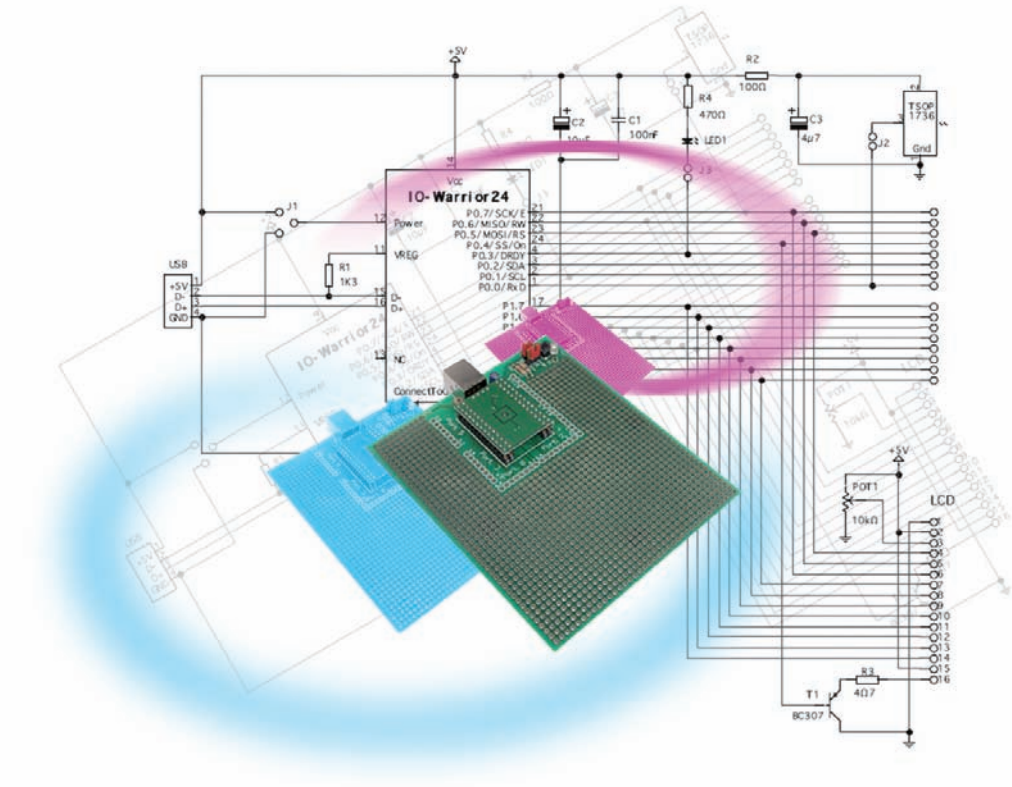
FRANZIS
PC & ELEKTRONIK

Jochen Ferger



Messen, Steuern und Regeln mit USB und C#

Die Warriors von Code Mercenaries

Auf CD-ROM:

- MSR-Anwendungsprogramme mit allen Quelltexten
- Datenblätter

FRANZIS
PC & ELEKTRONIK

Jochen Ferger

Messen, Steuern und Regeln mit USB und C#

Die Warriors von Code Mercenaries

Bibliografische Information der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte Daten sind im Internet über http://dnb.ddb.de abrufbar.

Satz: Fotosatz Pfeifer, 82166 Gräfelfing
art & design: www.ideehoch2.de
Druck: Bercker, 47623 Kevelaer
Printed in Germany

ISBN 978-3-7723-4268-4

Vorwort

C# ist als Programmiersprache in dem Bereich Messen, Steuern, Regeln kaum behandelt. Dabei bietet die Sprache viele Vorteile: syntaktisch einfacher und übersichtlicher Quelltext, hohe Performance, einfache Hardware-Unterstützung, sehr gute Dokumentationen und eine hervorragende Entwicklungsumgebung, die es als Express Edition sogar kostenlos im Internet zum Herunterladen gibt.

Hinzu kommt, dass in C# erstellte Bibliotheken auch in anderen .NET-Sprachen zur Verfügung stehen.

Als Hardwarebausteine wurden die Interfaces der Fima Code Mercenaries ausgewählt. Die Bausteine sind kostengünstig, bieten vielfältigste Möglichkeiten, realisieren Industriestandards und sind im Allgemeinen als einfach aufzubauende Starterkits zu erhalten. Sie eignen sich damit sowohl zum professionellen Einsatz in der Industrie oder anderen Bereichen der Steuerungstechnik als auch im semiprofessionellen Bereich zum Messen und zum Ansteuern einfacher Aktoren. Im Vordergrund stehen hierbei die Bausteine der IO-Warrior-Serie, aber auch der Spin-Warrior und der Joy-Warrior werden für messtechnische Projekte angesprochen.

Das Buch vermittelt die Grundlagen der Programmierung in C#. Anschließend erfolgt der Zugriff auf USB-Hardware mittels entsprechender DLLs (Dynamic Link Libraries). Auf einfache IO-Projekte folgt die Beschreibung der Spezialfunktionen wie die Ansteuerung von LCD-Displays, das Auslesen von Fernbedienungscodes sowie die Anwendung und Programmierung auf dem I²C- und dem SPI-Bus. Im Kapitel zur erweiterten Programmierung werden Netzwerkprojekte realisiert sowie die Speicherung und Aufbereitung von Messdaten dargestellt. Abschließend werden mit dem Spin-Warrior und dem Joy-Warrior weitere C#-Projekte realisiert und eine Übersicht auf weitere Bausteine der Firma Code Mercenaries gegeben.

Hinweis: Bei der Verwendung von Geräten und elektrotechnischen Bauteilen kann es durch falsche Handhabung und falsche Programmierung zur Zerstörung der Bauteile oder des PCs kommen. Hierfür übernimmt der Autor keine Haftung.

Bei der Verwendung von Daten im Netzwerk kann es zu Missbrauch der Daten kommen. Bitte beachten Sie die entsprechenden Vorschriften des Datenschutzes bei der Verwendung von Netzwerkdiensten. Für Missbrauch übernimmt der Autor keine Haftung.

Danksagung:

Ganz herzlicher Dank gilt der Firma Code Mercenaries, die mir alle Bausteine und Entwicklungskits zur Verfügung gestellt hat und mir mit Rat und Tat zur Seite stand, insbesondere Herrn Guido Körber vielen Dank.

Der Firma JenColor in Person von Frank Krumbein danke ich herzlich für die Bereitstellung der RGB-Sensoren, der Entwicklungssoftware sowie für die technische Unterstützung.

Technische Unterstützung habe ich von Herrn Eberhard Fahle, Herrn Christoph Schnedl und Herrn Ulrich Radig bekommen, wofür ich mich herzlich bedanke.

Ganz lieben Dank richte ich an Bine, die mich motiviert, unterstützt und die erste Korrektur übernommen hat.

Für Sophia und Paul.

Inhaltsverzeichnis

1 Grundlagen der C#-Programmierung

1.1 Installation der Entwicklungsumgebung

Die Entwicklungsumgebung von C# kann man in den unterschiedlichsten Versionen installieren. Für die im Buch beschriebenen Programmierbeispiele reicht die kostenlose Express Edition vollkommen aus. Sie können sie im Internet herunterladen und dann installieren. Sie benötigen für die Programmierung ausschließlich die C#-Variante der Express Edition. Zusätzlich sollte man die Hilfsbibliothek MSDN installieren. Auch diese können Sie im Internet herunterladen.

1.2 Projekterstellung einer Konsolenapplikation

Nachdem die Installationen durchgeführt wurden, kann man mit dem ersten Programmierprojekt beginnen. Die Konsolenapplikation erscheint vielleicht etwas antiquiert, jedoch ist bei der Steuerungstechnik die grafische Gestaltung von Programmen nicht wichtig. Gleichzeitig sind Konsolenapplikationen aber weitaus performanter und gerade die Leistung einer Software kann hier eine entscheidende Rolle spielen.

Nachfolgend wird beschrieben, wie man eine Konsolenapplikation erstellt.

Hierzu wird die Express Edition von C# per Doppelklick gestartet:

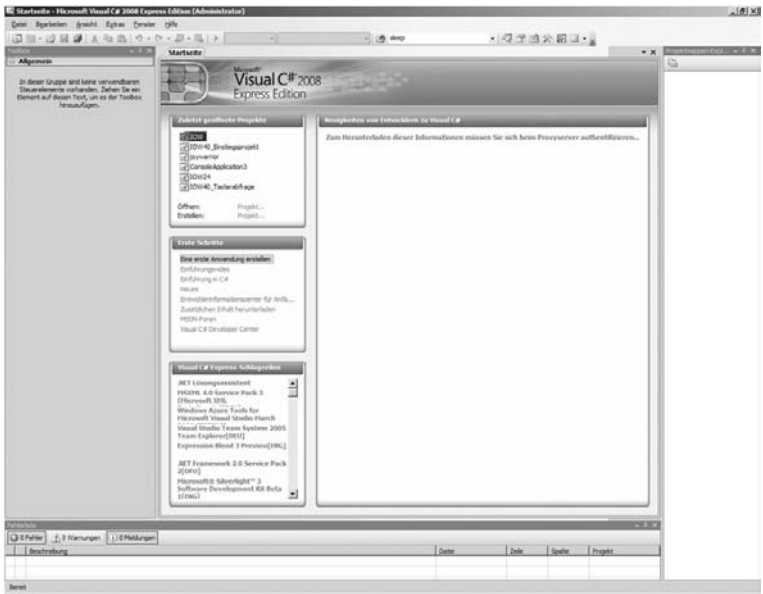


Abb. 1.1: Start der Entwicklungsumgebung

Im Menüpunkt *Datei* wird die Option *Neues Projekt* ausgewählt.

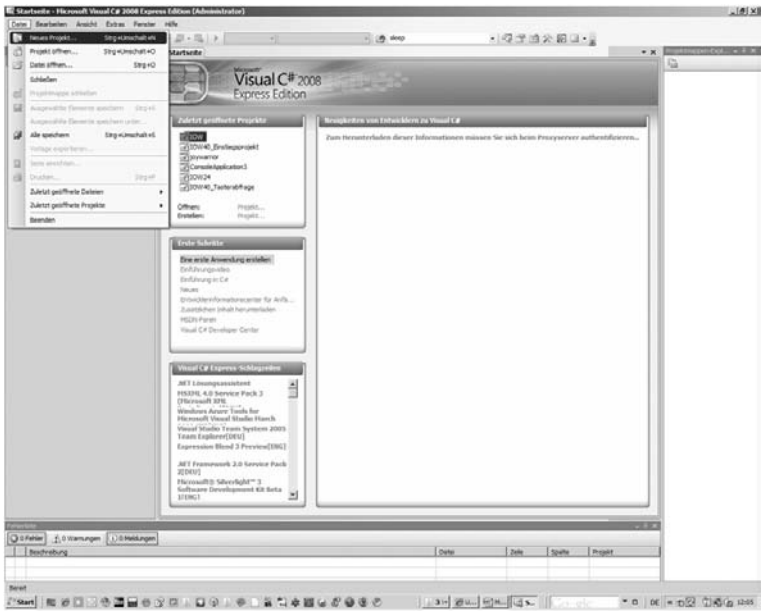


Abb. 1.2: Auswahldialog *Neues Projekt*

Im nächsten Auswahlfenster hat man die Möglichkeit, Programme mit grafischer Oberfläche, Klassenbibliotheken oder Konsolenapplikationen auszuwählen. Wählen Sie eine Konsolenanwendung aus und geben im unteren Textfeld einen repräsentativen Namen für Ihr Projekt ein. Die Eingabe wird mit einem Klick auf *OK* bestätigt.

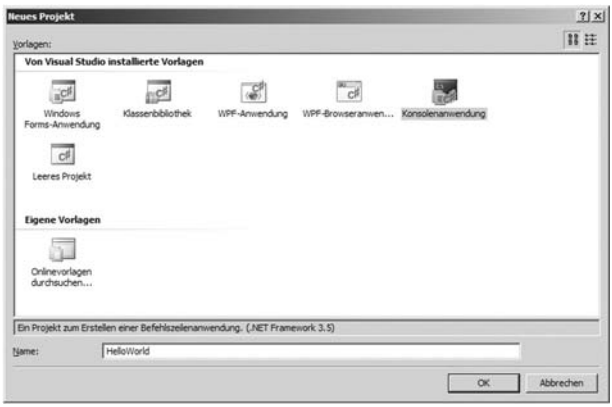


Abb. 1.3: Anlegen einer Konsolenapplikation

Die Entwicklungsumgebung generiert nun den Programmrumpf.

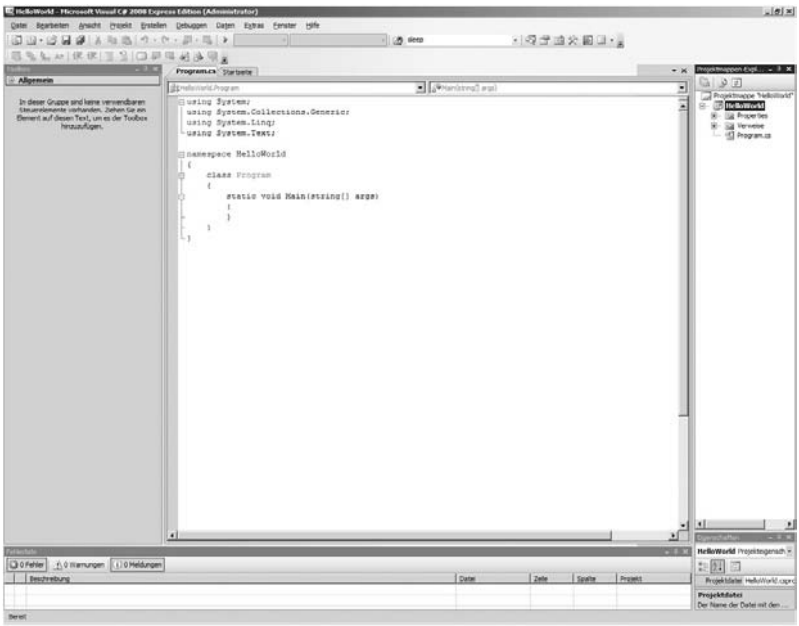


Abb. 1.4: Das erste lauffähige Programm

Geben Sie nun zwischen den beiden geschweiften Klammern unter *static void Main (string [] args)* nur die Buchstaben „Con“ ein. Das System reagiert sofort mit Vorschlägen zur automatischen Vervollständigung des Quelltextes.

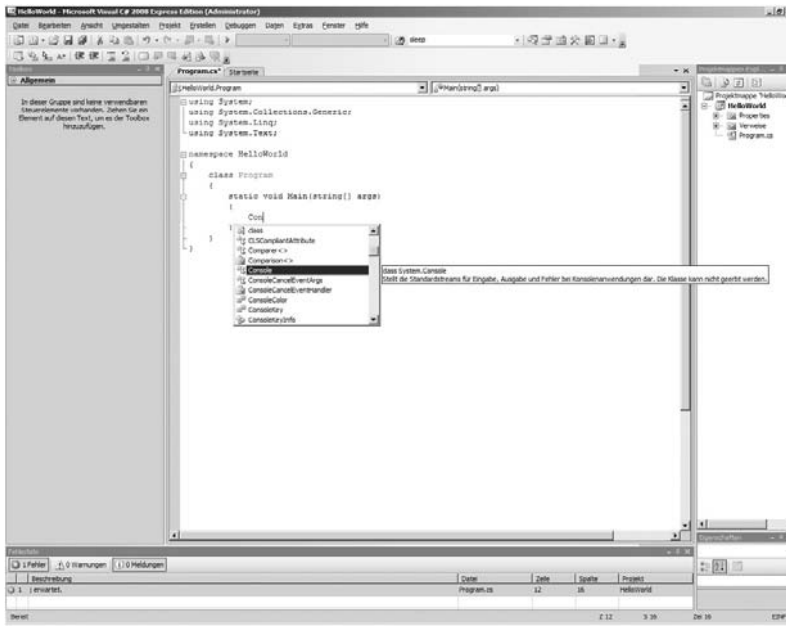


Abb. 1.5: Automatische Codevervollständigung

Geben Sie nun einen Punkt (.) ein, wird das Wort „Console“ automatisch vervollständigt und eine Auswahlliste von Funktionen passend zur Klasse *Console* vorgeschlagen. Vervollständigen Sie nun den Quelltext wie folgend beschrieben:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(“Hello World”);
            Console.ReadKey();
        }
    }
}
```

Durch einen Klick auf den grünen Pfeil in der oberen Menüzeile wird das Programm kompiliert und, sofern es fehlerlos ist, auch ausgeführt:

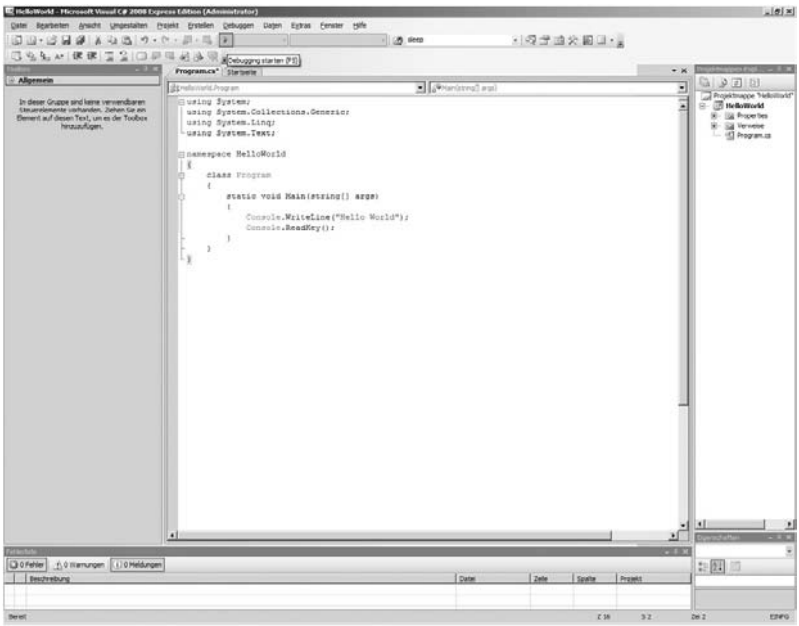


Abb. 1.6: Quellcode Hello World

Programmausgabe:

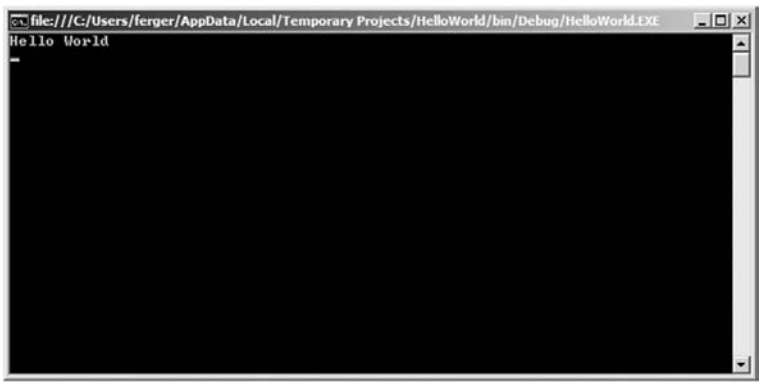


Abb. 1.7: Bildschirmausgabe Hello World

Mit Druck auf die Eingabetaste wird das Programm beendet.

1.3 Ausgaben

Auch in der Konsole kann man einigermaßen formatierte Ausgaben gestalten. Mithilfe der Escapezeichen *t* für einen Tabulator und *n* für einen Zeilenumbruch sind hier Gestaltungsmöglichkeiten vorhanden, will man beispielsweise Messreihen übersichtlich auf einer Konsolenseite darstellen.

Programmbeispiel:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ausgaben
{
    class Program
    {
        static void Main(string[] args)
        {
           Console.WriteLine("Beispiel für\tTabulator und \nZeilenumbruch");
            Console.ReadKey();
        }
    }
}
```

Programmausgabe:

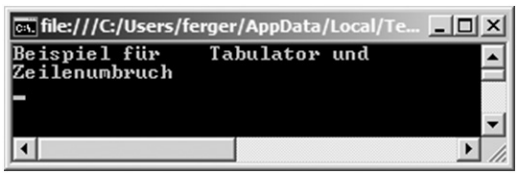


Abb. 1.8: Bildschirmausgabe

1.4 Variablentypen

Um während der Laufzeit eines Programms Daten zu speichern, benötigt man Variablen. Die gängigsten und für dieses Buch notwendigen sind in der nachfolgenden Tabelle aufgeführt:

Variable	Zahlenbereich
byte	0 bis 255
int	-231 bis 231 -1
double	5,0 * 10-324 bis 1,7 * 10308
string	Text bis zu 231 Zeichen
bool	true oder false

Eine Variable wird deklariert und initialisiert. Bei der Deklaration wird einem Bezeichner ein Variablentyp zugeordnet. Beispiel:

```
int i;
```

Bei der Initialisierung wird der Variablen ein Wert mitgeteilt. Beispiel:

```
i = 0;
```

1.5 Typecasting

Im Programm kann es aus unterschiedlichen Gründen vorkommen, dass ein Variablentyp umgewandelt werden muss. Man bezeichnet dies als *Typecasting*.

Beispiel 1:

Ein 8-Bit-AD-Wandler liefert je nach Eingangsspannung (Wertebereich 0 V bis 5 V) einen Wert von 0 bis 256, beispielsweise den Wert 100. Will man nun den dazugehörigen Spannungswert ermitteln, liefert die eigentliche Berechnung: $\frac{100}{256} \cdot 5V = 1{,}95V$

Verwendet man den folgenden Quellcode:

```
int wert1 = 100;
int wert2 = 256;
double ergebnis1 = wert1 / wert2*5;
Console.WriteLine(ergebnis1);
```

so ergibt die Ausgabe ein Ergebnis von 0.

Erklärung:

Die Operation zweier Integerzahlen hat als Ergebnis immer eine Integerzahl. Da aber $\frac{100}{256}$ einen Wert unter 1 ergibt und eine Integerzahl keinen Kommaanteil besitzt, ist das Ergebnis 0.

Über eine explizite Typenumwandlung kann Abhilfe geschaffen werden:

```
double ergebnis2 = (double)wert1 / wert2*5;
Console.WriteLine(ergebnis2);
```

Vor der Variablen *wert1* steht in Klammern der Variablentyp, der in diesem Fall zu verwenden ist. Die Rechenoperation mit einer Variablen vom Typ *double* ergibt auch immer eine Variable vom Typ *double*. Somit bekommt man hier das korrekte Ergebnis ausgegeben.

Beispiel 2:

Ein 8-Bit-DA-Wandler soll angesteuert werden. Die entsprechende Funktion erwartet einen Datentyp *byte* als Übergabeparameter (Wert 0 bis 255, die Funktion heißt beispielsweise *daAusgabe()*). Nacheinander sollen alle Werte von 0 bis 255 ausgegeben werden. Wird nun in einer *for*-Schleife eine Variable vom Typ *Integer* abgearbeitet, muss diese bei der Übergabe an die Funktion in eine Variable vom Typ *byte* gecastet werden.

Der Quelltext

```
for (int i = 0; i <= 255; i++)
 {
         daAusgabe(i);
 }
```

führt beim Kompilieren zu folgendem Fehler:

```
1-Argument: kann nicht von „int“ in „byte“ konvertiert werden.
```

Abhilfe schafft auch hier eine Typenumwandlung:

```
for (int i = 0; i <= 255; i++)
{
       daAusgabe((byte)i);
}
```

Jetzt lässt sich der Quelltext einwandfrei kompilieren.

1.6 Eingaben

Will man in einem Programm den Wert einer Variablen vom Benutzer eingeben lassen, funktioniert dies nicht ganz so einfach. Die Konsole bietet nur wenige Funktionen zum Einlesen von Werten. Mit der Funktion *Console.readLine()* wird eine ganze

Zeile (bis zur Betätigung der Eingabetaste) eingelesen und als *string* abgespeichert. Mittels der Klasse *Convert* kann man nun aus dem eingelesenen String den entsprechend gewünschten Datentyp zurückgeben lassen. Achtung: Gibt der Anwender Zeichen ein, die auf keinen Fall in einen numerischen Datentyp umgewandelt werden können, kommt es zu Fehlermeldungen. Gegebenenfalls muss eine Falscheingabe vom Programmierer abgefangen werden. Hierzu mehr im Punkt „Exceptions abfangen“.

Das folgende Programmbeispiel zeigt, wie man Variablen einlesen und umwandeln kann.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Variablen_einlesen
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Geben Sie eine Zahl mit einem Komma , ein: ");
            string strg = Console.ReadLine();
            double zahl1 = Convert.ToDouble(strg);
            Console.WriteLine("Geben Sie eine ganze Zahl ohne Komma ein: ");
            strg = Console.ReadLine();
            int zahl2 = Convert.ToInt16(strg);
            double ergebnis = zahl1 + zahl2;
            Console.WriteLine(zahl1 + " + " + zahl2 + " = " + ergebnis);
            Console.ReadKey();
        }
    }
}
```

Das Programm hat folgende Ausgabe:

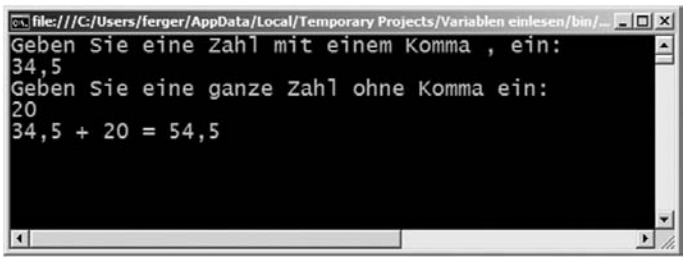


Abb. 1.9: Variablen einlesen

1.7 Mathematische Operationen

1.7.1 Standardoperationen

Um Datenverarbeitung überhaupt erst durchführen zu können, bedarf es vieler mathematischer Berechnungen, die unter anderem mit den folgenden Operatoren durchgeführt werden können:

Operator	Bedeutung	Beispiel
+	Addition	I = x + 2;
-	Subtraktion	i = z - 3;
*	Multiplikation	x = 5 * 2;
/	Division	e = 9 / 4;
%	Ganzzahliger Rest einer Division	i = 9 % 7;

++	Inkrementieren	i=0;
		i++;
		i hat hiernach den Wert 1.

--	Dekrementieren	i = 0; i--; i hat hiernach den Wert –1.

1.7.2 Boolesche Operationen

Operator	Bedeutung	Beispiel
<	ist kleiner als	i = 5 < 6; i hat hiernach den Wert true.
>	ist größer als	i = 5 > 6; i hat hiernach den Wert false.
<=	ist kleiner gleich als	i = 5 <= 5; i hat hiernach den Wert true.
>=	ist größer gleich als	i = 5 >= 5; i hat hiernach den Wert true.
==	ist gleich	i = 5 == 6; i hat hiernach den Wert false.

!=	ist ungleich	i = 5 != 6; i hat hiernach den Wert true.
&&	logisches UND	i = ((5 >6) && (6 >5)); i hat hiernach den Wert false.
||	logisches ODER	i = ((5 >6) || (6 >5)); i hat hiernach den Wert true.
&[1]	bitweise UND-Verknüpfung	i = (127 & 4); i hat hiernach den Wert 4
|	bitweise ODER-Verknüpfung	i = (127 | 128); i hat hiernach den Wert 128

1.8 Bits und Bytes

Digitale Signale bestehen grundsätzlich nur aus einer 1 (TRUE, HIGH) oder einer 0 (FALSE, LOW). Will man nun digitale Signale verarbeiten, ist es unumgänglich, etwas über die grundlegenden Funktionen UND, ODER und NICHT zu wissen. In diesem Abschnitt werden diese Funktionen/Bausteine vorgestellt und gleichzeitig die Anwendung in C# verdeutlicht.

1.8.1 UND

Der Baustein UND ermöglicht es dem Anwender, zu überprüfen, ob alle Eingangssignale gleichzeitig auf 1 (true) gesetzt sind. Ist dies der Fall, liegt am Ausgang des Bausteins auch das Signal 1 an, in allen anderen Fällen kann man am Ausgang eine 0 abnehmen.

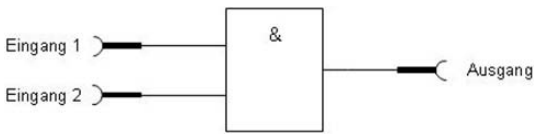


Abb. 1.10: Und-Baustein

Die folgende Tabelle, auch Wahrheitstabelle genannt, erläutert die verschieden Zustände:

1 Die bitweisen Verknüpfungen werden später noch genauer beleuchtet, da sie gerade für das Steuern und das Filtern digitaler Informationen sehr wichtig sind.

Eingang 1	Eingang 2	Ausgang
0	0	0
0	1	0
1	0	0
1	1	1

In der Programmiersprache C# kann die UND-Funktion beispielsweise in der bedingten Entscheidung verwendet werden:

```
bool e1 = true;
bool e2 = false;
bool e3;
if (e1 && e2)
e3 = true;
else
e3 = false;
```

Statt der bedingten Entscheidung können Sie die Zuweisung auch direkt notieren:

```
e3 = e1 && e2;
```

Auch ist es möglich, statt e1 und e2 andere Abfragen in der bedingten Entscheidung mit UND zu verknüpfen. Beispielsweise soll überprüft werden, ob eine Zahl im Bereich von 1 bis 10 liegt:

```
int zahl = 8;
if (zahl > 0 && zahl < 11)
Console.WriteLine („Zahl ist im Bereich von 1 bis 10“);
else
Console.WriteLine („Zahl ist nicht im Bereich von 1 bis 10“);
```

1.8.2 Bitweise UND-Verknüpfung

Diese Funktion ist für unsere Vorhaben sehr wichtig. Nehmen wir an, wir lesen von einem Interface fünf Eingänge gleichzeitig ein. Eine solche Maßnahme liefert als Ergebnis eine Zahl von 0 bis 31 zurück.

Will man aber nur wissen, ob das 3. Bit (Wertigkeit $2^2 = 4$) gesetzt ist oder nicht, muss man es herausfiltern. Dies kann man mit einer bitweisen UND-Verknüpfung realisieren. Wir benötigen hierfür ein Maskenbyte (Maske), in dem nur die Bits auf 1 gesetzt sind, die man filtern möchte – in unserem Fall also die Zahl 001002 oder umgerech-

net 4. Verknüpft man nun den eingelesenen Wert (die Bits dieses Werts werden im folgenden Beispiel mit „abcde“ bezeichnet) mit der Maske bitweise UND, passiert Folgendes:

Eingelesener Wert:	abcde
Maske:	00100
Operation:	&
Ergebnis	00c00

Man sieht, dass die Bits abde ignoriert werden. Im Ergebnis taucht nur noch das Bit c auf. Somit können wir also leicht den Zustand eines Bits herausfinden. In C# sähe die Umsetzung folgendermaßen aus:

```
int wert; //wert ist der Zustand meiner Eingänge des Interfaces
int maske = 4;
if ((wert & maske) == 4)
Console.WriteLine („Bit 3 liefert den Wert TRUE“);
else
Console.WriteLine („Bit 3 liefert den Wert FALSE“);
```

1.8.3 ODER-Verknüpfung

Die ODER-Verknüpfung ermöglicht es, zu überprüfen, ob mindestens einer der Eingänge den Wert *True* hat. Nur in dem Fall, dass alle Eingänge auf *False* gesetzt sind, besitzt auch der Ausgang den Wert *False*.

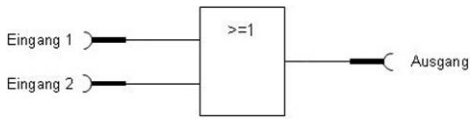


Abb. 1.11: ODER-Verknüpfung

Wahrheitstabelle:

Eingang 1	Eingang 2	Ausgang
0	0	0
0	1	1
1	0	1
1	1	1

In C# kann man die ODER-Verknüpfung dafür verwenden, eine von zwei Bedingungen abzufragen. Beispiel: Eine Alarmanlage hat zwei Sensoren und löst den Alarm aus, wenn einer der beiden Sensoren den Wert *TRUE* liefert:

```
if (s1 == true || s2 == true)
Console.WriteLine („Alarm“);
else
Console.WriteLine („Alles ruhig daheim“);
```

1.8.4 Nicht-Verknüpfung

Diese Verknüpfung invertiert das Eingangssignal. Beispielsweise liegen an Schnittstellen Signale oft invertiert an. Will man dann wissen, ob der Eingang gesetzt ist oder nicht, kann man das eingelesene Bit invertieren. In C# kann man die Nicht-Verknüpfung verwenden, um zu überprüfen, ob eine Bedingung nicht eingetroffen ist:

```
if (e1 != true)  // Alternativ geht: if (!e1)
{
Console.WriteLine („E1 ist nicht gesetzt“);
}
```

Es gibt noch jede Menge anderer Verknüpfungen, die aber in diesem Buch keine Verwendung finden. Bei Interesse kann man im Internet nach den Stichwörtern „Logische Verknüpfungen C#“ suchen oder die einschlägige Literatur verwenden. Hinweise hierzu finden Sie im Anhang.

1.8.5 Bytes verschieben

Mithilfe der Operationen <<x und >>x (x steht hier für eine ganze Zahl) kann man eine Zahl bitweise verschieben. Dies kann man beispielsweise dann nutzen, wenn man einen Zahlenwert erstellen will, der sich aus mehreren Bytes zusammensetzt. Beispielsweise setzt sich die IP-Adresse eines Rechners im Netzwerk aus vier Bytes zusammen. Die Operation <<x verschiebt eine Zahl um x Bits nach links, die dazukommenden Bits haben alle automatisch den Wert 0. Die Operation >>x verschiebt eine Zahl um x Bits nach rechts. Die Bits, die „rausgeschoben“ werden, gehen verloren.

1.9 Kontrollstrukturen

1.9.1 Die bedingte Entscheidung

Bei der *bedingten Entscheidung* wird eine Bedingung abgefragt. Trifft diese zu, wird ein Programmteil ausgeführt. Trifft die Bedingung nicht zu, wird ein anderer Programmteil ausgeführt. Die bedingte Entscheidung hat die Form:

```
if (Bedingung)
{
}
else
{
}
```

Anmerkung: Der else-Teil ist nicht zwingend erforderlich.

Beispielprogramm:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Bedingte_Entscheidung
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Geben Sie eine Zahl von 0 bis 100 ein:");
            string stri = Console.ReadLine();
            int i = int.Parse(stri);
            Console.WriteLine(i);
            if (i < 50)
            {
                Console.WriteLine("Die Zahl ist kleiner als 50!");
            }
            else
            {
                Console.WriteLine("Die Zahl ist größer als 49");
            }
            Console.ReadKey();
        }
    }
}
```

Das Programm hat folgende Ausgabe zur Folge:

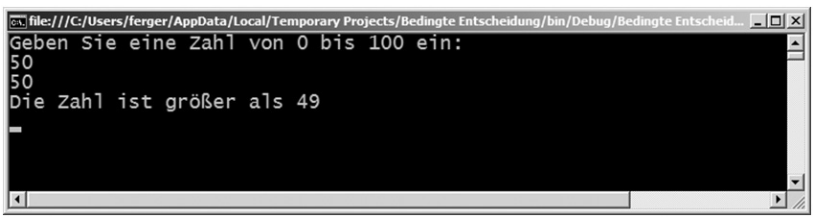


Abb. 1.12: Bedingte Entscheidung

1.9.2 Die Mehrfachentscheidung

Um mehrfachgeschachtelte if-else-Konstrukte zu vermeiden, kann man sich der switch-case-Mehrfachentscheidung bedienen.

Das Konstrukt beginnt mit der Anweisung:

```
switch(ganzzahlige Variable)
{
//Befehlsblock
}
```

Im Befehlsblock kann dann mittels verschiedener Fälle in case-Blöcken auf mögliche Werte der ganzzahligen Variablen reagiert werden. Im folgenden Beispiel wird in einer Mehrfachentscheidung unterschiedlich auf die Eingabe des Anwenders reagiert.

```
Console.WriteLine("Geben Sie eine ganze Zahl ohne Komma ein: ");
string strg = Console.ReadLine();
int zahl = Convert.ToInt16(strg);
switch (zahl)
{
    case 1: Console.WriteLine("Benutzer hat 1 eingegeben"); break;
    case 2: Console.WriteLine("Benutzer hat 2 eingegeben"); break;
    case 3: Console.WriteLine("Benutzer hat 3 eingegeben"); break;
    default: Console.WriteLine("Benutzer hat weder 1 noch 2 oder 3 eingege-
ben"); break;
}
```

1.10 Schleifen

Schleifen dienen dazu, Programmteile kontrolliert zu wiederholen. Gerade in der Messtechnik müssen Werte wiederholt eingelesen werden. Hierzu könnte man natürlich den Quellcode x-Mal hintereinander schreiben, jedoch führt dies zu keinem effizienten und sicherlich zu unübersichtlichem Quelltext. Eine Schleife läuft gewöhnlich so lange, bis eine im Quelltext definierte Bedingung erfüllt wurde. Die Bedingung kann am Anfang oder am Kopf der Schleife stehen. Man spricht dann von einer kopfgesteuerten Schleife. Steht die Bedingung am Ende, also am Fuß der Schleife, ist sie fußgesteuert. Auch kann man definieren, wie oft eine Schleife wiederholt werden soll. Hierzu dient die Zählschleife.

1.10.1 Fußgesteuerte Schleifen

Die Schleife hat immer die Form:

```
do
{
}while (Bedingung);
```

Beispielprogramm:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Fußgesteuerte_Schleife
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            do
            {//Start des Schleifenkörpers
                Console.WriteLine("Aktueller Wert von i: " + i);
                i++;
            } while (i < 10); ////Ende des Schleifenkörpers
            Console.ReadKey();
        }
    }
}
```

Das Programm führt zu folgender Bildschirmausgabe:

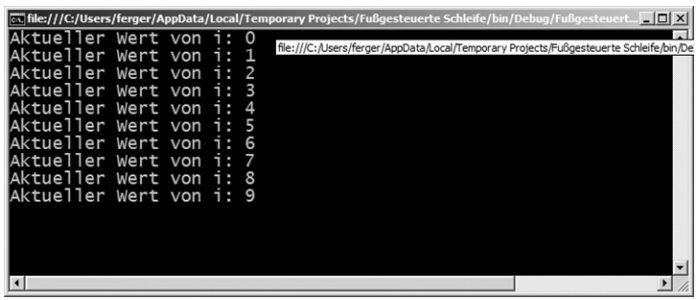


Abb. 1.13: Fußgesteuerte Schleife

1.10.2 Kopfgesteuerte Schleifen

Diese Schleife hat immer die Form:

```
while (Bedingung)
{

}
```

Beispielprogramm:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Kopfgesteuerte_Schleife
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            while (i < 10) //Die Schleife läuft, solange i kleiner 10 ist
            {  //Der Beginn des Schleifenkörpers
                Console.WriteLine("Aktueller Wert von i: " + i);
                i++;
            }  //Das Ende des Schleifenkörpers
            Console.ReadKey();
        }
    }
}
```

Das Programm führt zu folgender Ausgabe:

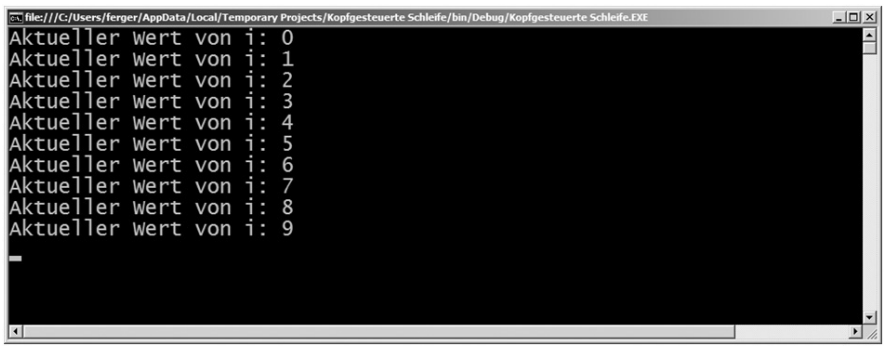


Abb. 1.14: Kopfgesteuerte Schleife

1.10.3 Zählschleifen

Die Zählschleife hat immer die Form:

```
for (Definition; Bedingung; Operation)
{

}
```

```
Beispielprogramm:
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Zählschleife
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("Aktueller Wert von i: " + i);
            }
            Console.ReadKey();
        }
    }
}
```

In dem for-Ausdruck wird i als Integervariable deklariert und mit dem Wert 0 initialisiert. Die Schleife läuft, solange i < 10 ist, und nach jedem Schleifendurchlauf wird i um den Wert 1 erhöht.

Das Programm hat folgende Bildschirmausgabe zur Folge:

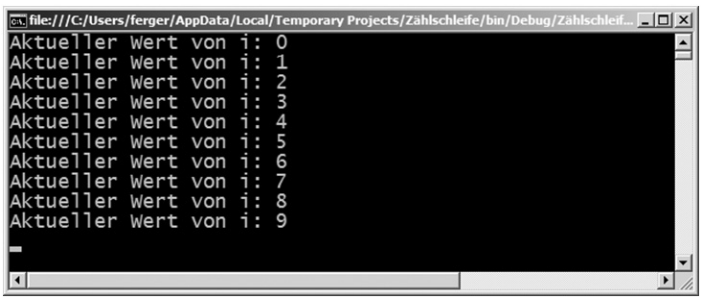


Abb. 1.15: Zählschleife

Welche der drei Schleifenarten man verwendet, hängt vom jeweiligen Anwendungsfall ab.

1.10.4 foreach-Schleife

Mittels dieser Schleife kann man beispielsweise alle Werte eines Arrays abarbeiten. Im nachfolgenden Beispiel wird ein Array mit fünf Integerwerten angelegt. In der Schleife werden alle Elemente des Felds nacheinander ausgegeben:

```
int[] feld = {2,4,6,8,10};
foreach (int wert in feld)
{
    Console.WriteLine(wert);
}
```

1.10.5 break, continue

Mittels *break* wird eine Schleife sofort beendet und verlassen. *continue* sorgt dafür, dass die Schleife sofort mit dem nächsten Durchlauf gestartet wird. Im nachfolgenden Beispiel wird der Benutzer in einer Endlosschleife aufgefordert, eine Integerzahl einzugeben. Wählt er *break*, wird das Programm beendet, wählt er *continue*, wird die Schleife sofort wiederholt, wählt er eine andere Zahl, erfolgt vor der Schleifenwiederholung eine Bildschirmausgabe:

```
while (true)
{
       Console.WriteLine("break(1) oder continue(2)");
       string strg = Console.ReadLine();
       int zahl = Convert.ToInt16(strg);
       if (zahl == 1)
       {
            break;
       }
       if (zahl == 2)
       {
            continue;
       }
       Console.WriteLine("Schleife wird wiederholt");
 }
```

1.11 Funktionen

In einem Programm kommt es häufig vor, dass Quelltextpassagen mehrfach wiederholt werden müssen. Um nicht jedes Mal diese Passagen neu tippen zu müssen, kann man Funktionen programmieren und diese dann nach Belieben aufrufen. Einer Funktion kann man Werte übergeben und die Funktion kann einen Wert zurückgeben. Die Funktion benötigt einen möglichst aussagekräftigen Namen. Im folgenden Beispiel wird eine Funktion erstellt, die das Quadrat eines übergebenen Werts berechnet und zurückgibt:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;

namespace Funktionen
{
    class Program
    {
        static double quadrat(double zahl)
        {
            double ergebnis;
            ergebnis = zahl * zahl;
            return ergebnis;
        }

        static void Main(string[] args)
        {
            string strzahl = Console.ReadLine();
            double zahl = double.Parse(strzahl);
            double qdr = quadrat(zahl);
            Console.WriteLine(qdr);
            Console.ReadKey();
        }
    }
}
```

Das Programm erzeugt diese Bildschirmausgabe:

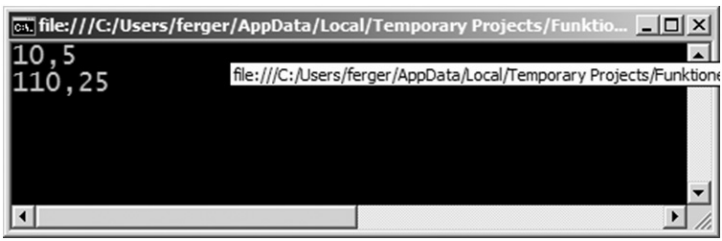


Abb. 1.16: Screenshot-Funktionen

Man beachte, dass die Kommazahlen mit einem echten Komma eingegeben werden! Auch kann der gleiche Funktionsname mehrfach verwendet werden, wenn in den einzelnen Funktionen die Art oder die Anzahl der übergebenen Parameter unterschiedlich ist.

1.11.1 Call by Value, Call by Reference

Einer Funktion können Werte übergeben werden. Hierbei ist es möglich, die Werte in der Funktion zu verändern, was aber die eigentliche Variable nicht berührt. Verwendet man das Schlüsselwort *ref*, wird nicht der Wert der Variablen (*value*), sondern die Va-

riable selbst (*reference*) übergeben. Im folgenden kleinen Programmbeispiel wird dies deutlich:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CallBy
{
    class Program
    {
        static void Addiere(int zahl)
        {
            zahl = zahl + 10;
        }

        static void AddiereReferenz(ref int zahl)
        {
            zahl = zahl + 10;
        }

        static void Main(string[] args)
        {
            int wert = 20;
            Addiere(wert);
            Console.WriteLine("Wert der Variablen: " + wert);
            AddiereReferenz(ref wert);
            Console.WriteLine("Wert der Variablen: " + wert);
            Console.ReadKey();
        }
    }
}
```

Das Programm erzeugt diese Bildschirmausgabe:

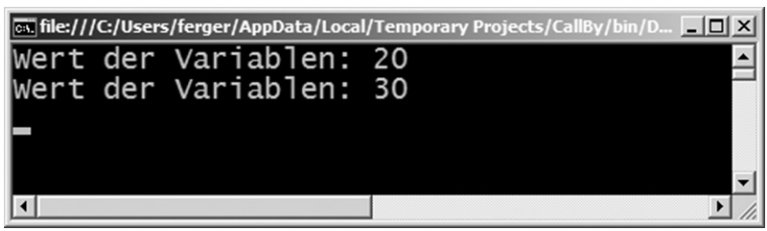


Abb. 1.17: *Call by value, call by reference*

Man sieht, dass beim Aufruf der Funktion *AddiereReferenz* der Wert der Variablen *wert* tatsächlich verändert wurde, wohingegen die Funktion *Addiere* die eigentliche Variable *wert* nicht ändert. Dies ist insbesondere dann wichtig, wenn externe Funktionen

beispielsweise aus der DLL zu den IO-Warriors eine Referenz erwarten und dort beim Aufruf auswertbare Daten schreiben.

1.12 Arrays/Felder

In Feldern können mehrere Werte vom gleichen Datentyp abgespeichert werden. Dies ist besonders für Messreihen geeignet. Aber auch die IO-Bausteine erwarten in den Reports Felder – hierzu später mehr.

Ein Feld, hier beispielsweise von 1.000 Integerzahlen, wird folgendermaßen angelegt:

```
int feld [ ] = new int [ 1000 ];
```

Anschließend kann man über den Index in den eckigen Klammern auf die einzelnen Feldelemente zugreifen.

```
feld [ 9 ] = 300;
```

Die Länge eines Felds bekommt man über folgenden Ausdruck:

```
Console.WriteLine (feld.Length); //hat hier 1000 als Ausgabe zur Folge
```

Der Index beginnt immer mit der Zahl 0. Somit hat das letzte Element des Felds den Index *feld.Length – 1* !

Die for-Schleife eignet sich ideal zum Füllen eines Arrays beispielsweise mit Messwerten.

Im nachfolgenden Beispielprogramm wird ein Feld aus 100 Integerzahlen mit Zufallswerten im Bereich von 0 bis 100 gefüllt. Die Zahlen werden auf dem Bildschirm ausgegeben. Anschließend werden das größte und das kleinste Element sowie das arithmetische Mittel aller Zahlen berechnet und ausgegeben.

Kommentierter Quellcode des Programms *Arrays:*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Arrays
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
        Random zufall = new Random();

        //Ein Feld für 100 Integerzahlen wird erzeugt
        int[] feld = new int[100];

        //Deklaration der notwendigen Variablen
        int summe = 0, kleinstes = 1000, groesstes = 0;
        double schnitt = 0;

        //Das Feld wird mit Zahlen gefüllt
        for (int i = 0; i < 100; i++)
          {
           //Das Objekt zufall aus der Klasse Random
           //liefert sehr leicht Zufallszahlen
           feld[i] = zufall.Next(101);
           //Ausgabe des Feldes
           Console.Write(feld[i] + "\t");
           //Bei jeder Zahl, die durch 10 teilbar ist, wird
           //ein Zeilenumbruch vorgenommen.
           if (((i + 1) % 10) == 0)
           {
                Console.WriteLine();
           }
          }
        for (int i = 0; i < 100; i++)
        {
           //Elemente werden summiert
           summe = summe + feld[i];
           //Das größte Element wird ermittelt
           if (feld[i] > groesstes)
            groesstes = feld[i];
           //Das kleinste Element wird ermittelt
           if (feld[i] < kleinstes)
             kleinstes = feld[i];
         }
         //Ausgabe der errechneten Ergebnisse
        Console.WriteLine("Groesstes Element: " + groesstes);
        Console.WriteLine("Kleinstes Element: " + kleinstes);
        //Der Schnitt wird errechnet. Da die Division durch 100
        //eine Kommazahl ergibt, summe aber vom Typ Integer ist
        //muss das Ergebnis der Rechnung explizit in eine Zahl
        //vom Typ double umgewandelt werden.
        schnitt = (double)summe / 100;
        Console.WriteLine("Schnitt: " + schnitt);
        Console.ReadKey();
      }
   }
}
```

1.13 Multithreading

Möchte man beispielsweise den Eingangszustand eines Hardwareinterfaces einlesen, kann dies im Hauptprogramm geschehen. Soll dieser Eingangszustand ständig ausgelesen werden, wird der Vorgang das Hauptprogramm blockieren oder stark verlangsamen. Hier bietet es sich an, das Hardwareinterface in einem parallelen Thread einzulesen und das Hauptprogramm somit nicht zu belasten. Ein Thread ist ein „kleiner" Prozess, der auf dem System abläuft.

In C# ist es denkbar einfach, parallele Prozesse zu erstellen. Im nachfolgenden Beispiel werden zwei unterschiedliche Texte von zwei Threads auf der Konsole ausgegeben:

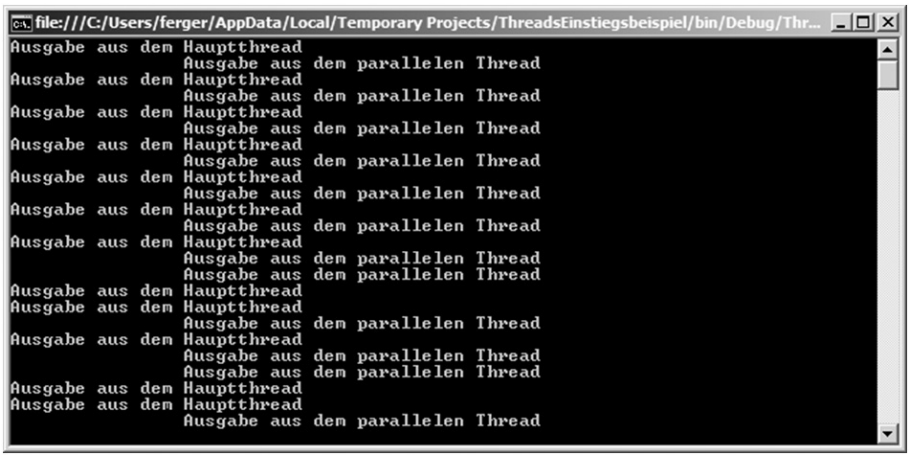


Abb. 1.18: Threads beim Arbeiten

Die Implementierung ist denkbar einfach:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//Die nachfolgende Zeile muss eingefügt werden,
//wenn Threads verwendet werden sollen
using System.Threading;

namespace ThreadsEinstiegsbeispiel
{
    class Program
    {
        static void Main(string[] args)
        {
            //Ein neuer Thread wird erzeugt. Dem ThreadStart wird eine
            //Funktion übergeben. Diese Funktion wird beim Start des
```

```
            //Threads ausgeführt
            Thread thread = new Thread(new ThreadStart(threadRunning));
            //Der Thread wird gestartet.
            thread.Start();
            while (true)
            {
                Console.WriteLine("Ausgabe aus dem Hauptthread");
                Thread.Sleep(200);
            }
        }

        static void threadRunning()
        {
            while (true)
            {
                Console.WriteLine("\t\tAusgabe aus dem parallelen Thread");
                Thread.Sleep(200);
            }
        }
    }
}
```

Das Programm hat den entscheidenden Nachteil, dass der parallele Thread endlos läuft und nur durch ein abruptes Beenden durch den User abgebrochen werden kann. Ein Thread kann durch den Aufruf der Methode *Abort()* beendet werden. Weiterhin sollte der Programmierer dafür sorgen, dass die Endlosschleife des Threads die Möglichkeit bekommt, von außen beendet zu werden. Im nachfolgenden Programm geschieht dies durch eine boolsche Variable *threadStop*:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//Die nachfolgende Zeile muss eingefügt werden,
//wenn Threads verwendet werden sollen
using System.Threading;

namespace ThreadsEinstiegsbeispiel
{
    class Program
    {
        //Die Variable threadStop erhält den Wert false.
        //Sie ist static deklariert, sodass die Main
        //als auch die threadRunning-Methode darauf
        //zugreifen kann.
        static bool threadStop = false;

        static void Main(string[] args)
        {
            //Ein neuer Thread wird erzeugt. Dem ThreadStart wird eine
            //Funktion übergeben. Diese Funktion wird beim Start des
            //Threads ausgeführt
```

```
            Thread thread = new Thread(new ThreadStart(threadRunning));
            //Der Thread wird gestartet.
            thread.Start();
            while (true)
            {
                String text = Console.ReadLine();
                int eingabe = Convert.ToInt16(text);
                if (eingabe == 0)
                {
                    threadStop = true;
                    break;
                }
            }
            //Der Thread wird beendet.
            thread.Abort();
            Console.ReadKey();
         }

        static void threadRunning()
        {
            //Der Thread stoppt, bis die Variable
            //threadStop den Wert true erhält.
            while (!threadStop)
            {
                Console.WriteLine("\t\tAusgabe aus dem parallelen Thread");
                Thread.Sleep(200);
            }
        }
    }
}
```

Das Programm erzeugt diese Bildschirmausgabe:

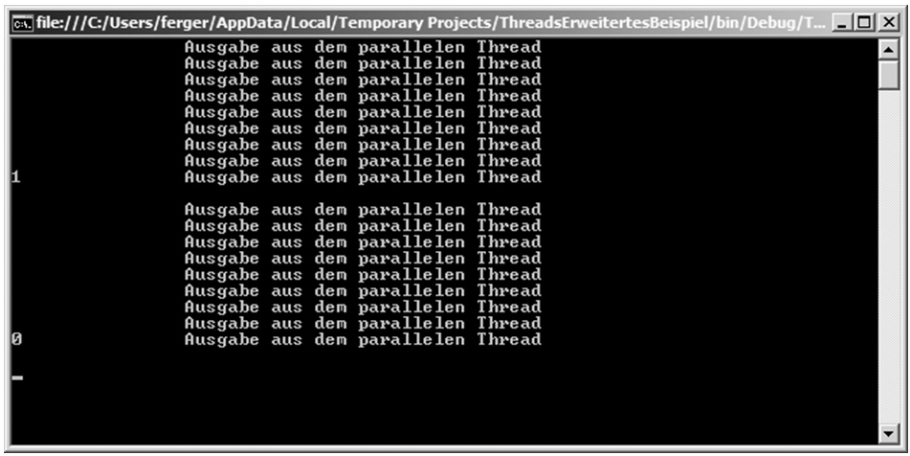


Abb. 1.19: Threads, die beendet werden

Wichtig:

Es gibt noch eine Menge Hinweise bezüglich Threads zu beachten. Um den Umfang des Buchs nicht zu sprengen, sei hier auf die einschlägige Literatur verwiesen (siehe Quellenangabe im Anhang).

1.14 Objektorientierung

Bis jetzt wurden Programme nur strukturiert erstellt. C# ist eine objektorientierte Sprache. Die Vorteile der Objektorientierung liegen in der Wiederverwertbarkeit von Quelltexten und in der sauber modellierten Struktur der Programme.

Grundbegriffe

Klassen:

Klassen sind die syntaktischen Mittel, um abstrakte Datentypen zu realisieren. Diese abstrakten Datentypen haben Eigenschaften (Attribute), aber auch Fähigkeiten (Methoden). In der Klassendeklaration wird der konkrete Bauplan der Klasse festgehalten. In der Klassendefinition wird der Bauplan umgesetzt. Wird der Bauplan umgesetzt, entsteht ein Objekt. Alle Klassennamen beginnen in C# mit einem großen Buchstaben.

Beispiel:

Der IO-Warrior 24 hat zwei digitale 8 Bit-Ports zum Einlesen und Ausgeben von Daten. Der Warrior wird durch eine Gerätenummer identifiziert.

Zur Erstellung der Klasse dieses Bausteins überlegt man nun, welche Daten vorhanden sind und welche Aktionen man mit diesem Interface ausüben können soll. An Daten sollen jeweils der letzte aus- oder eingegebene Zustand der Anschlüsse und die Gerätenummer gespeichert werden[2]. Passende Datentypen sind hier Integervariablen.

Als Aktionen soll der Warrior Daten digital an einen Port ausgeben und einlesen.

In einem Klassendiagramm würde der Warrior jetzt folgendermaßen aussehen:

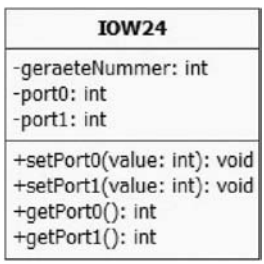


Abb. 1.20: Klassendiagramm IO-Warrior

2 Die Daten der Empfangsbausteine werden deshalb auch als Attribute gekapselt, da später beispielsweise im Hintergrund das Gerät selbstständig Daten an den Eingängen abfragt (siehe Threads).

Zur Erklärung:

In einem Klassendiagramm steht der Klassenname im oberen Bereich, in der Mitte stehen die Attribute und im unteren Teil stehen die Methoden der Klasse.

Im Quelltext sieht diese Klasse folgendermaßen aus:

```
class IOW24
    {
        int geraeteNummer;
        int port0;
        int port1;

        public void setPort0(int value)
        { IOW24.setPort(0, value); }

        public void setPort1(int value)
        { IOW24.setPort(1, value); }

        public int getPort0()
        { return IOW24.getPort(0); }

        public int getPort1()
        { return IOW24.getPort(1); }
    }
```

Anmerkung:

Die Funktionen *IOW24.getPort()* und *IOW24.setPOrt()* sind hier nur aus didaktischen Gründen aufgeführt. Wie der IOW24 tatsächlich auszulesen ist, wird später erläutert.

Will man nun ein solches Objekt erzeugen (instanziieren) und auf die Funktionen des Objektes zugreifen, geschieht dies folgendermaßen:

```
static void Main(string[] args)
   {
       //Ein Objekt wird erzeugt, indem
       //man den entsprechenden Konstruktor aufruft.
       //Das Objekt aus der Klasse IOW24 bekommt den
       //Namen iow.
       IOW24 iow = new IOW24();
       //Mittels des Punktoperators kann man auf
       //die einzelnen Funktionen des Objektes zugreifen.
       iow.setPort0(255);
       int wert = iow.getPort1();
   }
```

1.15 Grafische Benutzeroberfläche

1.15.1 Projekterstellung

Wählen Sie ein neues Projekt und hier als Vorlage die Windows *Forms*-Anwendung. Geben Sie dem Projekt einen Namen und bestätigen Sie mit einem Klick auf den Button *OK*.

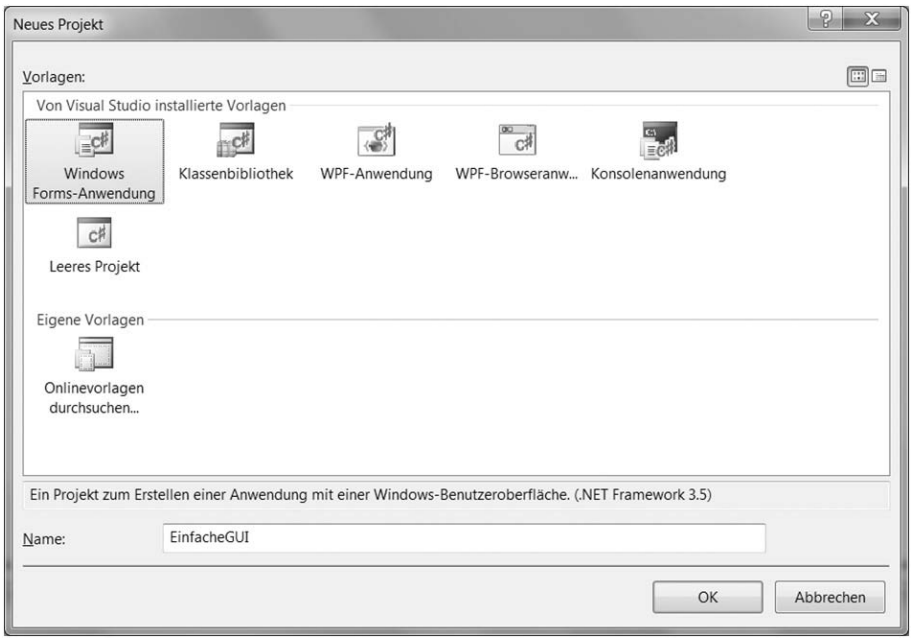


Abb. 1.21: GUI-Projekt erzeugen

Die Entwicklungsumgebung erzeugt nun eine Anwendung mit dem entsprechenden Quelltext. Auf dem Bildschirm erscheint ein Fenster (Form), das Sie nun gestalten können. Auf der linken Seite der Umgebung befinden sich die verschiedensten Steuerelemente, die Sie mit der Maus auf das Fenster ziehen und so Ihre grafische Benutzeroberfläche gestalten können. Auf der rechten Seite der Umgebung finden Sie das Eigenschaftsfenster. Klicken Sie ein Steuerelement auf Ihrer Form an, können Sie in dem Eigenschaftsfenster Änderungen am Steuerelement vornehmen.

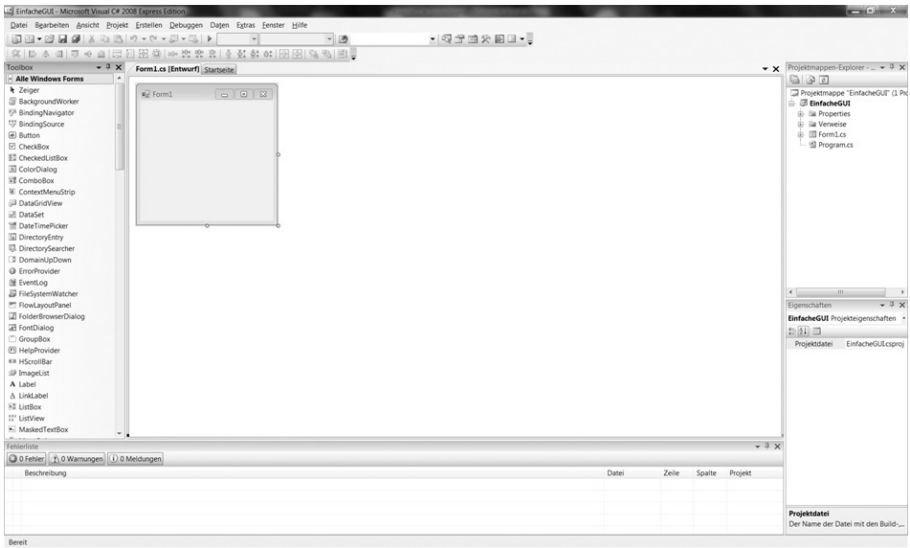


Abb. 1.22: Erste GUI

Das nachfolgende einfache Fenster wurde wie oben angegeben gestaltet.

Abb. 1.23: Gestaltete GUI

Das Programm ist sofort lauffähig, ein Klick auf den grünen Pfeil in der Entwicklungsumgebung startet das Programm. Allerdings funktionieren logischerweise nur die Optionen des Fensters an sich (minimieren, schließen).

Will man dem Programm „Leben“ einhauchen, muss man den Steuerelementen vorgeben, was im Fall eines Klicks geschehen soll.

1.15.2 Events von grafischen Komponenten:

Im Beispielprojekt soll sich beim Klick auf den Button der Text des Labels verändern. Klickt man doppelt auf den Button, öffnet die Umgebung automatisch ein Quelltextfenster an der richtigen Stelle und Sie können hier codieren, welche Aktionen passieren sollen:

```
 namespace EinfacheGUI
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            this.label1.Text = “Hallo GUI-World”;
        }
    }
}
```

Wurde der Quelltext fehlerfrei kompiliert und das Programm gestartet, sollte die gewünschte Aktion ausgeführt werden.

Abb. 1.24: GUI mit Events

Natürlich ist die obige Beschreibung der Programmiersprache und der Entwicklungsumgebung nicht ansatzweise vollständig. Für die Programmierung der Interfaces reichen die angesprochenen Themen jedoch aus. Für weitergehende Informationen sei auf die einschlägige Literatur sowie auf das Internet verwiesen.

2 Die IO-Warrior-Serie

Die Bausteine der Firma Code Mercenaries können als „Eier legende Wollmilchsäue" betrachtet werden. Verschiedenste Varianten dieser Serie von Bausteinen zum Messen, Steuern und Regeln werden angeboten. Alle Bausteine sind in Form eines Starterkits erhältlich. Das Starterkit muss zusammengebaut und gelötet werden, was schnell geschehen ist. Unter der Rubrik *Links* findet man auf der Webseite von Code Mercenaries Firmen, die auch fertig aufgebaute Kits anbieten. Die IO-Warrior sind in den folgenden Varianten verfügbar, die nachfolgend einzeln vorgestellt werden:

- IO-Warrior 24/IO-Warrior Power Vampire
- IO-Warrior 40
- IO-Warrior 56

Allen Bausteinen ist gemeinsam, dass ihre Pins als digitale Ein- oder Ausgänge verwendbar sind. Die Pins müssen nicht in 8-Bit-Ports gruppiert werden, was eine hohe Flexibilität für die Erstellung der eigenen Applikationen/Schaltungen gewährleistet. Neben den einfachen IO-Funktionen beinhalten die Warriors einige Spezialfunktionen (specialfunctions). Industriestandards wie I^2C und SPI sind genauso enthalten wie die Steuerung von LEDs/Schaltermatrizen oder LCD-Displays. Die Bausteine melden sich als HID(Human Interface Device)-Geräte beim Betriebssystem an, sodass keine speziellen Treiber notwendig sind. Im Lieferumfang der Starterkits ist eine CD mit den entsprechenden Programmbibliotheken und Beispielprogrammen für die Sprachen C++, Visual Basic und Delphi enthalten. Microsoft Windows PC, Linux und auch Apple Macs werden unterstützt[3]. Die Bibliotheken, Dokumentation und Beispielprogramme sind alle auch auf der Webseite der Firma Code Mercanaries, http://www.codemercs.com, zu finden. Auch empfiehlt es sich, bei Problemen das Forum der Webseite zu besuchen. Hier sind reichlich Problemlösungen und Hinweise zu finden.

Nachfolgend werden die einzelnen Varianten mit ihren Funktionalitäten vorgestellt:

IO-Warrior24

- 16 I/O Pins, die maximale Leserate beträgt 125 Hz
- IIC-Master-Funktion mit 100 kHz, der Durchsatz beträgt 750 Bytes/sec
- SPI-Master-Schnittstelle mit bis zu 2 MHz, der Durchsatz beträgt 750 Bytes/sec
- Die Ansteuerungsmöglichkeit von HD44780-kompatiblen Displaymodulen und einigen anderen Grafikmodulen ist implementiert.

3 Die Beispiele und Anwendungen in diesem Buch sind ausschließlich für Microsoft-Windows-PCs erstellt worden.

- Ansteuerung einer bis zu 8x32 großen LED Matrix ist möglich
- Empfang von RC5-kompatiblen Infrarot-Fernsteuerungen
- 2 Capture Timer
- Als Gehäuseform sind DIL24 oder SOIC24 verfügbar.

Die Pinbelegung des IOW 24

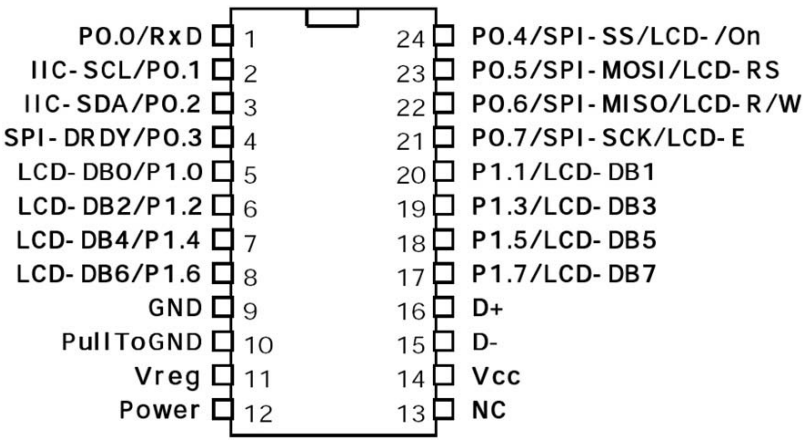


Abb. 2.1: Pinbelegung des IOW24

Beschreibung der Pins:

Pin	Beschreibung
1	Port 0.0 oder serieller Eingang für IR-Signale
2	Port 0.1 oder I^2C – Clock
3	Port 0.2 oder I^2C – Daten
4	Port 0.3 oder SPI DR DY
5	Port 1.0 oder LCD-Datenbit 0
6	Port 1.2 oder LCD-Datenbit 2
7	Port 1.4 oder LCD-Datenbit 4
8	Port 1.6 oder LCD-Datenbit 6
9	Ground
10	Ground
11	3 V geregelt für Pull-up-Widerstand von D-
12	Auswahl, ob USB im Hochstrom oder Niederstrommodus arbeitet

13	Nicht verbunden
14	Spannungsversorgung
15	differenzieller Dateneingang USB D-
16	differenzieller Dateneingang USB D+
17	Port 1.7 oder LCD Datenbit 7
18	Port 1.5 oder LCD Datenbit 5
19	Port 1.3 oder LCD Datenbit 3
20	Port 1.1 oder LCD Datenbit 1
21	Port 0.7 oder SPI Clock, oder LCD Enable
22	Port 0.6 oder SPI Master In Slave Out oder LCD Read/Write
23	Port 0.5 oder SPI Master Out Slave In oder LCD Reset
24	Port 0.4 oder SPI Slave Select oder LCD On

IO-Warrior24 Power Vampire

Eigenschaften des IO-Warrior 24 Power Vampire

- Spezialversion für Stromentnahme aus dem USB
- 4 spezielle Steuerleitungen für Ansteuerung von Power-Management-Chips
- 12 I/O-Pins, typ. 125 Hz Leserate
- IIC-Master-Funktion, 100 kHz, Durchsatz typ. 750 Bytes/sec
- SPI-Master-Schnittstelle mit bis zu 2 MHz, Durchsatz typ. 750 Bytes/sec
- Gehäuse DIL24 oder SOIC24

IO-Warrior40

Eigenschaften des IOW 40:

- 32 I/O-Pins, typische Leserate 125 Hz
- IIC-Master-Funktion, 100 kHz, typischer Durchsatz 750 Bytes/sec
- Ansteuerung von HD44780 kompatiblen Displaymodulen und einigen Grafikmodulen
- Ansteuerung einer bis zu 8x32 großen LED Matrix
- Ansteuerung einer 8x8 Schalter- oder Tastenmatrix
- Gehäuse SSOP48, DIL40 Modul (DIP40-Chip nur noch im Starterkit!)

Die Pinbelegung des IOW 40

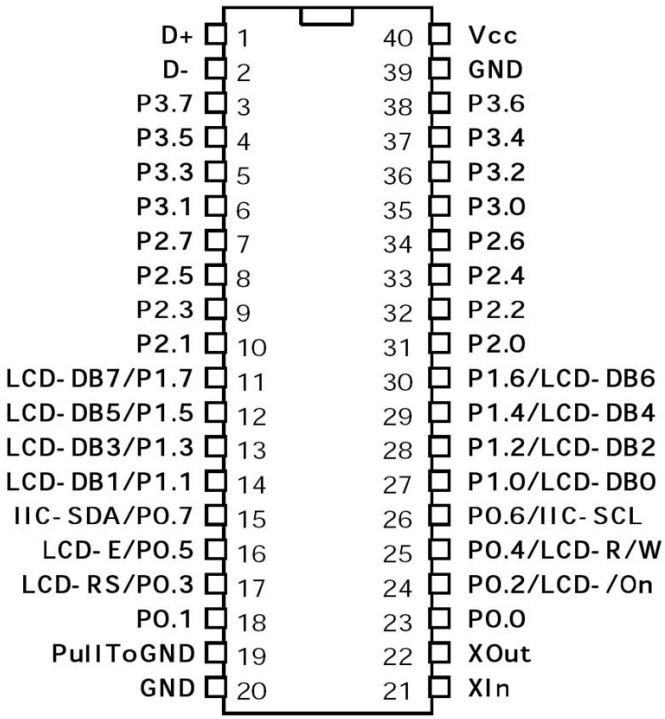


Abb. 2.2: Pinbelegung des IOW40

Die Bedeutung der Pins

Pin	Bedeutung
1	differenzieller Dateneingang USB D+
2	differenzieller Dateneingang USB D-
3	Port 3.7
4	Port 3.5
5	Port 3.3
6	Port 3.1
7	Port 2.7
8	Port 2.5
9	Port 2.3

10	Port 2.1
11	Port 1.7 oder LCD Datenbit 7
12	Port 1.5 oder LCD Datenbit 5
13	Port 1.3 oder LCD Datenbit 3
14	Port 1.1 oder LCD Datenbit 1
15	Port 0.7 oder I^2C Daten
16	Port 0.5 oder LCD Enable
17	Port 0.3 oder LCD Reset
18	Port 0.1
19	Ground
20	Ground
21	Oszillator Ausgang
22	Oszillator Eingang
23	Port 0.0
24	Port 0.2 oder LCD On
25	Port 0.4 oder LCD Read Write
26	Port 0.6 oder I^2C Clock
27	Port 1.0 oder LCD Datenbit 0
28	Port 1.2 oder LCD Datenbit 2
29	Port 1.4 oder LCD Datenbit 4
30	Port 1.6 oder LCD Datenbit 6
31	Port 2.0
32	Port 2.2
33	Port 2.4
34	Port 2.6
35	Port 3.0
36	Port 3.2
37	Port 3.4
38	Port 3.6
39	Ground
40	Spannungsversorgung

IO-Warrior56

Abb. 2.3: IO-Warrior 56

- Full-Speed-USB 2.0 Modus
- 50 I/O-Pins, typische Leserate 1.000 Hz
- IIC-Master-Funktion mit 50, 100 oder 400 kHz
- SPI-Master-Schnittstelle mit bis zu 12 MHz, der Durchsatz kann bis zu 62 KBytes/sec betragen
- Ansteuerung von diversen Displaymodulen inklusive der meisten Grafikmodule
- Ansteuerung einer bis zu 8x64 großen LED Matrix
- Ansteuerung einer 8x8 Schalter- oder Tastenmatrix
- Erweiterter Temperaturbereich: -10 bis +85 °C
- Bauform als Modul oder in MLFP56-Gehäuse

Die Pinbelegung ist dem Datenblatt des IO-Warrior 56 oder dem Anhang zu entnehmen.

Eine Übersicht der Funktionen der IO-Warrior

Type	I/O Pins	LCD	IIC	SPI	RC5 IR	Keys	LEDs	SSOP48	DIL24	SOIC24	MLFP56	Module	Starterkit
IO-Warrior 40	32	√	√	-	-	√	√	√	-	-	-	√	√
IO-Warrior 24	16	√	√	√	√	-	√	-	√	√	-	-	√
IO-Warrior 24 PV	12	-	√	√	-	-	-	-	√	√	-	-	-
IO-Warrior 56	50	√	√	√	√	√	√	-	-	-	√	√	√

2.1 Starterkits

Alle IO-Warriors sind als Starterkits verfügbar.

Starterkit IO-Warrior24

- 1/2 E-Karte (100x80 mm) mit großem Lochrasterfeld
- Alle für den Betrieb des IOW24 notwendigen Bauteile sind auf der Platine vorgesehen.
- Alle Ports mit Beschriftung sind am Lochrasterfeld herausgeführt.
- Ein IR-Empfängermodul (Fernsteuersoftware für Windows und MacOS im SDK) ist integriert.
- Ein Anschluss für ein LCD-Modul ist direkt auf der Platine vorhanden.
- Eine LED an Port 0.3 dient als Anzeige für erste Experimente.
- Die Lieferung erfolgt als Bausatz mit allen Teilen.
- Der Preis beträgt aktuell 49 € zzgl. MwSt.
- Sonderpreise für Schulen und Forschungseinrichtungen können erfragt werden.

Aufbau des Starterkits

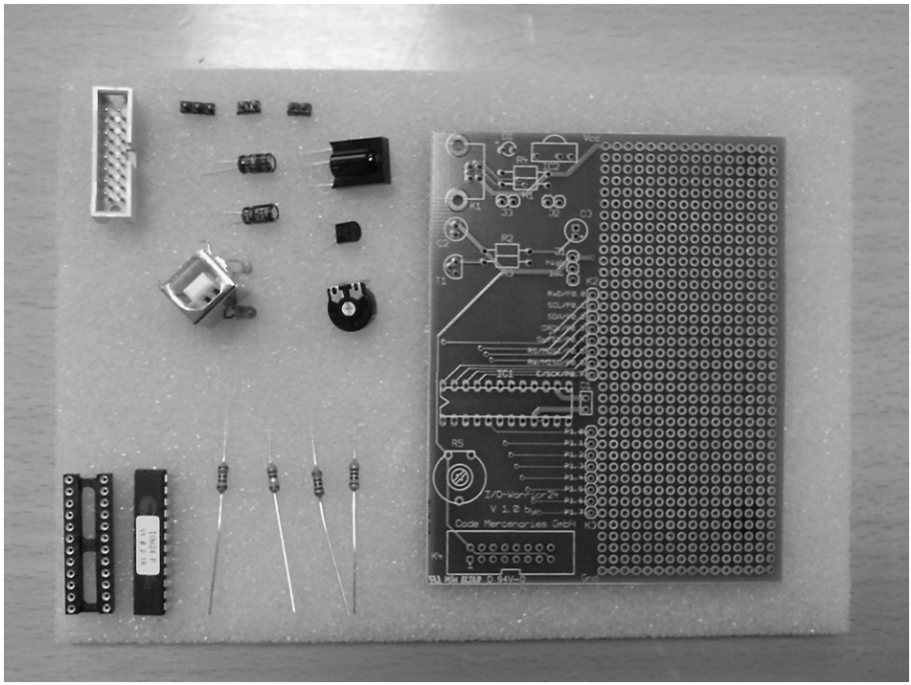


Abb. 2.4: Starterkit IOW24

Benötigt werden ein Lötkolben mit feiner Spitze, etwas Lötzinn und ein Seitenschneider. Sofern die Lötstellen bei den ersten Versuchen nicht sauber werden, bietet es sich an, etwas Entlötlitze zu verwenden. Die Platinenkontakte sind durchkontaktiert, sodass man sie mit einer Lötpumpe nicht immer freibekommt.

Als Erstes lötet man die Widerstände auf die Platine. Hierbei gilt es, die Farbcodes der Widerstände zu beachten:

R1 (1,3k) → braun, orange, schwarz, braun

R2 (100) → braun, schwarz, schwarz, schwarz

R3 (4,7) → gelb, violett, schwarz, silber

R4 (470) → gelb, violett, schwarz, schwarz

Nun erfolgt die Bestückung in folgender Reihenfolge:

IC-Sockel, Keramikkondensator, Potenziometer, LED (der längere Anschluss ist der +/-Pol), Jumper, Transistor, Pfostenstecker, Elektrolytkondensatoren und zuletzt der IR-Empfänger.

Die Jumper können alle gesetzt werden. Jumper 3 ist dafür zuständig, dass die LED zu Testzwecken angesteuert wird. Jumper 2 aktiviert den Datenzugang vom IR-Empfänger zum IOW. Jumper 1 entscheidet, ob der IOW im Hochstrom- oder im Niederstrombetrieb arbeitet.

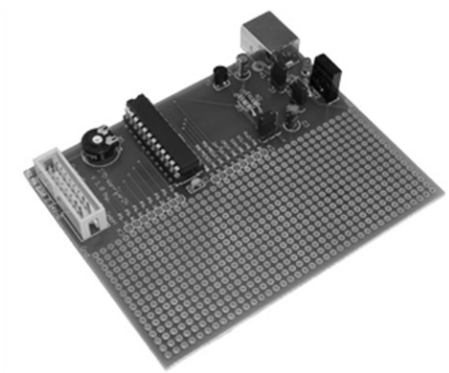


Abb. 2.5: Das fertig aufgebaute Starterkit

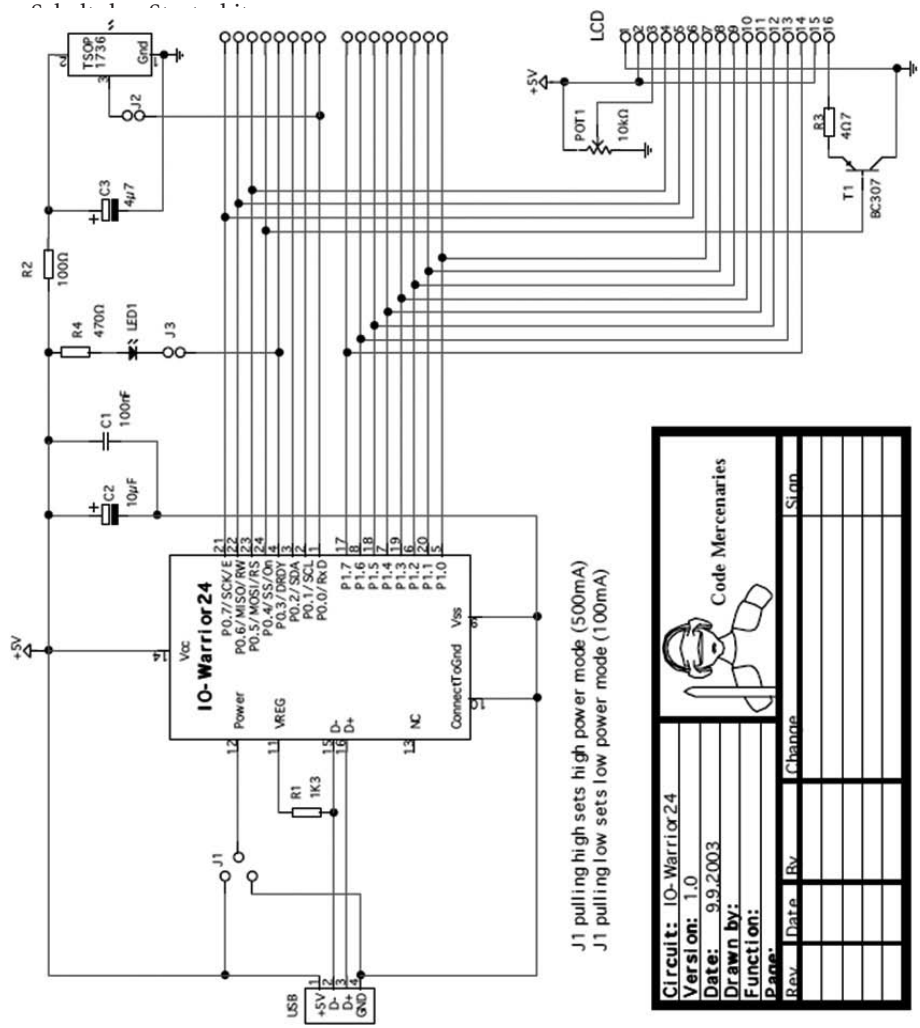


Abb. 2.6: Schaltplan des Starterkits

IO-Warrior40

- Das Starterkit wird auf einer 1/2 E-Karte (100x80 mm) mit großem Lochrasterfeld aufgebaut.
- Alle für den Betrieb des IOW40 notwendigen Bauteile sind auf der Platine vorgesehen.
- Alle Ports mit Beschriftung sind am Lochrasterfeld herausgeführt.
- 8 LEDs an Port 3 ermöglichen erste IO-Experimente.

- Ein Taster an Port 0.0 realisiert eine einfache Eingabe.
- Der aktuelle Preis beträgt 49 € zzgl. MwSt.
- Sonderpreise für Schulen und Forschungseinrichtungen sind zu erfragen. Ein 10er-Pack für Unterrichtszwecke ist ebenfalls verfügbar.

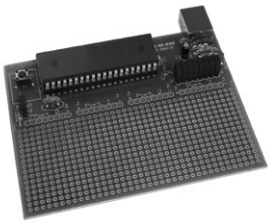


Abb. 2.7: Das fertig aufgebaute Starterkit des IOW40

IO-Warrior56

- Das Starterkit wird auf einer 3/4E-Karte (100x120 mm) mit großem Lochrasterfeld aufgebaut.
- Alle für den Betrieb des IOW56 notwendigen Bauteile sind auf der Platine vorgesehen.
- Ein Sockel für das IOW56-MOD-Modul ist beigefügt.
- Alle Ports mit Beschriftung sind am Lochrasterfeld herausgeführt.
- Eine LED an Port 6.7 realisiert einfache Ausgabeexperimente.
- Ein Taster an Port 6.0 dient als einfaches Eingangssignal.
- Der aktuelle Preis beträgt 69 € zzgl. MwSt.
- Sonderpreise für Schulen und Forschungseinrichtungen sind zu erfragen.

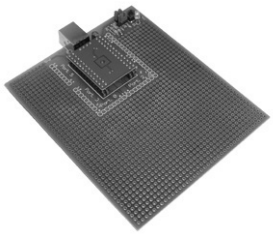


Abb. 2.9: Fertig aufgebautes Starterkit zum IOW56

Schaltplan Starterkit

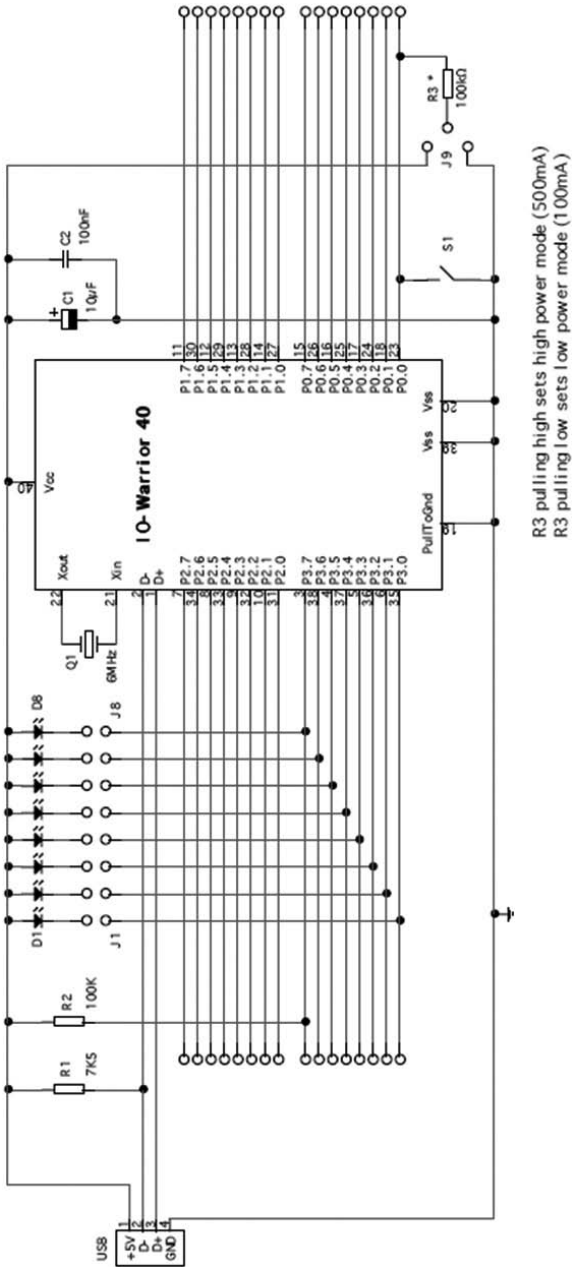


Abb. 2.8: Schaltplan des Starterkits zum IOW40

Schaltplan Starterkit

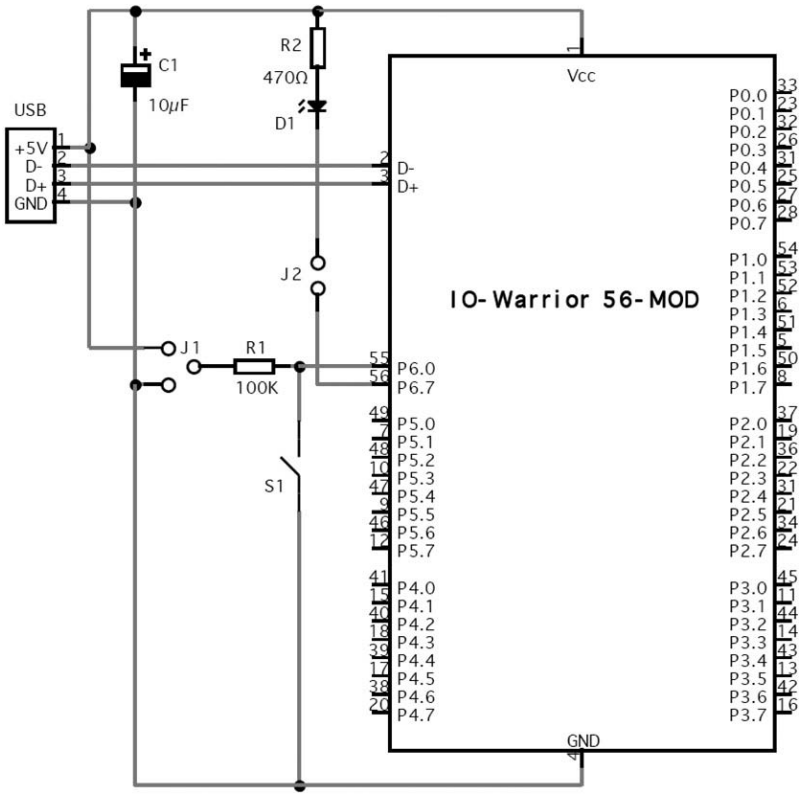


Abb. 2.10: Schaltplan des Starterkits zum IOW56

Die IO-Pins sind oft mehrfach belegt. Verwendet man die entsprechenden Spezialfunktionen, fungieren diese Pins nicht mehr als IO-Pins. Allerdings muss man sich bei der Programmierung keine Gedanken über deren Zustand machen, weil sie dann von IO-Operationen nicht mehr angetastet werden.

2.2 Einen IO-Warrior ansteuern, Kapselung in einer .net-Klasse

Wie immer, wenn man Hardware ansteuern möchte, benötigt man einen entsprechenden Treiber und einen Zugriff auf diesen in Form einer Universalbibliothek oder aber eine DLL, die die benötigten Funktionen zur Verfügung stellt. Die DLL für die IO-

Warriors gibt es auf der Webseite von Code Mercenaries im Downloadbereich. Hat man ein Starterkit gekauft, befindet sich die DLL auf der mitgelieferten CD. Da ständig neue Funktionen zu den Bausteinen hinzukommen und auch an der DLL immer weiter gearbeitet wird, empfiehlt es sich, hier doch öfter mal eine entsprechend aktuelle Version herunterzuladen. Die DLL kann in jedem C#-Programm eingebunden werden.

2.3 Die DLL zum Ansteuern

In der DLL iowkit.dll sind die folgenden Funktionen enthalten:

```
int IowKitOpenDevice ()
```

Die Funktion öffnet alle angeschlossenen IO-Warriors und liefert gleichzeitig einen Handle auf den ersten IO-Warrior zurück. Dies ist sinnvoll, da in den meisten Fällen wohl nur ein IO-Warrior angeschlossen sein wird.

```
IowKitSetLegacyOpenMode()
```

Die Funktion setzt den Modus, mit dem IO-Warriors angesprochen werden. Dies ist bei neueren IO-Warrior-Bausteinen nicht mehr notwendig. Daher wird hier nicht näher auf die Funktion eingegangen.

```
int IowKitGetProductId (int handle)
```

Es wird die Produktnummer des IO-Warriors zurückgegeben. Anhand dieser Nummer kann man im Programm unterscheiden, welcher IO-Warrior angeschlossen ist. Gerade bei mehreren Bausteinen mit unterschiedlichen Funktionalitäten ist dies wichtig. Der übergebene Parameter entspricht dem Handle, mit dem man den IO-Warrior identifiziert.

Der Rückgabewert entspricht 1500 bei einem IO-Warrior 40 und 1501 bei einem IO-Warrior 24.

```
int IowKitGetNumDevs ()
```

Die Funktion gibt die Anzahl der angeschlossenen IO-Warriors zurück. Vorher müssen natürlich die IO-Warriors über die entsprechende Funktion geöffnet werden.

```
int IowKitGetDeviceHandle (int device)
```

Die Funktion gibt eine Nummer (einen Handle) zurück, mit dem ein Baustein später identifiziert werden kann. Der Übergabeparameter entspricht der numerischen Reihenfolge des Bausteins.

```
int IowKitGetRevision (int handle)
```

Die Funktion gibt die Revisionsnummer des angeschlossenen Bausteins zurück. Hiermit kann beispielsweise überprüft werden, ob ein Baustein einer älteren Serie entstammt. Näheres liefert die Dokumentation der Library.

```
bool IowKitGetSerialNumber  (int handle, int & serial)
```

Die Funktion übergibt den Handle und eine Referenz auf eine Integervariable. Besitzt der angeschlossene Baustein eine Seriennummer, wird dies in der Variablen *serial* gespeichert und die Funktion gibt den Wert *true* zurück. Ist dies nicht der Fall, gibt die Funktion *false* zurück.

```
void IowKitCloseDevice(int handle)
```

Die Funktion schließt die angeschlossenen IO-Warrior.

In C#-Programmen ist es zwingend notwendig, diese Funktion am Ende des Programms aufzurufen, da sonst das Programm nicht sauber geschlossen wird.

```
int IowKitRead(IOWKIT_HANDLE devHandle, ULONG numPipe,
PCHAR buffer, ULONG length);
```

Die Funktion liest Daten vom entsprechenden IO-Warrior ein und speichert diese in einem Array. Wichtig:

Die Funktion blockiert das Programm, solange keine Änderungen der Eingangszustände vorliegen. Hierbei werden Zustandsänderungen nicht nur auf die IO-Pins bezogen, sondern auch etwaige Daten, die am I²C-Bus neu anliegen etc. Will man nichtblockierend die Eingänge auslesen, ist die Funktion *IowKitReadNonBlocking* zu verwenden.

```
ULONG IOWKIT_API IowKitReadNonBlocking(IOWKIT_HANDLE devHandle, ULONG numPipe,
PCHAR buffer, ULONG length);
```

Wie oben beschrieben, liest die Funktion die Eingangszustände des angeschlossenen IO-Warriors ein, ohne das laufende Programm zu blockieren.

```
BOOL IOWKIT_API IowKitReadImmediate(IOWKIT_HANDLE devHandle, PDWORD value);
```

Die Funktion liest die Eingangszustände des IO-Warriors ein und speichert das Ergebnis in der Variablen *value*. Allerdings betrifft dies nur die IO-Pins und nicht die aktivierten Spezialfunktionen wie I²C oder SPI.

```
BOOL IOWKIT_API IowKitSetTimeout(IOWKIT_HANDLE devHandle, ULONG timeout);
```

Die Funktion setzt den Lese-Time-out des angeschlossenen IO-Warriors in Millisekunden. Wird diese Zeit überschritten, bricht ein Lesevorgang ab.

```
BOOL IOWKIT_API IowKitSetWriteTimeout(IOWKIT_HANDLE devHandle, ULONG timeout);
```

Die Funktion setzt den Time-out beim Schreiben auf den IO-Warrior. Die Zeit wird in Millisekunden angegeben.

```
BOOL IOWKIT_API IowKitCancelIo(IOWKIT_HANDLE devHandle, ULONG numPipe);
```

Die Funktion bricht Lese- und Schreibvorgänge auf dem IO-Warrior ab.

```
ULONG IOWKIT_API IowKitWrite(IOWKIT_HANDLE devHandle, ULONG numPipe,
PCHAR buffer, ULONG length);
```

Die Funktion schreibt Daten an den IO-Warrior. Je nach dem Wert der Variablen *numPipe* werden hierdurch IO-Pins, aber auch Spezialfunktionen angesprochen.

```
HANDLE IOWKIT_API IowKitGetThreadHandle(IOWKIT_HANDLE iowHandle);
```

Die Funktion liefert eine Information, in welchem Thread Windows die Schreib- und Leseaktionen zum IO-Warrior verwaltet.

```
PCHAR IOWKIT_API IowKitVersion(void);
```

liefert die derzeitige Versionsnummer des IowKits.

2.4 Funktionen einer externen DLL in C# verwenden

Will man die Funktionen einer externen DLL in C# verwenden, muss diese folgendermaßen eingebunden werden:

Die DLL iowkit.dll ist im C#-Projektordner in den Ordner Debug zu kopieren oder sie liegt in einem vom Betriebssystem bekannten Pfad, beispielsweise im Ordner c:\Windows\System32.

Im Quelltext muss im Using-Bereich die Bibliothek *System.Runtime.InteropServices* eingebunden werden:

```
using System.Runtime.InteropServices;
```

Nun können innerhalb der Klasse die Funktionen der DLL eingebunden werden. Hierzu muss bekannt gegeben werden, dass es sich um eine externe Funktion handelt und aus welcher DLL sie stammt:

```
[DllImport(„iowkit“, SetLastError=true)]
public static extern int IowKitOpenDevice();
```

Es ist sehr wichtig, dass die einzelnen Variablentypen in den Funktionen der DLL auf die entsprechenden Variablentypen von C# angepasst werden, da es sonst zu Speicherfehlern kommen kann.

Im nachfolgenden Projekt wird demonstriert, wie einfache Funktionen aus der DLL verwendet werden. Benutzt wird ein IO-Warrior 40. Die IO-Pins von Port 3 werden ein und wieder abgeschaltet.

Der kommentierte Quelltext:

```
using System;
using System.Runtime.InteropServices;

namespace IOWarriorDLLVerwenden
{
    class Program
    {
        //Funktion zum Öffnen des IO-Warriors
        //Die Funktion gibt den handle auf den ersten angeschlossenen
        //IOWarrior zurück
        [DllImport(“iowkit”, SetLastError = true)]
        public static extern int IowKitOpenDevice();

        //Funktion zum Schließen des IO-Warriors
        //Der Funktion wird der handle des zu
        //schließenden IOWarriors übergeben
        [DllImport(“iowkit”, SetLastError = true)]
        public static extern void IowKitCloseDevice(int iowHandle);

        //Funktion zum Schreiben auf den IO-Warrior
        [DllImport(“iowkit”, SetLastError = true)]
        public static extern int IowKitWrite(int iowHandle, int numPipe, ref
byte buffer, int length);

        static void Main(string[] args)
        {
            byte[] daten = new byte[5];
            daten[0] = 0;
            daten[1] = 0;
            daten[2] = 0;
            daten[3] = 0;
            daten[4] = 0;
            //Der IO-Warrior wird geöffnet
            int handle = IowKitOpenDevice();
            //Das Array wird an den IO-Warrior geschrieben
            //Die numPipe ist 0, hiermit werden die einzelnen
            //IO-Pins des Warriors angesprochen
            //Alle LEDs leuchten
            IowKitWrite(handle, 0, ref daten[0], 5);
```

```
            System.Threading.Thread.Sleep(1000);
            daten[4] = 255;
            //Alle LEDS werden wieder ausgeschaltet
            IowKitWrite(handle, 0, ref daten[0], 5);
            //Der IO-Warrior wird geschlossen
            IowKitCloseDevice(handle);
        }
    }
}
```

Natürlich ist es wenig sinnvoll, die einzelnen Funktionen der DLL in jedes neue Projekt einzubinden. C# bietet die Möglichkeit, Klassenbibliotheken zu erstellen. Diese Klassenbibliotheken können einfach in neue Projekte eingebunden und verwendet werden. Die Klassenbibliothek befindet sich inklusive Quelltext auf der CD.

2.5 IO-Warrior identifizieren

Hat man mehrere IO-Warrior am PC angeschlossen, ist es notwendig, im Programm die einzelnen Bausteine unterscheiden zu können. Hierzu kann man sich der Produktnummer der Bausteine bedienen:

IO-Warrior	Produktnummer
IO-Warrior 24	Dezimal 5377, Hexadezimal: 1501
IO-Warrior 40	Dezimal 5376, Hexadezimal: 1500
IO-Warrior 56	Dezimal 5379, Hexadezimal: 1503

Über die oben beschriebenen Funktionen kann man herausfinden, wie viele IO-Warrior angeschlossen sind:

```
int anzahl = IowKitGetNumDevs();
```

Anschließend kann man in einer Schleife jeden einzelnen IO-Warrior ansprechen und die Produktnummer ermitteln:

```
int handle = IowKitGetDeviceHandle(i);
int product = IowKitGetProductId(handle);
```

i entspricht der Nummer des Gerätes, beginnend von 1.

Im nachfolgenden Projekt sind vier IO-Warrior am System angeschlossen.

Der kommentierte Quelltext (die externen Funktionen wurden wie oben beschrieben eingebunden. Das komplette Projekt befindet sich auf der beiliegenden CD):

```
 static void Main(string[] args)
 {
  //Die IO-Warrior werden geöffnet
  IowKitOpenDevice();
  //Die Anzahl der Warrior wird ermittelt
  int anzahl = IowKitGetNumDevs();
  Console.WriteLine("Es sind: " + anzahl + " IO-Warrior angeschlossen.\n");
  for (int i = 1; i <= anzahl; i++)
  {
    string warriorTyp = "";
    //Ein Handle zum Ansprechen des
    //jeweilgen IO-Warriors wird ermittelt
    int handle = IowKitGetDeviceHandle(i);
    //Die Produktnummer des IO-Warriors wird ermittelt
    int product = IowKitGetProductId(handle);
    if (product == 5376)
       warriorTyp = "IO-Warrior 40";
    if (product == 5377)
       warriorTyp = "IO-Warrior 24";
    if (product == 5379)
       warriorTyp = "IO-Warrior 56";
   Console.WriteLine("Es ist ein " + warriorTyp + " mit der ID: " + product+"
angeschlossen");
  }
  //Die IO-Warrior werden geschlossen
  IowKitCloseDevice(0);
  Console.ReadLine();
}
```

Das Programm erzeugt diese Bildschirmausgabe:

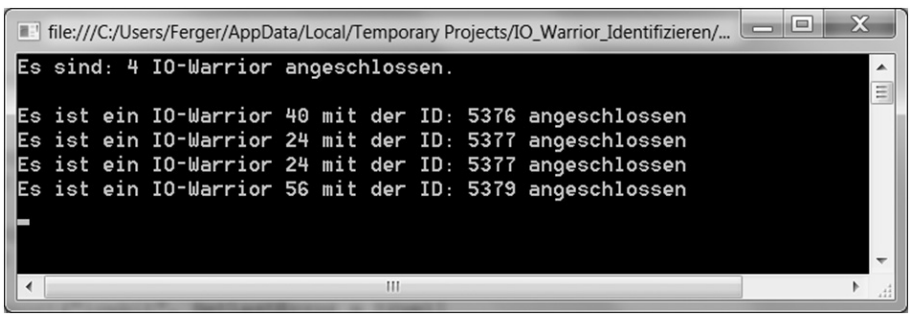


Abb. 2.11: IO-Warrior identifizieren

3 LCD-Display ansteuern und RC5-Fernbedienungscode auslesen

3.1 LCD-Display ansteuern

Oftmals ist es nicht notwendig, einen Monitor anzuschließen, wenn der PC seine messtechnischen oder steuerungstechnischen Aufgaben bewältigt. Gleichwohl kann es sinnvoll sein, Statusmeldungen oder andere kleinere Ergebnisse auf einem kleinen Display auszugeben.

Die IO-Warrior-Serie bietet in allen Varianten die Möglichkeit, sehr einfach ein LCD-Display anzusteuern. Das LCD-Display muss einen HD44780-Controller (Hitachi) oder aber einen zu diesem Standard kompatiblen Controller besitzen. Dies ist bei den meisten im Handel erhältlichen Displays der Fall.

Besonders die IOW24-Serie eignet sich hervorragend für die Ansteuerung eines Displays, da auf dem Starterkit eine Pfostenleiste vorbereitet ist. Über ein einfaches Flachbandkabel und zwei Pfostenstecker kann die Verbindung zwischen Warrior und Display hergestellt werden. Auch sehr viele Displays haben eine solche Pfostenleiste. Bestellnummern und Bezugsadressen sind im Anhang aufgeführt.

Die Programmierung des IO-Warriors ist verhältnismäßig einfach: Dem Datenblatt des IOW entnimmt man, dass über die Report-ID der Wert 4 und im ersten Byte der Wert 1 an den IOW übermittelt werden muss. Man aktiviert somit eine Spezialfunktion des IOW.

Anschließend wird, nachdem die Verbindung zum IOW hergestellt wurde, als Erstes ein 8 Byte großes Array erstellt, in das die entsprechenden Daten geschrieben werden:

```
IntPtr IowHandle = IowKitOpenDevice();
      byte[] daten = new byte[8];
      //Specialmode LCD aktivieren
      daten[0] = 0x4;
      daten[1] = 0x1;
      daten[2] = 0x0;
      daten[3] = 0x0;
      daten[4] = 0x0;
      daten[5] = 0x0;
```

```
        daten[6] = 0x0;
        daten[7] = 0x0;
        IowKitWrite(IowHandle, 1, daten, 8);
```

Wichtig:

Wenn Sie die Specialmode-Funktion des IOW nicht mehr verwenden, sollten Sie sie wieder ausschalten. Hierfür wird das Enableflag wieder auf den Wert 0 gesetzt:

```
  daten[0] = 0x4;
        daten[1] = 0x0; //Schaltet den LCD-Modus wieder aus
        daten[2] = 0x0;
        daten[3] = 0x0;
        daten[4] = 0x0;
        daten[5] = 0x0;
        daten[6] = 0x0;
        daten[7] = 0x0;
        IowKitWrite(IowHandle, 1, daten, 8);
```

Um nun Instruktionen an das Display zu senden, wird die Report-ID auf den Wert 5 gesetzt. Ansonsten sind erst einmal nur noch die beiden Bytes *daten[1]* und *daten[2]* entscheidend (vorausgesetzt, man möchte nur Standardzeichen oder Standardfunktionen an das Display senden).

Das Byte *daten[1]* setzt sich aus folgenden Bits zusammen:

Bit	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Funktion	Anzahl der zu sendenden Bytes	Anzahl der zu sendenden Bytes	Anzahl der zu sendenden Bytes	nicht benutzt, also 0	nicht benutzt, also 0	nicht benutzt, also 0	nicht benutzt, also 0	Auswahl des zu schreibenden Registers (0 für Funktionen, 1 für das Datenregister)

Das Byte *daten[2]* enthält dann das Datenbyte. Dieses setzt sich aus folgender Tabelle zusammen:

Befehl	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Funktion
Clear display	0	0	0	0	0	0	0	0	0	1	Anzeige löschen
Cursor home	0	0	0	0	0	0	0	0	1	*	Platziert den Cursor an DD-RAM-Adresse 0
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	I/D = 1: vorwärts/inkrementieren/rechts I/D = 0: rückwärts/dekrementieren/links S = 1: Die Anzeige wird nach dem Schreiben eines Zeichens jeweils um eine Stelle entsprechend I/D verschoben S = 0: Der Cursor wird nach dem Schreiben eines Zeichens jeweils um eine Stelle entsprechend I/D verschoben.
Display on/off	0	0	0	0	0	0	1	D	C	B	D = 1/0: Display ein/aus C = 1/0: Unterstrich-Cursor ein/aus B = 1/0: Blinkender Cursor ein/aus
Cursor/display shift	0	0	0	0	0	1	S/C	R/L	*	*	Verschiebt die Anzeige (S/C = 1) oder den Cursor (S/C = 0) um eine Stelle nach rechts (R/L = 1) oder nach links (R/L = 0).
System set	0	0	0	0	1	DL	N	F	*	*	DL = 0: 4 Bit Ansteuerung D4 bis D7 DL = 1: 8 Bit Ansteuerung N = 1: 2 oder 4 Displayzeilen N = 0: 1 Displayzeile F = 1: 5x10 Zeichenbox F = 0: 5x7 Zeichenbox

Tabelle von http://www.ulrichradig.de

Befehl	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Funktion
Set CG-RAM address	0	0	0	1	A5	A4	A3	A2	A1	A0	Stellt die Schreibadresse (0 bis 63) in den Zeichengenerator CG-RAM ein. Die nachfolgenden Zugriffe auf das Datenregister greifen auf das CG-RAM zu.
Set DD-RAM address	0	0	1	A6	A5	A4	A3	A2	A1	A0	Stellt die Schreibadresse (0 bis 39, 64 bis103) ins Display DD-RAM ein. Die nachfolgenden Zugriffe auf das Datenregister greifen auf das DD-RAM zu.
Read busy flag/ address counter	0	1	BF	A6	A5	A4	A3	A2	A1	A0	BF = 1: das Display ist beschäftigt/kein Schreib-/Lesezugriff möglich BF = 0: Display bereit/Zugriff möglich A6 bis A0: aktuelle Adresse im CG- oder DD-RAM
Write Data	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Schreibt Daten in CG- oder DD-RAM
Read Data	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Liest Daten aus dem CG- oder DD-RAM

Man beachte, dass das R/W-Bit schon durch die Report-ID des IOW gesetzt ist.

Im folgenden Beispielprogramm werden das Display initialisiert, der Cursor auf die Anfangsposition gesetzt, ein kurzer Text ausgegeben, das Display wieder ausgeschaltet und der IOW zurückgesetzt:

```
IntPtr IowHandle = IowKitOpenDevice();
      byte[] daten = new byte[8];
      //Specialmode LCD aktivieren
      daten[0] = 0x4;
      daten[1] = 0x1;
      daten[2] = 0x0;
      daten[3] = 0x0;
      daten[4] = 0x0;
      daten[5] = 0x0;
      daten[6] = 0x0;
      daten[7] = 0x0;
      IowKitWrite(IowHandle, 1, daten, 8);
      System.Threading.Thread.Sleep(1000); //Eine Sekunde warten

      //Display initialisieren
      daten[0] = 0x5;
      daten[1] = 0x4;
      daten[2] = 0x38;
      IowKitWrite(IowHandle, 1, daten, 8);
      System.Threading.Thread.Sleep(1000);

      //Cursor auf Anfangsposition setzen
      daten[0] = 0x5;
      daten[1] = 0x1;
      daten[2] = 0x2;
      IowKitWrite(IowHandle, 1, daten, 8);

      //Cursor um zwei Stellen nach rechts verschieben
      daten[0] = 0x5;
      daten[1] = 0x1;
      daten[2] = 0x14;

      IowKitWrite(IowHandle, 1, daten, 8);
      IowKitWrite(IowHandle, 1, daten, 8);
      System.Threading.Thread.Sleep(1000);

      //Text im Display ausgeben
      char[] text = new char[40];
text = “Kruemelchen”.ToCharArray(); //aus einem Text wird ein
//char-Array
      for (int i = 0; i < text.Length;i++ )
      {
          daten[0] = 0x5;
          daten[1] = 0x81;
          daten[2] = (byte)text[i];
          IowKitWrite(IowHandle, 1, daten, 8);
      }
```

```
System.Threading.Thread.Sleep(4000);

//Display löschen
daten[0] = 0x5;
daten[1] = 0x1;
daten[2] = 0x1;
IowKitWrite(IowHandle, 1, daten, 8);
System.Threading.Thread.Sleep(1000);


//Display ausschalten
daten[0] = 0x5;
daten[1] = 0x1;
daten[2] = 0x4;
IowKitWrite(IowHandle, 1, daten, 8);
System.Threading.Thread.Sleep(1000);

daten[0] = 0x4;
daten[1] = 0x0;
daten[2] = 0x0;
IowKitWrite(IowHandle, 1, daten, 8);
IowKitCloseDevice(IowHandle);
```

Gibt man einen längeren Text auf einem vierstelligen Display aus, fällt auf, dass er nicht richtig angezeigt wird. Dies liegt daran, dass die einzelnen Zeichenpositionen nicht durchgängig nummeriert sind. Die Adressen der einzelnen Zeichen entnimmt man den entsprechenden Tabellen im Anhang.

3.1.1 Eine fertige .Net-DLL verwenden: die DLL von Christoph Schnedl

Die DLL wurde für den speziellen Zweck geschrieben, einen PC als DVD-Player ohne Monitor zu verwenden. Die DLL wurde um einige Funktionen erweitert. Beispielsweise kann nun der IOW angesprochen werden, ohne dass Specialfunctions aktiviert werden. Weiter kann ein Report zum IOW geschrieben und vom IOW gelesen werden. Diese beiden Methoden ermöglichen es, nicht immer die von Code Mercenaries gelieferte DLL einbinden zu müssen.

Die DLL muss folgendermaßen eingebunden werden:

Im Projektmappenexplorer wird mit der rechten Maustaste auf den Projektnamen geklickt und die Option *Verweis hinzufügen* ausgewählt.

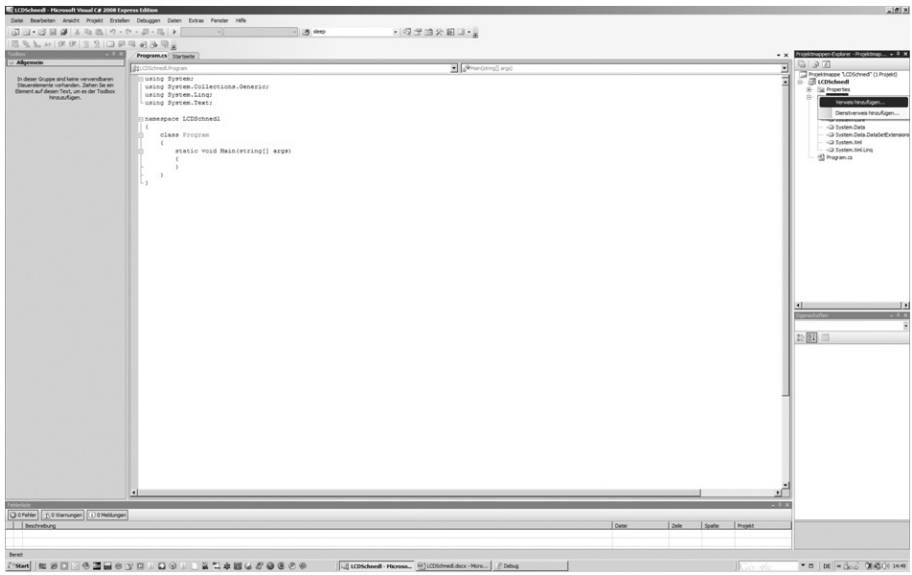


Abb. 3.1: Verweis hinzufügen

Die DLL (sie sollte vorher in den Ordner *Debug* des aktuellen Projekts kopiert werden) wird im Reiter *Durchsuchen* ausgewählt und per Klick auf den Button *OK* ausgewählt.

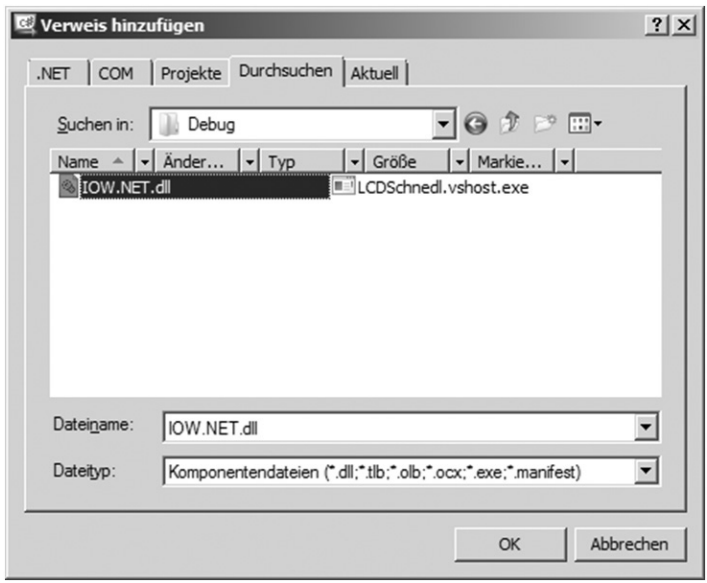


Abb. 3.2: Verweis auswählen

Damit ist die DLL ins Projekt eingebunden.

Um die Funktionen der DLL zu verwenden, muss sie im Using-Bereich eingebunden werden:

```
using IOWarrior;
```

Nun können die Funktionen und Objekte der DLL im Programm verwendet werden.

Zuerst instanziiert man ein Objekt der Klasse *IOWKit*:

```
IOWKit iow = new IOWKit();
```

Über das Objekt *iow* können jetzt alle implementierten Methoden verwendet werden.

Die Methoden der DLL iow.net.dll:

Methode:	Erklärung:
bool ClearLine(int zeile)	Die Zeile *zeile* des Displays wird gelöscht.
bool ClearScreen()	Das LCD-Display wird gelöscht.
bool DisableIR()	Der RC5-Modus wird ausgeschaltet, die Funktion gibt bei Erfolg *true* zurück.
bool EnableIR()	Der RC5-Modus wird eingeschaltet, die Funktion gibt bei Erfolg *true* zurück.
byte GenerateInitialFunctionSetValue()	Funktion zum Umrechnen der Zeilenposition bei Displays.
byte GenerateInitialModeValue()	Initialisierungswerte für die Displays können errechnet werden.
byte GetLCDByteValue(char)	Umrechnungsfunktion für Umlaute und ß.
bool Init(int zeichen, int zeilen)	Der IOW wird angesprochen und der LCD-Mode aktiviert. Übergeben werden die Daten des Displays.
bool Init(int zeichen, int zeilen, bool cursorSichtbar, bool cursorBlinkt)	Der IOW wird angesprochen und der LCD-Mode aktiviert. Übergeben werden die Daten des Displays sowie die Informationen, ob der Cursor sichtbar ist und blinken soll.
void Quit()	Der LCD-Mode wird beendet und die Verbindung zum IOW wird unterbrochen.
void ReadIrKeys()	Diese Funktion liest einen RC5-Code aus, den der IOW empfangen hat. Sie muss nicht aufgerufen werden, da sie intern schon über einen zusätzlichen Thread ständig aufgerufen wird.

Methode:	Erklärung:
int SendReportId(byte, byte, byte, byte, byte, byte, byte, byte)	Ein Report wird an den IOW gesendet.
bool SetCursorMode(bool cursorSichtbar, bool cursorBlinkt)	Es wird übergeben, ob der Cursor sichtbar ist und ob er blinken soll.
bool SetPosition(int zeichen, int zeile)	Die Schreibposition der nächsten Schreiboperation auf dem Display wird übermittelt.
bool UploadSpecialChar(int speicherNummer, bool[][] sonderzeichen)	Ein selbst definiertes Zeichen wird an die Speicheradresse *speicherNummer* (0-7) geschickt. Das Sonderzeichen wird über ein Array of bools definiert (siehe unten).
bool WriteAt(int positionX, int positionX, char zeichen)	Das Zeichen *zeichen* wird an die übergebene Position geschrieben und auf dem Display dargestellt.
bool WriteLine(int zeile, string text)	Der Text *text* wird in der Zeile *zeile* direkt ab Zeilenanfang geschrieben und dargestellt.
bool WriteLine(int zeile, string text, IOWarrior.IOWKit.AlignType)	Der Text *text* wird in der Zeile *zeile* geschrieben und dargestellt. Der Datentyp *AlignType* bestimmt, ob der Text linksbündig (left), zentriert (center) oder rechtsbündig (right) ausgegeben wird.
bool WriteString(string text)	Der Text *text* wird an der vorher definierten Position auf dem Display dargestellt.

Ein einfaches LCD-Projekt

Im nachfolgenden Projekt werden ein an den IO-Warrior angeschlossenes LCD-Display angesteuert und drei Zeilen ausgegeben. Drückt der Benutzer anschließend die *Enter*-Taste, werden die Ansteuerung und das Programm beendet.

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//Einbinden der Funktionen aus der DLL
using IOWarrior;

namespace EinfachesLCDProjekt
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            //Der IOWarrior wird instanziiert
            IOWKit iow = new IOWKit();
            //Der IOWarrior wird geöffnet und
            //der LCD-Mode aktiviert.
            //Es wird ein Display mit 20 Zeichen
            //und vier Zeilen angesprochen
            iow.Init(20, 4);
            //Drei Texte werden ausgegeben
            iow.WriteLine(1, "Sophia");
            iow.WriteLine(2, "Paul");
            iow.WriteLine(3, "Krümelchen");
            //Auf eine Eingabe des Benutzers wird
            //gewartet
            Console.ReadKey();
            //Der Displaymodus wird deaktiviert
            //und der IO-Warrior geschlossen
            iow.Quit();
        }
    }
}
```

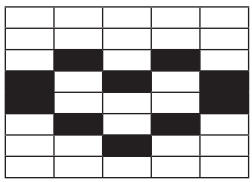
Abb. 3.3: Angesteuertes LCD-Display

3.1.2 Sonderzeichen selbst definieren, in das Display laden und wieder aufrufen

Soll ein eigenes definiertes Zeichen dargestellt werden, muss zuerst ein zweidimensionales Feld vom Datentyp *bool* deklariert und gefüllt werden:

```
bool[][] zeichen = new bool[5][];

for (int i = 0; i < 5; i++)
 zeichen[i] = new bool[8];
```

Dieses Feld entspricht einer Punktmatrix. Ein Element des Felds erhält den Wert *true*, sofern es schwarz dargestellt werden soll, ansonsten bekommt es den Wert *false*. Nachfolgend wird ein Sonderzeichen in Form eines Herzens erstellt:



Die letzte Zeile sollte auf *false* gesetzt werden, da sie für den Cursor reserviert ist.

Im Quelltext sieht die Wertezuweisung folgendermaßen aus:

```
zeichen[0][0] = false; zeichen[0][1] = false; zeichen[0][2] = false;
zeichen[0][3] = true; zeichen[0][4] = true; zeichen[0][5] = false;
zeichen[0][6] = false; zeichen[0][7] = false;

zeichen[1][0] = false; zeichen[1][1] = false; zeichen[1][2] = true;
zeichen[1][3] = false; zeichen[1][4] = false; zeichen[1][5] = true;
zeichen[1][6] = false; zeichen[1][7] = false;

zeichen[2][0] = false; zeichen[2][1] = false; zeichen[2][2] = false;
zeichen[2][3] = true; zeichen[2][4] = false; zeichen[2][5] = false;
zeichen[2][6] = true; zeichen[2][7] = false;

zeichen[3][0] = false; zeichen[3][1] = false; zeichen[3][2] = true;
zeichen[3][3] = false; zeichen[3][4] = false; zeichen[3][5] = true;
zeichen[3][6] = false; zeichen[3][7] = false;

zeichen[4][0] = false; zeichen[4][1] = false; zeichen[4][2] = false;
zeichen[4][3] = true; zeichen[4][4] = true; zeichen[4][5] = false;
zeichen[4][6] = false; zeichen[4][7] = false;
```

Anschließend wird das Sonderzeichen hochgeladen:

```
iow.UploadSpecialChar(1, zeichen);
```

In diesem Fall wurde das Sonderzeichen an die Speicherposition 1 geschickt, möglich sind hier die Positionen 0 bis 7. Nachdem ein Sonderzeichen hochgeladen wurde, werden auf dem Display alle acht Speicherplätze ausgegeben. Dies wurde im Quellcode von Christoph Schnedl so implementiert. Da die DLL aber quelloffen ist, sollte es kein Problem sein, diese Darstellung herauszunehmen.

Das Sonderzeichen kann einfach mit der Methode *WriteLine* dargestellt werden, das Sonderzeichen an der Speicherstelle x muss dann mittels des Strings {SPx} übergeben werden.

3.2 RC5-Code abfragen

3.2.1 Der RC5-Code

Der Code, ursprünglich von der Firma Philips entwickelt, besteht aus 14 seriell gesendeten Bits. Hierunter fallen ein Startbit, ein Togglebit, 5 Adressbits (die eine Gerätenummer bilden) und 7 Kommandobits. Die Bits sind in der Reihenfolge etwas durcheinander, da das 7. Kommandobit ursprünglich ein weiteres Startbit war. Aufgrund der gestiegenen Funktionsvielfalt von Geräten wurde das Startbit durch ein weiteres Kommandobit ersetzt. Ein übertragenes Signal verteilt sich folgendermaßen:

Startbit		Tooglebit	Adressbits					Kommandobits					
1	$\overline{C6}$	1 oder 0	A4	A3	A2	A1	A0	C5	C4	C3	C2	C1	C0

Abb. 5: Aufbau RC5-Code

Das Bit $\overline{C6}$ ist negiert, das Tooglebit wechselt bei jeder Übertragung seinen Wert.

Die gängigen Fernsehfernbedienungen unterstützen diesen Code. Mit dem IOW24 kann recht einfach der Code eines Fernbedienungsknopfes ausgewertet und anschließend zur Steuerung anderer Ereignisse genutzt werden.

Im nachfolgenden Programm wird Word mithilfe einer Fernbedienung gestartet. Die Fernbedienung wurde so eingestellt, dass die Geräteadresse 10 ist (zweiter Satellitenreceiver) und der Programmknopf 1 gewählt wurde.

Einige Gerätenummern können der folgenden Tabelle entnommen werden[4]:

Gerät	TV1	TV2	Videotext	Videorekorder 1
Nummer	0	1	2	5
Gerät	Videorekorder 2	Satreceiver 1	Satreceiver 2	CD Player
Nummer	6	8	10	20

4 Quelle: http://www.sprut.de/electronic/ir/rc5.htm

Die Kommandos entsprechen von 0 bis 10 den Programmtasten.

Ihre Geräteadresse können Sie natürlich auch von dem oben erwähnten Beispielprogramm auslesen lassen: Einfach das Programm starten und eine Fernbedienungstaste drücken. Auf diese Weise erfahren Sie auch, ob Ihre Fernbedienung RC5-kompatibel ist!

3.2.2 Beispielprogramm zum RC5-Code:

Im nachfolgend aufgeführten Programm wird ein Text auf dem Display ausgegeben, der Bildschirm wird gelöscht, ein Sonderzeichen wird definiert und auf dem Display ausgegeben. Zeitgleich empfängt der IOW parallel RC5-Codes und gibt diese auf der Konsole aus. Beim Druck auf die Taste 1 der Fernbedienung wird Word gestartet (Näheres hierzu im Kapitel „Erweiterte Programmierung“). Wird auf der Fernbedienung die Power-Off-Taste betätigt, wird das Programm beendet.

Das fertige Programm:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;
using Word = Microsoft.Office.Interop.Word;

namespace LCDSchnedl
{
    class Program
    {
        static IOWKit iow;

        static void Main(string[] args)
        {
            //Erzeugung eines Iow-Objekts
            iow = new IOWKit();
           // Den IO-Warrior ansprechen und den Specialmode Display aktivie-
ren
            iow.Init(20, 4);
            //Der RC5-Specialmode wird aktiviert
            iow.EnableIR();
            //Ein empfangener RC5-Code löst die Funktion irAusgabe aus
            iow.OnIREvent += new IOWKit.IREventHandler( irAusgabe );

            //Einen einfachen Text auf das Display ausgeben
            iow.WriteLine(1, “Nordhessen, wo”);
            iow.WriteLine(2, “jeder wohnen will!”);
            System.Threading.Thread.Sleep(2000);

           //Ab hier wird ein Sonderzeichen definiert und ins Display geladen
            bool[][] zeichen = new bool[5][];
```

```
            for (int i = 0; i < 5; i++)
                zeichen[i] = new bool[8];

            zeichen[0][0] = false; zeichen[0][1] = false; zeichen[0][2]
= false; zeichen[0][3] = true; zeichen[0][4] = true; zeichen[0][5]
= false; zeichen[0][6] = false; zeichen[0][7] = false;
            zeichen[1][0] = false; zeichen[1][1] = false; zeichen[1][2]
= true; zeichen[1][3] = false; zeichen[1][4] = false; zeichen[1][5]
= true; zeichen[1][6] = false; zeichen[1][7] = false;
            zeichen[2][0] = false; zeichen[2][1] = false; zeichen[2][2]
= false; zeichen[2][3] = true; zeichen[2][4] = false; zeichen[2][5]
= false; zeichen[2][6] = true; zeichen[2][7] = false;
            zeichen[3][0] = false; zeichen[3][1] = false; zeichen[3][2]
= true; zeichen[3][3] = false; zeichen[3][4] = false; zeichen[3][5]
= true; zeichen[3][6] = false; zeichen[3][7] = false;
            zeichen[4][0] = false; zeichen[4][1] = false; zeichen[4][2]
= false; zeichen[4][3] = true; zeichen[4][4] = true; zeichen[4][5]
= false; zeichen[4][6] = false; zeichen[4][7] = false;
            iow.ClearScreen();

            System.Threading.Thread.Sleep(2000);
            //Das Zeichen wird ins Display geladen
            iow.UploadSpecialChar(1, zeichen);
            iow.ClearScreen();
            System.Threading.Thread.Sleep(2000);
            //Das Sonderzeichen auf dem Display ausgeben
            iow.WriteLine(1,"{SP1}");
            System.Threading.Thread.Sleep(2000);

            //Der IO-Warrior wird nach Taste Eingabe
            //abgemeldet und das Display ausgeschaltet
            Console.ReadKey();
            iow.Quit();
        }

        //Die folgende Methode wird ausgeführt, sobald ein RC5-Code eintrifft
        static void irAusgabe(object sender, IREventArgs e)
        {
            //Das empfangene RC5-Signal wird auf dem Bildschirm ausgegeben
            Console.WriteLine(e.remoteControlID + " " + e.keyPressed + " " +
e.toggleBit);
            //Power off fordert den User auf,
            //das Programm per Eingabe zu beenden
            if (e.keyPressed == 12)
            {
                  Console.WriteLine("Programm wird nach einer Tasteneingabe
beendet");
                //Der IR-Mode wird beendet
                iow.DisableIR();
            }
            if (e.keyPressed == 1)
            {
                //Wurde eine 1 auf der Fernbedienung gedrückt,
                //wird Word gestartet
```

```
                Console.WriteLine("Word wird gestartet");
                  Word.Application word = new Microsoft.Office.Interop.Word.
Application();
                word.Visible = true;
            }
        }
    }
}
```

Eine mögliche Bildschirmausgabe:

Abb. 3.4: Einen PC fernsteuern

4 IOW-Ausgänge ansprechen

Die einzelnen Pins des IOW anzusprechen, ist mit wenigen Quelltextzeilen zu erreichen. Der Verweis auf die IOW.net.DLL wird hergestellt.

Zuerst instanziiert man ein Objekt der Klasse *IOWKit* und stellt mit der Methode *open()* die Verbindung zum IOW her:

```
IOWKit iow = new IOWKit();
iow.Open();
```

Ein Byte-Array mit fünf Elementen (beim IOW24 reicht ein Byte-Array mit drei Elementen) wird erstellt und mit den entsprechenden Daten gefüllt. Element 0 enthält den Wert 0. Die anderen vier Elemente stehen für je einen Port, an den die Daten von 0 bis 255 gesendet werden sollen.

Beispiel: Alle Ausgänge auf LOW setzen:

```
senden[0] = 0;
senden[1] = 0;
senden[2] = 0;
senden[3] = 0;
senden[4] = 0;
iow.SendReport(0, ref senden[0], 5);
```

Beispiel: Alle Ausgänge auf HIGH setzen

```
senden[0] = 0;
senden[1] = 255;
senden[2] = 255;
senden[3] = 255;
senden[4] = 255;
iow.SendReport(0, ref senden[0], 5);
```

Der kommentierte Quelltext des Programms:

Ausgänge setzen:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace Ausgänge_Setzen
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            iow.Open();
            //Array vorbereiten, alle Ausgänge
            //haben High Signal -> LEDs aus!!
            byte[] senden = new byte[5];
            senden[0] = 0;
            senden[1] = 255;
            senden[2] = 255;
            senden[3] = 255;
            senden[4] = 255;
            iow.SendReport(0, ref senden[0], 5);
            System.Threading.Thread.Sleep(2000);

            //Array vorbereiten, alle Ausgänge
            //haben Low Signal -> LEDs an!!
            senden[0] = 0;
            senden[1] = 0;
            senden[2] = 0;
            senden[3] = 0;
            senden[4] = 0;
            iow.SendReport(0, ref senden[0], 5);
            System.Threading.Thread.Sleep(2000);

            //LEDs blinken
            for (int i = 0; i < 256; i++)
            {
                senden[4] = (byte)i;
                iow.SendReport(0, ref senden[0], 5);
                System.Threading.Thread.Sleep(20);
                Console.Write(senden[4]);

            }
            Console.ReadKey();
            iow.Quit();
        }
    }
}
```

4.1 Ansteuern von Relais:

Bis zu einer definierten Stromstärke können Relais vom IO-Warrior direkt angesteuert werden. Hierbei ist zu beachten, dass die Relaisspule eine induktive Last darstellt und der Ausgang des IO-Warriors durch eine antiparallel geschaltete Diode geschützt werden muss.

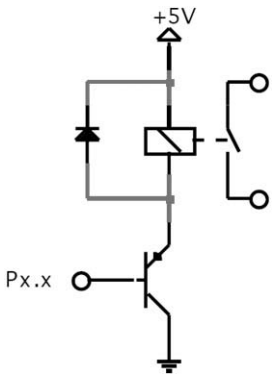


Abb. 4.1: Ein Relais direkt ansteuern[5]

Besser erscheint es doch, einen Treiberbaustein wie den ULN2803 zu verwenden. Der ULN hat acht Eingänge und acht Ausgänge. Gibt man an einen Eingang ein High-Signal, wird der entsprechende Ausgang auf Masse gezogen und kann jeweils 500 mA schalten, was für die meisten einfachen Relais ausreichen dürfte. Nachfolgend eine Schaltung zur Ansteuerung eines Gleichstrommotors mit Rechts-Links-Lauf.

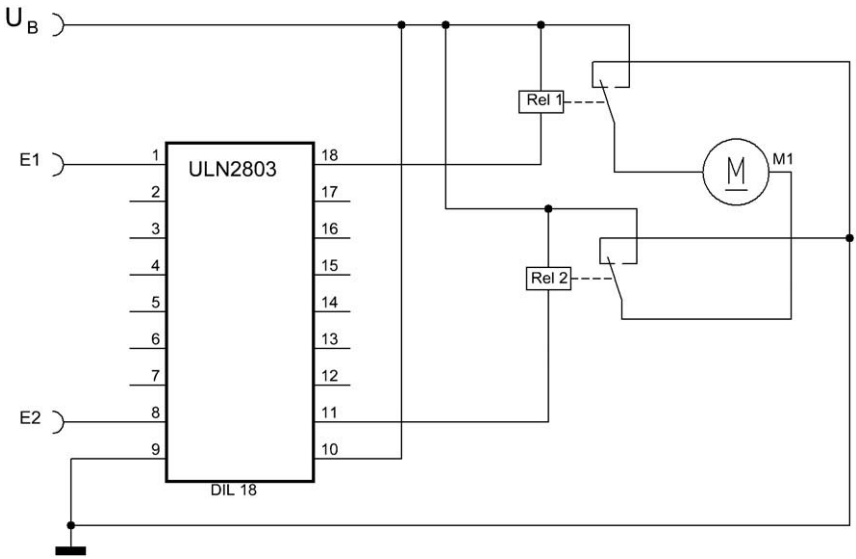


Abb. 4.2: Ansteuern von Relais mittels ULN2803

5 Schaltung aus dem Datenblatt der IO-Warrior

4.2 Einen Schrittmotor ansteuern

Schrittmotoren gibt es in verschiedenen Varianten. Hier werden unipolare und bipolare Schrittmotoren angesteuert. Schrittmotoren haben zwei Hauptwicklungen. Beim bipolaren Schrittmotor haben die einzelnen Wicklungen noch eine Mittelanzapfung.

Schrittmotoren sind günstig zu erwerben. Ein Blick auf die Webseiten der aktuellen Elektronikhändler oder den Restpostenbereich kann hilfreich sein.

Zur Ansteuerung der Schrittmotoren kann ein einfacher Treiberbaustein wie der ULN 2803 reichen, sofern der maximale Strom des Bausteins nicht überschritten wird.

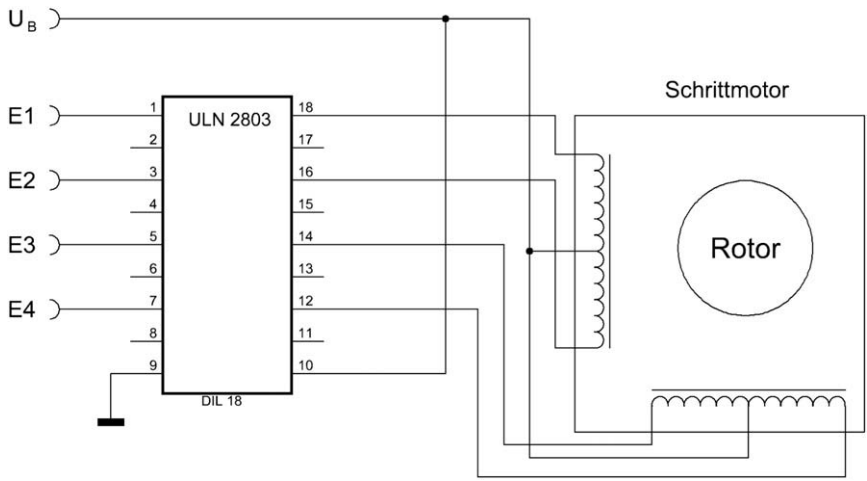


Abb. 4.3: Schrittmotor, bipolar ohne Relais

Ist dies doch der Fall, muss nach dem Ausgang des ULN2803 ein weiterer Treiber in Form eines Leistungstransistors oder Treibers folgen.

4.2.1 Das Prinzip des Schrittmotors

Beim Schrittmotor ist der Rotor ein Magnet. Die Spulen erzeugen nun je nach Ansteuerung ein gerichtetes Magnetfeld. Der Rotor richtet sich nach dem erzeugten Magnetfeld aus. Man bringt nun durch eine geeignete Schaltreihenfolge den Rotor dazu, sich schrittweise zu drehen. Dies kann im Halb- und im Vollschrittbetrieb geschehen. Genaue Anleitungen zur Schrittfolge und nähere Informationen zum Schrittmotor findet man unter *http://www.roboternetz.de/wissen/index.php/Schrittmotoren*.

4.2.2 Motoransteuerung mit dem L293D

Der L293D ist ein vierfacher Treiber, der Ströme bis 600 mA pro Treiber schalten kann. Zwei Treiber können jeweils einen Gleichstrommotor ansteuern, ein Schrittmotor kann ebenfalls über die vier Treiber angeschlossen werden. Interessant an dem Baustein ist die Tatsache, dass Treiber 1 und 2 von einem Enablepin freigeschaltet werden müssen. Hierdurch ist ohne Weiteres eine Pulsweitenmodulation zur Regulierung der Geschwindigkeit möglich.

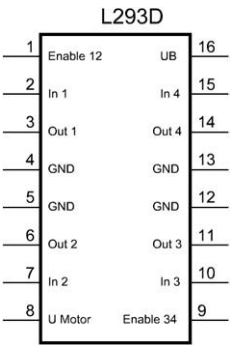


Abb. 4.4: Motortreiber L293D

Zur Funktionsweise:

Enable 12 schaltet Out 1 und Out 2 frei. Liegt dann an In 1 ein High-Signal an, schaltet Out 1 auf die Spannung U_{Motor}. U_B versorgt den IC mit Betriebsspannung und liegt bei um die 5 V. Besitzt man nur eine Spannungsquelle, können U_B und U_{Motor} auch verbunden werden. Laut Datenblatt verträgt der IC 36 V. Analog zur Abhängigkeit In 1 → Out 1 fungieren die Pins In 2 → Out 2, In 3 → Out 3 und In 4 → Out 4. Enable 34 schaltet die Ausgänge Out 3 und Out 4 frei. Alle Eingänge sind TTL-kompatibel und können mit dem IO-Warrior verwendet werden.

In der folgenden Schaltung wird über einen Jumper (J1) entweder U_B auf U_{Motor} gelegt oder aber U_{Motor} extern hinzugefügt:

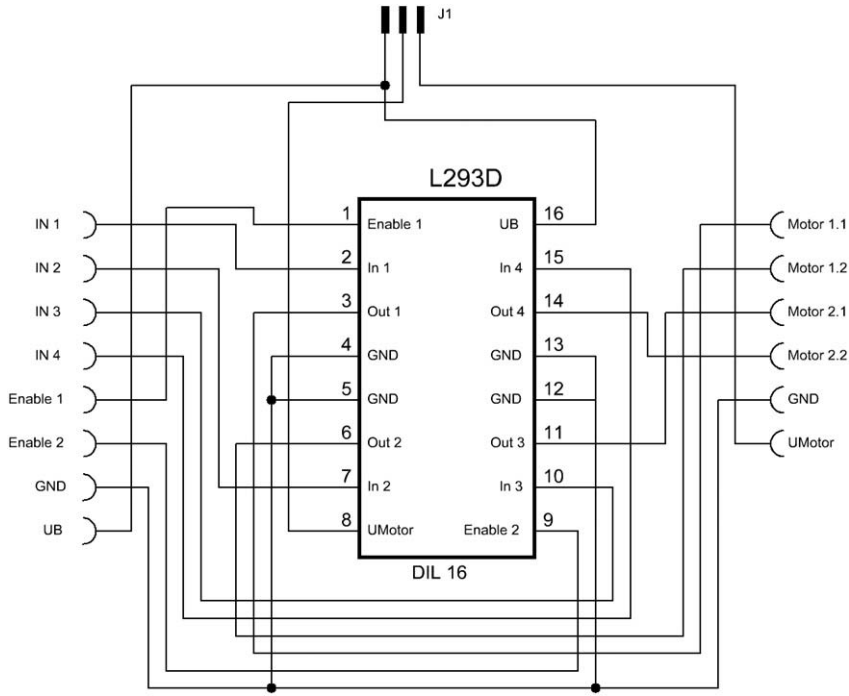


Abb. 4.5: Schaltung Motorsteuerung mit den L293D

Programm zur Ansteuerung des Schrittmotors

Verwendet wird ein IO-Warrior 56. Die vier Eingänge des IC sind mit den IO-Pins *Port0.0* bis *Port 0.3* verbunden. Die beiden *Enable*-Eingänge werden über die IO-Pins *Port1.0* und *Port1.1* ständig auf High geschaltet. Zwischen den einzelnen Schritten des Motors muss eine minimale Wartezeit (5 ms) eingeplant werden, da der Rotor des Motors träge ist. Über diese Wartezeit könnte später die Geschwindigkeit des Motors geregelt werden.

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace SchrittmotorBipolar
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            //Eine IO-Warriorinstanz anlegen
            IOWKit iow = new IOWKit();
            //Den IO-Warrior öffnen
            iow.Open();
            //Das Bytearray wird vorbereitet
            byte[] senden = new byte[8];
            senden[0] = 0;
            //Port 0
            senden[1] = 255;
            //Port 1 wird auf High gesetzt (Enable)
            senden[2] = 255;
            senden[3] = 255;
            senden[4] = 255;
            senden[5] = 255;
            senden[6] = 255;
            for (int i = 0; i < 1000; i++)
            {
                senden[1] = 5;
                iow.SendReport(0, ref senden[0], 8);
                Console.WriteLine(senden[1]);
                //Die Wartezeit ist erforderlich,
                //da der Rotor des Motors träge reagiert
                System.Threading.Thread.Sleep(5);
                senden[1] = 9;
                iow.SendReport(0, ref senden[0], 8);
                Console.WriteLine(senden[1]);
                System.Threading.Thread.Sleep(5);
                senden[1] = 10;
                iow.SendReport(0, ref senden[0], 8);
                Console.WriteLine(senden[1]);
                System.Threading.Thread.Sleep(5);
                senden[1] = 6;
                iow.SendReport(0, ref senden[0], 8);
                Console.WriteLine(senden[1]);
                System.Threading.Thread.Sleep(5);
             }
          iow.Quit();
        }
    }
}
```

4.3 IOW: digitale Eingänge einlesen

Die Eingänge eines IOW sind folgendermaßen einzulesen: Zuerst werden die ausgesuchten Eingänge über die Ausgabefunktion (SendReport) auf den Wert *HIGH* gesetzt. Anschließend erzeugt der IOW dann einen Report, wenn sich der Status an einem Pin ändert. Dieser Report kann auf zwei Arten eingelesen werden. Die Funktion

GetReport() liest nur dann ein, wenn ein Report vorliegt. Dies bedeutet, dass das Programm wartet (blockiert), bis ein neuer Report vorliegt. Die Funktion *GetReportImmediat()* liefert einen Report zurück, auch wenn sich im IOW an den Pins kein Zustand geändert hat.

Der eingelesene Report liefert die Eingangszustände portweise in Form eines Bytes zurück. Mithilfe der logischen Funktionen (Maskieren etc.) können dann einzelne Bits und somit einzelne Pins herausgefiltert werden.

Im nachfolgenden Programm werden die nichtblockierende und die blockierende Variante realisiert:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace Eingänge_lesen
{
    class Program
    {
        static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            iow.Open();
            //Array vorbereiten, alle Ausgänge
            //haben High-Signal
            byte[] senden = new byte[5];
            senden[0] = 0;
            senden[1] = 255;
            senden[2] = 255;
            senden[3] = 255;
            senden[4] = 255;

            //Array zum Einlesen vorbereiten
            byte[] empfangen = new byte[5];
            empfangen[0] = 0;
            empfangen[1] = 0;
            empfangen[2] = 0;
            empfangen[3] = 0;
            empfangen[4] = 0;

            //2000 mal wird HIGH auf alle
            //Pins geschrieben und anschließend
            //nicht blockierend eingelesen
            for (int i = 0; i < 2000; i++)
            {
                Console.Clear();
                Console.WriteLine("Eingänge lesen ohne Blockieren");
                iow.SendReport(0, ref senden[0], 5);
```

```
                iow.GetReportImmediat(0, ref empfangen[0], 5);
            Console.Write(empfangen[1] + "\t"+empfangen[2] + "\t"+empfangen[3]
 + "\t"+empfangen[4] + "\n");
                System.Threading.Thread.Sleep(10);
            }

            //20 mal wird HIGH auf alle
            //Pins geschrieben und anschließend
            //blockierend eingelesen
              for (int i = 0; i < 20; i++)
            {
                Console.Clear();
                Console.WriteLine("Eingänge lesen mit Blockieren");
                iow.SendReport(0, ref senden[0], 5);
                iow.GetReport(0, ref empfangen[0], 5);
                  Console.Write(empfangen[1] + "\t" + empfangen[2] + "\t" +
empfangen[3] + "\t" + empfangen[4] + "\n");
                System.Threading.Thread.Sleep(100);
            }
            Console.ReadKey();
            iow.Quit();
        }
    }
}
```

Schutzhinweise:

Die Eingänge des IO-Warriors können Spannungsspitzen ausgesetzt werden. Hier wird entweder der Schutz über zwei Dioden oder aber der Schutz über eine galvanische Trennung beispielsweise über entsprechende Optokoppler empfohlen. Hierzu wird der IC LTV 845 verwendet. Dieser realisiert vier optische Kopplungen in einem IC, für 16 Eingänge benötigt man also vier dieser ICs.

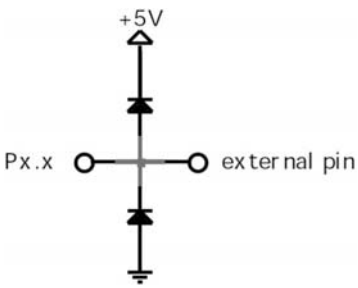


Abb. 4.6: Eingänge gegen Überspannungen schützen

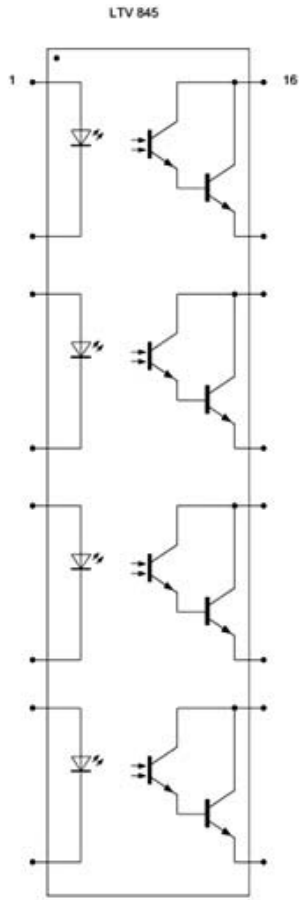


Abb. 4.7: LTV 845

Das Prinzip des IC entspricht etwa der oben beschriebenen Gabellichtschranke. Die Eingänge liegen an den Pins 1, 2 und 3, 4 usw. Die Ausgänge können jeweils gegenüber also an Pin 16, 15 usw. abgenommen werden. Die Schaltung wird beispielhaft für einen Eingang abgebildet:

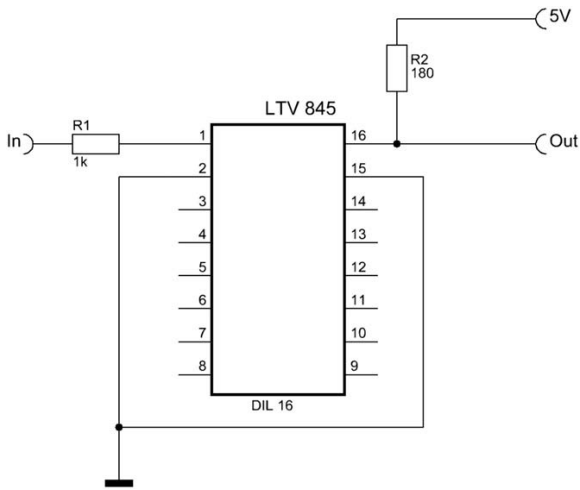


Abb. 4.8: Eingangsschaltung mit galvanischer Kopplung

Liegt an In ein High-Signal an, schaltet der ausgangsseitige Transistor durch und an Out werden ca. 0,8 V gemessen, was die digitalen Eingänge des IO-Warriors als *Low*-Signal erkennen. Liegt an In ein *Low*-Signal an, sperrt der ausgangsseitige Transistor und an Out liegen 5 V an. Der Eingangsbaustein invertiert also die zu messenden Signale. Auch hier sollte der Widerstand R2 so groß wie möglich gewählt werden, um den USB-Bus nicht zu stark zu belasten.

4.3.1 Komparator

Will man zwei analoge Spannungen vergleichen, kann man sich eines Komparators bedienen. Der IC LM339 integriert vier Komparatoren und lässt sich vielseitig einsetzen. In der folgenden Schaltung wird ein Komparator realisiert:

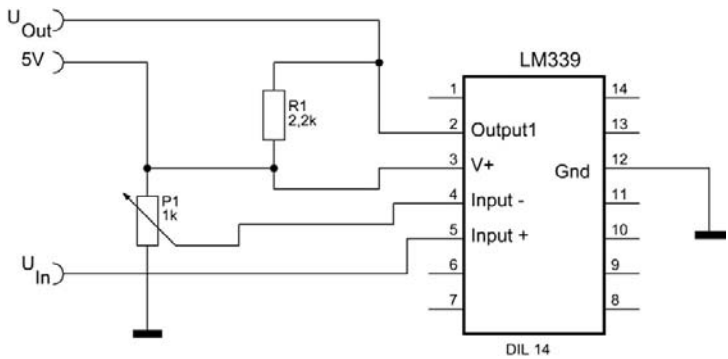


Abb. 4.9: Komparator realisiert mit einem LM339

Über das Potenziometer P1 wird die Vergleichsspannung eingestellt, die am IC an Input– anliegt. Liegt die Eingangsspannung angelegt an Input+ über der eingestellten Vergleichsspannung, liegen am Ausgang V+, also in unserem Fall 5 V an. Liegt die Eingangsspannung darunter, liegen am Ausgang 0 V an.

Diese Ausgangsspannung kann an einem IO-Pin des IO-Warriors eingelesen werden.

Im nachfolgenden Projekt wird über das Potenziometer eine Spannung von 2,5 V eingestellt. Die Eingangsspannung am Input + wird direkt über ein regelbares Netzteil eingestellt.

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace Komparator
{
    class Program
    {
        static void Main(string[] args)
        {
            //Anlegen eines IO-Objektes
            IOWKit iow = new IOWKit();
            //Öffnen des IO-Warriors
            iow.Open();
            //Vorbereiten des SendenArrays
            byte[] senden = new byte[5];
            senden[0] = 0;
            senden[1] = 255;
            senden[2] = 255;
            senden[3] = 255;
            senden[4] = 255;

            //Array zum Einlesen vorbereiten
            byte[] empfangen = new byte[5];
            empfangen[0] = 0;
            empfangen[1] = 0;
            empfangen[2] = 0;
            empfangen[3] = 0;
            empfangen[4] = 0;

            //20 mal werden die Eingänge eingelesen
            for (int i = 0; i < 20; i++)
            {
                Console.WriteLine("Eingangsspannung mit 2,5V vergleichen!");
                //Alle Ausgänge auf High setzen
                iow.SendReport(0, ref senden[0], 5);
                //Alle Eingänge einlesen
```

```
                iow.GetReportImmediat(0, ref empfangen[0], 5);
                //Rausfiltern des Bits 7 von Port 1
                if ((empfangen[2] & 128) == 128)
                 Console.WriteLine("Die gemessene Spannung liegt über 2,5V");
                else
                      Console.WriteLine("Die gemessene Spannung liegt unter
2,5V");
                System.Threading.Thread.Sleep(500);
            }
            iow.Quit();
            Console.ReadKey();
        }
    }
}
```

Das Programmfenster:

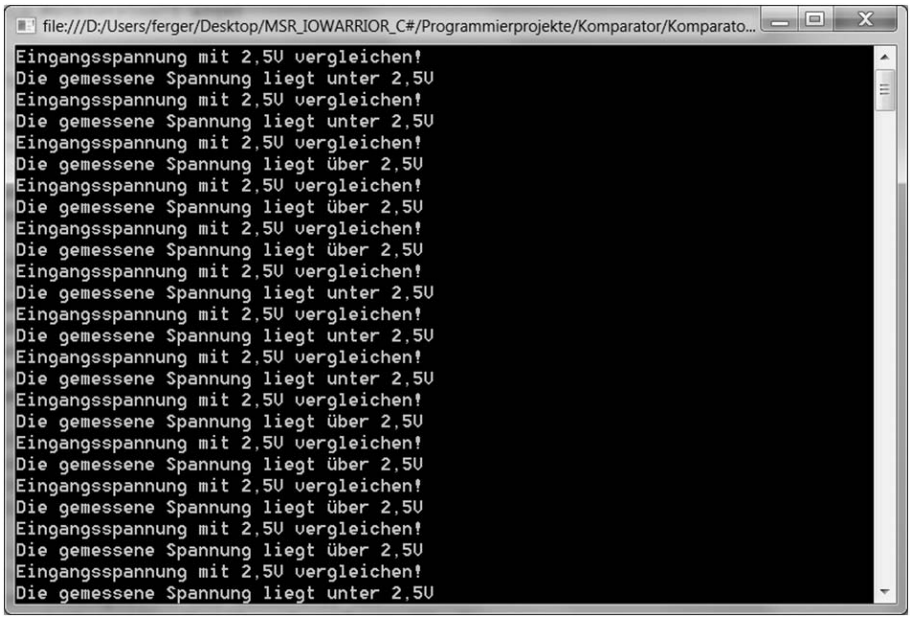


Abb. 4.10: Screenshot Komparator

Legt man an den einen Eingang des Komparators den Ausgang eines DA-Wandlers, kann man eine AD-Wandlung nach dem Prinzip der sukzessiven Approximation realisieren. Mehr dazu im Kapitel „SPI“.

4.4 Capture Timer mit dem IOW24, Frequenzmessung

Der IO-Warrior 24 besitzt in seiner neuesten Version zwei Capturetimer, mit denen man die Zeit messen kann, die zwischen einer ansteigenden und einer abfallenden Flanke oder umgekehrt vergeht. Somit kann die Periodendauer eines Signals gemessen werden – und damit die Frequenz. Die Auflösung der Timer beträgt 4 µs. Dies bedeutet, dass der Timerwert alle 4 µs inkrementiert wird. Der Timer speichert seinen Wert bei einer ansteigenden/abfallenden Flanke sowie den der nächsten abfallenden/ansteigenden Flanke und schickt einen Report auf den USB-Bus. Der Report muss dann noch umgerechnet werden, um die entsprechenden Zeiten oder Frequenzen auszurechnen. Wird der maximale Wert des Timers überschritten, kommt es zu einem Überlauf und der Timer beginnt bei 0 neu zu zählen. Der Überlauf kann im Report abgefragt werden.

Zum Messen dienen die Anschlüsse P0.0 (Capture Timer A) und P0.1 (Capture Timer B). Sofern diese Funktionalität aktiviert ist, können beim Verwenden von P0.0 keine RC5-Signale empfangen werden. Wird P0.1 verwendet, kann die I^2C-Funktion nicht benutzt werden.

Um die Capturetimer zu aktivieren, muss ein Report mit folgendem Inhalt an den IOW gesendet werden:

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x28	Flags	0	0	0	0	0	0

Das Byte Flags hat folgende Werte:

0 → Timer deaktiviert /Specialmode beendet

1 → Timer A aktiviert, Timer B deaktiviert

2 → Timer A deaktiviert, Timer B aktiviert

3 → Timer A aktiviert, Timer B aktiviert

Tritt nun ein Flankenwechsel am jeweiligen Port auf, wird ein Report generiert. Der Report besteht aus einem Array mit acht Bytes. Die Bytes teilen sich folgendermaßen auf:

ReportID	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x29 (Timer A) / 0x2A (Timer B)	Flags	Fallende Flanke Byte 0	Fallende Flanke Byte 1	Fallende Flanke Byte 2	Steigende Flanke Byte 0	Steigende Flanke Byte 1	Steigende Flanke Byte 2

Das Byte *Flag* setzt sich aus den folgenden Bits zusammen:

Bit 0: 1, wenn Timer A eine fallende Flanke entdeckt

Bit 1: 1, wenn Timer A eine steigende Flanke entdeckt

Bit 2: 1, wenn Timer B eine fallende Flanke entdeckt

Bit 3: 1, wenn Timer B eine steigende Flanke entdeckt

Bit 4: Überlauf im Timer A bei einer abfallenden Flanke

Bit 5: Überlauf im Timer A bei einer ansteigenden Flanke

Bit 6: Überlauf im Timer B bei einer abfallenden Flanke

Bit 7: Überlauf im Timer B bei einer ansteigenden Flanke

Um den exakten Wert des Timers zu berechnen, muss man die drei passenden Bytes mit ihrer berücksichtigten Wertigkeit addieren. Hierzu eignet sich das bitweise Verschieben von Variablen.

Beispiel

Der folgende Report wird eingelesen:

ReportID	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
41	1	34	56	210	64	92	211

Die ReportID ist 41, also umgerechnet 0x29. Somit kommt der Report vom Timer A.

Byte 1 mit dem Wert 1 besagt, dass der Report durch eine abfallende Flanke ausgelöst wurde.

Der Wert des Timers bei der abfallenden Flanke berechnet sich folgendermaßen:

Byte 2 = 34 → Wert0 = 34

Byte 3 = 56 → Wert1 = 14336 (der Wert 56 wurde um acht Bits nach links verschoben)

Byte 4 = 210 → Wert2 = 13762560 (der Wert 210 wurde um 16 Bits nach links geschoben)

Somit ergibt sich ein Timer-Stand von 34 + 14336 + 13762560 = 13776930

Der Wert des Timers bei der ansteigenden Flanke berechnet sich folgendermaßen:

Byte 2 = 64 → Wert0 = 64

Byte 3 = 92 → Wert1 = 23552 (der Wert 92 wurde um acht Bits nach links verschoben)

Byte 4 = 211 → Wert2 = 13828096 (der Wert 210 wurde um 16 Bits nach links verschoben)

Somit ergibt sich ein Timerstand von 64 + 23552 + 13828096 = 13851712

Es sind also von der fallenden Flanke bis zur ansteigenden Flanke 74782 Timerschritte vergangen. Da jeder Schritt 4 µs dauert, beträgt die Zeit 299,128 ms. Berechnet man auf diese Weise die nachfolgende Halbwelle eines periodischen Signals, kann man Periodendauer und Frequenz mit der entsprechenden Genauigkeit bestimmen.

Beispiel

Das nachfolgende Signal wird an den Eingang von Capturetimer A gelegt und ausgewertet, gleichzeitig werden die empfangenen Daten in eine Textdatei geschrieben:

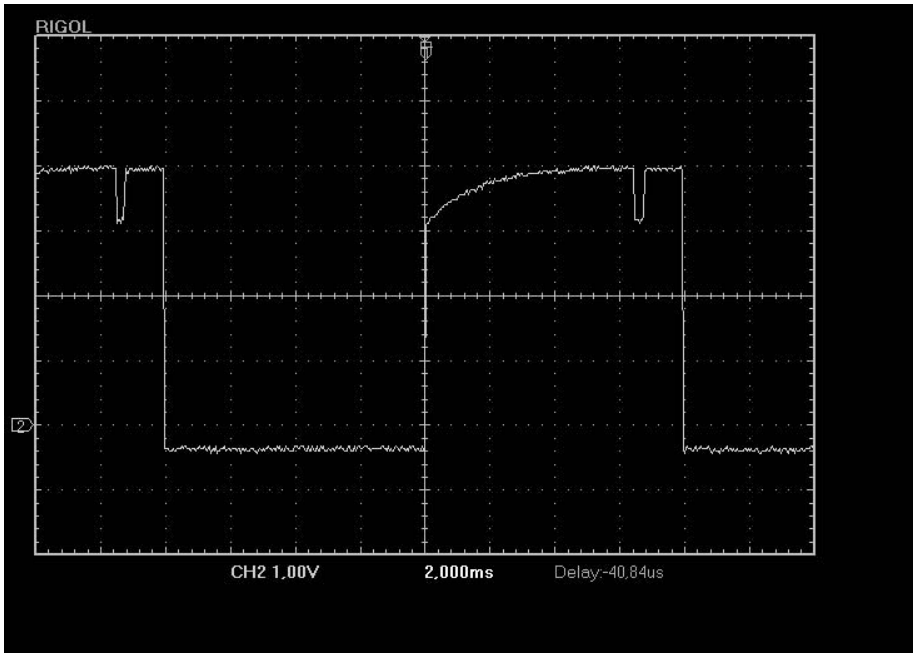


Abb. 4.11: Zu messende Spannung

```
Der kommentierte Quellcode des Programms:
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;
using System.IO;
namespace CaptureTimer
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            //Eine Textdatei wird geöffnet
            StreamWriter sw = new StreamWriter(@"NewFile.txt");
            Console.WriteLine("Datei geöffnet");
            Console.SetWindowSize(130, 20);
            //Die Verbindung zum IO-Warrior wird hergestellt
            IOWKit iow = new IOWKit();
            iow.Open();
            byte [] dataget = new byte[8];
            byte [] datasend = new byte[8];
            //Der Capture Timer A wird aktiviert
            datasend[0] = 0x28;
            datasend[1] = 1;
            iow.SendReport(1, ref datasend[0], 8);
            double halba=0,halbb=0,periodendauer,frequenz;
            //1000 Messungen erfolgen
            for (int i = 0; i < 1000; i++)
            {
               //Sofern eine Signaländerung an P0.0 anliegt, wird ein Report
abgefragt
                iow.GetReport(1, ref dataget[0], 8);
                Console.SetCursorPosition(0, 1);
                 //Wenn das Signal von High nach Low ging und kein Überlauf
vorhanden ist
                if ((dataget[1] == 1) && (dataget[7] >= dataget[4]))
                {
                    //Die Messdaten werden in die Textdatei geschrieben
                    sw.WriteLine(i + " " + dataget[0] + " " + dataget[1] +
" " + dataget[2] + " " + dataget[3] + " " + dataget[4] + " " + dataget[5] + "
" + dataget[6] + " " + dataget[7]);
                    //Berechnung der ersten Halbwelle
                    int zeita = dataget[2] + (dataget[3] << 8) + (dataget[4]
<< 16);
                    int zeitb = dataget[5] + (dataget[6] << 8) + (dataget[7]
<< 16);
                    double diff = zeita - zeitb;
                    halba = diff * 0.004;
                }
                else
                {
                   //Wenn das Signal von Low nach High ging und kein Überlauf
vorhanden ist
                    if ((dataget[1]==2)&&(dataget[7] >= dataget[4]))
                    {
                        //Die Messdaten werden in die Textdatei geschrieben
                        sw.WriteLine(i + " " + dataget[0] + " " + dataget[1]
+ " " + dataget[2] + " " + dataget[3] + " " + dataget[4] + " " + dataget[5] +
" " + dataget[6] + " " + dataget[7]);
                        //Berechnung der zweiten Halbwelle
                         int zeita = dataget[2] + (dataget[3] << 8) + (data-
get[4] << 16);
                         int zeitb = dataget[5] + (dataget[6] << 8) + (data-
get[7] << 16);
                        double diff = zeita - zeitb;
```

```
                        halbb = -1*diff * 0.004;

                    }
                }
                //Berechnung der Periodenauer und der Frequenz
                periodendauer = halba + halbb;
                frequenz = 1 / periodendauer;
                frequenz = Math.Round(frequenz, 4);
               sw.WriteLine("Halbwelle 1: " + halba + " Halbwelle 2: " + halbb
+ " Periodendauer: " + periodendauer + "ms\tFrequenz: " + frequenz + "kHz");
                Console.WriteLine("Halbwelle 1: " + halba + "\tHalbwelle 2: "
+ halbb + "\tPeriodendauer: " + periodendauer + "ms\tFrequenz: " + frequenz +
"kHz");
            }
            //Specialmode beenden und IOW schließen
            datasend[1] = 0;
            iow.SendReport(1, ref datasend[0], 8);
            Console.WriteLine("Specialmode beendet");
            iow.Quit();
            Console.WriteLine("IOWarrior geschlossen");
            //Textdatei schließen
            sw.Close();
            Console.WriteLine("Datei geschlossen");
            Console.ReadKey();
        }
    }
}
```

Das Programm hat folgende Bildschirmausgabe zur Folge:

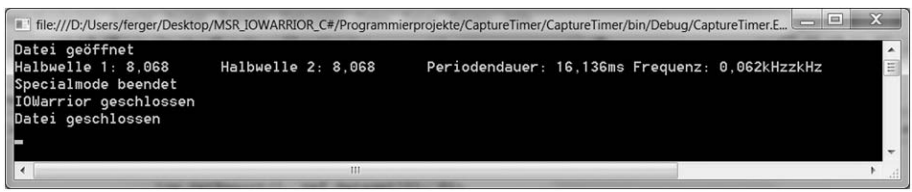


Abb. 4.12: Screenshot Frequenzmessung

5 SPI

Das Serial Peripheral Interface ist ein Bus-System, das von der Firma Motorola entwickelt wurde. Über ein Master-Slave-Prinzip kommunizieren digitale Bausteine miteinander. SPI ist voll duplexfähig, Daten können also gleichzeitig gesendet und empfangen werden. Mittlerweile gibt es Bausteine aus allen möglichen Bereichen mit SPI-Anschlussmöglichkeit.

Der Bus besteht aus vier Leitungen:

MOSI: Master Out Slave In, Daten werden vom Master an den Slave gesendet.

MISO: Master In Slave Out, Daten werden vom Slave an den Master gesendet.

SCK: Serial Clock, das Taktsignal

/SS: Slave Select auch als CS, Chip Select, bezeichnet, über ein LOW-Signal wird ein Slave ausgewählt.

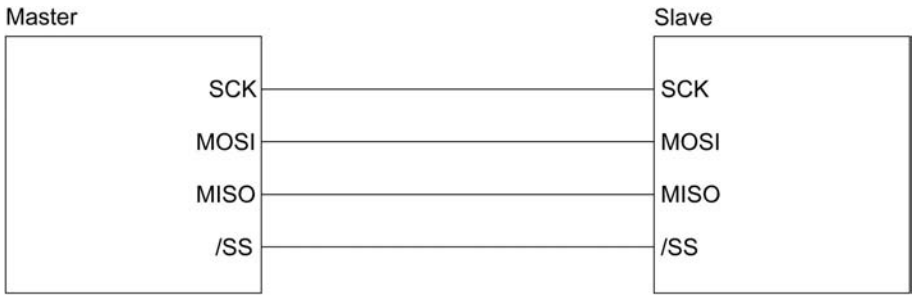


Abb. 5.1: SPI-Aufbau

Die Leitung MOSI wird nicht verwendet, sofern es sich um einen Baustein handelt, der ausschließlich Daten sendet, wie beispielsweise der später aufgeführte MCP3201. Umgekehrt muss die Leitung MISO nicht verwendet werden, wenn ein Baustein nur Daten empfängt. Empfänger werden über die Leitung /SS selektiert (der Anschluss wird bei den ICs oft als /CS bezeichnet). Ein LOW-Signal an diesem Pin wählt einen angeschlossenen IC aus, der dann Daten empfängt oder sendet. Will man mehrere ICs gleichzeitig anschließen, ist dies nur über eine entsprechende Logikschaltung und zusätzliche digitale Ausgänge am Master möglich, was aber mit dem IO-Warrior durchaus möglich ist.

Die IO-Warrior 24 und 56 haben die Masterfunktion des SPI integriert.

5.1 Verwenden der Special Mode Function SPI

Die Funktion des SPI wird aktiviert, indem man ein Array von Bytes mit den folgenden Werten an den IOW sendet:

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
8	Enable	Modus	0	0	0	0	0

Das Byte *Enable* wird auf den Wert 1 gesetzt, um SPI zu aktivieren, 0 deaktiviert die Spezialfunktion.

Das Byte *Modus* setzt sich aus den folgenden Bits zusammen:

Bit 0 und Bit 1 bestimmen die Datentransferrate:

Wert	Rate
00	2 MBit/s
01	1 MBit/s
10	0,5 MBit/s
11	0,0625 MBit/s

Bit 2 (CPHA) und Bit 3 (CPOL) bestimmen den Modus

Die genaue Datenübertragung ist beim SPI nicht definiert. Es haben sich in der Praxis aber vier leicht unterschiedliche Modi durchgesetzt, die durch die Einstellungen der Clock Polarity (CPOL) und der Clock Phase (CPHA) definiert werden:

Modus 0:	CPOL = 0	CPHA = 0
Modus 1:	CPOL = 0	CPHA = 1
Modus 2:	CPOL = 1	CPHA = 0
Modus 3:	CPOL = 1	CPHA = 1

CPOL definiert, welcher Flankenwechsel (also von LOW nach HIGH und umgekehrt) nach Aktivierung eines IC vom *Clock*-Signal zu Beginn anliegt. Bei CPOL = 0 liegt ein Flankenwechsel von LOW nach HIGH an. CPHA definiert, ob mit dem ersten Flankenwechsel (CPHA = 0) oder mit dem zweiten (CPHA = 1) begonnen wird, auf den Datenkanälen Daten zu übertragen.

Welcher Modus verwendet wird, hängt vom angeschlossenen Baustein ab. Seinem Datenblatt können die entsprechenden Informationen entnommen werden.

Datenübertragung bei den verschiedenen Modi[6]:

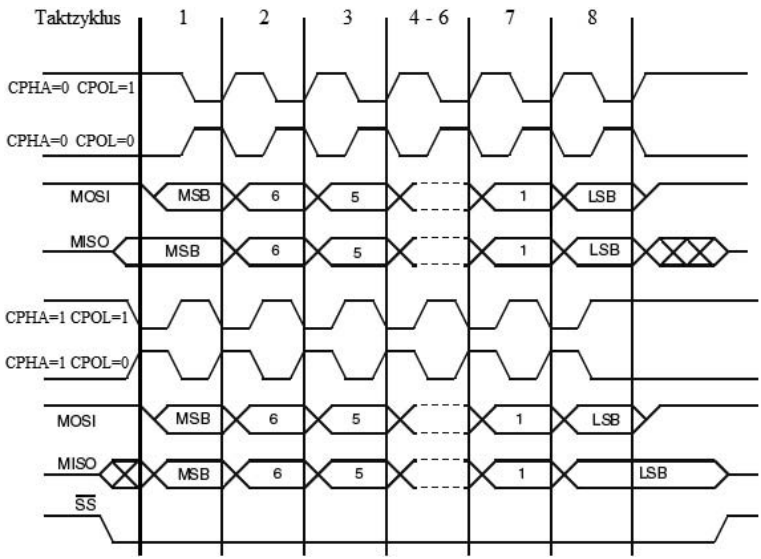


Abb. 5.2: Datenübertragung auf dem SPI-Bus

5.2 Porterweiterung mittels SPI

Der Baustein MCP 23S08 ermöglicht eine 8-Bit-Porterweiterung, wobei die Pins als Ein- oder Ausgänge verwendet werden können.

Die Beschaltung ist denkbar einfach:

Die SPI-Leitungen werden direkt mit den entsprechenden Pins des IO-Warriors verbunden. VDD ist der Betriebsspannungseingang und kann genau wie VSS (Masse) direkt vom Starterkit entnommen werden. Der Reset-Eingang ist auf High zu setzen. An den Pins 10 bis 17 können die Ausgangszustände abgenommen oder die Eingangszustände angelegt werden. Der Baustein besitzt interne Pull-up-Widerstände, die per Programmierung aktiviert werden müssen.

Über die Adresseingänge ist es möglich, vier MCP 23S08 gleichzeitig am SPI-Bus mit nur einem /*CS*-Ausgang zu betreiben.

Die Pinbelegung des MCP23S08:

6 Grafik aus: http://de.wikipedia.org/wiki/Serial_Peripheral_Interface

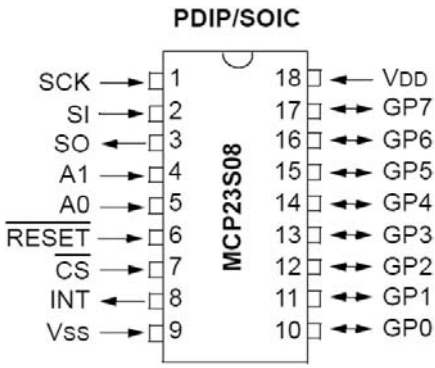


Abb. 5.3: MCP23S08

Die Pins *GP0* bis *GP7* stellen den IO-Port dar. Über die Pins *A0* und *A1* kann eine Adresse eingestellt werden, um mehrere Bausteine zu betreiben.

Die Programmierung ist denkbar einfach:

Dem Datenblatt entnimmt man, dass der Baustein mit bis zu 10 MHz im Modus 2 arbeitet. Entsprechend ist die Specialfunktion SPI des IO-Warriors zu konfigurieren:

```
datensend[0] = 8;
datensend[1] = 1;
datensend[2] = 8;
iow.SendReport(1, ref datensend[0], 8);
```

Beim Schreiben auf den Baustein werden drei Bytes auf den SPI-Bus geschrieben:

1. Byte: Adresse + Read/Write 2. Byte: Register 3. Byte: Daten

Die Adresse setzt sich folgendermaßen zusammen:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	0	A1	A0	1 zum Lesen 0 zum Schreiben

Im aktuellen Beispiel wurden A1 und A0 auf Masse gezogen. Somit ergibt sich der dezimale Wert 64 zum Schreiben und der Wert 65 zum Lesen des Bausteins.

Beim Lesen werden nur die Adresse und das Register auf den Bus geschrieben, der Baustein sendet anschließend das entsprechende Byte zurück.

Anschließend ist das Register *0* des Bausteins auf den Wert *0* zu setzen. Hiermit werden alle Pins als Ausgänge konfiguriert. Will man die Eingänge auslesen, muss der Wert 255 in das Register geschrieben werden:

```
datensend[0] = 9;
datensend[1] = 3;
datensend[2] = 64;
datensend[3] = 00;
datensend[4] = 0;
iow.SendReport(1, ref datensend[0], 8);
```

Ist eine Außenbeschaltung der Ausgangspins zunächst nicht erwünscht, werden die internen Pull-up-Widerstände aktiviert. Hierzu schreibt man in Register 6 des MCP den Wert 255:

```
datensend[3] = 6;
datensend[4] = 255;
iow.SendReport(1, ref datensend[0], 8);
```

Anschließend können Daten auf den Port geschrieben werden, indem sie in das Register 9 übertragen werden.

```
datensend[3] = 09;
datensend[4] = 255;
iow.SendReport(1, ref datensend[0], 8);
```

Im nachfolgenden Projekt wurden die Werte 0 bis 255 auf die Ausgänge gegeben.

Der kommentierte Quelltext:

```
using System;
using IOWarrior;

namespace SPIPorterweiterung
{
    class Program
    {
        static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            iow.Open();

            //Arrays zum Senden auf dem SPI- Bus
            //vorbereiten
            byte[] datensend = new byte[8];

            //SPI Mode aktivieren
            //Datenübertragungsgeschwindigkeit ist 2Mbit/s
            //CPHA wird auf Low gesetzt
            datensend[0] = 8;
            datensend[1] = 1;
            datensend[2] = 8;
            iow.SendReport(1, ref datensend[0], 8);

            //Vorbereitung des SPI - Reports
```

```
            //ID wird auf 9 gesetzt
            //3 Bytes werden gesendet
            //Ins Register 0 wird 0 geschrieben
            //Alle Pins sind Ausgänge
            datensend[0] = 9;
            datensend[1] = 3;
            datensend[2] = 64;
            datensend[3] = 00;
            datensend[4] = 0;
            iow.SendReport(1, ref datensend[0], 8);

            //Vorbereitung des SPI - Reports
            //Ins Register 6 wird 255 geschrieben
            //Der interne Pull-up wird verwendet
            datensend[3] = 6;
            datensend[4] = 255;
            iow.SendReport(1, ref datensend[0], 8);

            for (int i = 0; i < 256; i++)
            {
                //Der Wert i wird in das Register 9
                //geschrieben und steht am Ausgangsport
                //zur Verfügung
                datensend[3] = 09;
                datensend[4] = (byte)i;
                iow.SendReport(1, ref datensend[0], 8);
                Console.WriteLine(i);
            }
            iow.Quit();
            Console.ReadKey();
        }
    }
}
```

5.3 Ein programmierbares Potenziometer

Um Spannungen oder Ströme regeln zu können, kann es notwendig sein, nicht die eigentliche Betriebsspannung, sondern einen Widerstandswert zu variieren. Die Baureihe MCP 4xxx erfüllt diese Eigenschaft. Am Beispiel des MCP 4151-103 wird aufgezeigt, wie ein programmierbares Potenziometer verwendet werden kann:

Die Anschlussbelegung des MCP 4151-103

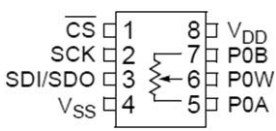


Abb. 5.4: MCP4151-103

Der MCP 4151-103 ist ein einfach verwendbarer einstellbarer Widerstand, der über den SPI-Bus angesteuert werden kann. Die Außenbeschaltung ist denkbar einfach (hier wird ein IO-Warrior 24 verwendet):

Pin 1 wird an den *Chip-Select*-Ausgang (Pin 0.4) des IO-Warriors angeschlossen, Pin 2 an den *Clock*-Ausgang (Pin 0.7), Pin 3 an den *MOSI*-Ausgang (Pin 0.5), Pin 4 wird auf Masse und Pin 8 auf 5 V gelegt. Masse und Betriebsspannung stehen auf den Starterkits zur Verfügung. An den Pins 6 bis 8 können die Potenziometeranschlüsse abgegriffen werden.

Der Baustein wird folgendermaßen programmiert:

Auf den SPI-Bus werden zwei Bytes geschrieben. Das erste Byte setzt sich folgendermaßen zusammen:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Register-adresse	Register-adresse	Register-adresse	Register-adresse	Kom-mando	Kom-mando	0	Daten Bit 8

Als Registeradresse ist im Beispiel 0000 zu wählen (andere Adressen sind für andere Bausteine aus dieser Reihe). Auch das Kommando ist 00 zu wählen. Hiermit wird auf den Baustein geschrieben. Es folgen die Daten, die 9 Bits breit sind, wobei das höchste Bit (Bit 8) noch im ersten Datenbyte mitgeschrieben wird.

Das zweite Datenbyte enthält die restlichen 8 Bits der zu schreibenden Daten.

Im nachfolgenden Projekt werden die Werte 0 bis 512 auf den Baustein geschrieben. Über ein Multimeter können die sich ändernden Widerstandswerte beobachtet werden:

Der kommentierte Quelltext:

```
using System;
using IOWarrior;

namespace SPIProgrammierbaresPoti
{
    class Program
    {
        static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            iow.Open();

            //Arrays zum Senden auf dem SPI- Bus
            //vorbereiten
            byte[] datensend = new byte[8];

            //SPI Mode aktivieren
```

```
            //Datenübertragungsgeschwindigkeit ist 2Mbit/s
            //CPHA wird auf High gesetzt
            datensend[0] = 8;
            datensend[1] = 1;
            datensend[2] = 4;
            iow.SendReport(1, ref datensend[0], 8);

            //Vorbereitung des SPI - Reports
            //ID wird auf 9 gesetzt
            //2 Bytes werden gesendet
            datensend[0] = 9;
            datensend[1] = 2;
            for (int i = 0; i < 512; i++)
            {
                System.Threading.Thread.Sleep(100);
                //Bit 9 der Daten wird gefiltert und als
                //Bit 0 im ersten Datenbyte gespeichert
                datensend[2] = (byte)((i&256)>>8);
                //Die restlichen Bits werden rausgefiltert
                //und im zweiten Datenbyte gespeichert
                datensend[3] = (byte)(i&255);
                iow.SendReport(1, ref datensend[0], 8);
                //Berechnung und Ausgabe des
                //Widerstandwertes
                double widerstand = (double)10000 * i / 256;
                Console.WriteLine(i +  " Widerstand:" + widerstand + "Ohm" );
            }
            iow.Quit();
            Console.ReadKey();
        }
    }
}
```

5.4 Einlesen einer analogen Spannung mit einer Auflösung von 12 Bit

Der Baustein MCP 3201 ist ein sehr einfach aufgebauter 12-Bit-AD-Wandler mit integriertem SPI. Eine Außenbeschaltung ist nicht notwendig, die Spannungsversorgung übernimmt der USB-Anschluss.

Pinbelegung des Bausteins

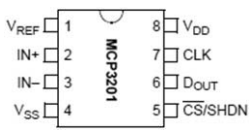


Abb. 5.5: MCP3201

Anschluss an den IO-Warrior

An V_{REF} wird wie an V_{DD} die Betriebsspannung von 5 V angelegt.

IN– und VSS werden mit Masse verbunden.

IN+ ist der Spannungseingang.

CLK wird mit Pin P0.7/SCK des IO-Warriors verbunden.

/CS wird mit Pin P0.4 / /SS des IO-Warriors verbunden.

DOUT wird mit Pin P0.6/MISO des IO-Warriors verbunden.

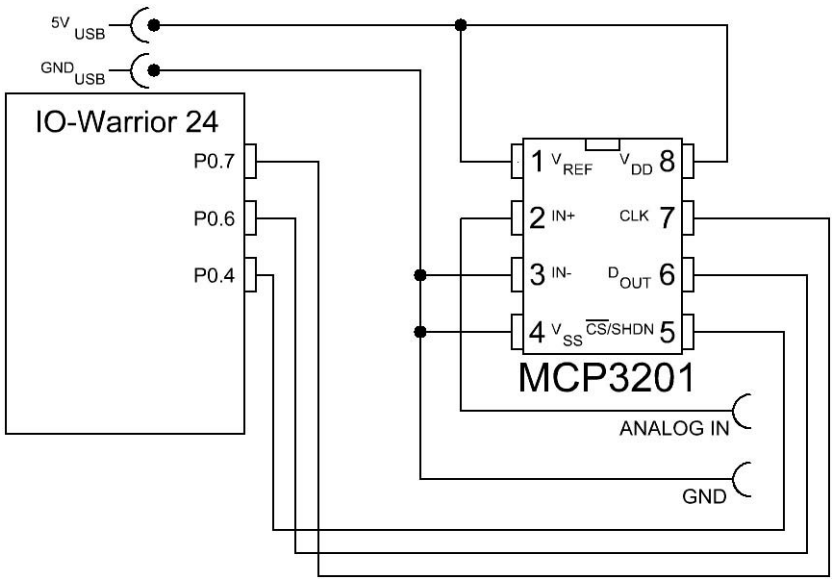


Abb. 5.6: Schaltung mit dem MCP3201

Der Baustein arbeitet nach dem Prinzip der sukzessiven Approximation. Hierbei erhöht ein Zähler einen Wert, den er digital an einen DA-Wandler weitergibt. Der DA-Wandler legt die Ausgangsspannung an einen Komparator. Die zu messende Eingangsspannung liegt auch am Komparator an. Sind die Spannungen gleich, gibt dies der Komparator an eine Logik weiter. Diese liest den Stand des 12-Bit-Zählers aus und gibt den entsprechenden Wert über ein Schieberegister auf den seriellen Ausgang. Wird der Baustein anschließend per LOW-Signal am Pin /CS selektiert, legt er seine Daten an den seriellen Ausgang, sodass der Master hierauf zugreifen kann.

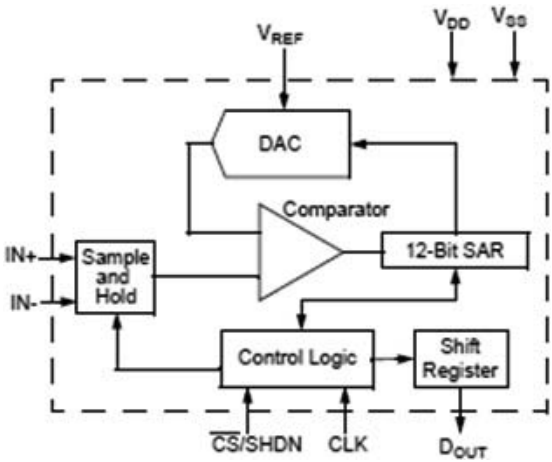


Abb. 5.7: Blockschaltbild des Bausteins

Im nachfolgenden Programm werden 10.000-mal die Daten des IC abgefragt. Da der Report des IOW immer nur Bytes speichert, muss der 12-Bit-Wert zusammengesetzt werden. Das niederwertige Byte liegt im Reportarray in Feld 3, das höherwertige in Feld 2. Dieses Byte wird mit dem Wert 15 maskiert, da nur die untersten vier Bit Werte liefern (12 Bit Auflösung). Anschließend wird das Byte um acht Bit nach links verschoben und zu dem niederwertigen Byte addiert. Der so entstandene Wert kann, bezogen auf die Referenzspannung von 5 V, in einen Spannungswert umgerechnet werden.

Der kommentierte Quellcode des Programms:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace SPI_MCP3201
{
    class Program
    {
        static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            if (iow.Open())
            {

                //Der SPI-Modus wird vorbereitet
                byte[] datensend = new byte[8];
                byte[] datenget = new byte[8];
```

```
                datensend[0] = 8;
                datensend[1] = 1;
                datensend[2] = 7;
                iow.SendReport(1, ref datensend[0], 8);
                for (int i = 0; i < 10000; i++)
                {
                  Console.SetCursorPosition(0, 0);
                  //Die Datenübertragung wird eingeleitet
                  datensend[0] = 9;
                  datensend[1] = 2;
                  datensend[2] = 0;
                  iow.SendReport(1, ref datensend[0], 8);
                  Console.WriteLine("Daten geschrieben");
                  //Die Daten vom Chip werden eingelesen
                  iow.GetReport(1, ref datenget[0], 8);
                  Console.Write(datenget[0] + "\t" + datenget[1]);
                 Console.WriteLine("\t" + datenget[2] + "\t" + datenget[3]);
                  //Berechnung der Eingangsspannung
                  double spannung = 0;
//Das niederwertige Datenbyte
                  int low = datenget[3];
//Das höherwertige Datenbyte wird mit dem Wert
//15 maskiert
                  int high = datenget[2] & 15;
//Das maskierte Datenbyte wird um acht Bit nach links
//verschoben
                  high = high << 8;
                  int data = low + high;
                  spannung = (double)data * 5 / 4096;
                  spannung = Math.Round(spannung, 3);
                  Console.WriteLine("Spannung: " + spannung + "V");
                }
                iow.Quit();
                Console.ReadKey();
            }
        }
    }
}
```

Abb. 5.8: Screenshot Spannungsmessung

5.5 Ansteuerung eines 12-Bit-DA-Wandlers mittels SPI

Von Linear Technology wird mit dem LTC 2630 ein DA-Wandler mit 12 Bit Auflösung angeboten, der über den SPI-Bus angesteuert werden kann. Über eine Adapterplatine, beispielsweise von der Firma Roth Elektronik (zu beziehen bei Reichelt), kann leicht ein Prototyp für Experimente und Untersuchungen aufgebaut werden. Eine Außenbeschaltung ist kaum notwendig. Die Betriebsspannung sollte über einen 0,1-µF-Kondensator vor Schwingungen geschützt werden.

Das Blockschaltbild[7] erklärt die Funktionsweise des Wandlers:

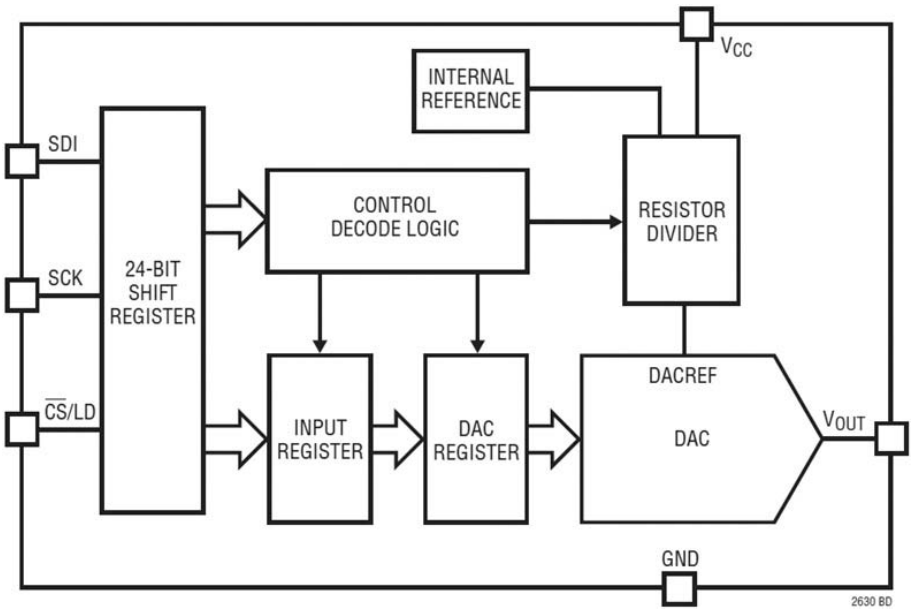


Abb. 5.9: Blockschaltbild LTC2603

Der Baustein wird folgendermaßen an den IO-Warrior angeschlossen:

7 Grafik aus dem Datenblatt des Bausteins LTC 2630

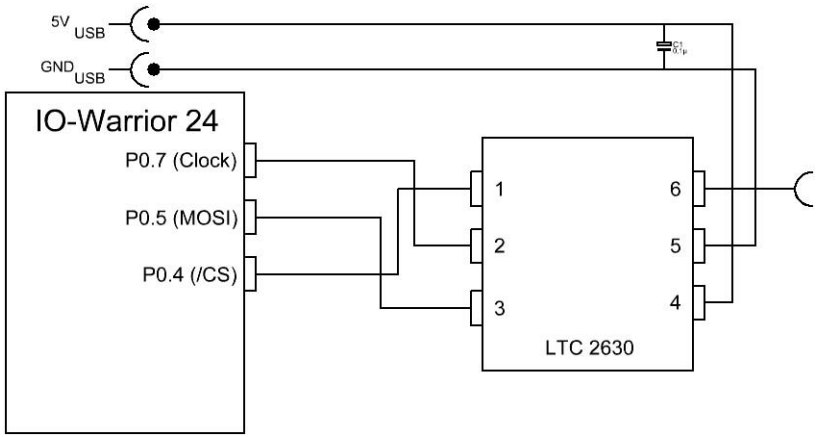


Abb. 5.10: Schaltplan

Programmierung des LTC 2630

Nach der Aktivierung der SPI-Spezialfunktion bekommt der Baustein drei Bytes übermittelt.

Im ersten Byte wird auf den Bits 4 bis 7 die Funktionalität des ICs festgelegt:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Funktion
0	0	0	0	0	0	0	0	In das Register des Bausteins wird geschrieben.
0	0	0	1	0	0	0	0	Der Ausgang wird gesetzt.
0	0	1	1	0	0	0	0	In das Register wird geschrieben und der Ausgang wird gesetzt.
0	1	0	0	0	0	0	0	Der Baustein wird in den Ruhezustand versetzt.
0	1	1	0	0	0	0	0	Die interne Referenzspannung wird verwendet.
0	1	1	1	0	0	0	0	Die Betriebsspannung wird als Referenz verwendet.

Die unteren vier Bits können auch den Wert 1 beinhalten. Sie werden nicht berücksichtigt.

Anschließend werden in zwei Bytes die auszugebenden Daten übermittelt. Hierbei enthält das erste zu sendende Byte die Bits 4 bis 11, das zweite an den Stellen Bit 4 bis Bit 7 die restlichen vier Bits. Alle anderen Bits werden nicht berücksichtigt.

Die folgende Grafik[8] verdeutlicht die Übertragung:

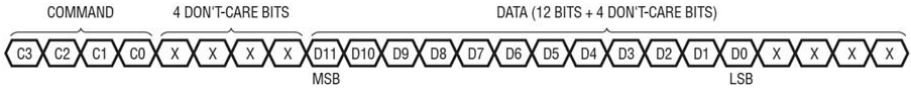


Abb. 5.11: Datenübertragung zum DA-Wandler

Das nachfolgende Programm erzeugt eine Dreiecksspannung am Ausgang des DA-Wandlers. Als Referenzspannung wird die Betriebsspannung verwendet, die dem USB-Bus entnommen wurde.

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace LTC2630
{
    class Program
    {
        static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            iow.Open();
            byte[] datensend = new byte[8];
            //SPI Mode aktivieren
            //Datenübertragungsgeschwindigkeit ist 2Mbit/s
            //CPHA un CPOL werden auf High gesetzt
            datensend[0] = 8;
            datensend[1] = 1;
            datensend[2] = 11;
            iow.SendReport(1, ref datensend[0], 8);
            //Vorbereitung des SPI - Reports
            //ID wird auf 9 gesetzt
            //3 Bytes werden gesendet
            datensend[0] = 9;
            datensend[1] = 3;
            //Als Referenz wird die Betriebsspannung
            //verwendet, die dem USB-Bus entnommen
            //wird
            datensend[2] = 112;
            datensend[3] = 0;
            datensend[4] = 0;
            iow.SendReport(1, ref datensend[0], 8);
            //Zum DA-Wadler wird geschrieben und
            //der Ausgang wird gesetzt.
```

8 Grafik aus dem Datenblatt des LTC 2630

```
            datensend[2] = 48;
            //Die unteren 4 Bit werden konstant
            //auf den Wert 15 gesetzt.
            byte dataout = 255;
            byte datahigh = 1;
            while (true)
            {
                datensend[3] = datahigh;//dataout;
                datensend[4] = dataout;//(byte)255;
                iow.SendReport(1, ref datensend[0], 8);
                //Die oberen 8 Bits werden aus
                //Performancegründen jeweils um
                //den Wert 4 erhöht.
                datahigh+=4;
            }
        }
    }
}
```

Die erzeugte Ausgangsspannung:

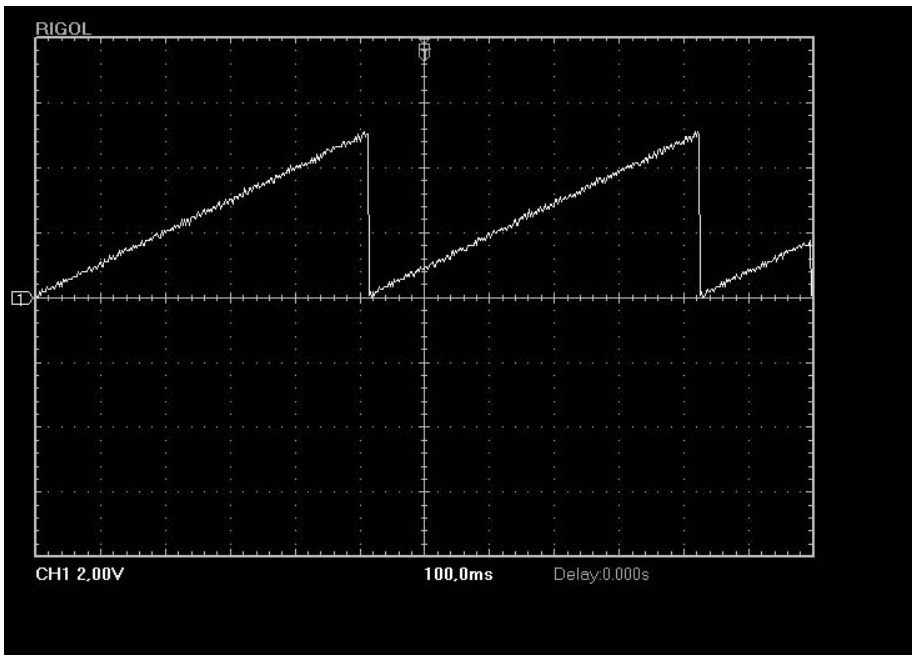


Abb. 5.12: Erzeugte Dreiecksspannung

5.6 Ansteuerung eines 16-Bit-DA-Wandlers mittels SPI

Der Baustein LTC 2602 der Firma Linear Technology ist ein zweifacher Digital-Analog-Wandler mit einer Auflösung von 16 Bit. Er eignet sich dadurch zur präzisen Spannungsausgabe. Der IC wird in SMD-Technik gefertigt. Entsprechende Adapterplatinen sind beispielsweise bei der Firma Reichelt erhältlich.

Die Beschaltung des Bausteins ist denkbar einfach:

Pin 1 wird an den Anschluss /SS des IO-Warriors, Pin 2 an den Anschluss SCK und Pin 3 wird an MOSI angeschlossen. Die Masse (Pin 7) sowie die Betriebsspannung (Pin 6) werden dem Starterkit entnommen. Die Referenzspannung (Pin 4) kann extern hinzugefügt werden. Im einfachsten Fall kann hier die Betriebsspannung VCC verwendet werden. An Pin 8 und Pin 5 können die gewandelten Spannungen abgegriffen und weiter verarbeitet werden.

Der Aufbau des DA-Wandlers:

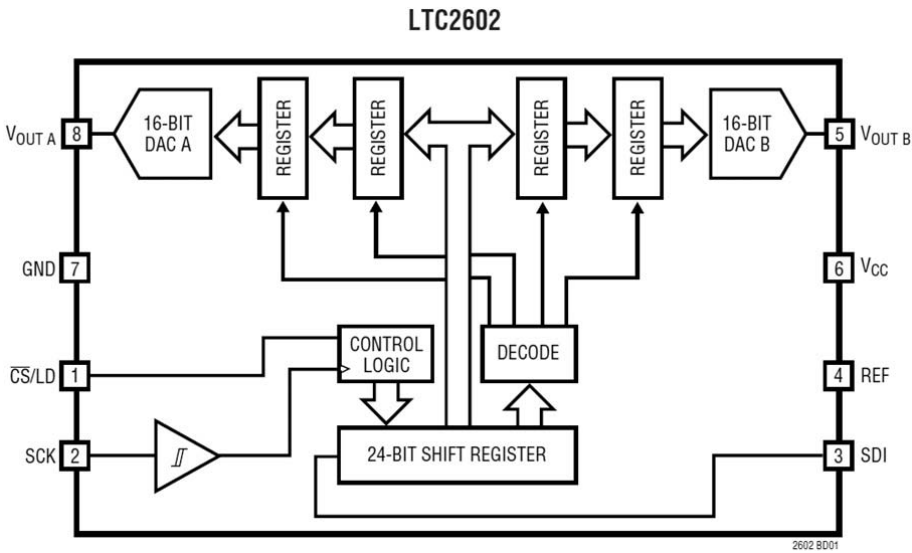


Abb. 5.13: Blockschaltbild des DA-Wandlers

Dem Blockschaltbild kann man entnehmen, dass der Baustein die empfangenen Daten in Register speichert und dann auf den Ausgang gibt. Über eine Logik kann der Ausgang ausgewählt werden. Außerdem ist es möglich, dass zuerst in die Register Daten geschrieben und in einem zweiten Schritt die Daten auf den Ausgang gegeben werden.

Programmierung des LTC 2602

Der LTC bekommt drei Bytes übermittelt:

Das erste Byte setzt sich aus vier Kommandobits und vier Adressbits zusammen:

Kommandobits

Bit 7	Bit 6	Bit 5	Bit 4	Funktion
0	0	0	0	Daten werden ins Register des ausgewählten Wandlers geschrieben.
0	0	0	1	Der Ausgang des ausgewählten Wandlers wird gesetzt.
0	0	1	0	Daten werden ins Register des ausgewählten Wandlers geschrieben und alle Ausgänge werden gesetzt.
0	0	1	1	Daten werden ins Register des ausgewählten Wandlers geschrieben und der entsprechende Ausgang wird gesetzt.
0	1	0	0	Der Baustein wird in den Ruhezustand versetzt.

Adressbits

Bit 3	Bit 2	Bit 1	Bit 0	Wandler
0	0	0	0	Wandler A
0	0	0	1	Wandler B
1	1	1	1	Beide Wandler

Will man beispielsweise auf den Wandler A schreiben und den entsprechenden Ausgang direkt aktivieren, muss das erste Byte den Wert 32 haben.

Die nachfolgenden Bytes enthalten den zu wandelnden Wert, wobei zuerst das höherwertige Byte gesendet wird.

Im nachfolgenden Programm wird eine sinusförmige Spannung erzeugt:

Der kommentierte Quellcode:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;
```

```
namespace LTC2602
{
    class Program
    {
        static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            iow.Open();
            byte[] datensend = new byte[8];
            byte[] datenget = new byte[8];
            //SPI Mode aktivieren
            //Datenübertragungsgeschwindigkeit ist 2Mbit/s
            //CPHA wird auf High gesetzt
            datensend[0] = 8;
            datensend[1] = 1;
            datensend[2] = 4;
            iow.SendReport(1, ref datensend[0], 8);
            //Vorbereitung des SPI - Reports
            //ID wird auf 9 gesetzt
            //3 Bytes werden gesendet
            //Es wird nur der DA-Wandler A
            //angesprochen
            datensend[0] = 9;
            datensend[1] = 3;
            datensend[2] = 32;
            //Variablen zur Sinusberechnung
            double winkel = 0;
            double step = 0.1;
            while (true)
                {
                    //Math.Sin liefert den Sinus des Winkels
                    //winkel. winkel muss im Bogenmaß angegeben werden
                    double sinus = Math.Sin(winkel);
                    //Umrechnung des Sinus in 16-Bit-Auflösung
                    int dataout = (int)((sinus + 1) * 32767);
                    //Die Augangsdaten werden in zwei Bytes
                    //aufgeteilt.
                    int datalow = dataout & 255;
                    int datahigh = dataout >> 8;
                    winkel = winkel + step;
                    //Die Daten werden an den LTC gesendet
                    datensend[3] = (byte)datahigh;
                    datensend[4] = (byte)datalow;
                    iow.SendReport(1, ref datensend[0], 8);
                }
        }
    }
}
```

Das Programm hat diesen Spannungsverlauf zur Folge:

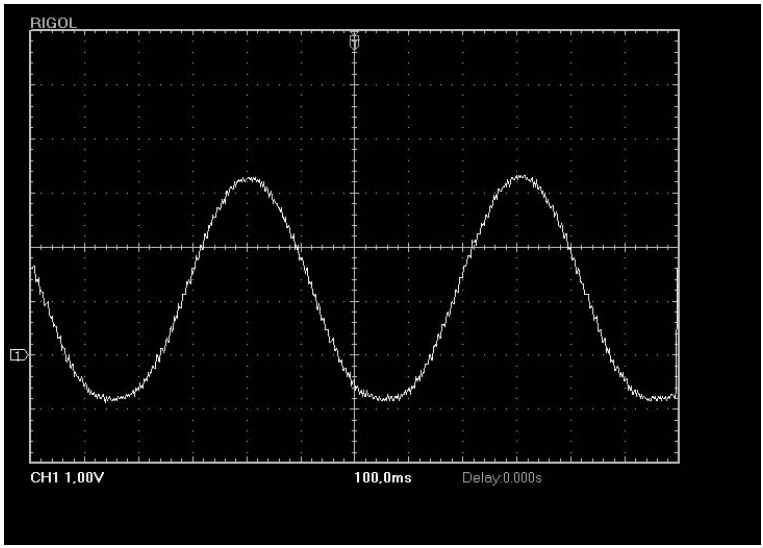


Abb. 5.14: Erzeugte Sinusspannung

Bei den einschlägigen Elektronikdistributoren findet man ICs mit SPI aus allen möglichen Bereichen: EPROMs, Sensoren, Wandler und LCD-Steuerungen können mittels SPI und damit mit den IO-Warrior Bausteinen angesteuert werden.

Eine umfangreiche Liste von SPI-fähigen Bausteinen findet man unter http://*www.mct.de/faq/spi.html*.

5.7 AD-Wandlung per sukzessiver Approximation

Ist kein AD-Wandler zur Hand, kann man mit einem DA-Wandler und einem Komparator einen solchen Wandler aufbauen. Das Prinzip ist denkbar einfach und wird im folgenden Schaubild verdeutlicht:

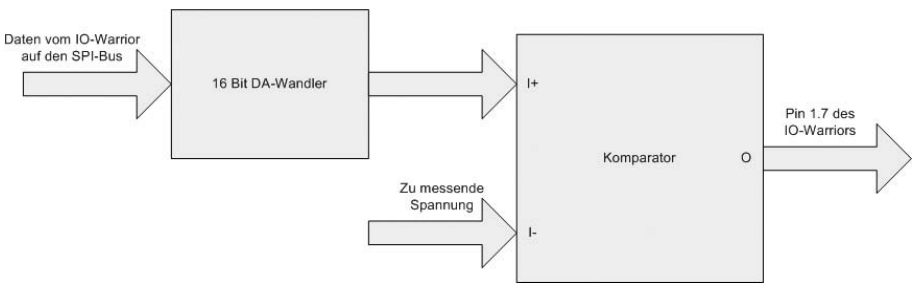


Abb. 5.15: Prinzip der sukzessiven Approximation

Am Komparator (beispielsweise dem LM 339 aus dem Kapitel „IO-Funktionen“) liegt die zu messende Spannung auf dem Eingang I– an. Am Eingang I+ liegt der Ausgang eines DA-Wandlers (hier der LTC2602) an. Der Ausgang des Komparators wird auf den IO-Pin 1.7 des IO-Warriors gelegt (natürlich ist auch jeder andere Pin, der nicht vom SPI-Modus betroffen ist, möglich). Der DA-Wandler bekommt nun per SPI Werte, die nach und nach erhöht werden. Erreicht die Ausgangsspannung des DA-Wandlers den Wert der zu messenden Spannung, kippt der Komparator seinen Ausgangszustand. Dieses Kippen kann über die IO-Funktionen des IO-Warriors wahrgenommen werden und somit weiß man, wie groß die zu messende Spannung ist. Ungenauigkeiten ergeben sich aus der Auflösung des DA-Wandlers und der Hysterese des Komparators.

Im nachfolgenden Projekt wird ein 16-Bit-DA-Wandler per SPI angesprochen und die Spannung mit einer Spannung, eingestellt am Potenziometer, an einem Komparator LM339 verglichen. Die Zeit der Wandlung wird gemessen.

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace SukzessiveApproximation16Bit
{
    class Program
    {
        static void Main(string[] args)
        {
            bool wertErreicht = false;
            IOWKit iow = new IOWKit();
            iow.Open();
            //Array zum Senden vorbereiten
            byte[] senden = new byte[5];
            senden[0] = 0;
            senden[1] = 255;
            senden[2] = 255;
            senden[3] = 255;
            senden[4] = 255;

            //Array zum Einlesen vorbereiten
            byte[] empfangen = new byte[5];
            empfangen[0] = 0;
            empfangen[1] = 0;
            empfangen[2] = 0;
            empfangen[3] = 0;
            empfangen[4] = 0;

            //Arrays zum Senden auf dem SPI- Bus
            //vorbereiten
```

```
            byte[] datensend = new byte[8];
            //SPI Mode aktivieren
            //Datenübertragungsgeschwindigkeit ist 2Mbit/s
            //CPHA wird auf High gesetzt
            datensend[0] = 8;
            datensend[1] = 1;
            datensend[2] = 4;
            iow.SendReport(1, ref datensend[0], 8);
            //Vorbereitung des SPI - Reports
            //ID wird auf 9 gesetzt
            //3 Bytes werden gesendet
            //Es wird nur der DA-Wandler A
            //angesprochen
            datensend[0] = 9;
            datensend[1] = 3;
            datensend[2] = 32;

            int ausgang = 0;
            DateTime vorWandlung = DateTime.Now;
            do
            {
                byte datahigh = (byte)(ausgang >> 8);
                byte datalow = (byte)(ausgang&255);
                //Die Daten werden an den LTC gesendet
                datensend[3] = (byte)datahigh;
                datensend[4] = (byte)datalow;
                iow.SendReport(1, ref datensend[0], 8);
                //Alle Eingänge auf HIGH setzen
                iow.SendReport(0, ref senden[0], 5);
                //Alle Eingänge einlesen
                iow.GetReportImmediat(0, ref empfangen[0], 5);
                //Rausfiltern des Bits 7 von Port 1
                if ((empfangen[2] & 128) == 128)
                {
                    Console.WriteLine(ausgang);
                    wertErreicht = true;
                    double spannung = (double)ausgang / 65534 * 5;
                    Console.WriteLine(spannung + " V");
                }
                else
                {
                    ausgang= ausgang + 1;
                }
            }while (!wertErreicht);
            DateTime nachWandlung = DateTime.Now;
            TimeSpan spanne = nachWandlung.Subtract(vorWandlung);
            Console.WriteLine(spanne);
            iow.Quit();
            Console.ReadKey();
        }
    }
}
```

Das Programm hat diese Bildschirmausgabe zur Folge:

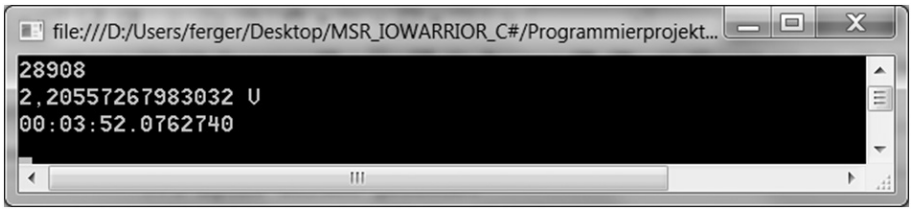


Abb. 5.16: Screenshot Spannungsmessung

5.7.1 Binäre Suche

Man sieht, dass die Wandlung sehr lange dauert. Abhilfe könnte hier eine Erhöhung der Schrittweite bringen, jedoch führt dies auch zu Ungenauigkeiten im Messwert. Eleganter ist es hier, das Prinzip der binären Suche anzuwenden:

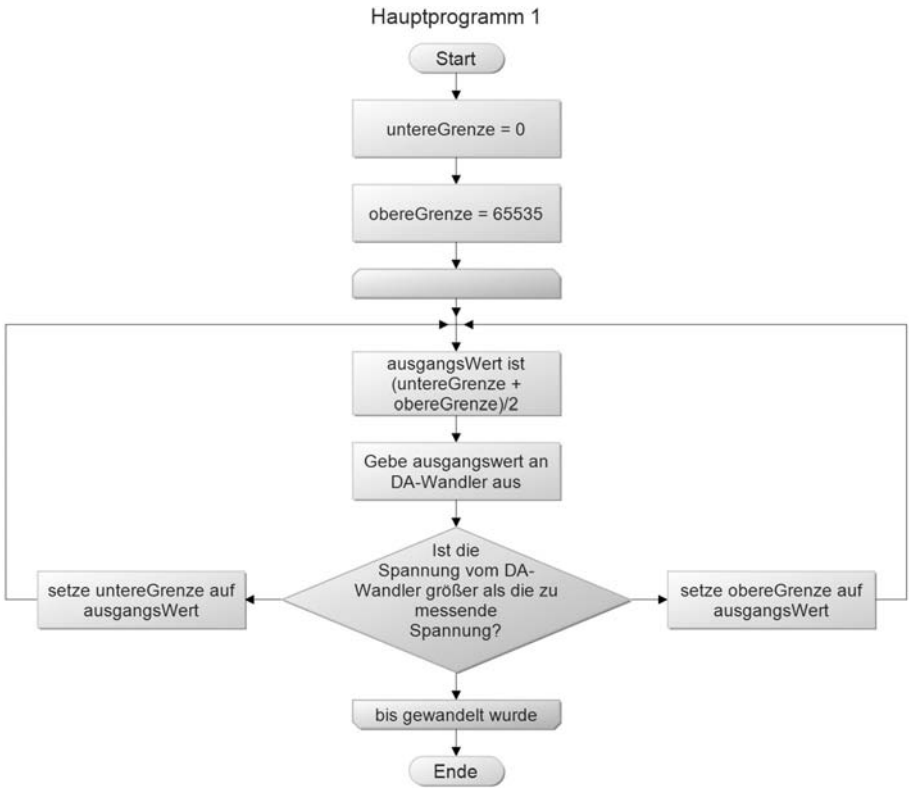


Abb. 5.17: Prinzip der binären Suche

Man gibt einen mittleren Spannungswert aus dem Startintervall von 0 V bis 5 V auf den DA-Wandler, also 2,5 V. War der Wert zu groß, nimmt man nun den mittleren Spannungswert aus dem unteren Intervall von 0 V bis 2,5 V, also 1,25 V. War der Wert zu klein, nimmt man den mittleren Spannungswert aus dem oberen Intervall von 2,5 V bis 5 V, also 3,75 V. Dieser Vorgang wiederholt sich so lange, bis die anliegende Spannung ermittelt wurde. Im unten abgebildeten Programm wird dies folgendermaßen ermittelt:

Der ausgegebene Spanungswert wird gespeichert. Gleicht der neu ausgegebene Spannungswert im nächsten Durchlauf dem alten, hat keine Veränderung mehr stattgefunden. Die Wandlung ist abgeschlossen.

Im nachfolgenden Projekt wird die Spannung nach obiger Schaltung gemessen, die Dauer der Wandlung wird am Ende ausgegeben.

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace SukzessiveApproximation16BitBinär
{
    class Program
    {
        static void Main(string[] args)
        {
           IOWKit iow = new IOWKit();
            iow.Open();
            //Array zum Senden vorbereiten
            byte[] senden = new byte[5];
            senden[0] = 0;
            senden[1] = 255;
            senden[2] = 255;
            senden[3] = 255;
            senden[4] = 255;

            //Array zum Einlesen vorbereiten
            byte[] empfangen = new byte[5];
            empfangen[0] = 0;
            empfangen[1] = 0;
            empfangen[2] = 0;
            empfangen[3] = 0;
            empfangen[4] = 0;

            //Arrays zum Senden auf dem SPI-Bus
            //vorbereiten
            byte[] datensend = new byte[8];
            //SPI Mode aktivieren
            //Datenübertragungsgeschwindigkeit ist 2Mbit/s
            //CPHA wird auf High gesetzt
```

```
            datensend[0] = 8;
            datensend[1] = 1;
            datensend[2] = 4;
            iow.SendReport(1, ref datensend[0], 8);
            //Vorbereitung des SPI-Reports
            //ID wird auf 9 gesetzt
            //3 Bytes werden gesendet
            //Es wird nur der DA-Wandler A
            //angesprochen
            datensend[0] = 9;
            datensend[1] = 3;
            datensend[2] = 32;

            int untereGrenze = 0;
            int obereGrenze = 65535;
            DateTime vorWandlung = DateTime.Now;
            int alt = -1;
            int ausgang;
            double spannung;
            do
            {
                //Die Mitte des aktuellen Intervalls wird berechnet
                ausgang = (obereGrenze + untereGrenze) / 2;
                byte datahigh = (byte)(ausgang >> 8);
                byte datalow = (byte)(ausgang & 255);
                //Die Daten werden an den LTC gesendet
                datensend[3] = (byte)datahigh;
                datensend[4] = (byte)datalow;
                iow.SendReport(1, ref datensend[0], 8);
                //Alle Eingänge auf HIGH setzen
                System.Threading.Thread.Sleep(50);
                iow.SendReport(0, ref senden[0], 5);
                //Alle Eingänge einlesen
                iow.GetReportImmediat(0, ref empfangen[0], 5);
                //Rausfiltern des Bits 7 von Port 1
                if ((empfangen[2] & 128) == 128)
                {
                    //Der ausgegebene Spannungswert wird berechnet
                    spannung = (double)ausgang / 65534 * 5;
                     Console.WriteLine("Ausgegebene Spannung: " + spannung +
"V war zu groß!!");
                    //Der Wert war zu groß, also ist das neue
                    //Intervall das Untere!
                    obereGrenze = ausgang;
                    //Es wird überprüft, ob eine Änderung am
                    //Ausgabewert sattgefunden hat. Wenn nein,
                    //wird die Schleife beendet.
                    if (ausgang == alt)
                    { break; }
                    alt = ausgang;
                }
                else
                {
                    //Der ausgegebene Spannungswert wird berechnet
```

```
                    spannung = (double)ausgang / 65534 * 5;
                     Console.WriteLine("Ausgegebene Spannung: " + spannung +
"V war zu klein!!");
                    //Der Wert war zu klein, also ist das neue
                    //Intervall das Obere!
                    untereGrenze = ausgang;
                    //Es wird geprüft, ob eine Änderung am
                    //Ausgabewert stattgefunden hat. Wenn nein,
                    //wird die Schleife beendet.
                    if (ausgang == alt)
                    { break; }
                    alt = ausgang;
                }
            } while (true);
            DateTime nachWandlung = DateTime.Now;
            //Die Dauer der Wandlung wird berechnet und ausgegeben
            TimeSpan spanne = nachWandlung.Subtract(vorWandlung);
           Console.WriteLine("Dauer der Messung: " + spanne.TotalMilliseconds
+"ms");
            Console.WriteLine("Gemessene Spannung: " + spannung);
            iow.Quit();
            Console.ReadKey();
        }
    }
}
```

Das Programm hat diese Ausgabe zur Folge:

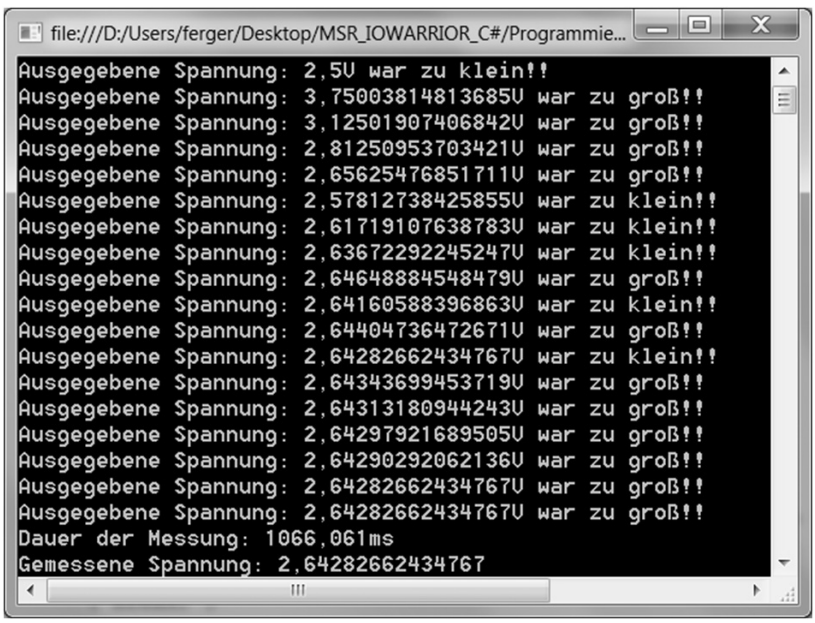


Abb. 5.18: Spannungsmessung mit binärer Suche

Das Programm ist ein klein wenig schneller, wenn die Bildschirmausgaben auf das Minimum reduziert werden:

Abb. 5.19: Spannungsmessung mit binärer Suche ohne Ballast

6 Der I²C-Bus

Der I²C-Bus ist ein serieller Bus mit nur zwei notwendigen Signalleitungen: Daten und Takt. Die benötigte Betriebsspannung und Masse werden vom USB-Bus geliefert, sodass man keinerlei sonstige Spannungsquellen verwenden muss.

Der Bus benötigt einen Masterbaustein, der den Datenfluss kontrolliert. Diese Funktion kann von den IO-Warrior übernommen werden. Der Datenausgang liegt beispielsweise beim IOW40 an Pin 0.6, der Taktausgang an Pin 0.7 an.

Das Busprinzip:

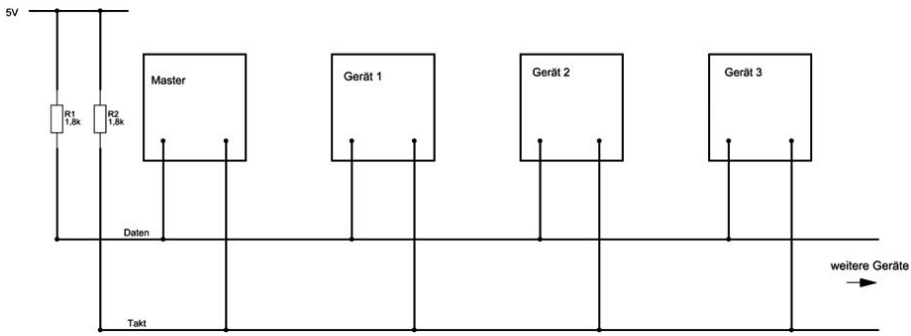


Abb. 6.1: Aufbau I²C-Bus

Wie man sieht, liegen die beiden Signalleitungen über jeweils einen 1,8-kΩ-Widerstand an 5 V. Diese Pull-up-Widerstände sind im IO-Warrior bereits integriert. Der Master sendet das Taktsignal, auch dies erledigt der IOW in den Frequenzen 50 kHz, 100 kHz und 400 kHz. IC-Bausteine wie der AT24C128 (EEPROM), DS1803 (programmierbares Potenziometer), LM75 (Temperatursensor), PCF8570 (256x8 Bit RAM), PCF8574A (8-Bit-IO-Erweiterung), PCF8591 (4-fach-AD-Wandler, 1 DA-Wandler) und SD20 (Servocontroller) sind leicht anschließ- und programmierbar. Am Beispiel des PCF8591 werden der Aufbau und die Programmierung dargestellt. Will man andere ICs in den Bus integrieren, muss man die Sendeabfolge auf dem Bus kennen:

Zu Beginn wird vom Master ein Adressbyte gesendet. Das Adressbyte setzt sich aus sieben Bit *Adresse* und einem Bit *Read/Write* zusammen. Bei den meisten ICs sind die vier höchsten Bits fest gesetzt, die drei niedrigsten sind frei einstellbar, sodass von einem IC acht Stück verwendet werden können.

Beispiel PCF8591:

Bit	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	R/W
Wert	1	0	0	1	x	x	x	0 Schreiben 1 Lesen

Setzt man also alle wählbaren Bits auf Low und will zum IC schreiben, muss das Byte 10010000 (Wert: 0x90) gesendet werden.

Anschließend werden ein oder mehrere *Control-Bytes* gesendet. Diese definieren, wie der entsprechende Baustein arbeiten soll.

6.1 Den I²C-Bus scannen

Möchte man wissen, welche Geräte an welcher Adresse auf dem Bus angeschlossen sind, kann man sich diese Informationen mit einem einfachen Programm leicht beschaffen. Auf dem I²C-Bus werden einfach alle möglichen Adressen „angefunkt". Ist eine solche Adresse im Bus nicht vorhanden, sendet der IOW einen Report mit dem Wert 128 zurück. Somit kann man alle auf dem Bus vorhandenen Adressen herausfiltern. Da die Adressen der einzelnen Bausteine teilweise festgelegt sind, kann man über eine Tabelle herausfinden, um welchen IC es sich hier handeln könnte. Eine entsprechende Tabelle ist im Anhang zu finden.

Das nachfolgende Programm scannt alle Adressen des I²C-Busses und gibt im Fall eines vorhandenen Bausteins die Adresse auf dem Bildschirm aus:

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace I²C_Scan
{
    class Program
    {
        static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            //Der Kontakt zum IOWarrior wird hergestellt.
            iow.Open();
            //Vorbereitung der Byte-Arrays zum Senden und Empfangen
```

```
            byte[] senden = new byte[8];
            for (byte i = 0; i < 8; i++)
                senden[i] = 0;
            byte[] empfangen = new byte[8];
            for (byte i = 0; i < 8; i++)
                empfangen[i] = 0;

            //I²CMode aktivieren
            senden[0] = 1;
            senden[1] = 1;
            iow.SendReport(1, ref senden[0], 8);
            //Report ID 2 zum Senden auf dem I²C Bus
            senden[0] = 0x2;
            //Start und Stop Bit gesetzt, 1 Byte wird nachfolgend gesendet
            senden[1] = 0xC1;
            for (int i = 0; i < 256; i++)
            {
                //Alle möglichen Adressen auf dem I²C-Bus werden "angefunkt"
                senden[2] = (byte)i;
                iow.SendReport(1, ref senden[0], 8);
                //Der Report vom IOW wird abgefragt
                iow.GetReport(1, ref empfangen[0], 8);
                 //hat das empfangene Byte den Wert 128 liegt ein Fehler auf
dem Bus vor
                if (empfangen[1] != 128)
                {
                    int adresse = i & 254;
                    Console.Write("\nAdresse: " + adresse);
                }
              }
            //I²CMode deaktivieren
            senden[0] = 1;
            senden[1] = 0;
            senden[2] = 0;
            senden[3] = 0;
            senden[4] = 0;
            iow.SendReport(1, ref senden[0], 8);
            iow.Quit();
            Console.ReadKey();
        }
    }
}
```

Bildschirmausgabe des laufenden Programms. Angeschlossen sind vier PCF8591:

Abb. 6.2: Screenshot I²C-Bus

Es wird jeweils zwei Mal die gleiche Adresse ausgegeben, da auf jeden IC geschrieben und gelesen werden kann und sich dann das erste Bit ändert.

6.2 PCF8591

Der Baustein PCF8591 beinhaltet vier AD-Wandler sowie einen DA-Wandler jeweils in 8 Bit Auflösung. Die Betriebsspannung (Anschluss V_{DD} Pin 16) entspricht in unserem Fall der Referenzspannung (Anschluss V_{REF} Pin 14) von 5 V und wird vom USB-Bus abgenommen. Die negative Betriebsspannung (VSS Pin 8) wird auf Masse gelegt.

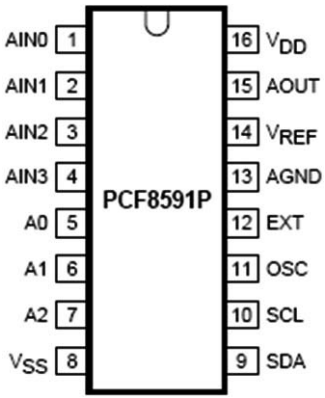


Abb. 6.3: Pinbelegung des PCF8591

An Pin 1 bis 4 können die externen analogen Signale angelegt werden. Über Pin 5 bis 7 wird die Adresse des Bausteins eingestellt. Die einzelnen Pins werden entweder auf Masse oder an 5 V gelegt. In der späteren Schaltung sollte die Möglichkeit geschaffen werden, die Adresse per Jumper einzustellen. Die Masse der analogen Eingänge kann herausgeführt werden, wird aber in diesem Anwendungsfall auf die Masse des USB-Busses gelegt. Mit Pin 12 wird eingestellt, ob man einen externen Oszillator verwenden will. Liegt dieser Pin auf Masse, kann man auch auf den Anschluss eines externen Takts (Pin 11) verzichten. An Pin 15 kann der analoge Ausgang abgenommen werden.

Bleiben nur noch die beiden I^2C-Bus-Signalleitungen. Die Datenleitung wird an Pin 9 und die Taktleitung an Pin 10 angelegt.

Die analogen Eingänge können in vier verschiedenen Modi arbeiten:

Modus 0:	Alle vier Eingänge arbeiten single ended. Das bedeutet: Alle Eingänge messen separat die Spannung gegen Masse.
Modus 1:	Drei differenzielle Eingänge sind vorhanden, wobei die Eingänge 0 bis 2 jeweils gegen Eingang 3 gemessen werden.
Modus 2:	Eingang 0 und Eingang 1 arbeiten single ended, Eingang 2 und Eingang 3 arbeiten differenziell gegeneinander.
Modus 3:	Jeweils Eingang 0, Eingang 1 und Eingang 2 und Eingang 3 arbeiten differenziell gegeneinander.

Der Modus und die Arbeitsweise des Bausteins werden mittels eines Controlbytes eingestellt, das sich folgendermaßen zusammensetzt:

Bit 0 und Bit 1: Sie bilden zusammen die Adressierung, welcher AD-Eingang ausgewählt wird. Hierbei entspricht der zusammengesetzte Zahlenwert der Eingangsnummer (also 11 für Eingang *AIN3* usw.).

Bit 2: Das Autoincrement-Flag; wird es gesetzt, erhöht sich beim Einlesen die Kanalnummer automatisch.

Bit 3: Wird fest auf den Wert 0 gesetzt.

Bit 4 und Bit 5: Diese beiden Bits bestimmen den Einlesemodus wie oben beschrieben. Auch hier entspricht der Zahlenwert dem Modus.

Bit 6: Wird dieses Bit auf 1 gesetzt, ist der analoge Ausgang aktiv.

Bit 7: Wird fest auf den Wert 0 gesetzt.

Beispiel:

Will man die vier Eingänge alle separat, also im Modus 0 verwenden und den analogen Ausgang benutzen, hat das Byte den Wert 01000000 also 0x40.

Der Schaltplan eines möglichen Analogboards sieht folgendermaßen aus:

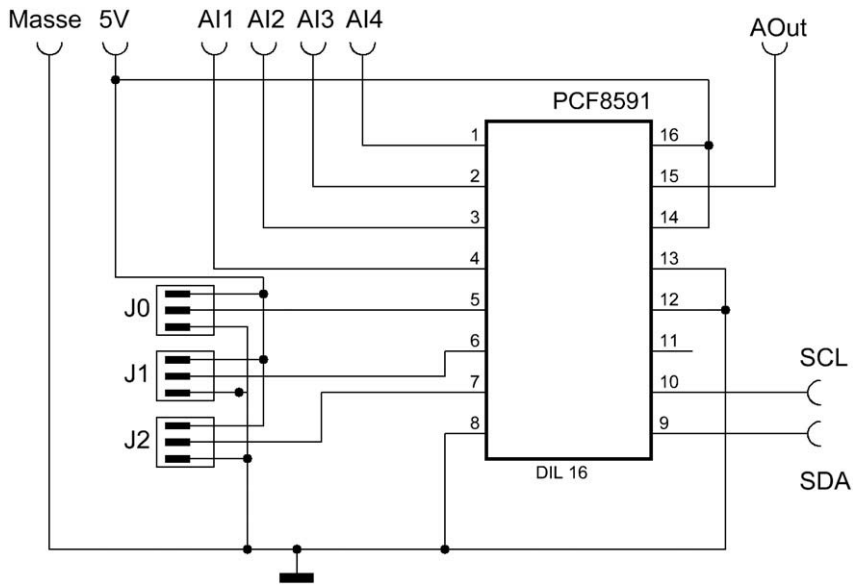


Abb. 6.4: Schaltplan Analogboard mit einem PCF8591

Die Schaltung kann leicht auf einer Rasterplatine mit etwas Silberdraht aufgebaut werden. Natürlich kann man auch eine Platine ätzen oder fräsen. Das Layout wurde mit Sprint Layout erstellt, die Projektdatei sowie die Grafik befinden sich auf der CD. Das Layout ist zweiseitig. Eine einseitige Variante findet man im Buch „Messen, Steuern und Regeln mit Visual C++“ vom Franzis Verlag auf Seite 162. Will man eingangsseitig eine eigene Masse verwenden, kann man dies mit Pin 13 realisieren. Dieser Pin wird dann nicht an Masse, sondern nach außen geführt.

6.3 Die Programmierung des IO-Warriors zur Nutzung des I²C-Busses in C#

Um den I²C-Modus zu nutzen, muss er eingeschaltet werden. Dem Datenblatt des IOW entnimmt man, dass das folgende Array an den IOW übertragen werden muss:

Byte	0	1	2	3	4	5	6	7
Wert	1	1	0	0	0	0	0	0

Will man den Modus wieder abschalten, muss sich der Wert von Byte 1 auf 0 ändern.

Kontrolldaten schreibt man nun in Form eines Arrays in den IOW. Das Array ist acht Byte groß und setzt sich folgendermaßen zusammen:

Byte	ReportID (0)	1	2	3	4	5	6	7
Inhalt	0x02	Flags	Daten	Daten	Daten	Daten	Daten	Daten

Der Wert des Bytes Flags setzt sich folgendermaßen zusammen:

Bit	0	1	2	3	4	5	6	7
Bedeutung	Anzahl der zu sendenden Bytes	Anzahl der zu sendenden Bytes	Anzahl der zu sendenden Bytes	nicht benutzt (0)	nicht benutzt (0)	nicht benutzt (0)	Stoppsequenz erzeugen	Startsequenz einleiten

Bit 0 (Wertigkeit 2^0) bis Bit 2 (Wertigkeit 2^2) ergeben zusammen die Anzahl der Bytes, die im Report gesendet werden. Bit 7 wird gesetzt, wenn eine neue Datenübertragung gestartet wird. Wird nur ein Report (also maximal sechs Datenbytes) auf den Bus geschrieben, wird auch die Stoppsequenz erzeugt. Die Stoppsequenz kann auch offen gehalten werden, wenn mehrere Reports geschrieben werden sollen. Sollen also z. B. 3 Bytes auf den Bus geschrieben werden, muss das Flag Byte den Wert 11000011 haben oder entsprechend 0xC3.

Soll also auf dem Wandler mit der Adresse 0 die Spannung 5 V (digitaler Wert 255 bei einer Auflösung von 8 Bit) ausgegeben werden, muss der Report folgendermaßen aussehen:

Byte	ReportID (0)	1	2	3	4	5	6	7
Inhalt	0x02	0xC3	0x90	0x40	0xFF	0x00	0x00	0x00

Folglich muss unter Verwendung der IOW.net.DLL die folgende Programmsequenz ausgeführt werden, um die verlangte Spannung am Ausgang des ICs zu erzeugen:

```
IOWKit iow = new IOWKit();
iow.Open();
byte[] senden = new byte[8];
for (byte i = 0; i < 8;i++)//Das Array mit Nullen füllen
     senden[i] = 0;

//I²CMode aktivieren
senden[0] = 1;
senden[1] = 1;
```

```
iow.SendReport(1, ref senden[0], 8);

//Spannung am Ausgang ausgeben
senden[0] = 0x2; //Report ID 2 zum Senden auf dem I²C Bus
senden[1] = 0xC3; //Start und Stop Bit gesetzt, 3 Bytes werden gesendet
senden[2] = 0x90; // Adresse 00, auf den IC wird geschrieben
senden[3] = 0x40; //Controllbyte: 01000000: DA wird aktiviert
senden[4] = 0xff; //Controllbyte: 01000000: DA wird aktiviert

//I²CMode deaktivieren
senden[0] = 1;
senden[1] = 1;
senden[2] = 0x00; // Adresse 00, auf den IC wird geschrieben
senden[3] = 0x00; //Controllbyte: 01000000: DA wird aktiviert
senden[4] = 0x00; //Controllbyte: 01000000: DA wird aktiviert
iow.SendReport(1, ref senden[0], 8);
//Den IOW abmelden
iow.Quit();
```

Wichtig:

Immer wenn ein Report auf den I²C-Bus geschrieben wird, antwortet der Bus mit einer Quittierung. Dieser Quittierungsreport wird im IO-Warrior gespeichert und muss abgerufen werden. Will man beispielsweise Daten vom Bus lesen und sendet entsprechende Daten auf den Bus, bekommt man zuerst eine Quittierung und erst im nächsten Report die verlangten Daten.

Der Quittungsblock setzt sich folgendermaßen zusammen:

ReportID	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x02	Flags	0x00	0x00	0x00	0x00	0x00	0x00

Das Flagbyte hat im Bit 7 ein Errorbit. Ist dieses gesetzt, hat ein Fehler stattgefunden. Die Bits 3 bis 6 sind auf den Wert 0 fest gesetzt. Die Bits 0 bis 2 ergeben zusammen die Nummer des letzten Bytes, das erfolgreich übertragen wurde.

6.4 Lesen vom I²C-Bus

Um vom Bus zu lesen, muss auch dies mit einem Report eingeleitet werden. Der Report setzt sich folgendermaßen zusammen:

Byte	ReportID	1	2	3	4	5	6	7
Bedeutung	0x03	Anzahl der Bytes, die empfangen werden sollen	Kommando	0x00	0x00	0x00	0x00	0x00

Soll also ein Wert vom AD-Eingang 0 eingelesen werden, ist der folgende Report an den IOW zu senden:

```
//Die AD-Wandlung wird eingeleitet
senden[0] = 0x03;
senden[1] = 0x02;
senden[2] = 0x91;
senden[3] = 0x00;
iow.SendReport(1, ref senden[0], 8);
```

Um die Daten abzurufen, wird ein Report nach folgender Zusammensetzung aus dem IOW ausgelesen:

ReportID	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x03	Flags	Daten	Daten	Daten	Daten	Daten	Daten

Im Byte 1 kann wieder das Bit 7 über eine Maskierung herausgefiltert werden. Ist dieses auf 1 gesetzt, lag ein Fehler vor.

Im Quellcode sieht der Auslesevorgang also beispielsweise folgendermaßen aus:

```
byte[] empfangen = new byte[8];
for (byte i = 0; i < 8; i++)
       empfangen[i] = 0;
empfangen[0] = 0x03;
iow.GetReport(1, ref empfangen[0], 8);
```

Im per Referenz übergebenen Array stehen nun ab dem Element mit der ID 2 die eingelesenen Werte und können weiterverarbeitet werden.

Im nachfolgenden Programm wird an den I²C-Bus ein AD/DA-Wandler in Form des Bausteins PCF 8591 angeschlossen. Zuerst wird auf den analogen Ausgang des Bausteins die Spannung linear von 0 auf 5 V erhöht. Anschließend wird mehrfach die Spannung am Kanal 0 eingelesen und der gemessene Wert sowie die umgerechnete Spannung auf dem Bildschirm ausgegeben.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace ADDA_Wandlung
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            IOWKit iow = new IOWKit();
            iow.Open();
            byte[] senden = new byte[8];
            for (byte i = 0; i < 8;i++)
                senden[i] = 0;
            byte[] empfangen = new byte[8];
            for (byte i = 0; i < 8; i++)
                empfangen[i] = 0;

            //I²CMode aktivieren
            senden[0] = 1;
            senden[1] = 1;
            iow.SendReport(1, ref senden[0], 8);

            senden[0] = 0x2; //Report ID 2 zum Senden auf dem I²C Bus
             senden[1] = 0xC3; //Start und Stopp Bit gesetzt, 3 Bytes werden
nachfolgend gesendet
            senden[2] = 0x90; // Adressbyte: 10010000: Adresse 00, auf den IC
wird geschrieben
            senden[3] = 0x40; //Controllbyte: 01000000: DA wird aktiviert
            for (byte wert = 0; wert < 255; wert++)
            {
                 senden[4] = wert;  //Wert der Spannung bezogen auf 8 Bit ->
255 sind ca 5V
                iow.SendReport(1, ref senden[0], 8);
                empfangen[0] = 0x03;
                iow.GetReport(1, ref empfangen[0], 8);
                System.Threading.Thread.Sleep(2);
            }
            //Lesen wird vorbereitet
            senden[0] = 0x02;
            senden[1] = 0x82;
            senden[2] = 0x90;
            senden[3] = 0x40;
            senden[4] = 0x00;
            //Die AD-Wandlung wird eingeleitet
            iow.SendReport(1, ref senden[0], 8);
            senden[0] = 0x03;
            senden[1] = 0x02;
            senden[2] = 0x91;
            senden[3] = 0x00;
            iow.SendReport(1, ref senden[0], 8);

            //Ein erster Report wird abgefragt,
            //dieser enthält allerdings noch die Quittierung
            //vom letzten vorbereitenden Befehl
            //Die AD-Wandlung wird zweimal angestoßen!
            empfangen[0] = 0x03;
            iow.GetReport(1, ref empfangen[0], 8);
            for (int i = 0; i < 8; i++)
                Console.Write(empfangen[i] + " ");
            Console.Write("\t");
            Console.ReadKey();
```

```
            //Nachfolgend wird 5000 mal ein Spannungswert eingelesen und
ausgegeben
            for (int z = 0; z < 5000; z++)
            {
               iow.SendReport(1, ref senden[0], 8); //Aufforderung an den PCF
zu wandeln
                empfangen[0] = 0x03;
                 iow.GetReport(1, ref empfangen[0], 8); //Der Report vom IOW
wird abgefragt
                Console.Write("\nEmpfangene Daten): ");
                for (int i = 0; i < 8; i++)
                    Console.Write(empfangen[i] + " ");
               double spannung = (double)empfangen[2] * 5 / 255; //Die Span-
nung wird errechnet und ausgegeben
                Console.Write("Spannung: " + spannung);
            }
            //I²CMode deaktivieren
            senden[0] = 1;
            senden[1] = 0;
            senden[2] = 0;
            senden[3] = 0;
            senden[4] = 0;
            iow.SendReport(1, ref senden[0], 8);
            iow.Quit();
            Console.ReadKey();
        }
    }
}
```

6.5 Farbmessung mit dem PCF 8591 und einem MTCS – TIAM 2

Die Firma Mazet stellt Chips zur RGB-Messung und zur LED-Regelung her. Die Chips zur RGB-Messung sind in unterschiedlichsten Ausführungen, vom reinen Sensor bis zum fertig aufgebauten USB-Modul, und für wenige Euro erhältlich. In diesem Projekt wird ein MTSC – TIAM 2 verwendet.

Abb. 6.5: Der Farbsensor

Der Baustein besitzt drei analoge Spannungsausgänge, die, je nach Farbanteil, eine entsprechende Spannung ausgeben. Drei interne Verstärker sorgen für die entsprechenden Signale. Der Verstärkungsfaktor kann über die Eingänge am IC in acht Stufen verändert werden. Die Außenbeschaltung des Bausteins:

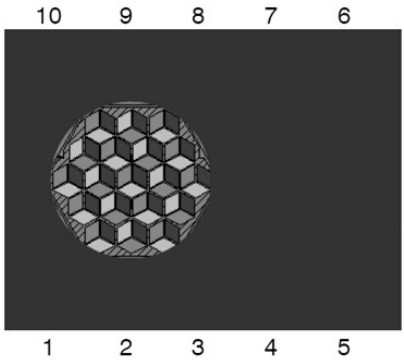


Abb. 6.6: Anschlussbelegung von oben

Pin	Beschreibung
1	Power Down: Ein Low-Signal versetzt den Baustein in den Stand-by-Modus.
2	Ausgangsspannung Y (Grün)
3	Ausgangsspannung Z (Blau)
4	Ausgangsspannung X (Rot)
5	Verstärkungsfaktoreingang 3
6	Betriebsspannung: Spannungen von 3,3 V bis 5,5 V sind hier möglich.
7	Verstärkungsfaktoreingang 2
8	Verstärkungsfaktoreingang 1 (Tabelle siehe unten)
9	Masse
10	Referenzspannung: Eine Spannung von 2,4 V ist anzulegen.

Die Verstärkungsfaktoren können über IO-Ports mit digitalen Signalen eingestellt werden. Hierbei wird die entsprechende Rückkopplungsimpedanz der internen Operationsverstärker folgendermaßen eingestellt:

Eingang 1	Eingang 2	Eingang 3	Impedanz
HIGH	HIGH	HIGH	20 MΩ
LOW	HIGH	HIGH	10 MΩ
LOW	HIGH	LOW	5 MΩ
HIGH	LOW	HIGH	2 MΩ
LOW	LOW	HIGH	1 MΩ
HIGH	LOW	LOW	500 kΩ
HIGH	HIGH	LOW	100 kΩ
LOW	LOW	LOW	25 kΩ

Im nachfolgenden Projekt wird die Impedanz 25 kΩ gewählt, indem alle Eingänge auf Masse gelegt werden. Die Betriebsspannung wird direkt vom USB-Bus entnommen. Die drei Spannungsausgänge werden auf die Eingänge eines PCF-8591-AD-Wandlers geschaltet, sodass alle drei Farbwerte über den I^2C-Bus ermittelt werden können. Genauere Messungen können mit hochauflösenden Wandlern, wie sie im Kapitel „SPI“ beschrieben sind, durchgeführt werden.

Im Programm wird ein Fenster erzeugt, das die gemessene Farbe annimmt. Die Farbwerte werden in der Konsole angezeigt.

Um die drei AD-Wandler des PCF 8591 auszulesen, muss man im Controlbit nacheinander die Bits 6 und 7 auf die Werte 00, 01 und 10 setzen. Anschließend kann man einen Report mit der ID 3 an den IO-Warrior senden und das Byte vom entsprechenden Wandler empfangen.

Nachfolgend ein kommentierter Auszug aus dem Farbmessungsprojekt. Hier wird dargestellt, wie die drei Wandler ausgelesen werden. Das komplette Projekt befindet sich auf der beiliegenden CD.

```
//Nachfolgend werden 1000 mal
//die Spannungswerte der Wandler eingelesen
//, die Farbwerte ausgegeben und die
//Farbe des Fensters verändert
for (int z = 0; z < 1000; z++)
{
     Console.SetCursorPosition(0, 0);
     //Wandler 0 wird angesprochen
     senden[0] = 0x02;
     senden[1] = 0xC2;
     senden[2] = 0x90;
     senden[3] = 0x00;
     senden[4] = 0;
     iow.SendReport(1, ref senden[0], 8);
```

```
    senden[0] = 0x03;
    senden[1] = 0x02;
    senden[2] = 0x91;
    senden[3] = 0x00;
    //Aufforderung an den PCF zu wandeln
    iow.SendReport(1, ref senden[0], 8);
    //Der Report vom IOW wird abgefragt
    iow.GetReport(1, ref empfangen[0], 8);
    rot = empfangen[2] * 2;

    //Wandler 1 wird angesprochen
    senden[0] = 0x02;
    senden[1] = 0xC2;
    senden[2] = 0x90;
    senden[3] = 01;
    //Aufforderung an den PCF zu wandeln
    iow.SendReport(1, ref senden[0], 8);

    //Einlesen des Wandlerbytes
    senden[0] = 0x03;
    senden[1] = 0x02;
    senden[2] = 0x91;
    senden[3] = 0x00;
    iow.SendReport(1, ref senden[0], 8);
    //Der Report vom IOW wird abgefragt
    iow.GetReport(1, ref empfangen[0], 8);
    gruen = empfangen[2] * 2;

    //Wandler 2 wird angesprochen
    senden[0] = 0x02;
    senden[1] = 0xC2;
    senden[2] = 0x90;
    senden[3] = 0x02;
    senden[4] = 0;
    iow.SendReport(1, ref senden[0], 8);

    senden[0] = 0x03;
    senden[1] = 0x02;
    senden[2] = 0x91;
    senden[3] = 0x00;
    iow.SendReport(1, ref senden[0], 8);
    //Der Report vom IOW wird abgefragt
    iow.GetReport(1, ref empfangen[0], 8);
    blau = empfangen[2] * 2;
    //Die Farbe des Fensters wird erzeugt und gesetzt
    Color farbe = Color.FromArgb(rot,gruen,blau);
    form.BackColor = farbe;
}
```

Die folgende Ausgabe ist möglich:

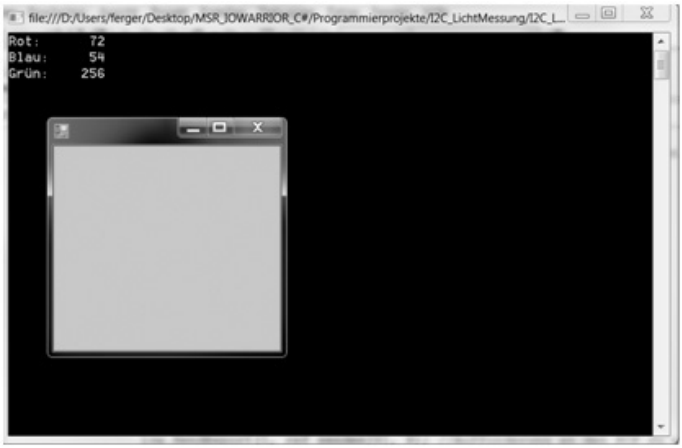


Abb. 6.7: Screenshot der Farbmessung, das Fenster hat die Farbe grün!

Eine reine Farbmessung ohne den IO-Warrior ist beispielsweise durch das Modul MTSC – C2 Colorimeter möglich. Eine kommentierte DLL ist erhältlich, Software zur direkten Messung ebenso.

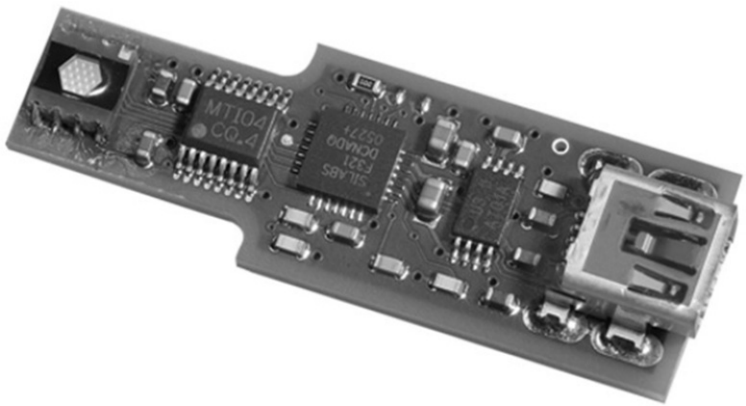


Abb. 6.8: Colorimeter mit USB-Anschluss

6.6 Temperaturmessung mit dem LM75

Ein kostengünstiger Chip für den I²C-Bus ist der LM75. Mit ihm kann man leicht ohne Außenbeschaltung eine Temperaturmessung durchführen.

Die Features des Bausteins:

- Temperaturmessung in 0,5°-Schritten
- Arbeitstemperatur von -55 °C bis 125 °C
- Betriebsspannung von 2,7 V bis 5,5 V
- I2C-fähig
- alarmfähig
- 8 Bausteine an einem Bus möglich
- Stand-by-Modus möglich
- geringer Stromverbrauch: 250 mA und 1 mA im Stand-by-Modus
- geringer Preis (ca. 1,50 €)

Verwendung des LM75 mit dem IO-Warrior und Programmierung in C#

Der Baustein wird mit einem IOW40 verwendet, ist aber auch mit allen anderen IOWs verwendbar. Er hat folgende Anschlussbelegung:

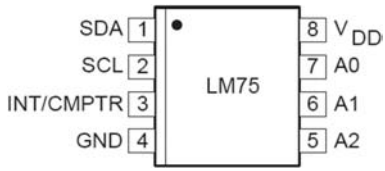


Abb. 6.9: LM75

V_{DD} wird an die 5-V-Leiste des Starterkits angeschlossen. An A0 bis A2 kann die Adresse des Bausteins eingestellt werden. In meiner Schaltung liegen diese Pins auf Masse, daher die Adresse *90* zum Schreiben und *91* zum Lesen.

SDA liegt an Pin 0.7 und SCL an Pin 0.6 des IOW. Der Steueranschluss INT/CMPTR wird nicht verwendet.

Programmierung des LM75

Verwendet wurde hier wieder die IOW.net.DLL. Zuerst wird nach dem Instanziieren eines Objekts der Specialmode I²C im IOW aktiviert.

```
byte[] senden = new byte[8];
for (byte i = 0; i < 8; i++)
    senden[i] = 0;
//I²CMode aktivieren
senden[0] = 1;
senden[1] = 1;
iow.SendReport(1, ref senden[0], 8);
```

Nun wird dem LM75 mitgeteilt, welches Register angesprochen werden soll. Dafür muss der LM75 adressiert werden und die Information erhalten, ob geschrieben

oder gelesen werden soll. Analog zum PCF 8591 hat der LM75 folgendes Adressbyte:

Bit	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	R/W
Wert	1	0	0	1	x	x	x	0 schreiben 1 Lesen

Da geschrieben werden soll und die Adresse auf 00 eingestellt ist (Pins A0 bis A2 des LM75 auf Masse), wird der Wert 0x90 übergeben.

Dann wird dem LM75 mitgeteilt, welches seiner Register angesprochen werden soll. Für die Temperaturmessung ist das Register mit der Adresse *00* verantwortlich. Dem Datenblatt des LM75 entnimmt man, dass alle anderen Bits auf 0 zu setzen sind. Also lautet der erforderliche Quelltext:

```
senden[0] = 0x2; //Report ID 2 zum Senden auf dem I²C-Bus
senden[1] = 0xC2; //Start- und Stopp-Bit gesetzt, 2 Bytes werden
                  //nachfolgend gesendet
senden[2] = 0x90; // Adressbyte: 10010000: Adresse 00, auf den
                 //IC wird geschrieben
senden[3] = 0x00; //Register 0 wird selektiert
iow.SendReport(1, ref senden[0], 8);
```

Nun wird das Lesen eingeleitet. Das Temperaturregister besteht aus zwei Bytes, deren Auswertung später erläutert wird.

Zum Lesen muss dem IOW die *ReportID 3* mitgegeben werden. Es sollen 2 Bytes empfangen werden. Die Adresse des entsprechenden Geräts ist 0x91. Dementsprechend wird der folgende Code verwendet:

```
 //Das Lesen der Temperatur wird vorbereitet
senden[0] = 0x03;//Lesen
senden[1] = 0x02;//Zwei Bytes empfangen
senden[2] = 0x91;//Andresse + Read!
iow.SendReport(1, ref senden[0], 8);
```

Nun kann mit dem üblichen Verfahren vom I²C-Bus gelesen werden.

```
byte[] empfangen = new byte[8];
for (byte i = 0; i < 8; i++)
  empfangen[i] = 0;
empfangen[0] = 0x03;
iow.GetReport(1, ref empfangen[0], 8); //Der Report vom IOW
                                       //wird abgefragt
```

Auswertung und Berechnung der Temperatur:

Das erste empfangene Byte enthält die ganzzahlige Temperatur sowie das Vorzeichen. Das zweite Byte enthält am Bit 7 nur die Information, ob zur Temperatur 0,5 °C hinzuaddiert werden müssen oder nicht.

Durch eine einfache Maskierung kann dieses Bit herausgefiltert und entsprechend der Temperaturwert ergänzt werden.

Ist beim ersten Byte das Bit 7 = 0, kann der Wert einfach direkt umgerechnet werden. Ist das nicht der Fall, muss das Bit 7 zuerst über eine Maskierung mit 0x7F genullt werden. Der Temperaturwert berechnet sich dann nach der Formel: Temperatur = 256° – Temperatur (gegebenenfalls –0,5 °C).

Im nachfolgenden Programm werden 500-mal die Daten des LM75 ausgelesen und die berechnete Temperatur wird auf dem Bildschirm ausgegeben. Beachten Sie, dass nach jedem Auslesen eine ausreichend lange Wartezeit einzufügen ist, da der LM75 nur ca. 10-mal in der Sekunde messen kann!

Der kommentierte Quellcode des Programms LM75:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using IOWarrior;

namespace LM75
{
    class Program
    {
       public static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            iow.Open();
            byte[] senden = new byte[8];
            for (byte i = 0; i < 8; i++)
                senden[i] = 0;
            byte[] empfangen = new byte[8];
            for (byte i = 0; i < 8; i++)
                empfangen[i] = 0;

            //I²CMode aktivieren
            senden[0] = 1;
            senden[1] = 1;
            iow.SendReport(1, ref senden[0], 8);

            senden[0] = 0x2; //Report ID 2 zum Senden auf dem I²C Bus
             senden[1] = 0xC2; //Start und Stop Bit gesetzt, 2 Bytes werden
nachfolgend gesendet
           senden[2] = 0x90; // Adressbyte: 10010000: Adresse 00, auf den IC
wird geschrieben
```

```
            senden[3] = 0x00; //Register 0 wird selektiert
            iow.SendReport(1, ref senden[0], 8);

            //Das Lesen der Temperatur wird vorbereitet
            senden[0] = 0x03;//Lesen
            senden[1] = 0x02;//Zwei Bytes empfangen
            senden[2] = 0x91;//Andresse + Read!
            iow.SendReport(1, ref senden[0], 8);

           //Nachfolgend wird 500-mal eine Temperatur gemessen und ausgegeben
            for (int z = 0; z < 500; z++)
            {
                Console.Clear();
                double halbe = 0;
                double temperatur = 0;
                  iow.SendReport(1, ref senden[0], 8); //Aufforderung an den
                  PCF, zu wandeln
                 empfangen[0] = 0x03;
                 iow.GetReport(1, ref empfangen[0], 8); //Der Report vom IOW
                 wird abgefragt
                Console.Write("Empfangene Daten): ");
                for (int i = 0; i < 8; i++)
                    Console.Write(empfangen[i] + " ");
                //Überprüfung, ob 0,5° dazuaddiert werden müssen
                if ((empfangen[3] & 0x80) == 0x80)
                    halbe = 0.5;
                else
                    halbe = 0;
                //Überprüfung, ob positives oder negatives
                //Vorzeichen!
                if ((empfangen[2] & 0x80) != 0x80)
                {
                    temperatur = empfangen[2];
                    temperatur = temperatur + halbe;
                }
                else
                {
                    temperatur = empfangen[2] -128;
                    temperatur = -1 * (256 - empfangen[2]);
                    temperatur = temperatur - halbe;
                }

                Console.Write("\nTemperatur: " + temperatur);
                //Der Baustein kann maximal 10 Messungen in der Sekunde vor-
                nehmen
                System.Threading.Thread.Sleep(100);
            }
            //I²CMode deaktivieren
            senden[0] = 1;
            senden[1] = 0;
            senden[2] = 0;
            senden[3] = 0;
            senden[4] = 0;
            iow.SendReport(1, ref senden[0], 8);
```

```
            iow.Quit();
            Console.ReadKey();
            return;

        }
    }
    }
```

6.7 Porterweiterung mit dem I²C-Bus und dem Baustein PCF8574A

Der IO-Warrior 24 hat zwei 8-Bit-Ports für digitale Ein- und Ausgaben. Benötigt man zusätzliche Ports, z. B. weil man schon andere Spezialfunktionen des IO-Warriors verwendet, bietet sich eine Erweiterung mittels I²C-Bus an. Der IC PCF 8574 bietet sich hier an, da für den Aufbau keinerlei Außenschaltung notwendig ist. Der Baustein kostet ca. 1 €.

Der Baustein realisiert einen 8-Bit-IO-Port. Es ist also möglich, sowohl Daten zu schreiben als auch zu lesen.

Die Pin-Belegung des Bausteins:

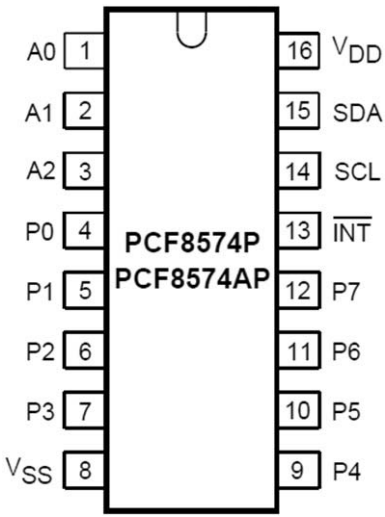


Abb. 6.10: PCF8574

Über die Pins *A0* bis *A2* wird die Adresse des Bausteins bestimmt. Acht verschiedene Kombinationen ermöglichen es, acht Porterweiterungen zu realisieren, womit ein Zu-

griff auf 64 digitale Ein- oder Ausgänge in einer Schaltung gestaltet werden kann. Im nachfolgenden Beispiel werden A0, A1 und A2 an Masse gelegt.

An VDD liegt die Betriebsspannung von 5 V an, an VSS Masse.

SDA wird an den I²C-Datenausgang des IOW angeschlossen, SCL an den I²C-Clock-Ausgang.

An P0 bis P7 können digitale Daten ausgegeben oder aber eingelesen werden.

Die Adressierung des PCF 8574

Wie bei den anderen I²C-Bausteinen hat der PCF 8574 eine 7-Bit-Adresse, wobei vier Bits fest definiert sind. Das letzte Bit im Adressbereich bestimmt, ob auf den Baustein geschrieben (LOW-Signal) oder vom Port gelesen werden soll (HIGH-Signal).

Bit	7	6	5	4	3	2	1	0
Wert	0	1	1	1	A2	A1	A0	R/W

Will man also ein Datenbyte auf den Baustein schreiben und A0 bis A2 sind auf LOW gesetzt, ist der Wert 0x70 und anschließend das Datenbyte auf den I²C-Bus zu schreiben.

Will man ein Datenbyte vom Baustein lesen, ist der Wert 0x71 auf den I²C-Bus zu schreiben. Anschließend kann man das Datenbyte empfangen.

Wichtig:

Es können immer nur acht Bits gleichzeitig gelesen oder geschrieben werden. Benötigt man Ein- und Ausgänge, kann man sich leicht mit mehreren ICs und unterschiedlichen Adressen behelfen.

Die Programmierung des PCF8574 mittels C#

Im nachfolgenden Programm werden die Ausgänge des PCF8574 10-mal wechselnd auf Low und High gesetzt. Anschließend werden die Ausgänge zu Eingängen. Es werden 1.000-mal die Eingänge eingelesen und deren Zustand auf der Konsole ausgegeben.

Der kommentierte Quelltext des Programms zum PCF 8574:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace PCF8574A_I²C_Porterweiterung
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            iow.Open();
            byte[] senden = new byte[8];
            for (byte i = 0; i < 8; i++)
                senden[i] = 0;
            byte[] empfangen = new byte[8];
            for (byte i = 0; i < 8; i++)
                empfangen[i] = 0;

            //I²CMode aktivieren
            senden[0] = 1;
            senden[1] = 1;
            iow.SendReport(1, ref senden[0], 8);
            //Report ID 2 zum Senden auf dem I²C Bus
            senden[0] = 0x2;
            //Start und Stop Bit gesetzt, 3 Bytes werden nachfolgend gesendet
            senden[1] = 0xC2;
            // Adressbyte: 01110001: Adresse 0x70, auf den IC wird geschrieben
            senden[2] = 0x70;

            for (int i = 0; i < 10; i++)
            {
                //Alle Ausgänge werden auf Low gesetzt
                senden[3] = 0x00;
                iow.SendReport(1, ref senden[0], 8);
                System.Threading.Thread.Sleep(500);
                //Alle Ausgänge werden auf High gesetzt
                senden[3] = 0xFF;
                iow.SendReport(1, ref senden[0], 8);
                System.Threading.Thread.Sleep(500);

            }
            //Report ID 3 zum Empfangen vom I²C BUs
            senden[0] = 0x03;
            //Ein Byte soll empfangen werden
            senden[1] = 0x01;
            //Adresse 0x71 zum Lesen vom Baustein
            senden[2] = 0x71;
            //senden[3] wird nicht benötigt
            senden[3] = 0x00;
            Console.SetWindowSize(140, 20);
            for (int z = 0; z < 1000; z++)
            {
                iow.SendReport(1, ref senden[0], 8);
                //Der Report vom IOW wird abgefragt
                iow.GetReport(1, ref empfangen[0], 8);
                Console.Write("\nEmpfangene Daten: ");
                for (int i = 0; i < 3; i++)
```

```
                Console.Write(empfangen[i] + " ");
            if ((empfangen[2] & 128) == 128)
                Console.Write(" Bit7 gesetzt");
            else
                Console.Write(" Bit7  nicht gesetzt");
            if ((empfangen[2] & 64) == 64)
                Console.Write(" Bit6 gesetzt");
            else
                Console.Write(" Bit6  nicht gesetzt");
            if ((empfangen[2] & 32) == 32)
                Console.Write(" Bit5 gesetzt");
            else
                Console.Write(" Bit5  nicht gesetzt");
            if ((empfangen[2] & 16) == 16)
                Console.Write(" Bit4 gesetzt");
            else
                Console.Write(" Bit4  nicht gesetzt");
            if ((empfangen[2] & 8) == 8)
                Console.Write(" Bit3 gesetzt");
            else
                Console.Write(" Bit3  nicht gesetzt");
            if ((empfangen[2] & 4) == 4)
                Console.Write(" Bit2 gesetzt");
            else
                Console.Write(" Bit2  nicht gesetzt");
            if ((empfangen[2] & 2) == 2)
                Console.Write(" Bit1 gesetzt");
            else
                Console.Write(" Bit1  nicht gesetzt");
            if ((empfangen[2] & 1) == 1)
                Console.Write(" Bit0 gesetzt");
            else
                Console.Write(" Bit0  nicht gesetzt");
        }

        //I²CMode deaktivieren
        senden[0] = 1;
        senden[1] = 0;
        senden[2] = 0;
        senden[3] = 0;
        senden[4] = 0;
        iow.SendReport(1, ref senden[0], 8);
        iow.Quit();
        Console.ReadKey();
    }
  }
}
```

Die resultierende Bildschirmausgabe (Bit 4 wurde auf Masse gelegt):

Abb. 6.11: Screenshot Porterweiterung

Wichtig:

Der Eingangszustand des Bausteins ist davon abhängig, welchen Zustand die Pins vor dem Einlesen hatten. Im obigen Fall wurden die Pins vor dem ersten Einlesen auf *High*-Signal gesetzt – daher die entsprechende Ausgabe.

6.8 Messen der Beleuchtungsstärke mit dem BH1710FVT auf dem I²C-Bus

Der BH1710FVT ist ein kleiner, aber feiner Lichtsensor, der Beleuchtungsstärken von 0 bis 65535 Lux messen kann. Der direkte Anschluss an den I²C-Bus ist realisiert, eine Außenbeschaltung ist nicht notwendig. Der Baustein ist im Handel als einzelner IC erhältlich, jedoch aufgrund seiner Abmessungen (SMD-Technik) nicht einfach zu verlöten. Da es den Baustein aber fertig aufgebaut als Zubehör des J-Controls gibt, sind einfache Beleuchtungsmessungen kein Problem.

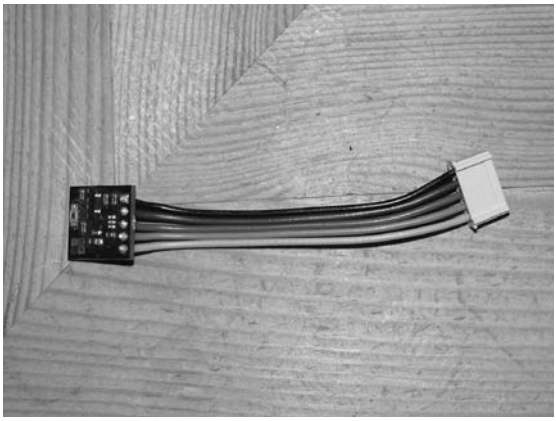


Abb. 6.12: BH1710FVT als J-Controlmodul

Die Pinbelegung des BH1710FVT:

Pin 1: Spannungsversorgung

Pin 2: Adresse; hier können je nach High oder Low zwei Adressen eingestellt werden

Pin 3: Ground

Pin 4: I^2C- Daten

Pin 5: I^2C-Referenzspannung oder Reset

Pin 6: I^2C-Clock

Ist das Adressbit auf High gelegt, besitzt der Baustein die Adresse 1011100, bei Low lautet die Adresse 0100011.

Der Baustein kann in verschiedenen Modi messen. Die Modi unterscheiden sich je nach Auflösung und Messzeit. Der hochauflösende Modus benötigt ca. 120 ms, der mittel auflösende Modus ca. 16 ms und der niedrig auflösende Modus ca. 2,9 ms pro Messung.

Die Programmierung des Bausteines ist einfach:

Zuerst wird der Baustein über ein Byte adressiert. Hierbei wird an die eingestellte Adresse eine 0 zum Schreiben und eine 1 zum Lesen vom Baustein angehängt. Da der Baustein zuerst einen Modus übermittelt bekommen muss, wird zu Beginn der Wert 0 an das Adressbyte gehängt. Anschließend wird ein Operationsbyte gesendet, das der folgenden Tabelle zu entnehmen ist:

Byte	Bedeutung
00000000	Der Baustein wird in den Stand-by-Modus versetzt.
00000001	Der Baustein wird aktiviert (nach dem Stand-by-Modus notwendig).
00000111	Der Baustein wird resettet.
00010000	Der Baustein misst kontinuierlich im hochauflösenden Modus.
00010011	Der Baustein misst kontinuierlich im mittleren auflösenden Modus.
00010110	Der Baustein misst kontinuierlich im niedrig auflösenden Modus.
00010000	Der Baustein misst einmalig im hochauflösenden Modus.
00100011	Der Baustein misst einmalig im mittleren auflösenden Modus.
00100110	Der Baustein misst einmalig im niedrig auflösenden Modus.

Nachdem die beiden Bytes auf dem I^2C-Bus abgesetzt wurden, können Daten eingelesen werden. Hierzu schickt man dem Baustein auf seiner Adresse eine angehängte 1 und kann dann ein Array von Bytes einlesen.

Der Baustein schickt die Daten mit einer Auflösung von 16 Bit, also zwei Bytes. Das höherwertige Byte steht im Array im Index 2, das niederwertige im Index 3. Die beiden Bytes müssen zusammengesetzt werden. Der zusammengesetzte Wert wird durch 1,2 geteilt. So erhält man den Wert für die Beleuchtungsstärke.

Im nachfolgenden Projekt wird 20-mal die Beleuchtungsstärke gemessen.

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;

namespace Lichtsensor
{
    class Program
    {

        static void Main(string[] args)
        {
            IOWKit iow = new IOWKit();
            iow.Open();
            byte[] senden = new byte[8];
            for (byte i = 0; i < 8; i++)
                senden[i] = 0;
            byte[] empfangen = new byte[8];
            for (byte i = 0; i < 8; i++)
                empfangen[i] = 0;

            //I²CMode aktivieren
            senden[0] = 1;
            senden[1] = 1;
            iow.SendReport(1, ref senden[0], 8);

            //Report ID 2 zum Senden auf dem I²C Bus
            senden[0] = 0x2;
           //Start und Stop Bit gesetzt, 3 Bytes werden nachfolgend gesendet
            senden[1] = 0xC2;
            // Adressbyte: 01000110 +0, auf den IC wird geschrieben
            senden[2] = 0x46;
            //Controllbyte: 00010000: Hochauflösender Modus
            senden[3] = 0x10;
            iow.SendReport(1, ref senden[0], 8);

            //Der Report vom IOW wird abgefragt
            iow.GetReport(1, ref empfangen[0], 8);
            Console.Write("\nEmpfangene Daten): ");
            for (int i = 0; i < 8; i++)
                Console.Write(empfangen[i] + " ");
            //Report ID 3 zum Lesen vom I²C Bus
```

```
            senden[0] = 0x03;
            senden[1] = 0x2;
            // Adressbyte: 0100011: + 1 vom IC wird gelesen
            senden[2] = 0x47;
            senden[3] = 0x00;
            for (int z = 0; z < 20; z++)
            {
                System.Threading.Thread.Sleep(200);
                iow.SendReport(1, ref senden[0], 8);
                //Der Report vom IOW wird abgefragt
                iow.GetReport(1, ref empfangen[0], 8);
                //das niederwertige Byte wird gelesen
                int lsb = empfangen[3];
                //das höherwertige Byte wird gelesen
                //und umgerechnet
                int msb = empfangen[2]<<8;
                //die beiden Bytes werden zusammengesetzt
                int data = lsb + msb;
                //Umrechnung in LUX
                double lux = (double)data / 1.2;
                lux = Math.Round(lux, 2);
                //Ausgabe der Daten
                Console.Write("\nEmpfangene Daten: ");
                for (int i = 0; i < 4; i++)
                    Console.Write(empfangen[i] + " ");
                Console.Write(" " + lux + " Lux");
            }
            //I²CMode deaktivieren
            senden[0] = 1;
            senden[1] = 0;
            senden[2] = 0;
            senden[3] = 0;
            senden[4] = 0;
            iow.SendReport(1, ref senden[0], 8);
            iow.Quit();
            Console.ReadKey();
        }
    }
}
```

Das Programm hat beispielsweise diese Bildschirmausgabe zur Folge:

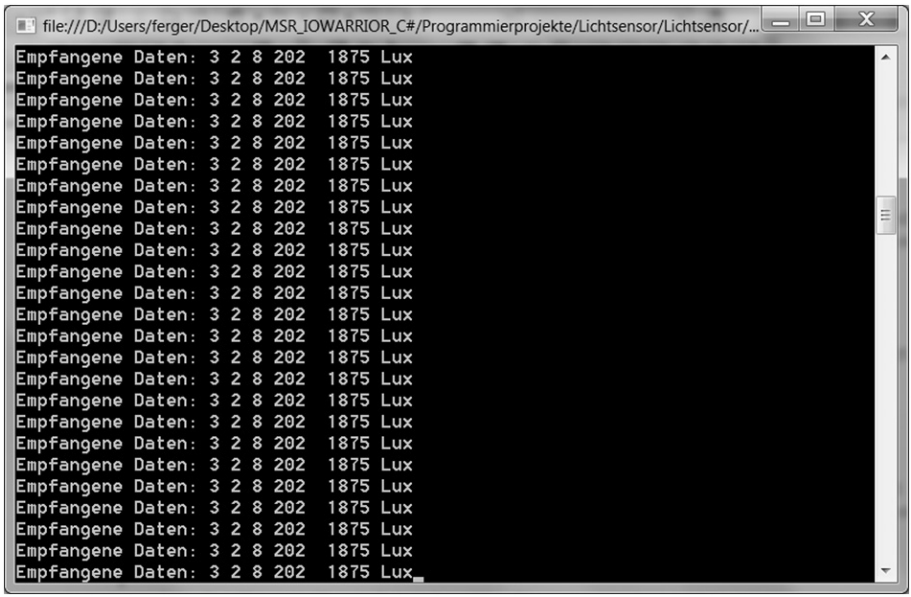


Abb. 6.13: Screenshot Lichtmessung

Eine einfache und elegante Variante des IO-Warrior 24 zur Verwendung mit dem I²C-Bus ist der I²C-Dongle:

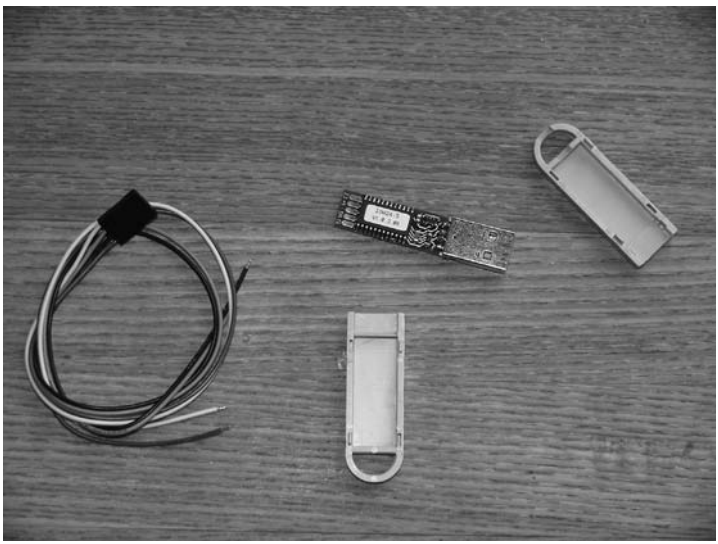


Abb. 6.14: I²C-Dongle

7 Erweiterte Programmierung mit C#

In diesem Kapitel geht es um einige Programmierhinweise, die bei der Messwertaufbereitung oder dem Zugriff auf Messwerte eine große Rolle spielen. Hierbei spielen der Zugriff auf Messwerte über das Netzwerk, die Benachrichtigung über Messereignisse per E-Mail sowie die Speicherung von Messwerten an den unterschiedlichsten Orten die größten Rollen. Weiterhin werden die Erfassung von Zeiten sowie das Abfangen von Ausnahmen behandelt.

7.1 Programmierung im Netzwerk

Die Programmierung im Netzwerk erlangt für die Mess- und Steuerungstechnik immer mehr Bedeutung. Im Grunde gibt es drei spezifische Anwendungsfälle:

1. das Ansteuern eines Interfaces über das Netzwerk,
2. das Auslesen von Daten eines Interfaces über das Netzwerk,
3. die Benachrichtigung über ein Messereignis per E-Mail empfangen.

Die beiden ersten Fälle werden im unten aufgeführten Beispielprojekt IOWarriorServer dargestellt.

Der dritte Fall wird in einem separaten Projekt behandelt.

Die Kommunikation über das Netzwerk ist dank der Entwicklung von Sockets (Programmierschnittstellen zur einfachen Verbindungserstellung im Netzwerk) denkbar einfach geworden. Ein Serversocket wird an eine IP-Adresse und an einen Port im Netzwerk gebunden (binding). Anschließend wird der Serversocket gestartet und er „hört" nun in das Netzwerk hinein (listening). Geht eine Verbindung auf dem entsprechenden Port ein, akzeptiert der Server sie (accept) und stellt einen Arbeits-Socket zur Verfügung. Über den Arbeits-Socket kann der Server nun mit dem Client Daten austauschen. Als Client dient beispielsweise ein einfacher Telnetclient, den man wohl auf jedem gängigen Betriebssystem finden wird. Bei Windows Vista und Windows 7 muss dieser Dienst in der Systemsteuerung unter *Programme und Funktionen -> Windows Features aktivieren oder deaktivieren* aktiviert werden.

Um die Netzwerkklassen nutzen zu können, müssen die Bibliotheken *System.Net* und *System.Net.Sockets* eingebunden werden. Für die *Stream*-Operationen muss die Bibliothek *System.IO* eingebunden werden.

Der Serversocket nennt sich in C# *TCPListener*. Er stellt nach der Instanziierung die Methoden *Start, AcceptTCPClient* und *Stop* zur Verfügung.

Die Klasse *TCPClient* erlaubt es, einen Datenstrom (NetworkStream) über die Methode *GetStream* zu generieren. Über diesen Datenstrom kann ungepuffert mit dem Client kommuniziert werden. Für die Datenausgabe ist dies ausreichend. Die Dateneingabe vom Client zum Server wird über einen gepufferten Strom realisiert, sodass mit *Enter* bestätigte Zeilen vom Client auch zeilenweise eingelesen werden können. Die Vorbereitung der eigentlichen Kommunikation könnte also folgendermaßen lauten:

```
//Der Listener wird an den Localhost auf Port 10000 gebunden
TcpListener  serverSocket  =  new  TcpListener(IPAddress.Parse("127.0.0.1"),
10000);
TcpClient client = default(TcpClient);
//Der ServerSocket wird gestartet (listen)
serverSocket.Start();
//eine eingehende Verbindung wird angenommen
client = serverSocket.AcceptTcpClient();
//Vorbereitung der Streams, der Eingabestrom
//ist gepuffert
NetworkStream netzwerkStrom = client.GetStream();
BufferedStream gepuffert = new BufferedStream(netzwerkStrom);
StreamReader reader = new StreamReader(gepuffert);
```

Das Objekt aus der Klasse *BufferedStream* stellt die Methode *ReadLine* zur Verfügung, die die eingegebene Zeile vom Client als String zurückgibt. Das Objekt der Klasse *NetworkStream* stellt die Methode *write* zur Verfügung. Diese Methode bekommt ein Bytearray, die erste Position und die Länge des Arrays übergeben. Eine kleine Hilfsfunktion wandelt den zu sendenden String in ein Bytearray um (siehe Projekt *IOWarriorServer*).

7.2 Das Projekt *IOWarriorServer*

In diesem Projekt wartet der Server auf eine eingehende Verbindung. Ist diese zustande gekommen, hat der Client die Möglichkeit, bei einem IO-Warrior die Ausgänge zu setzen, den Zustand der Eingangspins abzufragen oder die Kommunikation zu beenden.

Der kommentierte Quellcode des Projekts *IOWarriorServer*:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Sockets;
using System.IO;
```

```
using System.Net;
using IOWarrior;

namespace IOWarriorServer
{
    class Program
    {
        //Hilfsfunktion, die einen übergebenen String in
        //ein Byte-Array umwandelt.
        static byte[] getBytes(String s)
        {
            return Encoding.ASCII.GetBytes(s);
        }

        static void Main(string[] args)
        {
            //Variable zum Beenden der Verbindung
            bool endCommunication = false;
            //IO-Warrior Objekt erzeugen und öffnen
            IOWKit iow = new IOWKit();
            iow.Open();
            //Der Listener wird an den Localhost auf Port 10000 gebunden
                 TcpListener serverSocket = new TcpListener(IPAddress.Par-
se("127.0.0.1"), 10000);
            TcpClient client = default(TcpClient);
            //Der ServerSocket wird gestartet (listen)
            serverSocket.Start();
            while (true)
            {
                //eine eingehende Verbindung wird angenommen
                client = serverSocket.AcceptTcpClient();
                //Vorbereitung der Streams, der Eingabestrom
                //ist gepuffert
                NetworkStream netzwerkStrom = client.GetStream();
               BufferedStream gepuffert = new BufferedStream(netzwerkStrom);
                StreamReader reader = new StreamReader(gepuffert);
                //Das byte-Array senden dient der Zeichenübertragung
                //an den Client
                byte[] senden;
                while (!endCommunication)
                {
                    //Ausgabe des Menüs an den Client
                senden = getBytes("\n\rWillkommen beim IOWarrior-Server\n\r");
                    netzwerkStrom.Write(senden, 0, senden.Length);
                    senden = getBytes("Wollen Sie:\n\r");
                    netzwerkStrom.Write(senden, 0, senden.Length);
                    senden = getBytes("Ausgaenge setzen          (1)\n\r");
                    netzwerkStrom.Write(senden, 0, senden.Length);
                    senden = getBytes("Eingaenge einlesen        (2)\n\r");
                    netzwerkStrom.Write(senden, 0, senden.Length);
                    senden = getBytes("Die Kommunikation beenden (0)\n\r");
                    netzwerkStrom.Write(senden, 0, senden.Length);
                    senden = getBytes("Ihre Auswahl bitte: ");
                    netzwerkStrom.Write(senden, 0, senden.Length);
```

```
                    //Auf die Eingabe des Clients wird gewartet
                    string eingang = reader.ReadLine();
                    //Ist die Eingabe eine 0, wird die Schleife verlassen
                    if (eingang.Equals("0"))
                    {
                        endCommunication = true;
                        //end = true;
                        break;
                    }
                    else
                    {
                       //Die Auswahl war 1, die Ausgänge sollen gesetzt werden
                        if (eingang.Equals("1"))
                        {
                            //Instruktionen an den Client
                                    senden = getBytes("\n\rAusgaenge setzen
(1)\n\r");
                            netzwerkStrom.Write(senden, 0, senden.Length);
                              senden = getBytes("\n\rGeben Sie die Werte der
vier Ports\n\r");
                            netzwerkStrom.Write(senden, 0, senden.Length);
                             senden = getBytes("\n\rgetrennt durch ein Leer-
zeichen ein: ");
                            netzwerkStrom.Write(senden, 0, senden.Length);
                            eingang = reader.ReadLine();
                               //Der eingelesene String wird in vier Strings
zerlegt
                            String []sports = eingang.Split(' ');
                            if (sports.Length == 4)
                            {
                              //Die Ausgänge des IO-Warriors werden entspre-
chend
                                //der Eingabe des Clients gesetzt
                                byte[] ports = new byte[5];
                                ports[0] = 0;
                                ports[1] = Convert.ToByte(sports[0]);
                                ports[2] = Convert.ToByte(sports[1]);
                                ports[3] = Convert.ToByte(sports[2]);
                                ports[4] = Convert.ToByte(sports[3]);
                                iow.SendReport(0, ref ports[0], 5);
                                System.Threading.Thread.Sleep(1000);
                            }
                        }
                        else
                            if (eingang.Equals("2"))
                            {
                              //Die Auswahl war 2. Nachdem alle Ausgänge des
IO-Warriors
                                 //auf High gesetzt wurden, werden alle Ein-
gangsports
                                //nicht blockierend eingelesen
                                   senden = getBytes("\n\rEingaenge einlesen
(2)\n\r");
                              netzwerkStrom.Write(senden, 0, senden.Length);
```

```
                                byte[] einlesen = new byte[5];
                                einlesen[0] = 0;
                                einlesen[1] = 255;
                                einlesen[2] = 255;
                                einlesen[3] = 255;
                                einlesen[4] = 255;
                                iow.SendReport(0, ref einlesen[0], 5);
                               iow.GetReportImmediat(0, ref einlesen[0], 5);
                               //Die eingelesenen Daten werden formatiert an
den
                                //Client ausgegeben
                                String ausgabe = "Port0: " + einlesen[1]
+ " Port1: " + einlesen[2] +" Port2: " + einlesen[3] +" Port3: " + einlesen[4]+"\
n\r";
                                senden = getBytes(ausgabe);
                              netzwerkStrom.Write(senden, 0, senden.Length);
                            }
                            else
                            {
                               //Der Client hat eine falsche Auswahl getrof-
fen
                           senden = getBytes("\n\rFalsche Auswahl!!!!\n\r");
                              netzwerkStrom.Write(senden, 0, senden.Length);
                           }
                    }
                }
                //Die Verbindng zum Client wird getrennt
                client.Close();
                //Eine neue Verbindung kann eingegangen werden,
                //da der ServerSocket immer noch am "hören" ist
                endCommunication = false;
            }
        }
    }
}
```

Damit das Programm funktionieren kann, muss die interne Firewall des Servers den betroffenen Port (hier 10000) zulassen.

Kommunikation über das Netzwerk und über Streams ist immer fehleranfällig. Beispielsweise kann der Client die Kommunikation unerwartet abbrechen oder das Netzwerk kann plötzlich nicht mehr zur Verfügung stehen. In diesen Fällen werden Ausnahmen (Exceptions) das Programm unterbrechen. Diese Exceptions können über try-catch-Blöcke abgefangen werden. Näheres hierüber erfahren Sie in der einschlägigen Literatur.

Das Programm könnte folgenden Ablauf beim Client haben:

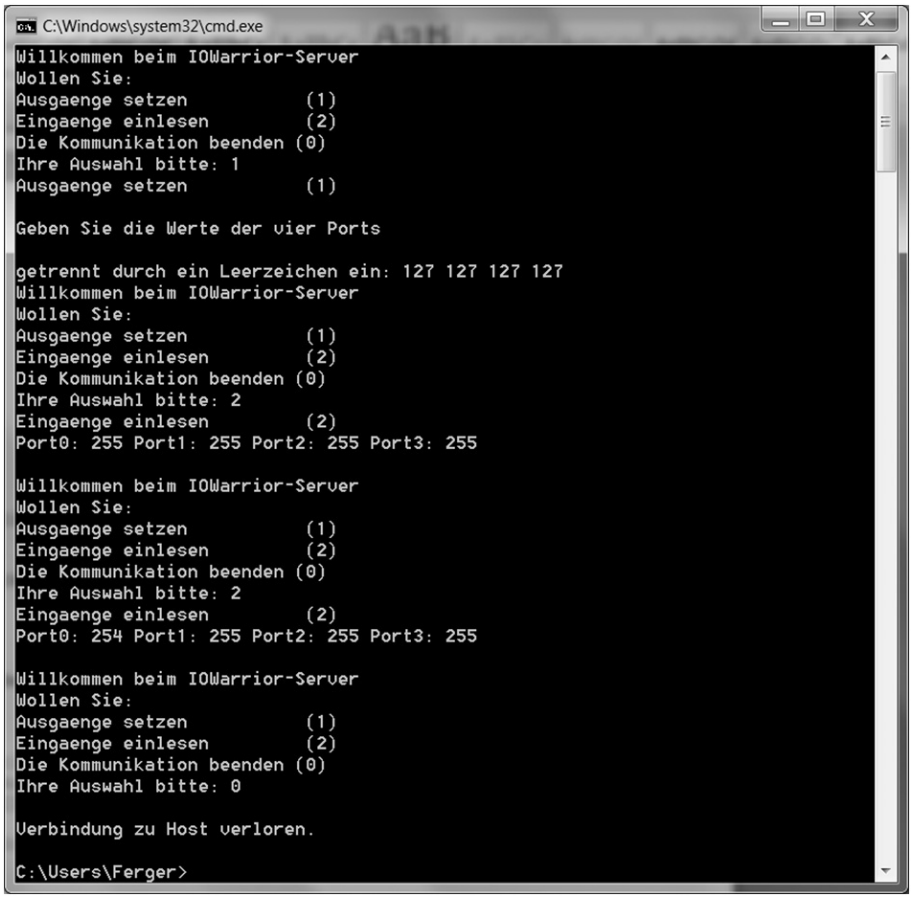


Abb. 7.1: Screenshot des IO-Warriorservers

7.3 Versenden einer Mail in Abhängigkeit eines Mess-Ereignisses

Auch das Versenden einer Mail ist unter C# ein Kinderspiel. Benötigt werden die Zugangsdaten der entsprechenden Server und die Bibliothek *System.Net.Mail.* Anschließend werden die Benutzerinformationen (Benutzername und Passwort) in den entsprechenden Attributen gesetzt.

Der Mail-Versand wird über das Protokoll *SMTP* realisiert. Hierzu bietet uns C# eine Klasse *SmtpClient*. Beim Aufruf des Konstruktors werden die SMTP-Serveradresse (als

String) und der Port (gewöhnlich 587) übergeben. Mittels der Klasse *NetworkCredentials* wird ein entsprechendes Objekt instanziiert.

Nun bekommt der SMTP-Client die Nachricht in der Methode *Send* übergeben. Eine Nachricht wird in der Klasse *MailMessage* gekapselt. Die Attribute *Subject, Body,IsBodyHtml* und *Priority* sind selbsterklärend. Einzig im Konstruktor werden zwei Mail-Adressen übergeben: der Sender und der Empfänger. Die Mail-Adressen werden in der Klasse *MailAddress* realisiert, übergeben werden im Konstruktor die Mail-Adresse und der Alias als String.

Der allgemeine Ablauf beim Verschicken einer Mail mit C#:

```
MailAddress from = new MailAddress(„joferger@web.de“, „Jochen Ferger“);

//Anlegen der Empfangsadresse
MailAddress to = new MailAddress(“ferger@fds.limburg.de”, “Jochen Ferger”);

//Instanziieren ener neuen Nachricht
//Parameter werden gesetzt
MailMessage mailmessage = new MailMessage(from, to);
mailmessage.Subject = “Warnung”;
mailmessage.Body = “Die gemessene Temperatur liegt über 27 °Celsius”;
mailmessage.IsBodyHtml = false;
mailmessage.Priority = MailPriority.High;

// SMTP Server benötigt ggf. eine Authentfizierung!
SmtpClient emailClient = new SmtpClient(“smtp.web.de”, 587);
System.Net.NetworkCredential SMTPUserInfo = new System.Net.
NetworkCredential(“joferger”, “passwort”);
emailClient.UseDefaultCredentials = false;
emailClient.Credentials = SMTPUserInfo;
//Die Nachricht wird verschickt
emailClient.Send(mailmessage);
```

Im nachfolgenden Projekt ist ein Temperatursensor über den I²C-Bus an einen IO-Warrior angeschlossen. Die Temperatur wird ständig erfasst, beispielsweise wie in einem Serverraum. Übersteigt die Temperatur einen Wert von 26,5 °C, wird dem Administrator eine entsprechende Mail geschickt.

Der kommentierte Quellcode des Projekts *Temperaturserver*:

```
using System;
using System.Linq;
using IOWarrior;
using System.Net.Mail;

namespace TemperaturServer
{
    class Program
    {
```

```
static void Main(string[] args)
{
    IOWKit iow = new IOWKit();
    iow.Open();
    byte[] senden = new byte[8];
    for (byte i = 0; i < 8; i++)
        senden[i] = 0;
    byte[] empfangen = new byte[8];
    for (byte i = 0; i < 8; i++)
        empfangen[i] = 0;
    //I²CMode aktivieren
    senden[0] = 1;
    senden[1] = 1;
    iow.SendReport(1, ref senden[0], 8);
    //Vorbereitungen zur Datenkommunikation
    senden[0] = 0x2;
    senden[1] = 0xC2;
    senden[2] = 0x90;
    senden[3] = 0x00;
    iow.SendReport(1, ref senden[0], 8);
    iow.GetReport(1, ref empfangen[0], 8);
    //Das Lesen der Temperatur wird vorbereitet
    senden[0] = 0x03;//Lesen
    senden[1] = 0x02;//Zwei Bytes empfangen
    senden[2] = 0x91;//Andresse + Read!
    iow.SendReport(1, ref senden[0], 8);
    iow.GetReport(1, ref empfangen[0], 8);
    double temperatur = 0;
    do{
        System.Threading.Thread.Sleep(100);
        double halbe = 0;

        //Aufforderung an den Sensor zu messen
        iow.SendReport(1, ref senden[0], 8);

        //Der Report vom IOW wird abgefragt
        iow.GetReport(1, ref empfangen[0], 8);
        Console.Write("Empfangene Daten: ");
        for (int i = 0; i < 4; i++)
            Console.Write(empfangen[i] + " ");
        Console.WriteLine();

        //Überprüfung, ob 0,5° dazu addiert werden müssen
        if ((empfangen[3] & 0x80) == 0x80)
            halbe = 0.5;
        else
            halbe = 0;

        //Überprüfung, ob positives oder negatives
        //Vorzeichen!
        if ((empfangen[2] & 0x80) != 0x80)
        {
            temperatur = empfangen[2];
```

```
                    temperatur = temperatur + halbe;
                }
                else
                {
                    temperatur = empfangen[2] - 128;
                    temperatur = -1 * (256 - empfangen[2]);
                    temperatur = temperatur - halbe;
                }
              }
              while (temperatur < 27) ;

            //Anlegen der Absendeadresse
              MailAddress from = new MailAddress("joferger@web.de", "Jochen
Ferger");

            //Anlegen der Empfangsadresse
           MailAddress to = new MailAddress("ferger@fds.limburg.de", "Jochen
Ferger");

            //Instanziieren einer neuen Nachricht
            //Parameter werden gesetzt
            MailMessage mailmessage = new MailMessage(from, to);
            mailmessage.Subject = "Warnung";
            mailmessage.Body = "Die gemessene Temperatur liegt über 27° Cel-
sius";
            mailmessage.IsBodyHtml = false;
            mailmessage.Priority = MailPriority.High;

            // SMTP Server benötigt ggf. eine Authentfizierung!
            SmtpClient emailClient = new SmtpClient("smtp.web.de", 587);
                System.Net.NetworkCredential SMTPUserInfo = new System.Net.
NetworkCredential("joferger", "passwort");
            emailClient.UseDefaultCredentials = false;
            emailClient.Credentials = SMTPUserInfo;
            //Die Nachricht wird verschickt
            emailClient.Send(mailmessage);
            Console.WriteLine("Message Sent");

            //I²CMode deaktivieren
            senden[0] = 1;
            senden[1] = 0;
            senden[2] = 0;
            senden[3] = 0;
            senden[4] = 0;
            iow.SendReport(1, ref senden[0], 8);
            iow.Quit();
            Console.ReadKey();
        }
    }
}
```

Bildschirmausgabe des Mail-Projekts:

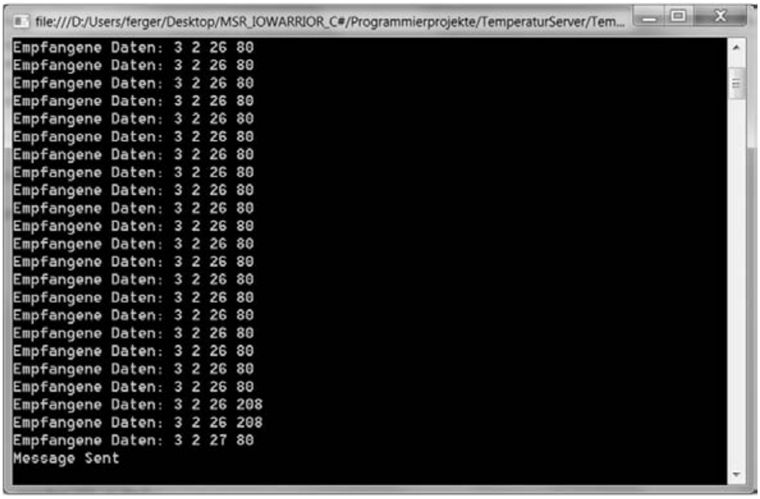


Abb. 7.2: Screenshot Temperaturmessung mit Mail-Versand

Empfangene Mail:

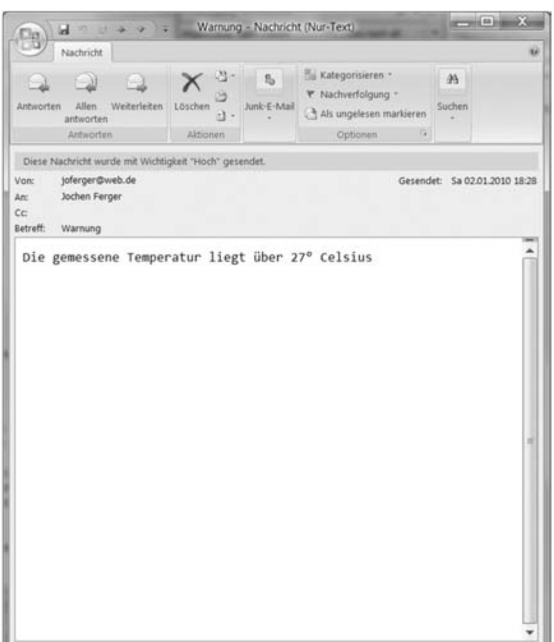


Abb. 7.3: Empfangene E-Mail

7.4 Messdaten erfassen

7.4.1 Messwerte in eine Textdatei schreiben

Messdaten können mittels C# in Textdateien geschrieben werden, um sie später auszuwerten. Die Bibliothek *System.IO* muss im *Using*- Bereich eingebunden werden.

```
using System.IO;
```

Anschließend wird eine Datei geöffnet oder angelegt:

```
FileStream datei = new FileStream(„Messdaten.txt“, FileMode.OpenOrCreate);
```

Ein Datenstrom wird an die Datei gekoppelt, um Schreibzugriffe zu vereinfachen:

```
StreamWriter stream = new StreamWriter(datei);
```

In die Datei können nun über die Methode *WriteLine* Strings geschrieben werden:

```
stream.WriteLine(“Messdaten”);
```

Sind die Dateioperationen beendet, müssen der Datenstrom und die Datei jeweils über die Methode Close geschlossen werden:

```
stream.Close();
datei.Close();
```

Im nachfolgenden Projekt wird über den I^2C-Bus die Temperatur gemessen und in eine Textdatei geschrieben.

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using IOWarrior;
using System.IO;

namespace MesswerteInEineTextdateiSchreiben
{
    class Program
    {
        static void Main(string[] args)
        {
            //IOwarrior vorbereiten
            IOWKit iow = new IOWKit();
            iow.Open();
```

```
            byte[] senden = new byte[8];
            for (byte i = 0; i < 8; i++)
                senden[i] = 0;
            byte[] empfangen = new byte[8];
            for (byte i = 0; i < 8; i++)
                empfangen[i] = 0;
            //I²CMode aktivieren
            senden[0] = 1;
            senden[1] = 1;
            iow.SendReport(1, ref senden[0], 8);
            //Vorbereitungen zur Datenkommunikation
            senden[0] = 0x2;
            senden[1] = 0xC2;
            senden[2] = 0x90;
            senden[3] = 0x00;
            iow.SendReport(1, ref senden[0], 8);
            iow.GetReport(1, ref empfangen[0], 8);
            //Das Lesen der Temperatur wird vorbereitet
            senden[0] = 0x03;//Lesen
            senden[1] = 0x02;//Zwei Bytes empfangen
            senden[2] = 0x91;//Andresse + Read!
            iow.SendReport(1, ref senden[0], 8);
            iow.GetReport(1, ref empfangen[0], 8);
            double temperatur = 0;

            //Eine Datei wird angelegt bzw. geöffnet
               FileStream datei = new FileStream("Messdaten.txt", FileMode.
OpenOrCreate);
            //An die Datei wird ein Datenstrom zum Schreiben gekoppelt.
            StreamWriter stream = new StreamWriter(datei);
            stream.WriteLine("Messdaten");
            for (int i = 0; i < 20; i++)
            {
                System.Threading.Thread.Sleep(500);
                double halbe = 0;

                //Aufforderung an den Sensor zu messen
                iow.SendReport(1, ref senden[0], 8);

                //Der Report vom IOW wird abgefragt
                iow.GetReport(1, ref empfangen[0], 8);

                //Überprüfung, ob 0,5° dazu addiert werden müssen
                if ((empfangen[3] & 0x80) == 0x80)
                    halbe = 0.5;
                else
                    halbe = 0;

                //Überprüfung, ob positives oder negatives
                //Vorzeichen!
                if ((empfangen[2] & 0x80) != 0x80)
                {
                    temperatur = empfangen[2];
                    temperatur = temperatur + halbe;
```

```
                }
                else
                {
                    temperatur = empfangen[2] - 128;
                    temperatur = -1 * (256 - empfangen[2]);
                    temperatur = temperatur - halbe;
                }
                //Der String zur Messung wird zusammengesetzt
              string messung = (i + 1) + ". Messung: " + temperatur + "°Celsius";
                //und in die Datei geschrieben
                stream.WriteLine(messung);
            }

            //Datenstrom schließen
            stream.Close();
            //Datei schließen
            datei.Close();

            //I²CMode deaktivieren
            senden[0] = 1;
            senden[1] = 0;
            senden[2] = 0;
            senden[3] = 0;
            senden[4] = 0;
            iow.SendReport(1, ref senden[0], 8);
            iow.Quit();

        }
    }
}
```

Die Datei sieht nun folgendermaßen aus:

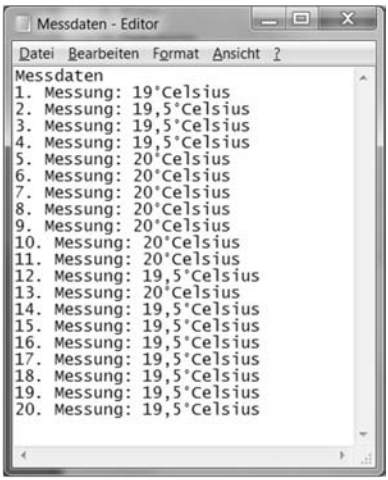


Abb. 7.4: Messdaten in der Textdatei

7.4.2 Messdaten in eine Excel-Datei schreiben

Dank des *Com*-Objektmodells von Microsoft ist es einfach, Applikationen wie Word, Excel etc. von anderen Programmen aus zu verwenden. Beispielsweise können sehr einfach Messwerte in eine *Excel*-Datei geschrieben werden.

Hierzu muss in einem neuen Projekt der Verweis hinzugefügt werden, dass die *Excel-Com*-Objekte verwendet werden. Man findet diese im Menüpunkt *Verweis hinzufügen* (im Projektmappenexplorer mit der rechten Maustaste auf das aktuelle Projekt klicken) und dort im Reiter COM den Komponentennamen *Microsoft Excel 12.0 Object Library*.

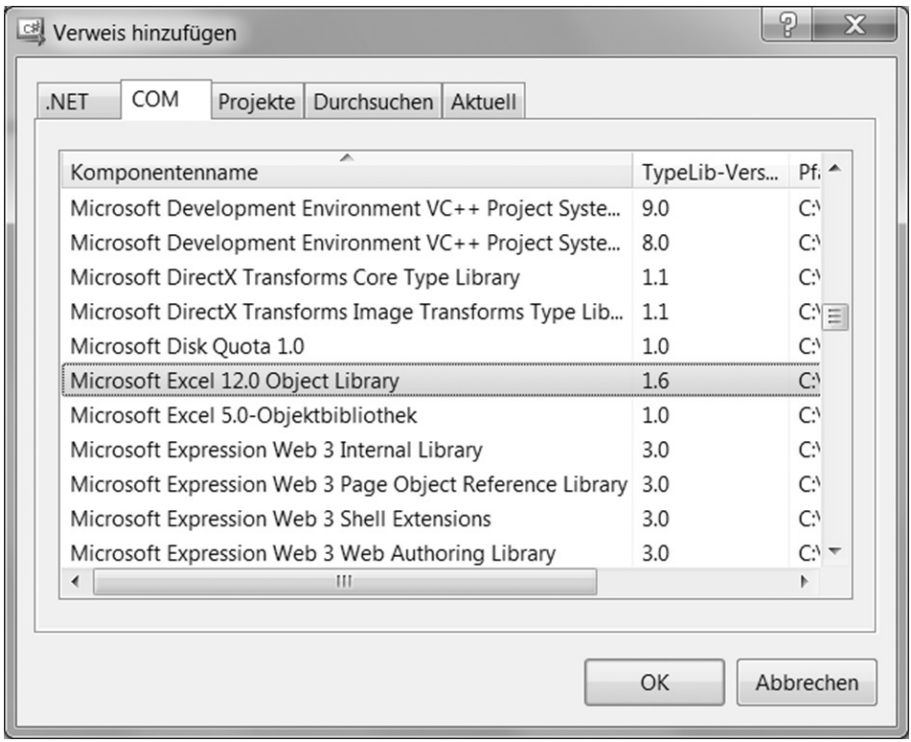


Abb. 7.5: *Excel*-Komponente hinzufügen

Dieses Projekt bezieht sich auf Excel 2007. Für andere Versionen müssen kompatible *Com*-Bibliotheken eingebunden werden.

Die Bibliothek muss im *Using*-Bereich eingebunden werden:

```
using Excel = Microsoft.Office.Interop.Excel;
```

Die Verwendung der einzelnen Objekte und deren Methoden ist einfach:

Excel wird gestartet:

```
Excel.Application applikation = new Excel.Application();
```

Ein Dokument wird angelegt:

```
Excel.Workbook dokument = (Excel.Workbook)(applikation.Workbooks.Add(System.
Reflection.Missing.Value)); ;
```

Ein Datenblatt wird im Dokument aktiviert:

```
Excel.Worksheet datenBlatt = (Excel.Worksheet)dokument.ActiveSheet; ;
```

Daten können in das Sheet geschrieben werden:

```
datenBlatt.Cells[1, 1] = "Uhrzeit";    // Zelle A1
datenBlatt.Cells[1, 2] = "Temperatur";   // Zelle A2
.
.
.
```

Das Sheet wird gespeichert und geschlossen:

```
dokument.Close(true, "messwerte.xlsx", System.Reflection.Missing.Value);
```

Excel wird beendet:

```
applikation.Quit();
```

Im nachfolgenden Projekt wird die Temperatur über den I^2C-Bus gemessen und alle 0,5 Sekunden mit der entsprechenden Zeit in eine *Excel*-Datei geschrieben. Wie die Uhrzeit aktualisiert und als String zusammengesetzt wird, kann man dem Quelltext entnehmen.

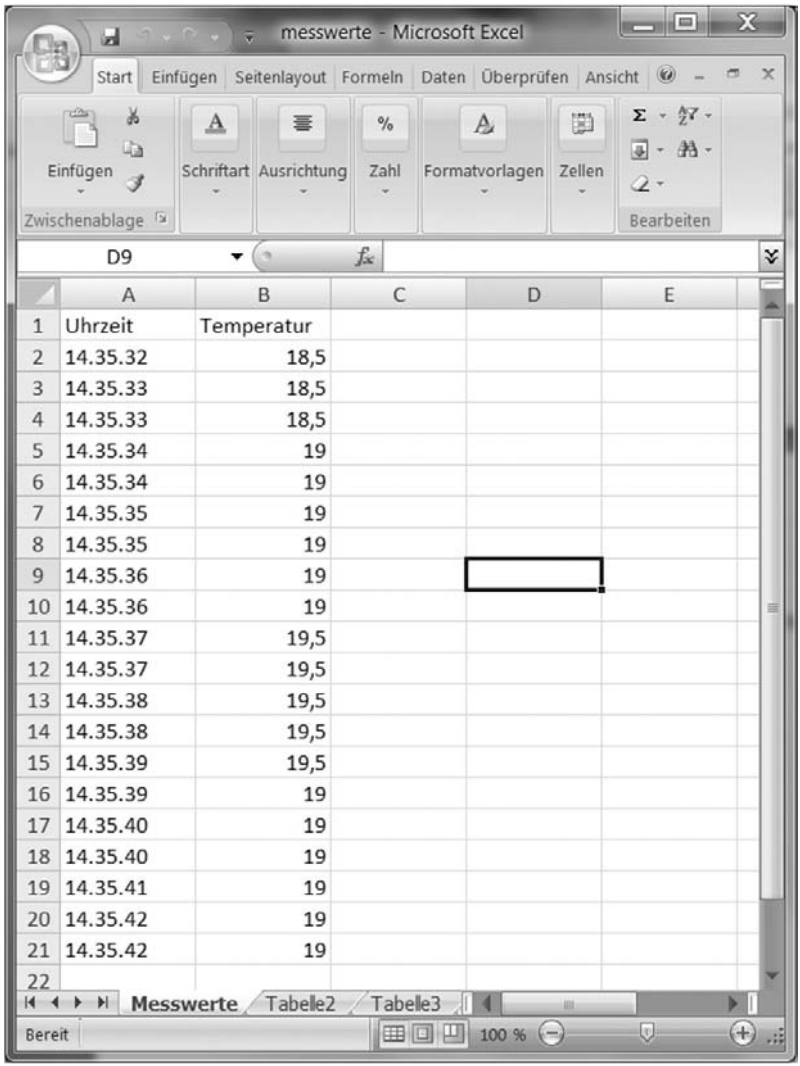
Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Excel = Microsoft.Office.Interop.Excel;
using IOWarrior;

namespace MesswerteInExcelSchreiben
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            //IoWarrior vorbereiten
            IOWKit iow = new IOWKit();
            iow.Open();
            byte[] senden = new byte[8];
            for (byte i = 0; i < 8; i++)
                senden[i] = 0;
            byte[] empfangen = new byte[8];
            for (byte i = 0; i < 8; i++)
                empfangen[i] = 0;

            //I²CMode aktivieren
            senden[0] = 1;
            senden[1] = 1;
            iow.SendReport(1, ref senden[0], 8);

            //Vorbereitungen zur Datenkommunikation
            senden[0] = 0x2;
            senden[1] = 0xC2;
            senden[2] = 0x90;
            senden[3] = 0x00;
            iow.SendReport(1, ref senden[0], 8);
            iow.GetReport(1, ref empfangen[0], 8);

            //Das Lesen der Temperatur wird vorbereitet
            senden[0] = 0x03;//Lesen
            senden[1] = 0x02;//Zwei Bytes empfangen
            senden[2] = 0x91;//Andresse + Read!
            iow.SendReport(1, ref senden[0], 8);
            iow.GetReport(1, ref empfangen[0], 8);

            //Excel Prozess aktivieren
            Excel.Application applikation = new Excel.Application();
            //Dokument anlegen
           Excel.Workbook dokument = (Excel.Workbook)(applikation.Workbooks.
Add(System.Reflection.Missing.Value)); ;
            //Datenblatt im Dokument anlegen
          Excel.Worksheet datenBlatt = (Excel.Worksheet)dokument.ActiveSheet;
;
            // Überschriften eingeben
            datenBlatt.Cells[1, 1] = "Uhrzeit";    // Zelle B2
            datenBlatt.Cells[1, 2] = "Temperatur";   // Zelle C2

            double temperatur = 0;
            for (int i = 2; i < 22;i++)
            {
                //Aktuelle Uhrzeit und Datum
                DateTime datum = DateTime.Now;
                //String für die Uhrzeit zusammensetzen
                string uhrzeit = datum.Hour.ToString() + "." + datum.Minute.
ToString() + "." + datum.Second.ToString();
```

```
                //Uhrzeit ins Datenblatt schreiben
                datenBlatt.Cells[i, 1] = uhrzeit;
                System.Threading.Thread.Sleep(500);
                double halbe = 0;

                //Aufforderung an den Sensor zu messen
                iow.SendReport(1, ref senden[0], 8);

                //Der Report vom IOW wird abgefragt
                iow.GetReport(1, ref empfangen[0], 8);

                //Überprüfung, ob 0,5° dazu addiert werden müssen
                if ((empfangen[3] & 0x80) == 0x80)
                    halbe = 0.5;
                else
                    halbe = 0;

                //Überprüfung, ob positives oder negatives
                //Vorzeichen!
                if ((empfangen[2] & 0x80) != 0x80)
                {
                    temperatur = empfangen[2];
                    temperatur = temperatur + halbe;
                }
                else
                {
                    temperatur = empfangen[2] - 128;
                    temperatur = -1 * (256 - empfangen[2]);
                    temperatur = temperatur - halbe;
                }
                datenBlatt.Cells[i, 2] = "" + temperatur;
            }

            //Datenblatt benennen
            datenBlatt.Name = "Messwerte";
            //Dokument speichern
           dokument.Close(true, "messwerte.xlsx", System.Reflection.Missing.
Value);
            //Excel beenden
            applikation.Quit();

            //I²CMode deaktivieren
            senden[0] = 1;
            senden[1] = 0;
            senden[2] = 0;
            senden[3] = 0;
            senden[4] = 0;
            iow.SendReport(1, ref senden[0], 8);
            iow.Quit();
        }
    }
}
```

Das Ergebnis in der *Excel*-Datei:



	A	B	C	D	E
1	Uhrzeit	Temperatur			
2	14.35.32	18,5			
3	14.35.33	18,5			
4	14.35.33	18,5			
5	14.35.34	19			
6	14.35.34	19			
7	14.35.35	19			
8	14.35.35	19			
9	14.35.36	19			
10	14.35.36	19			
11	14.35.37	19,5			
12	14.35.37	19,5			
13	14.35.38	19,5			
14	14.35.38	19,5			
15	14.35.39	19,5			
16	14.35.39	19			
17	14.35.40	19			
18	14.35.40	19			
19	14.35.41	19			
20	14.35.42	19			
21	14.35.42	19			
22					

Abb. 7.6: Messdaten in einer *Excel*-Datei

7.4.3 Messdaten in eine *MySQL*-Datenbank schreiben

MySQL steht für nichtkommerzielle Zwecke kostenlos zur Verfügung und kann im Internet heruntergeladen werden. Gerade Internetzugriffe mittels PHP sind einfach zu

gestalten. Schreibt man Messwerte in eine solche Datenbank, können sie leicht im Internet zugänglich gemacht werden.

Der Zugriff von C# auf MySQL erfordert neben der eigentlichen Datenbankserverinstallation noch ein paar kleinere Vorbereitungen:

Zuerst muss ein Connector installiert werden, der den Zugriff von *.NET*-Sprachen ermöglicht. Man kann ihn unter *http://dev.mysql.com/downloads/* herunterladen. Das Paket wird entpackt und die Installation anschließend per Doppelklick gestartet:

Abb. 7.7: Installation des *MySQL*-Connectors

Nun muss im C#-Projekt noch der Verweis auf den Connector hinzugefügt werden.

Die entsprechende DLL befindet sich im Ordner *C:\Program Files\MySQL\MySQL Connector Net 6.2.2\Assemblies* (es sei denn, Sie haben ein anderes Installationsverzeichnis gewählt). Eingebunden werden muss die Datei *MySQL.Data.dll*.

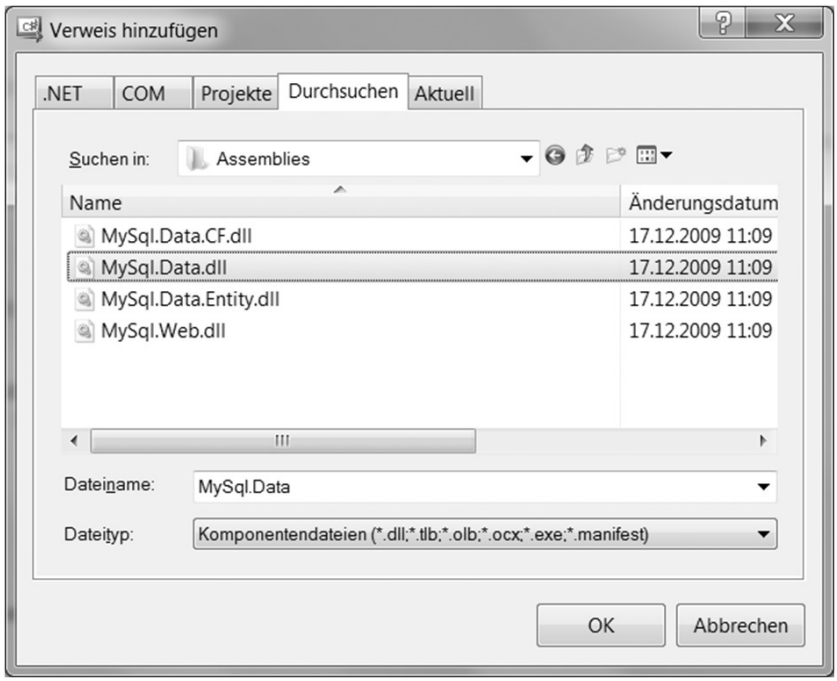


Abb. 7.8: Verweis auf den MySQL-Connector

Ist der Verweis hinzugefügt, ist die Verbindung zur Datenbank einfach zu erstellen:

Zuerst muss ein Verbindungs-String zusammengebaut werden. Dieser enthält den Servernamen (hier der localhost), den Datenbanknamen, den Usernamen und dessen Passwort:

```
string mysqlConnectionString = "SERVER=localhost;" + "DATABASE=messdaten;"
+ "UID=root;" + "PASSWORD=;";
```

Dann wird ein Objekt aus der Klasse MySQLConnection instanziiert, der Verbindungsstring wird hierbei übergeben.

```
MySqlConnection myCon = new MySqlConnection(mysqlConnectionString);
```

Die Datenbankverbindung wird geöffnet:

```
myCon.Open();
```

Ein Objekt aus der Klasse *MySQLCommand* wird erzeugt, um SQL-Statements abzusetzen:

```
MySqlCommand command = myCon.CreateCommand();
```

Das SQL-Statement wird in das Attribut *command.CommandText* geschrieben:

```
command.CommandText = „Insert into daten values (0,20.5);“;
```

Ausgeführt wird das Statement über die Methode *ExecuteNonQuery():*

```
command.ExecuteNonQuery();
```

Werden Daten in die Datenbank geschrieben, ist die Methode *ExecuteNonQuery* zu verwenden. Sollen Daten gelesen werden, steht die Methode *ExecuteReader* zur Verfügung, nähere Informationen hierzu liefern entsprechende Internetseiten.

Sind alle Daten in die Datenbank geschrieben, ist die Verbindung zu beenden:

```
myCon.Close();
```

Im nachfolgenden Projekt werden 20 Temperaturdaten in eine *MySQL*-Datenbank geschrieben[9]. Die Datenbank trägt den Namen *messdaten*, der User ist der *root*, sein Passwort ist leer und die entsprechende Tabelle heißt *daten*.

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MySql.Data.MySqlClient;
using IOWarrior;

namespace MessdatenInEineDatenbankSchreiben
{
    class Program
    {
        static void Main(string[] args)
        {
            //IOwarrior vorbereiten
            IOWKit iow = new IOWKit();
            iow.Open();
            byte[] senden = new byte[8];
            for (byte i = 0; i < 8; i++)
                senden[i] = 0;
```

9 Die Datenbank und die entsprechende Tabelle werden über die folgenden Statements erstellt: create database messwerte; use messwerte; create table daten (id int not null auto_increment,werte decimal(8,2),primary key(id));

```
            byte[] empfangen = new byte[8];
            for (byte i = 0; i < 8; i++)
                empfangen[i] = 0;
            //I²CMode aktivieren
            senden[0] = 1;
            senden[1] = 1;
            iow.SendReport(1, ref senden[0], 8);
            //Vorbereitungen zur Datenkommunikation
            senden[0] = 0x2;
            senden[1] = 0xC2;
            senden[2] = 0x90;
            senden[3] = 0x00;
            iow.SendReport(1, ref senden[0], 8);
            iow.GetReport(1, ref empfangen[0], 8);
            //Das Lesen der Temperatur wird vorbereitet
            senden[0] = 0x03;//Lesen
            senden[1] = 0x02;//Zwei Bytes empfangen
            senden[2] = 0x91;//Andresse + Read!
            iow.SendReport(1, ref senden[0], 8);
            iow.GetReport(1, ref empfangen[0], 8);
            double temperatur = 0.0;

            //Connection String zusammenbauen
                     string mysqlConnectionString = "SERVER=localhost;" +
"DATABASE=messdaten;" + "UID=root;" + "PASSWORD=;";
            //Die Datenbankverbindung vorbereiten
             MySqlConnection myCon = new MySqlConnection(mysqlConnectionStri
ng);
            //Die Datenbankverbindung wird hergestellt
            myCon.Open();
            //Ein Objekt zum Absetzen von SQL-
            //Statements wird erstellt
            MySqlCommand command = myCon.CreateCommand();
            //Das Statement wird vorbereitet
            string sqlState = "insert into daten values (0,";
            //20 Messungen werden in die Datenbank geschrieben
            for (int i = 0; i < 20; i++)
            {
                System.Threading.Thread.Sleep(500);
                double halbe = 0.0;

                //Aufforderung an den Sensor zu messen
                iow.SendReport(1, ref senden[0], 8);

                //Der Report vom IOW wird abgefragt
                iow.GetReport(1, ref empfangen[0], 8);

                //Überprüfung, ob 0,5° dazu addiert werden müssen
                if ((empfangen[3] & 0x80) == 0x80)
                    halbe = 0.5;
                else
                    halbe = 0.0;
```

```
                //Überprüfung, ob positives oder negatives
                //Vorzeichen!
                if ((empfangen[2] & 0x80) != 0x80)
                {
                    temperatur = empfangen[2];
                    temperatur = temperatur + halbe;
                }
                else
                {
                    temperatur = empfangen[2] - 128;
                    temperatur = -1 * (256 - empfangen[2]);
                    temperatur = temperatur - halbe;
                }
                string sTemp = "" + temperatur;
                //Das Komma muss in MySQL als . dargestellt werden!!
                sTemp = sTemp.Replace(",", ".");
                Console.Write(sTemp + " ");
                //Das Statement wird fertiggestellt
                command.CommandText = sqlState + sTemp + ");";
                //und an die Datenbank geschrieben
                command.ExecuteNonQuery();
            }
            //Die Datenbankverbindung wird geschlossen
            myCon.Close();

            //I²CMode deaktivieren
            senden[0] = 1;
            senden[1] = 0;
            senden[2] = 0;
            senden[3] = 0;
            senden[4] = 0;
            iow.SendReport(1, ref senden[0], 8);
            iow.Quit();
        }
    }
}
```

Ist das Programm abgelaufen, können die Daten auf dem Datenbankserver abgerufen werden:

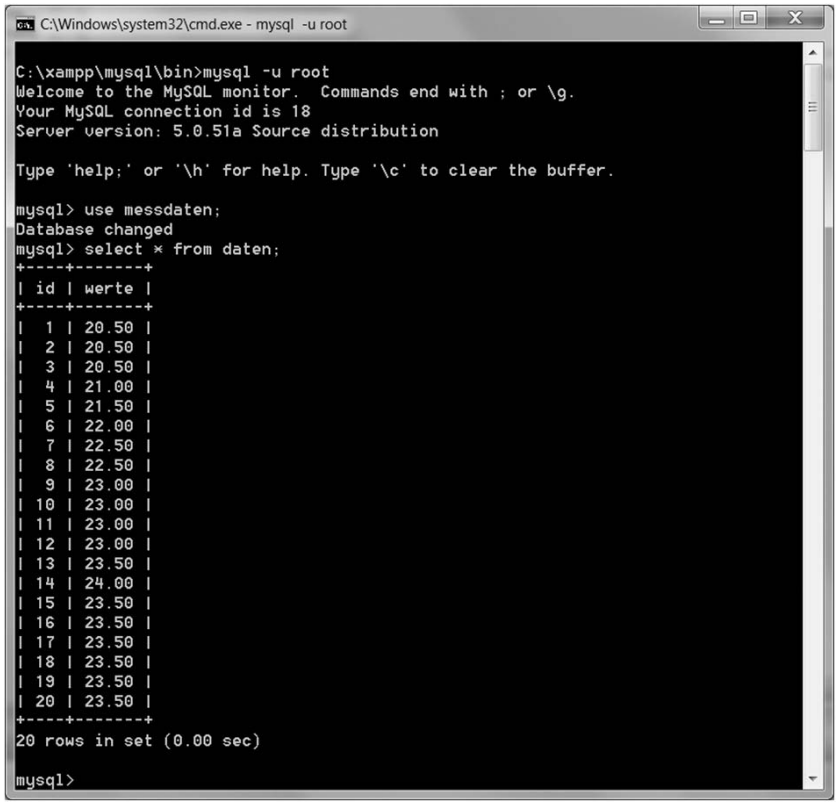


Abb. 7.9: Daten in einer *MySQL*-Datenbank

7.5 Zeiterfassung

Oftmals ist es bei Messungen notwendig, Zeitspannen aufzunehmen. Zeiten werden in C# grundsätzlich in Objekten der Klasse *DateTime* gekapselt. Über die Zeile

```
DateTime zeit1 = DateTime.Now;
```

werden im Objekt *zeit1* das aktuelle Datum und die aktuelle Uhrzeit gespeichert. Diese Informationen können einfach ausgegeben werden. Über diverse Attribute kann man Stunden, Minuten etc. ermitteln und theoretisch so auch Zeitspannen errechnen. Hierzu gibt es allerdings eine treffendere Klasse: *TimeSpan*. Diese Klasse kapselt eine Zeitspanne. Ein Objekt kann beispielsweise über die Subtraktion zweier Zeiten instanziiert werden:

```
TimeSpan spanne = zeit2.Subtract(zeit1);
```

Das Objekt *spanne* enthält nun alle Informationen der Zeitspanne von zeit1 bis zeit2. Die Informationen können direkt ausgegeben werden, über diverse Attribute können auch Werte zum Weiterverarbeiten ausgefiltert werden. Im Kapitel „SPI“ wird über die Zeitspannenberechnung die Wandlungszeit der sukzessiven Approximation mit und ohne binäre Suche berechnet.

Im nachfolgenden Projekt wird demonstriert, wie eine Zeitspanne berechnet und ausgegeben wird.

Der kommentierte Quelltext:

```
using System;

namespace ZeitspanneMessen
{
    class Program
    {
        static void Main(string[] args)
        {
            //Die aktuelle Zeit wird im Objekt
            //zeit1 gespeichert
            DateTime zeit1 = DateTime.Now;
            Console.WriteLine("Start der Zeitmessung: " + zeit1);
            //Zwei Sekunden Wartezeit
            System.Threading.Thread.Sleep(2000);
            //Die aktuelle Zeit wird im Objekt
            //zeit2 gespeichert
            DateTime zeit2 = DateTime.Now;
            //Die Zeitspanne wird mittels der zwei
            //gespeicherten Zeiten im Objekt
            //spanne gesichert und ausgegeben
            TimeSpan spanne = zeit2.Subtract(zeit1);
            Console.WriteLine("Ende der Zeitmessung: " + zeit2);
            Console.WriteLine("Zeitspanne: " + spanne);
            Console.ReadKey();
        }
    }
}
```

Das Programm hat folgende Bildschirmausgabe:

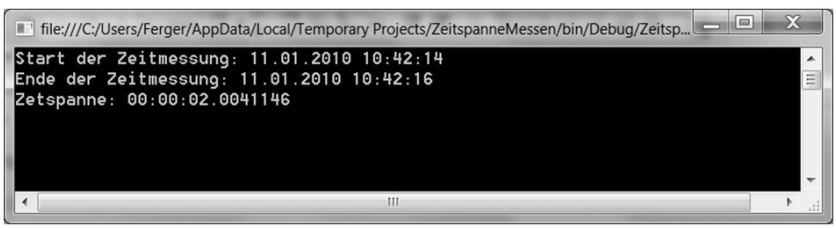


Abb. 7.10: Screenshot Zeitmessung

7.6 Abfangen von Ausnahmen/Exceptions

Syntaktische Fehler werden in der Entwicklungsumgebung abgefangen oder das Programm wird erst gar nicht gestartet. Es können aber auch Fehler während der Laufzeit auftreten. Diese Fehler lösen sogenannte Exceptions aus.

Im nachfolgenden Programm wird ein Benutzer aufgefordert, eine Zahl einzugeben. Die Zahl soll anschließend wieder ausgegeben werden:

```
static void Main(string[] args)
{
    Console.Write("Eine Zahl eingeben: ");
    string szahl = Console.ReadLine();
    int zahl = Convert.ToInt32(szahl);
    Console.Write(zahl);
    Console.ReadLine();
}
```

Die Funktion *Console.ReadLine()* liefert allerdings einen String zurück. Somit können vom Benutzer Falscheingaben gemacht werden. Ist dies der Fall, reagiert das System folgendermaßen:

Wird das Programm aus der Entwicklungsumgebung heraus aufgerufen, hält diese das Programm an und gibt eine Rückmeldung auf den Fehler:

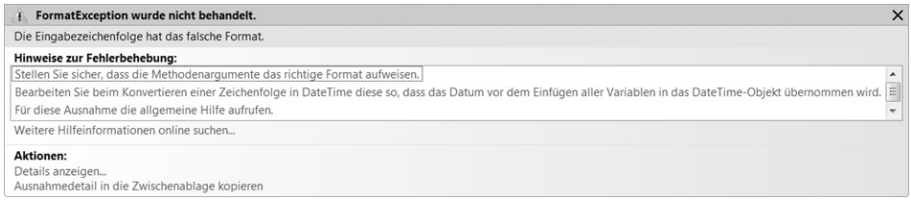


Abb. 7.11: Exception

Wird das Programm direkt per Doppelklick auf die *.exe*-Datei aufgerufen, kann dies folgende Ausgabe zur Folge haben:

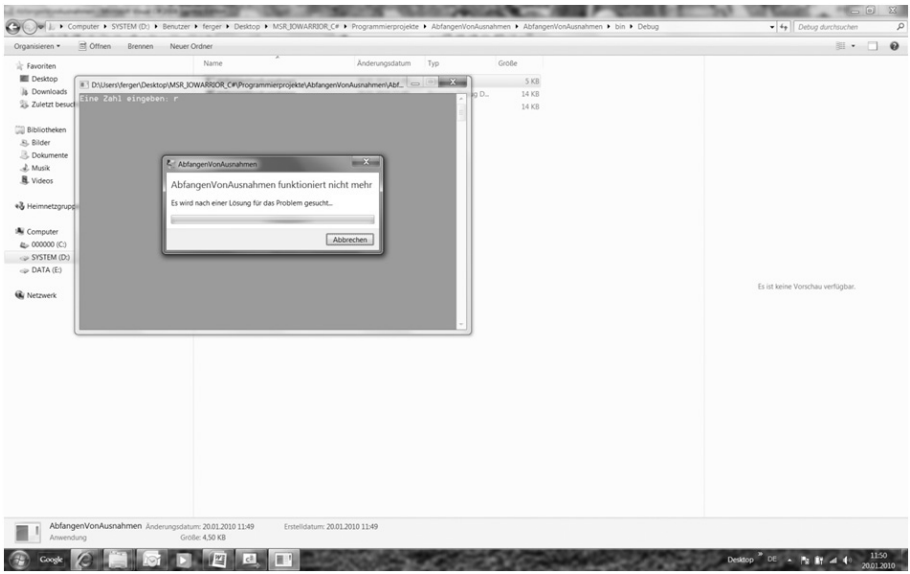


Abb. 7.12: Exception löst einen Absturz aus.

Um dies zu vermeiden, kann man das Programm anweisen, einen Quelltextblock zu „probieren". Geschieht dies erfolgreich, läuft das Programm weiter. Tritt ein Fehler (eine Exception) auf, kann im Quelltext reagiert werden. Mittels eines *try-catch*-Blocks kann dem Programm mitgeteilt werden, welcher Quelltext versucht wird und wie im Fehlerfall reagiert werden soll:

```
try
{
     zahl = Convert.ToInt32(szahl);
}
catch (FormatException fe)
{
     Console.WriteLine("Sie haben keine Zahl eingegeben.");
     Console.WriteLine("i wird auf -1 gesetzt");
     zahl = -1;
}
```

Das Programm hat nun bei einer Falscheingabe folgende Bildschirmausgabe:

Abb. 7.13: Abgefangene Exception

Es gibt sehr viele unterschiedliche Exceptions. Kennt man den genauen Namen der *Exception* nicht, kann ganz allgemein ein Objekt der Klasse Exception abgefangen werden. Über Methoden kann die genaue Art der Ausnahme herausgefunden werden (siehe unten).

7.6.1 Eigene Exceptions erstellen

Schreibt man Funktionen mit kritischem Quelltext, ist es sinnvoll, eigene Exceptions zu erzeugen, die dann von anderen Funktionen abgefangen werden. Möchte man eine eigene Exception erstellen, kann man mit dem Schlüsselworts *throw* eine solche werfen.

In der nachfolgenden Funktion soll der Anwender eine Zahl von 0 bis 255 eingeben. Grundsätzlich wird eine Falscheingabe in Form eines Buchstabens o. ä. abgefangen. Liegt der eingegebene Wert jedoch außerhalb des gewünschten Bereichs, wird eine Exception „geworfen“:

```
static int zahlenEingabe()
 {
     Console.WriteLine("Bitte einen Wert von 0 bis 255 eingeben: ");
     string szahl = Console.ReadLine();
     int wert = -1;
     try
     {
         wert = Convert.ToInt32(szahl);
     }
     catch (FormatException fe)
     {
         return 0;
     }
     if ((wert < 0) || (wert > 255))
         throw new Exception("Falscher Wert eingegeben");
     else
         return wert;
 }
```

Im Quelltext können die Informationen der „geworfenen“ Exception leicht über diverse Methoden ermittelt werden. Beispielsweise gibt die Methode *toString()* Informationen über den Fehler an sich und über die Programmzeilen und den Funktionsnamen aus, in der der Fehler ausgelöst wurde:

```
static void Main(string[] args)
        {
            try
            {
                zahlenEingabe();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
               Console.ReadLine();
        }
```

Folgende Bildschirmausgabe wäre möglich:

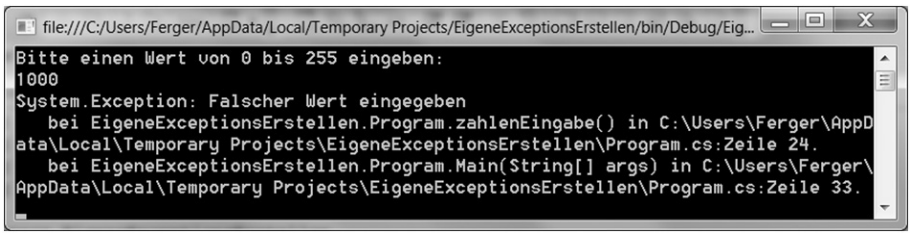


Abb. 7.14: Eigene Exception

8 Der SpinWarrior – Bewegung im Griff

In diesem Kapitel werden zum einen Chips vorgestellt, mit denen man Drehbewegungen messen kann. Zum anderen wird auf die Programmierung von Events eingegangen, mit denen man auf gemessene Ereignisse im Programm reagieren kann.

Der SpinWarrior nimmt Wegstrecken, Bewegungen und Drehzahlen auf. Er eignet sich daher sehr gut für Frontplatten, an denen mit Drehreglern Eingaben getätigt werden sollen. Aber auch andere Messungen können getätigt werden. Ohne großen schaltungstechnischen Aufwand kann dieser Chip in Schaltungen eingesetzt und an den USB-Bus angeschlossen werden.

8.1 Varianten des SpinWarriors

SpinWarrior 24R4

Der SpinWarrior 24R4 hat Anschlussmöglichkeiten für vier inkrementelle Drehgeber und sieben Schalter. Er kann geeignete Drehgeber mit niedrigem Stromverbrauch direkt mit Spannung versorgen. Die Signalfrequenz der Drehgeber kann bis zu 3,5 kHz betragen. Die Bewegungsdaten der Achsen werden relativ mit 8 Bit Auflösung gemessen. Der USB-Anschluss ist vorbereitet.

SpinWarrior 24R6

Der SpinWarrior 24R6 hat Anschlussmöglichkeiten für sechs inkrementelle Drehgeber und sieben Schalter. Er kann geeignete Drehgeber mit niedrigem Stromverbrauch direkt mit Spannung versorgen. Die Signalfrequenz der Drehgeber kann bis zu 2,5 kHz betragen. Die Bewegungsdaten der Achsen werden relativ mit 8 Bit Auflösung gemessen. Der USB-Anschluss ist vorbereitet.

SpinWarrior 24A3

Der SpinWarrior 24A3 hat Anschlussmöglichkeiten für drei inkrementelle Drehgeber und sechs Schalter. Er kann geeignete Drehgeber mit niedrigem Stromverbrauch direkt mit Spannung versorgen. Die Signalfrequenz der Drehgeber kann bei einem Geber bis zu 5 kHz, bei zwei Gebern bis zu 4,4 kHz und bei drei angeschlossenen Gebern bis zu 3,9 kHz betragen. Die Bewegungsdaten der Achsen werden absolut mit einer 16-Bit-Auflösung gemessen. Der USB-Anschluss ist vorbereitet.

Alle drei Bausteine sind in DIL24- und in SOIC24-Gehäusen verfügbar. Die Pinbelegungen der Bausteine sind im Anhang ebenso zu finden wie die entsprechenden Schaltpläne. Ist man im Besitz eines IOW24-Starterkits, kann man durch einfaches Wechseln der ICs die aufgebaute Grundplatine verwenden.

Auf der Webseite des Herstellers kann man das SDK inklusive der DLL zum Ansteuern des SpinWarriors herunterladen. Auch finden sich hier Beispielapplikationen sowie Datenblätter und Dokumentationen.

8.2 Die DLL zum Ansteuern des SpinWarriors

Die DLL *spinkit.dll* stellt alle notwendigen Funktionen zur Verfügung, die nachfolgend dargestellt werden:

```
SPINKIT_HANDLE SPINKIT_API SpinKitOpenDevice(void);
```

Mit dieser Funktion wird die Verbindung zu allen SpinWarriors hergestellt. Eine HandleID auf den ersten Baustein wird zurückgegeben. Hat man nur einen SpinWarrior am USB-Bus angeschlossen, reicht diese Funktion zum Verbindungsaufbau vollkommen aus.

```
void SPINKIT_API SpinKitCloseDevice(SPINKIT_HANDLE spinHandle);
```

Die Funktion schließt alle angeschlossenen Warriors. Gerade unter C# ist der Aufruf dieser Funktion am Ende der Software notwendig, da sonst das Programm nicht sauber beendet werden kann.

```
BOOL SPINKIT_API SpinKitRead(SPINKIT_HANDLE spinHandle, PSPINKIT_DATA
SpinData);
```

Die Funktion liest die Daten des angesteuerten Warriors nur dann ein, wenn eine Signaländerung am Baustein anliegt. Ansonsten blockiert diese Funktion das laufende Programm. Es bietet sich hier an, diese Funktion in einem separaten Thread aufzurufen. Da die Funktion blockiert, kann man den Thread gut dazu verwenden, Events zu erzeugen. Diese Events können dann später, beispielsweise im Programm mit grafischer Benutzeroberfläche, grafische Komponenten verändern.

Wichtig:

Der C-Struct *PSINKIT_DATA* ist als Mischung von Integerzahlen und booleschen Werten definiert. Dieser Struct kann in C# nicht so einfach erzeugt und übergeben werden. Es reicht allerdings, der Funktion in C# eine Referenz auf ein ausreichend großes Integer-Array zu übergeben.

```
BOOL SPINKIT_API SpinKitReadNonBlocking(SPINKIT_HANDLE spinHandle,
PSPINKIT_DATA SpinData);
```

Die Funktion liest die Daten des angesteuerten Warriors immer ein, ohne das laufende Programm zu blockieren.

```
ULONG SPINKIT_API SpinKitGetNumDevs(void);
```

Die Funktion liefert die Anzahl der angeschlossenen Spinwarriors.

```
SPINKIT_HANDLE SPINKIT_API SpinKitGetDeviceHandle(ULONG numDevice);
```

Über diese Funktion kann die HandleID eines Bausteins gewonnen werden. Hat man beispielsweise zwei SpinWarriors am USB-Bus angeschlossen, bekommt man die erste HandleID über die Funktion *SpinKitOpenDevice*. Die ID des zweiten Bausteins bekommt man zurückgegeben, wenn man der Funktion *SpinKitGetDeviceHandle* den Wert 2 übergibt.

```
ULONG SPINKIT_API SpinKitGetProductId(SPINKIT_HANDLE spinHandle);
```

Die Funktion liefert die Produktnummer des SpinWarriors zurück. Die HandleID muss übergeben werden. Ist ein SpinWarrior24R4 angeschlossen, wird der Wert *1200h* zurückgegeben. Der SpinWarrior 24R6 hat die Produktnummer 1201h und der SpinWarrior 24A3 kann über den Wert *1202h* identifiziert werden.

```
ULONG SPINKIT_API SpinKitGetRevision(SPINKIT_HANDLE spinHandle);
```

Die Funktion liefert die Revisionsnummer des Spinwarriors zurück. Die HandleID muss übergeben werden.

```
BOOL SPINKIT_API SpinKitGetSerialNumber(SPINKIT_HANDLE spinHandle, PWCHAR
serialNumber);
```

Die Funktion liefert die Seriennummer des Spinwarriors zurück. Die HandleID muss übergeben werden. Die Seriennummer wird in ein Array von Characters per Referenz geschrieben. In C# kann statt des Char-Arrays eine Referenz auf das erste Element eines Short-Arrays übergeben werden.

```
BOOL SPINKIT_API SpinKitSetTimeout(SPINKIT_HANDLE spinHandle, ULONG timeout);
```

Die Funktion setzt die Time-out-Zeit des Bausteins. Wird diese Zeit beim Auslesen überschritten, bricht der Baustein den Lesevorgang ab.

```
PCHAR SPINKIT_API SpinKitVersion(void);
```

Die Funktion liefert die Versionsnummer der API zurück.

Ein einfaches Beispielprojekt

Nachdem alle Funktionen der DLL eingebunden wurden, kann man folgendermaßen auf den SpinWarrior zugreifen:

Der SpinWarrior wird geöffnet, ein Handle auf ihn wird zurückgegeben:

```
int handle = SpinKitOpenDevice();
```

Ob mehr als ein SpinWarrior angeschlossen wurde, kann über die Funktion *SpinKitGetNumDevs()* abgefragt werden. Wir gehen davon aus, dass nur ein Warrior angeschlossen ist.

Ein Integerarray mit 14 Elementen wird erzeugt, um die aktuellen Daten des SpinWarriors zu speichern:

```
Int32[] daten = new Int32[14];
```

Die aktuellen Daten des Warriors werden ausgelesen und im Array gespeichert. Daten liegen nur an, wenn an den Eingängen des Warriors Änderungen vorliegen. Die Funktion blockiert also:

```
SpinKitRead(handle, ref daten[0]);
```

Der Warrior muss nach Abschluss aller Messungen geschlossen werden:

```
SpinKitCloseDevice(handle);
```

Im nachfolgenden Einstiegsprojekt werden die Daten des Spinwarriors eingelesen und auf der Konsole ausgegeben. Man beachte, dass die Daten der Drehregler relativ ausgegeben werden (bei einer Rechtsdrehung erscheint eine 1, bei einer Linkdrehung eine –1).

Der kommentierte Quelltext:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;

namespace SpinWarriorEinstiegsprojekt
{
    class Program
    {
        [DllImport("spinkit", SetLastError=true)]
```

```
            public static extern int SpinKitOpenDevice();

        [DllImport("spinkit", SetLastError = true)]
        public static extern void SpinKitCloseDevice(int spinHandle);

        [DllImport("spinkit", SetLastError = true)]
        public static extern int SpinKitGetNumDevs();

        [DllImport("spinkit", SetLastError = true)]
       public static extern bool SpinKitRead(int spinHandle, ref Int32  spi);

        [DllImport("spinkit", SetLastError = true)]
         public static extern int SpinKitReadNonBlocking(int spinHandle, int
numPipe, ref Int32 buffer, int length);

        [DllImport("spinkit", SetLastError = true)]
               public static extern int SpinKitGetDeviceHandle(int numDe-
vice);

        static void Main(string[] args)
        {
            //Vorbereitung des Datenarrays
            Int32[] daten = new Int32[14];
            //Der JoyWarrior wird geöffnet
            int handle = SpinKitOpenDevice();
            Console.WriteLine(handle);
            Console.WriteLine(SpinKitGetNumDevs());
            //200-mal wird der Warrior eingelesen
            for (int z = 0; z < 200; z++)
            {
                SpinKitRead(handle, ref daten[0]);
                   for (int i = 0; i < 14; i++ )
                        Console.Write(" " + daten[i]);
                    Console.WriteLine();
            }
            //Der Warrior wird geschlossen
            SpinKitCloseDevice(handle);
            Console.ReadKey();
        }
    }
}
```

Eine mögliche Bildschirmausgabe:

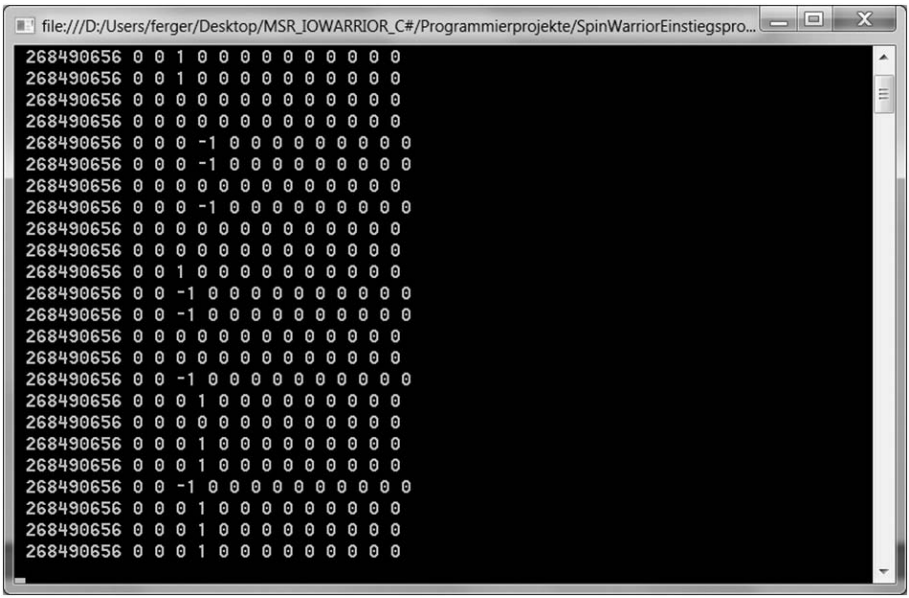


Abb. 8.1: Screenshot Spinwarrior

8.3 Kapselung des Spinwarriors in einer C#-Klasse

Nachfolgend wird eine Klasse programmiert, mit der man leichten Zugriff auf den SpinWarrior hat und die das Einbinden der API-DLL schon integriert. Die Klasse ist für einen SpinWarrior 24R4 geschrieben. Möchte man mehrere Warriors oder einen anderen Baustein aus der *Spin*-Serie verwenden, ist ein Umschreiben der Klasse leicht möglich. Die Projektdatei und die Quelltexte liegen auf der beiliegenden CD vor.

Die Klasse soll folgendermaßen konzipiert werden:

Beim Instanziieren eines Objekts soll die Verbindung zum Baustein hergestellt werden. Ein Thread, in dem ständig die Daten am Baustein eingelesen werden, wird gestartet. *Get*-Methoden stellen die Zugriffsmöglichkeit auf die Daten am Warrior dar. Eine *Close*-Methode stoppt den Mess-Thread und schließt die Verbindung zum Warrior. Zusätzlich soll ein Event erzeugt werden, sofern geänderte Daten am SpinWarrior anliegen. Mehr ist hier nicht notwendig.

Das Klassendiagramm der Klasse SpinWarrior:

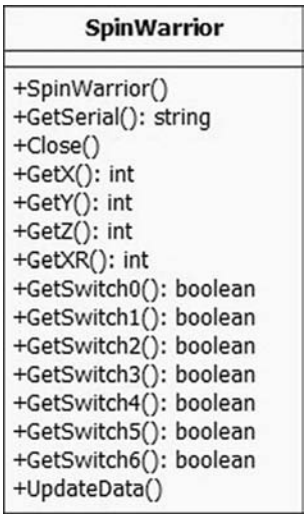


Abb. 8.2:Klassendiagramm

Erstellung der *.NET*-**DLL**

Zuerst erstellt man eine Klasse *Spinlib.cs*, die den C#-Zugriff auf die Funktionen der spinkit.dll herstellt. Die DLL liegt während der Projekterstellung im Ordner */bin/Debug*. Der Klasse können dann noch weitere Methoden hinzugefügt werden, die die Programmierarbeit später erleichtert, jedoch ist dies nicht unbedingt notwendig.

Der Quellcode der Klasse *Spinlib.cs*:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
namespace Spin
{
    public class Spinlib
    {
        [DllImport(“spinkit”, SetLastError = true)]
        public static extern int SpinKitOpenDevice();

        [DllImport(“spinkit”, SetLastError = true)]
        public static extern void SpinKitCloseDevice(int spinHandle);

        [DllImport(“spinkit”, SetLastError = true)]
        public static extern int SpinKitGetNumDevs();
```

```
        [DllImport("spinkit", SetLastError = true)]
       public static extern bool SpinKitRead(int spinHandle, ref Int32 data);

        [DllImport("spinkit", SetLastError = true)]
        public static extern bool SpinKitReadNonBlocking(int spinHandle, ref
Int32 data);

        [DllImport("spinkit", SetLastError = true)]
        public static extern int SpinKitGetDeviceHandle(int numDevice);

        [DllImport("spinkit", SetLastError = true)]
        public static extern int SpinKitProductId(int spinHandle);

        [DllImport("spinkit", SetLastError = true)]
        public static extern int SpinKitGetRevision(int spinHandle);

        [DllImport("spinkit", SetLastError = true)]
        public static extern bool SpinKitGetSerialNumber(int spinHandle, ref
short serial);

        [DllImport("spinkit", SetLastError = true)]
       public static extern bool SpinKitSetTimeout(int spinHandle, int time-
out);

        [DllImport("spinkit", SetLastError = true)]
        public static extern string SpinKitVersion();


        public static int Open()
        {
            return SpinKitOpenDevice();
        }

        public static void Close(int handle)
        {
            SpinKitCloseDevice(handle);
        }

        public static void GetData(int handle,ref int [] datenArray)
        {
            SpinKitRead(handle, ref datenArray[0]);
        }

       public static void GetDataNonBlocking(int handle, ref int[] datenArray)
        {
            SpinKitReadNonBlocking(handle, ref datenArray[0]);
        }

    }
}
```

Um ein Event-Handling zu ermöglichen, benötigt man eine Klasse, die von EventArgs erbt. Eine solche Klasse kapselt die Informationen, die vom SpinWarrior später verwendet werden sollen. Dies sind also vier Integerwerte für die Drehgeber und sieben boolesche Werte für die Schalter. Die Klasse bekommt den Namen *SpinEventArgs*. Die einzelnen Attribute werden mit dem Schlüsselwort *public* versehen, sodass ein leichter Zugriff möglich ist.

Der Quellcode der Klasse *SpinEventArgs.cs*:

```
using System;

namespace Spin
{
    /// <summary>
    /// Klasse zum Erzeugen eines Events
    /// </summary>
    public class SpinEventArgs: EventArgs
    {
        public int X,Y,Z,XR;
        public bool sw0,sw1,sw2,sw3,sw4,sw5,sw6;

          public SpinEventArgs(int X, int Y, int Z, int XR, bool sw0, bool
sw1,bool sw2,bool sw3,bool sw4,bool sw5,bool sw6)
        {
            this.X = X;
            this.Y = Y;
            this.Z = Z;
            this.XR = XR;
            this.sw0 = sw0;
            this.sw1 = sw1;
            this.sw2 = sw2;
            this.sw3 = sw3;
            this.sw4 = sw4;
            this.sw5 = sw5;
            this.sw6 = sw6;
        }
     }
}
```

Die eigentliche Hauptklasse *SpinWarrior.cs* soll hier nicht komplett dargestellt werden, da viele Methoden (SET, GET) trivial sind. Wichtig sind die Attribute, der Konstruktor sowie einige Vorbereitungen zur Erzeugung von Threads und Events und die Methode *close()*.

Die Attribute:

```
private string serial = “unknown”;
        private bool connect = false;


        private int spinX;//Wert des ersten Drehreglers
```

```
        private int spinY; //Wert des zweiten Drehreglers
        private int spinZ; //Wert des dritten Drehreglers
        private int spinXr; //Wert des vierten Drehreglers

        private bool switch0; //Wert des ersten Schalters
        private bool switch1; //Wert des zweiten Schalters
        private bool switch2; //Wert des dritten Schalters
        private bool switch3; //Wert des vierten Schalters
        private bool switch4; //Wert des fünften Schalters
        private bool switch5; //Wert des sechsten Schalters
        private bool switch6; //Wert des siebten Schalters

        private int handle = 0; //Die HandleID des SpinWarriors
        private int[] data;   //Das Datenarray zum Speichern aller Werte
        private Thread dataUpdate; //Der Thread zum Einlesen der Daten
        private bool updateRunning = false; //boolscher Wert um den Thread
                                                              //sauber zu been-
den

//Dem Delegat SpinEventHandler wird die Funktion OnSpinEvent zugeordnet. //
Die Funktion muss selbst nicht geschrieben werden und kann später //beispiels-
weise in einer Software mit grafischer Benutzeroberfläche //verwendet werden.
         public delegate void SpinEventHandler(object source, SpinEventArgs
args);
        public event SpinEventHandler OnSpinEvent;
```

Der Konstruktor:

```
public SpinWarrior()
{
    handle = Spinlib.Open();//Der IOWarrior wird angesteuert
    if (handle !=0)
    connect = true;//Abfragemöglichkeit wird geschaffen
                        //Wenn die Verbindung fehlgeschlagen ist, hat
 //connect den Wert false
    short[] sn = new short[8];//Array für die Seriennummer
    if (Spinlib.SpinKitGetSerialNumber(handle, ref sn[0]))
    {
          serial = string.Empty;
          foreach (char c in sn)//Die Seriennummer wird in den String
             serial += c;         //serial geschrieben
    }
    data =   new int[14];//Ein IntegerArray mit 14 Werten ist notwendig
       //um die Daten aus dem SpinWarrior zu speichern
    updateRunning = true;//Diese Variable wird vom Thread abgefragt. Ist
 //der Wert true, werden Daten aus dem
 //SpinWarrior abgeholt. Im anderen Fall kann der
 //Thread sauber beendet werden.
    dataUpdate = new Thread(new ThreadStart(UpdateData)); //Der Thread,
//der ständig Daten vom SpinWarrior einliest,
//wird instanziiert. Die Methode, die im
//eigenständigen Thread gestartet wird, heißt
```

```
//UpdateData

    dataUpdate.Start(); //Der Thread wird gestartet
}
```

Die vom Thread aufgerufene Methode zur Aktualisierung der vom SpinWarrior gelesenen Daten:

```
   private void UpdateData()
        {
            while (updateRunning)//Über die Variable updateRunning vom
   //Datentyp bool kann die Schleife beendet
   //werden.
            {
                if (!connect)
                    return;
                else
                {
                    Spinlib.SpinKitRead(handle, ref data[0]);
                                        //Daten werden nur dann vom SpinWar-
rior
//eingelesen, wenn eine Datenänderung anliegt. //Ansonsten blockiert die Me-
thode.
                    spinX += data[1];//Da die Drehgeber inkrementell
                    spinY += data[2];// arbeiten, müssen die Eingangsdaten
                    spinZ += data[3];//aufsummiert werden!!!
                    spinXr += data[4];
                    //Console.WriteLine(data.Length);
                    if (data[13] == 1)
                        switch6 = true;
                    else
                        switch6 = false;
                    if (data[12] == 1)
                        switch5 = true;
                    else
                        switch5 = false;
                    if (data[11] == 1)
                        switch4 = true;
                    else
                        switch4 = false;
                    if (data[10] == 1)
                        switch3 = true;
                    else
                        switch3 = false;
                    if (data[9] == 1)
                        switch2 = true;
                    else
                        switch2 = false;
                    if (data[8] == 1)
                        switch1 = true;
                    else
                        switch1 = false;
                    if (data[7] == 1)
```

```
                    switch0 = true;
                else
                    switch0 = false;
                SpinEventArgs args = new SpinEventArgs(spinX, spinY, spinZ,
 spinXr, switch0, switch1, switch2, switch3, switch4, switch5, switch6);
            //Ein Event wird vorbereitet und die Methode
//OnSpinEvent kann außerhalb der Klasse //verwendet werden.
                    OnSpinEvent(this, args);
                }
            }
        }

public void Close()
        {
            if (handle != 0)
            {
                updateRunning = false;
                Thread.Sleep(50);
                dataUpdate.Abort();
                //Console.WriteLine("Thread aborted");
                Spinlib.SpinKitCloseDevice(handle);
                //Spinlib.Close(handle);
                //Console.WriteLine("Spin Warrior abgemeldet");

            }
        }
```

8.4 Erstellen einer Konsolenapplikation mit dem SpinWarrior

Man erstellt ein neues Projekt. Um die obige DLL zu verwenden, kopiert man sie und die spinkit.dll in den Ordner */bin/Debug* des entsprechenden Projekts. Sollten die Ordner nicht vorhanden sein, können sie über den Menüpunkt *Alle speichern* erzeugt werden.

Im Projektmappen-Explorer fügt man den Verweis auf die SpinWarrior.net.dll hinzu.

Bei den Using-Direktiven muss *using Spin*; hinzugefügt werden.

Im nachfolgenden Programm wird der Zustand am SpinWarrior ständig auf dem Bildschirm ausgegeben. Im Programm wird ein Objekt aus der Klasse *SpinWarrior* instanziiert und in einer Endlosschleife werden die entsprechenden Attribute des Objekts auf der Konsole ausgegeben. Wird der Switch0 am SpinWarrior gedrückt, bricht die Schleife ab und das Programm wird sauber beendet.

Der Quelltext des Konsolenprogramms:

```
class Program
{
    static void Main(string[] args)
    {
        SpinWarrior spi = new SpinWarrior();
        while (true)
        {
            Console.Clear();
            Console.WriteLine(spi.GetSerial());
            Console.WriteLine(spi.GetSwitch0());
            Console.WriteLine(spi.GetSwitch1());
            Console.WriteLine(spi.GetSwitch2());
            Console.WriteLine(spi.GetSwitch3());
            Console.WriteLine(spi.GetSwitch4());
            Console.WriteLine(spi.GetSwitch5());
            Console.WriteLine(spi.GetSwitch6());
            Console.WriteLine(spi.GetX());
            Console.WriteLine(spi.GetY());
            Console.WriteLine(spi.GetZ());
            Console.WriteLine(spi.GetXR());
            if (spi.GetSwitch0())
                break;
        }
        spi.Close();
        Console.WriteLine("Ende");
    }
}
```

Das Programm hat beispielsweise folgende Bildschirmausgabe:

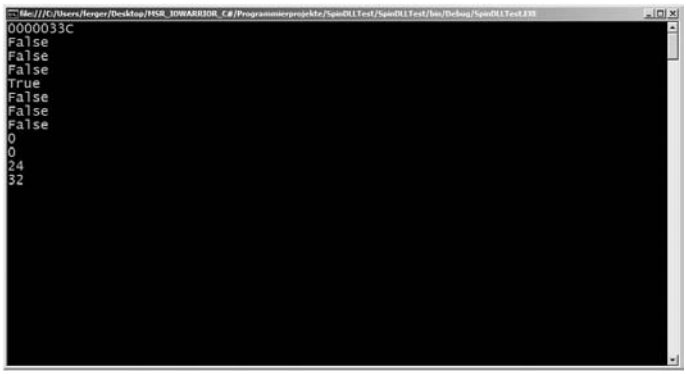


Abb. 8.3: Screenshot Spinwarrior

Im Konsolenprogramm wurde noch nicht mit Events gearbeitet, was schleunigst nachgeholt werden muss – allerdings in einer Software mit grafischer Oberfläche!

8.5 Erstellen einer Software mit grafischer Oberfläche unter Verwendung von Events

Zuerst erstellt man ein neues Projekt mit grafischer Oberfläche (Windows-Form-Anwendung) und kopiert die beiden DLLs wieder in das *Debug*-Verzeichnis. Nachdem

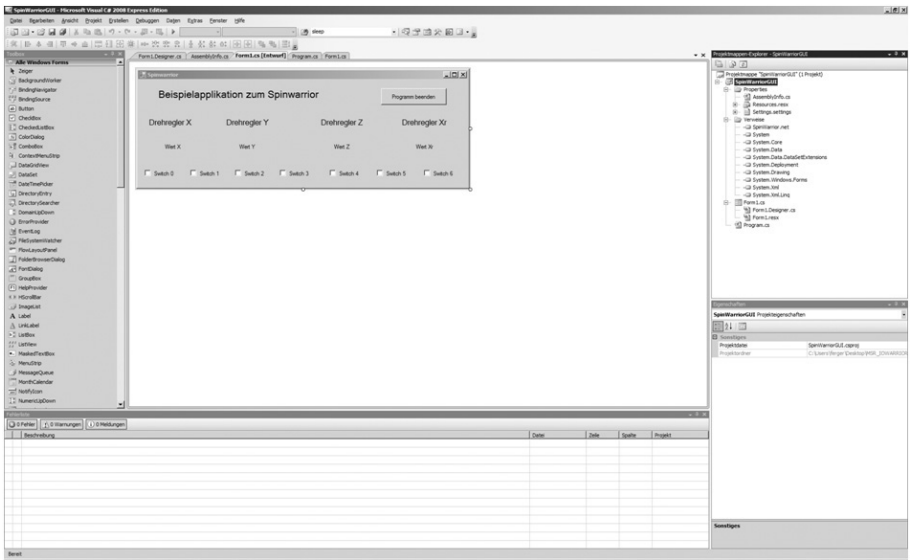


Abb. 8.4: Erstellung der GUI

der Verweis auf die .Net.DLL im *Objekteigenschaften*-Explorer hinzugefügt wurde, kann die grafische Oberfläche gestaltet werden:

Im Konstruktor wird ein Objekt der Klasse *SpinWarrior* instanziiert. Vergessen Sie nicht, im Bereich der *Using*-Direktiven *using Spin*; hinzuzufügen.

```
public Form1()
{
    InitializeComponent();
    spi = new SpinWarrior();
}
```

Wichtig: Durch das Erzeugen eines Objekts aus der Klasse *SpinWarrior* wird ein Thread gestartet!

Der Button beendet das Programm. Durch einen Doppelklick auf den Button kommt man zum Quelltext der richtigen Methode:

```
private void btExit_Click(object sender, EventArgs e)
 {
    spi.Close();
    Application.Exit();
    Environment.Exit(0);
 }
```

In dieser Methode wird der Messthread über den Aufruf von *spi.Close()* beendet und die Verbindung zum SpinWarrior unterbrochen. *Application.Exit()* beendet die grafische Oberfläche und *Enviroment.Exit(0)* räumt alle noch laufenden Threads aus dem Weg.

- In der Methode *Form1_Load* wird der Event-Handler hinzugefügt. Ein Event vom SpinWarrior löst dann die Methode *MyOnSpinEvent* aus:

```
private void Form1_Load(object sender, EventArgs e)
{
   spi.OnSpinEvent += new SpinWarrior.SpinEventHandler(MyOnSpinEvent);
}
```

Die Methode *MyOnSpinEvent* bekommt die Informationen des SpinWarriors über ein Objekt aus der Klasse *SpinEventArgs* (siehe oben). Je nach Zustand der Drehregler und Switches wird eine entsprechende Methode aufgerufen, um das jeweilige Label oder die entsprechende Textbox abzuändern.

Wichtig:

Es wurde für jedes Steuerelement, das von den Daten des SpinWarriors beeinflusst wird, eine eigene Methode geschrieben. Greift man direkt auf die Steuerelemente zu, wird eine Exception ausgelöst.

Der Quelltext der Methode MyOnSpinEvent:

```
public void MyOnSpinEvent(object o, SpinEventArgs args)
  {
      if (args.sw0)
          this.SetcbSwitch0(true);
      else
          this.SetcbSwitch0(false);
      if (args.sw1)
          this.SetcbSwitch1(true);
      else
          this.SetcbSwitch1(false);
      if (args.sw2)
          this.SetcbSwitch2(true);
      else
          this.SetcbSwitch2(false);
      if (args.sw3)
```

```
            this.SetcbSwitch3(true);
        else
            this.SetcbSwitch3(false);
        if (args.sw4)
            this.SetcbSwitch4(true);
        else
            this.SetcbSwitch4(false);
        if (args.sw5)
            this.SetcbSwitch5(true);
        else
            this.SetcbSwitch5(false);
        if (args.sw6)
            this.SetcbSwitch6(true);
        else
            this.SetcbSwitch6(false);
        this.SetlblX(args.X);
        this.SetlblY(args.Y);
        this.SetlblZ(args.Z);
        this.SetlblXr(args.XR);
    }
```

Wie oben beschrieben, ist es nicht ratsam, direkt aus der Methode *MyOnSpinEvent* auf die Steuerelemente zuzugreifen. Dies liegt daran, dass die Methode von einem anderen Thread aufgerufen wurde und nicht von dem Thread, der die Steuerelemente erzeugt hat. Dies könnte zu Problemen während der Laufzeit führen. Anhand der Methode *SetcbSwitch0* soll dargestellt werden, wie man sauber auf die Steuerelemente zugreifen kann:

```
    public void SetcbSwitch0(bool value)
      {
        if (this.cbSwitch0.InvokeRequired)
        {
           SetCbSwitch0Callback d = new SetCbSwitch0Callback(SetcbSwitch0);
            this.Invoke(d, new object[] { value });
        }
        else
        {
            if (value)
                this.cbSwitch0.Checked = true;
            else
                this.cbSwitch0.Checked = false;
        }
     }
```

In der Methode wird zuerst geprüft, ob ein erneuter Aufruf der Methode notwendig ist (this.cbSwitch0.InvokeRequired). Ist dies der Fall, wird die Methode über den Quelltext im ersten *if*-Teil erneut aufgerufen. Hiermit garantiert die Laufzeitumgebung, dass nicht gleichzeitig von zwei verschiedenen Threads auf ein Steuerelement zugegriffen wird.

Darf der Thread auf das Steuerelement zugreifen, tritt der erste *else*-Fall ein und das Häkchen der Checkbox wird je nach Wert des entsprechenden Switches vom SpinWarrior gesetzt oder nicht gesetzt.

Das Prinzip dieser Methode ist auf alle anderen Steuerelemente übertragbar.

Die letzte wichtige Methode ist *Dispose* aus der Klasse *Form1*. Hier muss ebenfalls die Verbindung zum SpinWarrior unterbrochen und die Threads müssen beendet werden (siehe Button), da das Programm schließlich auch über das x des Fensters beendet werden kann.

```
protected override void Dispose(bool disposing)
 {
     if (disposing && (components != null))
     {
         components.Dispose();
     }
     base.Dispose(disposing);
     spi.Close();
     Application.Exit();
     Environment.Exit(0);

 }
```

Das Programm hat beispielsweise folgendes Fenster zur Folge:

Abb. 8.5: GUI mit dem SpinWarrior

Sie finden den kompletten Quelltext inklusive der entsprechenden Projektdatei auf der beiliegenden CD.

9 Der JoyWarrior

Der JoyWarrior ist ein Chip, der beim Betriebssystem als Gamecontroller angemeldet wird. Es ist daher nicht notwendig, über eine DLL auf Funktionen zuzugreifen und diese auszuwerten.

Der JoyWarrior kann als Gamecontroller und damit zum Steuern, jedoch auch zum Messen verwendet werden. Die Chips sind allesamt sehr preiswert.

9.1 Varianten des JoyWarriors

Folgende Varianten stehen in 8-Bit-Auflösung zur Verfügung[10]:

JoyWarrior20 A8-8

- Vier analoge Achsen (Potenziometer oder Spannungseingang) mit jeweils 8 Bit
- Bis zu 8 Tasten, direkt an den Chip angeschlossen
- Gehäusevarianten: PDIP 20 oder SOIC 20

JoyWarrior20 A8-16

- Vier analoge Achsen (Potenziometer oder Spannungseingang) mit jeweils 8 Bit
- Bis zu 16 Tasten in einer 4x4-Matrix angeschlossen
- Gehäusevarianten: PDIP 20 oder SOIC 20

JoyWarrior24 A8-8

- Drei Potenziometerachsen mit 8 Bit Auflösung
- Bis zu 8 Tasten, direkt an den Chip angeschlossen
- Selbstkalibrierend und selbstzentrierend (abschaltbar)
- Minimale externe Beschaltung: 1 Widerstand, 2 Kondensatoren
- Gehäusevarianten: PDIP 24 oder SOIC 24, Modul verfügbar

JoyWarrior24 A8-16

- Drei Potenziometerachsen mit 8 Bit Auflösung
- Bis zu 16 Tasten in einer 4x4-Matrix angeschlossen
- Selbstkalibrierend und selbstzentrierend (abschaltbar)
- Minimale externe Beschaltung: 1 Widerstand, 2 Kondensatoren
- Gehäusevarianten: PDIP 24 oder SOIC 24, Modul verfügbar

10 Quelle: http://www.codemercs.com/index.php?id=80&L=0

JoyWarrior24 A8-Modul

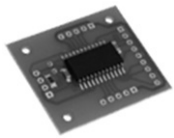


Abb. 9.1: Joywarrior

Basierend auf JW24A8-8 und JW24A8-16, sind einsatzbereite Module verfügbar. An sie müssen nur noch die Potenziometer und Tasten sowie ein USB-Kabel angeschlossen werden, um einen Gamecontroller mit 3 Achsen und bis zu 8 oder 16 Tasten zu haben.

Die Module sind unter den Bezeichnungen JW24MOD-A8-8 und JW24MOD-A8-16 verfügbar.

Folgende Varianten stehen in 10-Bit-Auflösung zur Verfügung:

JoyWarrior20 A10-8

- Drei analoge Achsen (Potenziometer oder Spannungseingang) mit jeweils 10 Bit
- Bis zu 8 Tasten, direkt an den Chip angeschlossen
- Gehäusevarianten: PDIP 20 oder SOIC 20

JoyWarrior20 A10-16

- Drei analoge Achsen (Potenziometer oder Spannungseingang) mit jeweils 10 Bit
- Bis zu 16 Tasten in einer 4x4 Matrix angeschlossen
- Gehäusevarianten: PDIP 20 oder SOIC 20

Interessant ist auch die Variante *JoyWarrior24 F8.* Bei dieser Variante ist auf dem Modul schon ein 3-Achsen-Beschleunigungssensor integriert. Weiter können acht Taster eingelesen werden. Das Modul ist auch als Quakecatcher Kit erhältlich. Das Quakecatcher Network ist ein Projekt der Universität von Kalifornien zur verteilten Messung seismischer Aktivitäten. Die Software kann unter *http://qcn.stanford.edu/* heruntergeladen werden. Das Kit besteht aus dem JoyWarrior20 A10-16-Modul, einem USB-Anschlusskabel und einem kleinen Gehäuse. Es ist sehr schnell aufgebaut und dann sofort einsatzbereit.

Abb. 9.2: Zusammenbau des Quakecatchers

Abb. 9.3: Der fertige Quakecatcher

Die Pinbelegungen aller Bausteine und Module befinden sich im Anhang. In den nachfolgenden Beispielen wird der JoyWarrior mit drei 100-kOhm-Potenziometern und acht Schaltern folgendermaßen beschaltet:

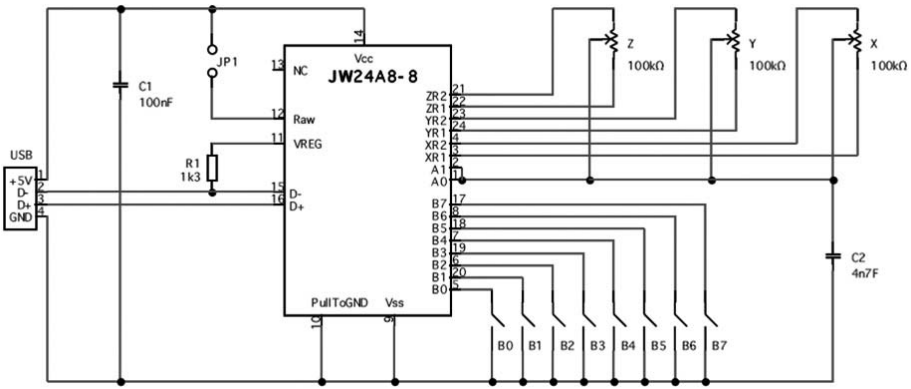


Abb. 9.4: Schaltplan JoyWarrior

Ist der JoyWarrior angeschlossen, wird er in der Systemsteuerung als Joystick erkannt

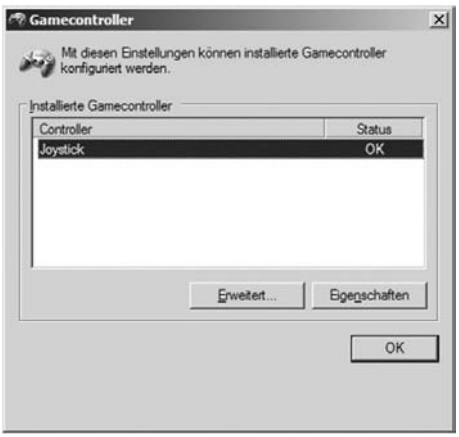


Abb. 9.5: JoyWarrior in der Systemsteuerung

Die einzelnen Werte können schon auf der Eigenschaftsseite beobachtet werden:

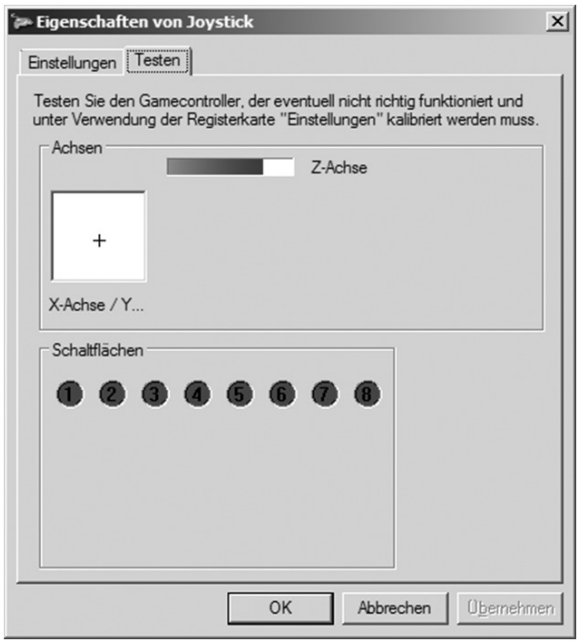


Abb. 9.6: Eigenschaften des JoyWarriors

9.2 JoyWarrior, C# und DirectX

Mit DirectX liefert Microsoft eine Schnittstelle speziell zur Spieleprogrammierung. Hier finden sich unter anderem auch Bibliotheken, um einen Gamecontroller auszulesen. Genau diese Technik findet nachfolgend Anwendung.

Zuerst wird dargestellt, wie ein JoyWarrior A8-8 mittels DirectX in einer Konsolenapplikation ausgelesen wird. Anschließend wird eine Klassenbibliothek erstellt, um die Verwendung des JoyWarriors zu vereinfachen. In einem Programm mit grafischer Oberfläche wird dies letztendlich verwendet.

9.2.1 Installation von DirectX

DirectX ist frei erhältlich. Man kann das Paket unter *http://msdn.microsoft.com/de-de/directx* herunterladen. Beachten Sie, dass Sie das SDK (Software Development Kit) herunterladen, denn nur hiermit kann programmiert werden. Das Softwarepaket ist sehr umfangreich (>500 MB).

9.2.2 Projekterstellung einer Konsolenapplikation

Legen Sie ein neues Projekt an und wählen Sie eine Konsolenapplikation. Im Projektexplorer muss nun ein Verweis auf DirectX hinzugefügt werden (Installation des SDK vorausgesetzt).

Wählen Sie im Fenster unter dem Reiter *.NET* den Punkt *Microsoft.DirectX.DirectInput* aus und bestätigen Sie mit dem Button *OK*.

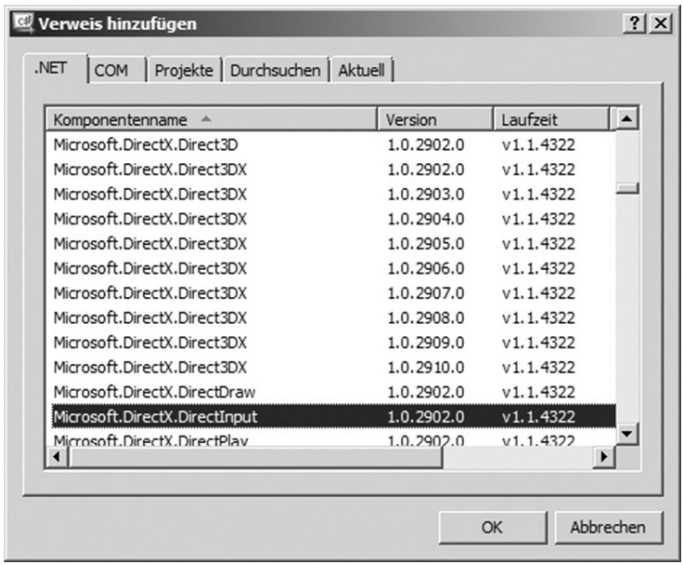


Abb. 9.7: Einbindung von DirectX

Im Bereich der Using-Direktiven muss nun die Verwendung des Paketes angemeldet werden:

```
using Microsoft.DirectX.DirectInput;
```

Nun kann mit der eigentlichen Programmierung begonnen werden.

Um den JoyWarrior anzusprechen und auszulesen, sind folgende Schritte notwendig:

Zuerst wird eine Liste aller Gamecontroller vom System abgerufen:

```
DeviceList gameControllerList = Manager.GetDevices(DeviceClass.GameControl,
EnumDevicesFlags.AttachedOnly);
```

Ist nur ein Gamecontroller angeschlossen, kann folgendermaßen auf ihn zugegriffen werden:

Über

```
gameControllerList.MoveNext();
```

wird ein Zeiger auf den ersten Gamecontroller der Liste gesetzt.

Der Gamecontroller wird instanziiert:

```
Device joystickDevice = new Device (((DeviceInstance)gameControllerList.
Current).InstanceGuid );
```

Dem Gamecontroller wird mitgeteilt, dass er als Joystick zu behandeln ist:

```
joystickDevice.SetDataFormat(DeviceDataFormat.Joystick);
```

Der JoyWarrior wird verwendet:

```
joystickDevice.Acquire();
```

Der aktuelle Status des JoyWarriors wird ermittelt:

```
joystickDevice.Poll();
```

Der Status des Joysticks wird in einem Objekt der Klasse *JoystickState* gekapselt:

```
JoystickState state = joystickDevice.CurrentJoystickState;
```

In dem Objekt aus der Klasse *JoystickState* sind die Neigungswerte der drei Achsen in den Attributen X, Y und Z gespeichert und können beispielsweise auf der Konsole ausgegeben werden:

```
Console.WriteLine(state.X + „   „ + state.Y + „   „ + state.Z + „   „);
```

Im folgenden Programm werden 1.000-mal die Daten des JoyWarriors abgefragt und auf der Konsole ausgegeben.

Der kommentierte Quellcode der Main-Methode:

```
static void Main(string[] args)
{
     Device joystickDevice;
     JoystickState state;
     try
     {
      // Ausgehend, dass nur ein JoyWarrior und auch sonst kein anderer
      //Gamecontroller angeschlossen ist!
```

```
      //Liefert eine Liste der angeschlossenen Gamecontroller
     DeviceList gameControllerList = Manager.GetDevices(DeviceClass.GameCon-
trol, EnumDevicesFlags.AttachedOnly);
               Console.WriteLine(gameControllerList.Count + " Joystick ange-
schlossen");

      //Wenn mindestens ein Gamecontroller angeschlossen ist:
      if (gameControllerList.Count > 0)
      {
      //Zeiger auf das erste (und einzige!) Element in der Liste setzen
        gameControllerList.MoveNext();

        //Joywarrior initialisieren
                   joystickDevice = new Device(((DeviceInstance)gameControl-
lerList.Current).InstanceGuid);
        Console.WriteLine(joystickDevice.ToString());

        //DirectX bekommt mitgeteilt, dass der Gamecontroller ein Joystick
        //ist

        joystickDevice.SetDataFormat(DeviceDataFormat.Joystick);
        //DirectX bekommt mitgeteilt, dass der Joystick verwendet wird
        joystickDevice.Acquire();

        byte[] buttons; //Array für die Daten der Buttons
        for (int z = 0; z < 1000; z++)
        {
          Console.Clear();
          //Die Joystickinformationen werden aktualisiert
          joystickDevice.Poll();
          state = joystickDevice.CurrentJoystickState;
           Console.WriteLine(state.X + "    " + state.Y + "    " + state.Z + "
" );
          buttons = state.GetButtons();
          Console.WriteLine(buttons.Length + " Buttons vorhanden");
          for (int i = 0; i < 8; i++)
               Console.Write(buttons[i]+"\t");
          System.Threading.Thread.Sleep(200);
        }
      }
    }
   catch (Exception err)
   {
     Console.WriteLine(err.Message);
     Console.WriteLine(err.StackTrace);
   }
  Console.ReadKey();
}
```

Das Programm erzeugt die folgende Bildschirmausgabe:

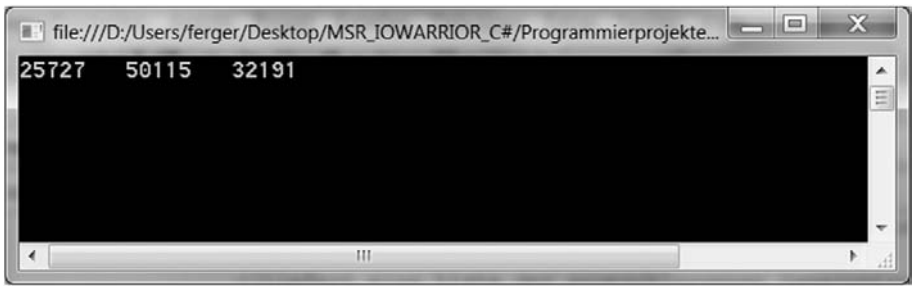


Abb. 9.8: Screenshot der Konsolenapplikation

9.3 Eine .NET-Klasse des JoyWarriors erstellen

Ziel ist es, eine Klassenbibliothek zu entwickeln, die in einem eigenen Thread ständig die Daten aus dem JoyWarrior ausliest. Über ein Eventhandling wird Zugriff auf die aktualisierten Daten des JoyWarriors ermöglicht.

Der Quellcode der Bibliothek ist auf der CD enthalten. Die Klassenbibliothek stellt folgende Funktionen zur Verfügung:

Funktion/Klassen	Beschreibung
JoyWarrior()	Konstruktor, der einen JoyWarrior anspricht und die Messung der aktuellen Werte in einem eigenen Thread startet. Ändern sich Daten, wird ein Event mit Übergabe eines Objekts aus der Klasse *JoyWarriorEventArgs* geworfen.
Close()	Die Kommunikation zum JoyWarrior wird beendet.
JoyWarriorEventArgs	Das Objekt kapselt in den Attributen *x, y, z* sowie *b0* bis *b7* die Neigungsdaten oder den Zustand der Buttons.

Nachfolgend wird die Klassenbibliothek in ein Konsolenprojekt eingebunden. Hierzu wird die DLL wieder in den *Debug*-Ordner des Projekts kopiert, ein Verweis im Projektexplorer wird erstellt und über die Using-Anweisung in den Quellcode integriert.

Der kommentierte Quellcode des Konsolenprojekts:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using Joy;

namespace JoyWarriorDLLTest
{
    class Program
    {
        static void Main(string[] args)
        {
            //Objekt der Klasse JoyWarrior instanziieren,
            //damit wird ein zusätzlicher Thread gestartet,
            //der den JoyWarrior ständig abfragt
            JoyWarrior joy = new JoyWarrior();
            //EventHandler hinzufügen
            //bei Eintreffen eines Events wird die Methode
            //MyOnJoyEvent aufgerufen
              joy.OnJoyWarriorEvent += new JoyWarrior.JoyWarriorEventHandler
(MyOnJoyEvent);
        }

        static void MyOnJoyEvent(Object o, JoyWarriorEventArgs args)
        {
            //Ausgabe der Daten des JoyWarriors auf die
            //Konsole
            Console.WriteLine(args.toString());
            //Wenn der Button B0 gedrückt wird
            if (args.B0)
            {
                //Alle Threads werden beendet
                //Damit endet das Programm
                Environment.Exit(0);
            }
        }
    }
}
```

Das Programm hat folgende Ausgabe:

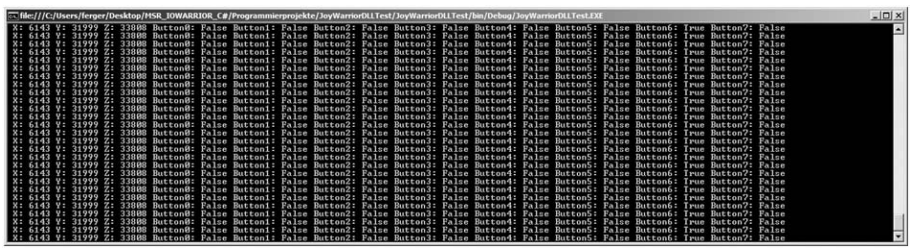


Abb. 9.9: Screenshot der Konsolenapplikation

10 Anhang

10.1 Ergänzungen und Ausblicke

Die Produktpalette der Warriors ist weitaus größer. Nicht alle Bausteine können hier behandelt werden, verdienen es aber, erwähnt zu werden:

Mit den Mousewarriors ist es möglich, Eingabegeräte zu kreieren, die wahlweise an den Ports USB, PS2, aber auch an den seriellen Eingang oder einen ADB-Anschluss angeschlossen werden können.

„KeyWarrior packt das gesamte Protokoll- und Elektronik-Know-how, das für den Bau einer Tastatur mit oder ohne Mausfunktion notwendig ist, in eine Familie von Chips. Egal, ob eine große oder kleine Zahl Tasten oder programmierbare Makros notwendig ist, KeyWarrior bedient die Anforderungen am USB, PS/2 und ADB für die Maus sogar seriell.[11]“

Es lohnt sich, die Webseite von Code Mercenaries öfter zu besuchen.

10.2 Pinbelegungen

Die Pinbelegung und Beschreibung des IO-Warriors 56 zeigt das folgende Bild.

11 aus http://www.codemercs.com/index.php?id=37

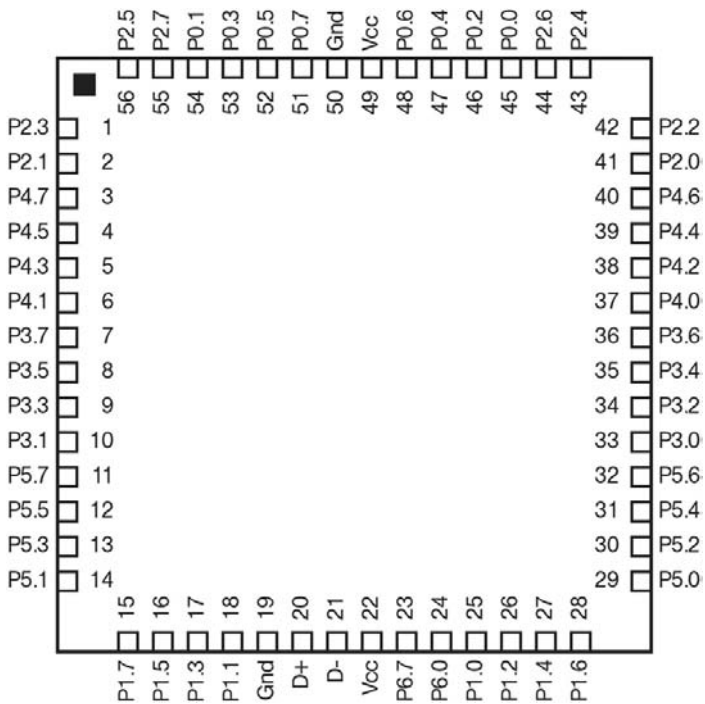


Abb. 10.1: Pinbelegung des IO-Warriors 56 MLFP

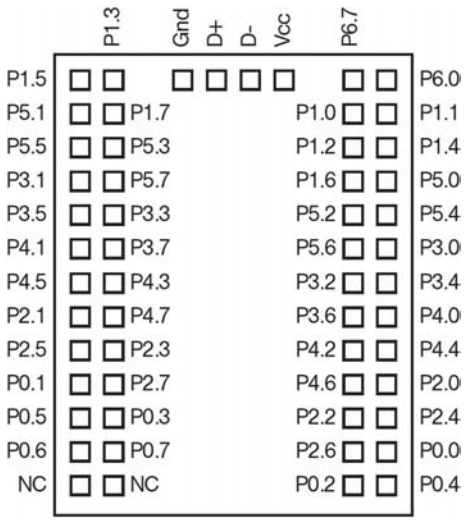


Abb. 10.2: Pinbelegung des IOW56-Moduls

Pi # MLFP56	Pin# Module	Type	Name	Special function
1	22	I/O	P2.3	X3
2	19	I/O	P2.1	X2
3	20	I/O	P4.7	
4	17	I/O	P4.5	LCD-CS1 (not driven by special mode function)
5	18	I/O	P4.3	LCD-E-/RE
6	15	I/O	P4.1	LCD-RS
7	16	I/O	P3.7	LCD-Data7
8	13	I/O	P3.5	LCD-Data5
9	14	I/O	P3.3	LCD-Data3
10	11	I/O	P3.1	LCD-Data1
11	12	I/O	P5.7	
12	9	I/O	P5.5	
13	10	I/O	P5.3	SPI-/DRDY
14	7	I/O	P5.1	SPI-/SS
15	8	I/O	P1.7	IIC-SCL
16	5	I/O	P1.5	IIC-SDA
17	6	I/O	P1.3	LED-/OE
18	53	I/O	P1.1	LED-Clk, undefined state during start up
19	4	power	Gnd	
20	3	USB	D+	
21	2	USB	D-	
22	1	power	Vcc	
23	56	I/O	P6.7	
24	55	I/O	P6.0	Power select during start up
25	54	I/O	P1.0	LED-Data, undefined state during start up
26	52	I/O	P1.2	LED-Strobe
27	51	I/O	P1.4	
28	50	I/O	P1.6	
29	49	I/O	P5.0	SPI-SCK
30	48	I/O	P5.2	SPI-MOSI
31	47	I/O	P5.4	SPI-MISO
32	46	I/O	P5.6	
33	45	I/O	P3.0	LCD-Data0
34	44	I/O	P3.2	LCD-Data2
35	43	I/O	P3.4	LCD-Data4
36	42	I/O	P3.6	LCD-Data6
37	41	I/O	P4.0	LCD-/On
38	40	I/O	P4.2	LCD-R/W-/WE
39	39	I/O	P4.4	LCD-E2-/RES
40	38	I/O	P4.6	LCD-CS2 (not driven by special mode function)
41	37	I/O	P2.0	X0
42	36	I/O	P2.2	X2
43	35	I/O	P2.4	X4
44	34	I/O	P2.6	X6
45	33	I/O	P0.0	Y0
46	32	I/O	P0.2	Y2
47	31	I/O	P0.4	Y4
48	27	I/O	P0.6	Y6
49	-	power	Vcc	
50	-	power	Gnd	
51	28	I/O	P0.7	Y7
52	25	I/O	P0.5	Y5
53	26	I/O	P0.3	Y3
54	23	I/O	P0.1	Y1
55	24	I/O	P2.7	X7
56	21	I/O	P2.5	X5

Abb. 10.3: Beschreibung der Pinfunktionalitäten des IOW56

10.3 Die Adressen der Spalten und Zeilen[12] von LCD-Displays

1 LINE X 8 CHARACTERS PER LINE

2 LINES X 8 CHARACTERS PER LINE

Character	1	2	3	4	5	6	7	8
Line 1	80	81	82	83	84	85	86	87
Line 2	C0	C1	C2	C3	C4	C5	C6	C7

1 LINE X 16 CHARACTERS PER LINE

Character	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Line 1	80	81	82	83	84	85	86	87	C0	C1	C2	C3	C4	C5	C6	C7

1 LINE X 16 CHARACTERS PER LINE

2 LINES X 16 CHARACTERS PER LINE

Character	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Line 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
Line 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF

2 LINES X 16 CHARACTERS PER LINE

4 LINES X 16 CHARACTERS PER LINE

Character	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Line 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
Line 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
Line 3	90	91	92	93	94	95	96	97	98	99	8A	9B	9C	9D	9E	9F
Line 4	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF

12 Tabelle von http://www.ulrichradig.de

1 Line X 20 Characters per Line

2 Lines X 20 Characters per Line

Character	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Line 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	90	81	92	93
Line 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3

1 Line X 24 Characters per Line

2 Lines X 24 Characters per Line

Character	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Line 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	90	91	92	93	94	95	96	97
Line 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3	D4	D5	D6	D7

4 Lines X 20 Characters per Line

Character	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Line 1	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	90	81	92	93
Line 2	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3
Line 3	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	90	81	92	93
Line 4	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3

10.4 Tabelle von I²C-Adressen[13]:

	LSB							
MSB	000x	001x	010x	011x	100x	101x	110x	111x
0010	SAA5240	SAA5240	SAA5240	SAF1134	SAA5252	SAA9020	SAA9020	SAA9020
	SAA4700	SAA5241	SAB9070	SAF1135	SAA9020			
	SAF1134	SAA5243	SAF1134	SAF1135				
	SAF1135	SAA5244	SAA5244					
		SAA5246						
		SAA9041						
		SAA4700						
		SAF1134						
		SAA1135						
0011	SAA7250	SAA7250			SAA1136	PCF1810	PCF1810	PCF1810
	PCB5020	PCB5020			PCF1810		SAA1770	SAA1770
	PCB5021	PCB5021						
	PCB5032	PCB5032						
0100	SAA1137	SAA1137	PCA1070	PCA1070	PCD3311	PCD3311	SAB3028	PCD5002
	PCD4430	PCD4430			PCD3312	PCD3312		
	SAA7194	SAA7194	PCF8574	PCF8574	PCF8574	PCF8574	PCF8574	PCF8574
	PCF8574	PCF8574	TDA8444	TDA8444	TDA8444	TDA8444	TDA8444	TDA8444
	TDA8444	TDA8444						
0101								
0110								
0111	PCF8576	PCF8576	PCF8577	PCF8577A	PCF8578	PCF8566	PCF8566	
	SAA1064	SAA1064	SAA1064	SAA1064	PCF8579	PCF8574A	PCF8574A	
	PCF8574A	PCF8574A	PCF8574A	PCF8574A	PCF8574A			
1000	TEA6320	TDA8424	TDA8425					
	TEA6330	TDA8425	TDA8415					
	TDA8420	TDA8426	TDA8416					
	TDA8421	TDA8420	TDA8440	TDA6360				
	TDA8960	TDA8421	TDA8940	TDA8480				
	NE5751	TDA9860	TDA8417					
		NE5751	TDA6360					
			TDA8480					

13 Tabelle aus: http://www.elektronik-magazin.de/page/I²C-bus-adressliste-23

	LSB							
MSB	000x	001x	010x	011x	100x	101x	110x	111x
1001	TDA8440	TDA8440	TDA8440	TDA8440	TDA8440	TDA8440	TDA8440	TDA8440
	TDA8540	TDA8540	TDA8540	TDA8540	TDA8540	TDA8540	TDA8540	TDA8540
	PCF8591	PCF8591	PCF8591	PCF8591	PCF8591	PCF8591	PCF8591	PCF8591
1010	PCF8570	PCF8570	PCF8570	PCF8570	PCF8570	PCF8570	PCF8570	PCF8570
	PCF8571	PCF8571	PCF8571	PCF8571	PCF8571	PCF8571	PCF8571	PCF8571
	PCF8580	PCF8580	PCF8580	PCF8580	PCF8580	PCF8580	PCF8580	PCF8580
	PCF8581	PCF8581	PCF8581	PCF8581	PCF8581	PCF8581	PCF8581	PCF8581
	PCF8582	PCF8582	PCF8582	PCF8582	PCF8582	PCF8582	PCF8582	PCF8582
	PCF8583	PCF8583						
1011	SAA7199	SAA7199	TDA8416		TDA2518	PCA8510	SAA7186	SAA9065
	PCF8570	SAA7152	PCF8570	PCF8570	SAA7186	PCA8516	PCF8570	PCF8570
		PCF8570			PCF8570	PCF8570		
1100	TSA5510	TSA5510	TSA5510	TSA5510	TSA5510	TSA5510	TSA5510	TSA5510
	TSA5511	TSA5511	TSA5511	TSA5511	TSA5511	TSA5511	TSA5511	TSA5511
	TSA5512	TSA5512	TSA5512	TSA5512	TSA5512	TSA5512	TSA5512	TSA5512
	TSA5514	TSA5514	TSA5514	TSA5514	TSA5514	TSA5514	TSA5514	TSA5514
	TSA5519	TSA5519	TSA5519	TSA5519	TSA5519	TSA5519	TSA5519	TSA5519
	SAB3035	SAB3035	SAB3035	SAB3035	SAB3035	SAB3035	SAB3035	SAB3035
	SAB3036	SAB3036	SAB3036	SAB3036	SAB3036	SAB3036	SAB3036	SAB3036
	SAB3037	SAB3037	SAB3037	SAB3037	SAB3037	SAB3037	SAB3037	SAB3037
	UMA1009	UMA1009	UMA1009	UMA1009	UMA1009	UMA1009	UMA1009	UMA1009
	UMA1010	UMA1010	UMA1010	UMA1010	UMA1010	UMA1010	UMA1010	UMA1010
		TEA6000	TSA6057	TSA6057				
		TEA6100	PCA8516	PCA8516				
			TSA6060	TSA6060				
			TSA6061	TSA6061				

10.5 Die Pinbelegung und Beschreibung des SpinWarriors 24

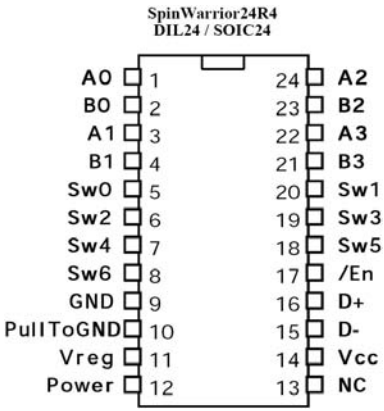


Abb. 10.4: Pinbelegung des SP24

Zur Erklärung der Pins:

Name	I/O	Type	Pins	Description
D+, D-	I/O	special	16, 15	USB differential data lines
A0, B0	I	Input, internal pullup	1, 2	Quadrature signals for X axis encoder
A1, B1	I	Input, internal pullup	3, 4	Quadrature signals for Y axis encoder
A2, B2	I	Input, internal pullup	24, 23	Quadrature signals for Z axis encoder
A3, B3	I	Input, internal pullup	22, 21	Quadrature signals for Rx axis encoder
Sw0, Sw1, Sw2, Sw3, Sw4, Sw5, Sw6	I	Input, internal pullup	5, 20, 6, 19, 7, 18, 8	Switch inputs, contacts must close to ground
/En	O	OpenDrain, internal pullup	22, 21	Enable signal for encoders. Encoders can draw power when this signal is low
Power	I	Input internal pull down	12	Used to set high or low power mode
PullToGND	I		10	Used during manufacturing, connect to GND
GND		Power supply	9	Ground
Vcc		Power supply	14	Supply voltage
Vreg	O	Regulated 3V out	11	Power for D- pullup resistor
NC	-		13	do not connect

Abb. 10.5: Pinbeschreibung

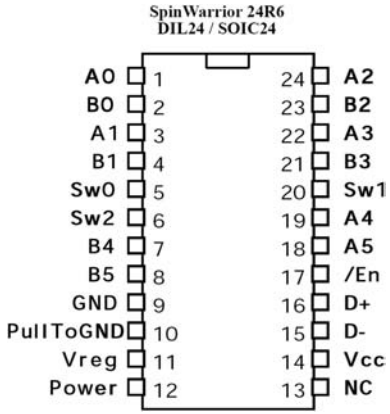


Abb. 10.6: Pinbelegung des SP24R6

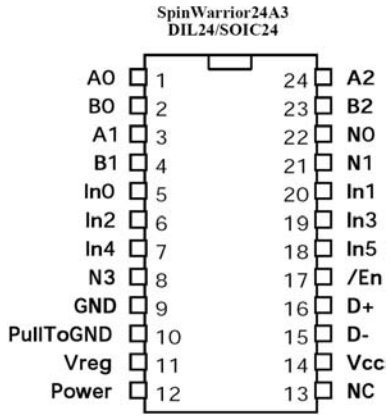


Abb. 10.7: Pinbelegung des SP24A3

Schaltplan:

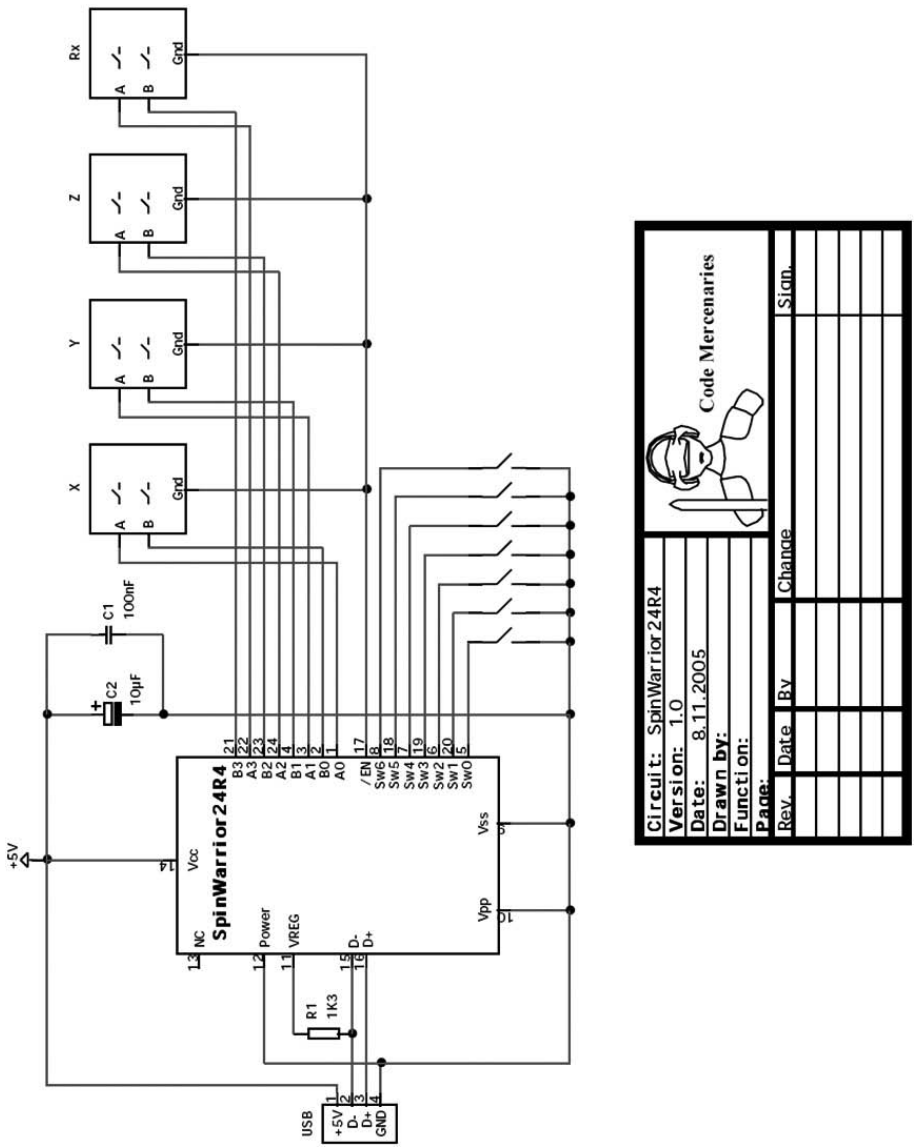


Abb. 10.8: Schaltplan zum SP24R4 mit mechanischen Drehgebern

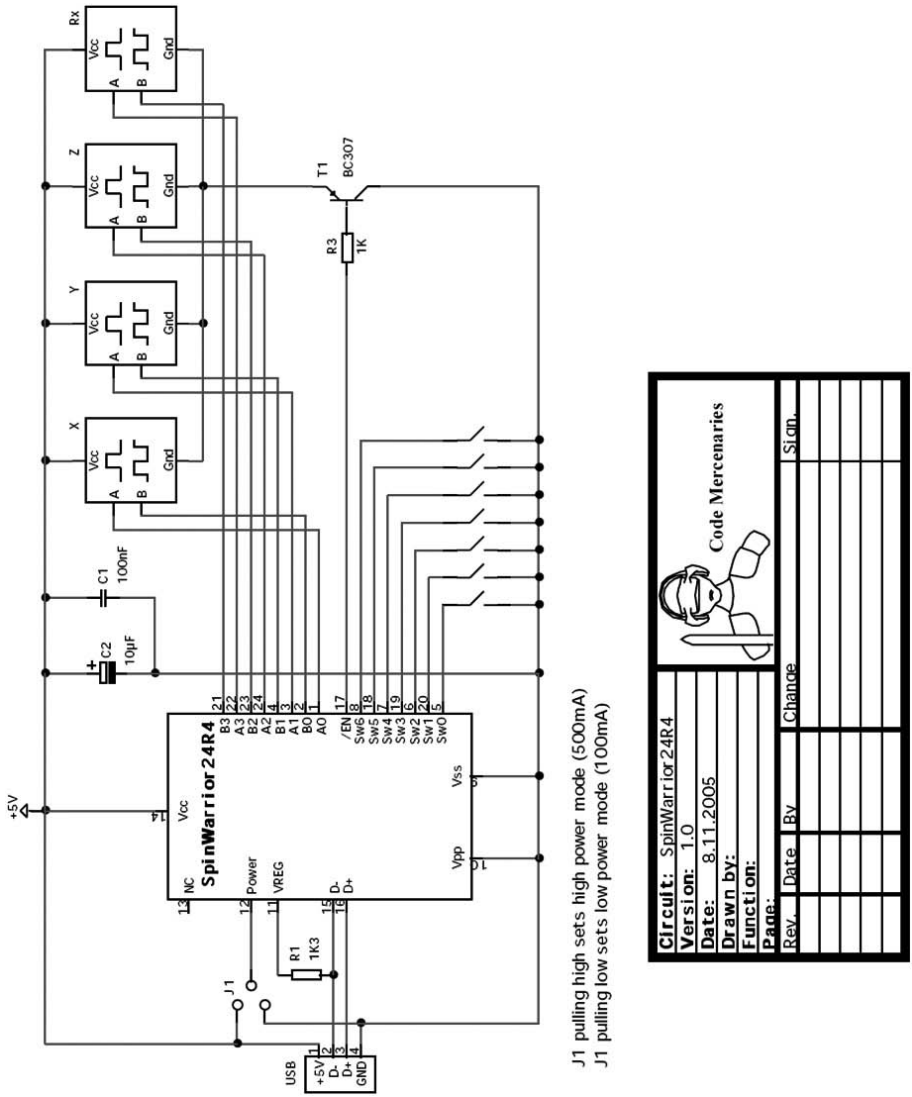


Abb. 10.9: Schaltplan zum SP24R4 mit optischen Drehgebern

10.6 Quellenangabe

Internetseite der Firma Code Mercanaries:

http://www.codemercs.com

Internetseite der Firma JenColor:

http://www.MAZeT.de

Visual C#, das umfassende Handbuch als Openbook:

http://openbook.galileocomputing.de/visual_csharp/

Elektronikbauteile:

http://www.reichelt.de

www.hbe-berlin.de

Webseite von Ralf Greinert mit einem All In One Tool für die Io-Warrior:

http://www.greinert-dud.de

Webseite mit einer C#-Library für die IOWs:

http://www.wayoda.org

Unser Dank gilt den Firmen Code Mercenaries und JenColor sowie Eberhard Fahle, Christoph Schnedl und Ulrich Radig für die technische Unterstützung.

Sachverzeichnis

FRANZIS
PC & ELEKTRONIK

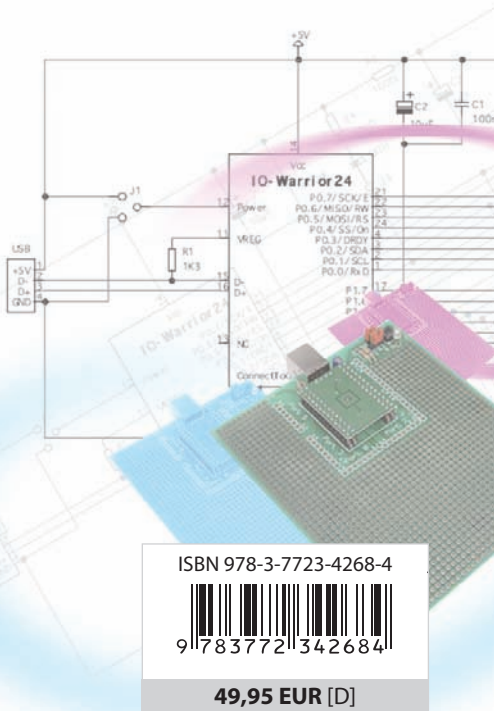
Jochen Ferger

Messen, Steuern und Regeln mit USB und C#

C# als Programmiersprache bietet viele Vorteile im Bereich Messen, Steuern, Regeln mit der USB-Schnittstelle: syntaktisch einfacher und übersichtlicher Quelltext, hohe Performance, einfache Hardware-Unterstützung, sehr gute Dokumentationen und eine hervorragende Entwicklungsumgebung, die es in Form der Express Edition sogar kostenlos im Internet zum Herunterladen gibt. Hinzu kommt, dass in C# erstellte Bibliotheken auch in anderen .NET-Sprachen zur Verfügung stehen.

Dieses Buch vermittelt die Grundlagen der Programmierung in C#. Anschließend erfolgt der Zugriff auf USB-Hardware mittels entsprechender DLLs (Dynamic Link Libraries). Auf einfache IO-Projekte folgt die Beschreibung der Spezialfunktionen wie die Ansteuerung von LCD-Displays, das Auslesen von Fernbedienungscodes sowie die Anwendung und Programmierung auf dem I^2C- und dem SPI-Bus. Im Kapitel zur erweiterten Programmierung werden Netzwerkprojekte realisiert sowie die Speicherung und Aufbereitung von Messdaten dargestellt. Abschließend werden mit dem Spin-Warrior und dem Joy-Warrior weitere C#-Projekte realisiert und eine Übersicht auf weitere Bausteine der Firma Code Mercenaries gegeben.

Aus dem Inhalt:



ISBN 978-3-7723-4268-4

49,95 EUR [D]