Automatische Autofernsteuerung mittels 802.11b und Bildverarbeitung auf einem PC

Diplomarbeit WS03

R. Guldener T. Balczarczyk

Hochschule Rapperswil, 10. Dezember 2003

Inhaltsverzeichnis

1 Abstract **1**

2 Bestandteile der Bildverarbeitung **2**

2.1 Programmieren mit 2-dimensionalen Feldern 2
2.2 Beibehaltung der Bildgrössen 2
2.3 Programmieren mit Bitmaps 3
2.4 Die Bildverarbeitungsklasse *CImageProcessing* 3
2.4.1 Bilddaten in Vektor-und Matrixform 3
2.4.2 Abtasten 4
2.4.3 Kantendetektion 4
2.4.4 Normalisierung 5
2.4.5 Polynomapproximation 5
2.4.6 Radsteuerung 5
2.4.7 Erkennen von Hindernissen und Verzweigungen 6
2.4.8 Bild speichern 7
2.4.9 Bild analysieren 7

3 Bestandteile des Hauptprogramms **8**

3.1 Grafische Ein- und Ausgabe mit Device Context 8
3.2 Die Klasse *CMy80211CarDlg* 8
3.2.1 Initialisierung der Variablen 8
3.2.2 Starten der Bildverarbeitung 8
3.2.3 Reset 9
3.2.4 Stoppen der Bildverarbeitung 9
3.2.5 Programm beenden 10
3.2.6 Bilder von der Kamera holen 10
3.2.7 Bilder anzeigen 10
3.2.8 Entscheidung zwischen Hindernis und Verzweigung 11
3.2.9 Benutzer-Eingabe bei einer Verzweigung 11
3.2.10 Fahren in der Verzweigung 12
3.2.11 Hindernis erkannt 12
3.2.12 Benutzer-Eingabe bei einem Hindernis 13
3.2.13 Manuelle Steuerung als neuer Prozess 13
3.2.14 Erstellen der Nachricht an das Auto 14
3.2.15 Senden einer Nachricht an das Auto 14
3.2.16 Auto anhalten 14
3.2.17 Geschwindigkeitsregelung beim Anfahren 14
3.2.18 Aktualisieren der Dialogelemente 15
3.2.19 Batterie-Spannung anzeigen 15

4 PIC Board für die Cardbus-Wirelesskarte 16

4.1 Zusammenfassung der Änderungen . . . 16
4.2 Version mit I^2C-BUS . . . 16
4.3 Version mit RS232 . . . 18
4.4 Ansteuerung der Servos . . . 19
4.5 Taktfrequenz und Pulsweitenmodulation bei der Version mit I^2C . . . 19
4.6 Taktfrequenz und Pulsweitenmodulation bei der Version mit RS232 . . . 20
4.7 Empfangen und weiterleiten der UDP Nachrichten . . . 20
4.8 Hauptprogramm mittels I^2C-Bus . . . 23
4.9 Hauptprogramm mittels RS232 . . . 23
4.10 Die Funktion `set_pwm()` . . . 24
4.11 Automatische Abschaltung mittels Watchdog-Timer . . . 26
4.12 Die Verbindung der zwei PICs . . . 27
4.13 Die Batteriespannungsüberwachung . . . 27
4.14 Der Schrittmotor . . . 27
4.15 Endstufe für den Schrittmotor . . . 28
4.16 Konfiguration . . . 28
4.16.1 Chipweb Board . . . 28
4.16.2 Wirelesskamera . . . 29
4.16.3 Wirelesskarte beim PC . . . 29

5 Bildsensoren anderer Hersteller 30

5.1 CCD-CMOS Sensor . . . 30
5.2 CCD-Sensor von National . . . 30
5.3 Die Software . . . 31
5.4 Die Geschwindigkeit . . . 31
5.5 CCD-Sensor von Omnivision . . . 32

A Installation der Programme 35

A.1 ActiveX Komponente und Treiber für die Kamera installieren . . . 35
A.2 Bildverarbeitung . . . 35

B Die Control-Unit des Autos 36

Abbildungsverzeichnis

2.1 Matrix in C/C++ dynamisch erstellt 2
2.2 Erweiterung der Randzonen für z.B. den Kantendetektor 3
2.3 Linien-Detektoren 6

4.1 Ansteuerung der Servomotoren 19
4.2 Linearisierung der Steuerung 26
4.3 Ansteuerung einer Phase beim Schrittmotor 28

5.1 Einbindung des CCD-Sensors in das bestehende Projekt 30

B.1 Oberfläche der Control-Unit 36

Listings

2.1 Matrix in C/C++ dynamisch erstellen und löschen . . . 2
2.2 void CImageProcessing::buffer2array(unsigned char* pBuffer, float** array, const int pHEIGHT, const int pWIDTH) . . . 3
2.3 unsigned char* CImageProcessing::array2buffer(float** picArray, const int pHEIGHT, const int pWIDTH, const bool drawLine=0) . . . 4
2.4 void CImageProcessing::edgeDetection(int type, float** source, float** destination, const int pHEIGHT, const int pWIDTH) . . . 5
3.1 BOOL CMy80211CarDlg::OnInitDialog() . . . 8
3.2 void CMy80211CarDlg::OnStart() . . . 9
3.3 void CMy80211CarDlg::OnReset() . . . 9
3.4 void CMy80211CarDlg::OnStop() . . . 9
3.5 void CMy80211CarDlg::OnExit() . . . 10
3.6 unsigned char* CMy80211CarDlg::extractPicture(const int pHEIGHT, const int pWIDTH) . . . 10
3.7 void CMy80211CarDlg::displayBitmap(unsigned char* pBuffer, const int pHEIGHT, const int pWIDTH) . . . 10
3.8 void CMy80211CarDlg::makeDecisions() . . . 11
3.9 bool CMy80211CarDlg::displayMessageOnDetectedBypass() . . . 11
3.10 void CMy80211CarDlg::handlingBypass(bool pDirection) . . . 12
3.11 void CMy80211CarDlg::handlingLineInterruption() . . . 12
3.12 bool CMy80211CarDlg::displayMessageOnLineInterruption() . . . 13
3.13 void CMy80211CarDlg::OnStartManualCarControl() . . . 13
3.14 char* CMy80211CarDlg::createMessage(int pAngle) . . . 14
3.15 void CMy80211CarDlg::sendMessage(char* pMessage) . . . 14
3.16 int CMy80211CarDlg::adjustSpeed(int actualSpeed) . . . 14
3.17 CMy80211CarDlg::getBatteryVoltage() . . . 15
3.18 DWORD WINAPI processBatteryVoltage(LPVOID lpParam) . . . 15
4.1 void write_pic(BYTE faddress, BYTE fdata, BYTE fdata2) . . . 16
4.2 BYTE get_pic() . . . 17
4.3 void ssp_interupt() . . . 17
4.4 int_rda void serial_isr() . . . 18
4.5 byte bgetc() . . . 18
4.6 Taktfrequenz und Pulsweitenmodulation bei der Version mit I^2C . . . 19
4.7 Taktfrequenz und Pulsweitenmodulation bei der Version mit RS232 . . . 20
4.8 BOOL udp_recv(void) . . . 21
4.9 Frog-Port mit I^2C-Bus . . . 22
4.10 void set_pwm(DIRECTION fdirection) . . . 24
4.11 void set_pwm(byte axes, long value, boolean oper) . . . 25

1 Abstract

Das Auto fährt ohne Probleme der Linie entlang, Verzweigungen und Hindernisse werden auch erkannt. Bei den Verzweigungen wurde zuerst ein Template Matching implementiert, was aber dann schlechte Ergebnisse lieferte aufgrund des mechatronischen Aufbaus des Autos. Konkret heisst das, dass die Bildverarbeitung mit Template Matching zu langsam war. Es wurde dann eine andere Variante in Betracht gezogen, die viel weniger Rechenaufwand zu bewältigen hatte. Es wurden sog. Linien-Detektoren implementiert, die auf Helligkeitsunterschiede der Bildwerte im Helligkeits- und Kantenbild auf den Linien achten. Somit können Verzweigungen und Hindernisse in einem Schritt erkannt werden. Bei einem Hindernis hat der Benutzer Gelegenheit, das Auto selbst zu steuern. Bei einer Verzweigung kann der Benutzer die Richtung wählen, danach fährt das Auto in die vorgegebene Richtung selbstständig weiter. Bei der Fahrt in der Verzweigung werden andere Kantendetektoren in der Bildverarbeitung benutzt, dies hat den Effekt, dass nur Kanten mit einem bestimmten Richtungsvektor detektiert werden. Zum Schluss wurde die Bildverarbeitung noch in eine Klasse *CImageProcessing* gepackt, damit sind alle Funktionen zentral zugreifbar.
Die Kommunikation zwischen den Applikationen wurde schon am Anfang auf der Basis des Wireless Netzwerkes 802.11b verwirklicht. Die Kamera unterstützt nur TCP. Somit war nur das Protokoll zwischen Auto und PC wählbar. Für Realtime[1]-Anwendungen eignet sich das UDP Protokoll. Somit sendet die Wireless-Kamera die Bilder dem PC und wertet diese aus. Danach sendet der PC dem Auto die Steuerbefehle, welche im Empfänger verarbeitet werden. Der PIC[2], welcher die Wireless-Karte steuert, hat nicht zwei PWM[3] Ausgänge. Somit wird ein zweiter PIC mit zwei PWM Ausgängen benutzt, welcher die Steuerdaten über die serielle Schnittstelle bzw. über den I^2C-Bus empfängt. Die Variante mit dem I^2C-Bus funktionierte jedoch nicht stabil. Um eine stabile Geschwindigkeit zu erreichen, wurde versucht einen Schrittmotor einzubauen. Dies misslang, weil der Motor das nötige Drehmoment nicht erreichte.

[1]Definition Realtime: Eingangsdatenrate = Ausgangsdatenrate
[2]PIC = Microcontroller
[3]PWM = Pulsweitenmodulation

2 Bestandteile der Bildverarbeitung

2.1 Programmieren mit 2-dimensionalen Feldern

Die Bilddaten stehen anfangs in einem Vektor. Zwecks einfacherer und übersichtlicher Weiterverarbeitung werden diese Bilddaten in den Vektoren in 2-dimensionale Felder, also Matrizen, gespeichert.
Folgender Codeausschnitt zeigt die Erstellung solch einer Matrix in C, die Abbildung 2.1 verdeutlicht dies nochmals.

```
// create
float** pBuffer;
pBuffer = new float*[4];

for( int row=0; row<4; row++ ){
    pBuffer[row] = new float[3];
}

// access pixel: pBuffer[row][column]

// delete
for( row=0; row<4; row++ ){
    delete [] (pBuffer[row]);
}
delete [] pBuffer;
```

Listing 2.1: Matrix in C/C++ dynamisch erstellen und löschen

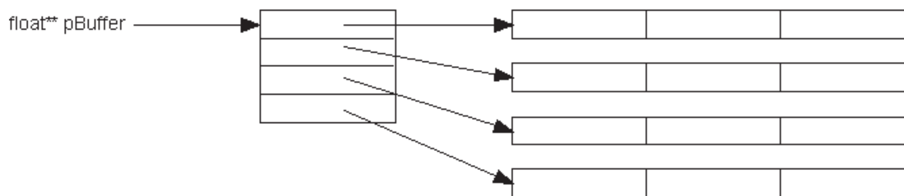


Abbildung 2.1: Matrix in C/C++ dynamisch erstellt

2.2 Beibehaltung der Bildgrössen

Das Problem stellte sich für die Polynomapproximation als das Helligkeits- und das Kantenbild nicht gleich gross waren. Das Kantenbild war kleiner als das Helligkeitsbild, was in diesem Fall nach der Kantendetektion (=Hochpass-Filterung) völlig normal war. Das war umständlich, denn man musste in den Berechnungen auf die Bildgrössen achten, sonst erfolgte ein Speicherzugriff ins Leere und somit ein Programmabsturz.
Dieses Problem wurde nun so gelöst, dass das Helligkeitsbild vor der Kantendetektion in den Randzonen herum vergrössert wird. Bei einem 3x3 Kantendetektor macht man einen Rand, bei einem 5x5 Kantendetektor macht man dementsprechend zwei Ränder um das Helligkeitsbild. Das Prinzip einer solchen Bildvergrösserung ist in Abbildung 2.2 schematisch dargestellt. Bei der Implementierung vergrössert man die Randzonen nur so viel wie nötig damit das Kantenbild schlussendlich gleich gross ist wie das Helligkeitsbild, sonst ist man die ganze Zeit nur am Bilder kopieren.

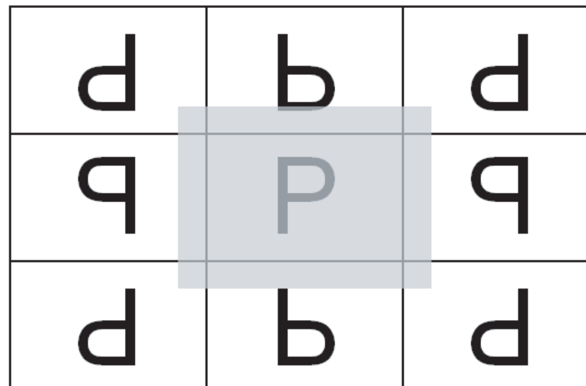


Abbildung 2.2: Erweiterung der Randzonen für z.B. den Kantendetektor

2.3 Programmieren mit Bitmaps

Ein- und Ausgabe im Windows Programm geschieht mittels Bitmaps. Es gibt ein gutes Tutorial im Internet unter `http://www.runicsoft.com/bmp.php`, wo Schritt für Schritt die Programmierung mit Bitmaps anhand eines Beispiels erklärt wird.

2.4 Die Bildverarbeitungsklasse *CImageProcessing*

Bis kurz vor Ende der Diplomarbeit war die Bildverarbeitung noch nicht in einer Klasse implementiert, sondern in Form einer Headerdatei und dessen Implementierung. Die definierten Variablen und Funktionen wurden nun als eine neue Klasse *CImageProcessing* erstellt. Damit kann in einem Programm ein Bildverarbeitungsobjekt erzeugt werden und alle bekannten Bildverarbeitungsalgorithmen der bisherigen Implementierung können nun auf das Objekt angewendet werden. Sicherlich kann die Bildverarbeitungsklasse noch verbessert oder erweitert werden, sie bietet damit eine Grundlage für weitere Semesterarbeiten oder sogar Diplomarbeiten.

2.4.1 Bilddaten in Vektor-und Matrixform

Die Bilddaten stehen als erstes in einem Vektor zur Verfügung. Um im weiteren Programmablauf bequem auf einzelne Bildelemente zugreifen zu können, werden die Bilddaten aus dem Vektor in eine Matrix überführt. Damit kann später auf ein beliebiges Pixel zugegriffen werden.

```
void CImageProcessing::buffer2array(...) {
    ...

    for( row=0; row<pHEIGHT; row++ ){
        for( int x=0; x<3*pWIDTH; x+=3, tempX++ ){
            bufpos = (pHEIGHT - row - 1) * psw + x;
            // We read the pixel data from the buffer and write it to our array,
            // which holds our gray-scaled picture
            column = tempX%pWIDTH;
            array[row][column] = (float)(0.299*pBuffer[bufpos] + 0.587*pBuffer[bufpos + 1]
                                        + 0.114*pBuffer[bufpos + 2]);
        }
    }
    ...
}
```

Listing 2.2: void CImageProcessing::buffer2array(unsigned char* pBuffer, float** array, const int pHEIGHT, const int pWIDTH)

Umgekehrt ist es auch von Vorteil, wenn man z.B. ein Bild auf dem Bildschirm ausgeben will, eine entsprechende Funktion zu schreiben, welche das Bild in der Matrixform wieder in einen Bildvektor überführt.

```
unsigned char* CImageProcessing::array2buffer(...)
{
    ...

    // Now we loop trough our matrix and write data to the new buffer
    for(  row=0; row<pHEIGHT; row++ ){
        for( x=0; x<3*pWIDTH; x+=3, tempX++ ){
            // Position in padded buffer
            newpos = ( pHEIGHT - row - 1 ) * psw + x;
            column = tempX%pWIDTH;

            // Extract the gray-scaled color information from our matrix
            grayValue = picArray[row][column];

            // GrayValue is assigned to Red, Green and Blue,
            // this results in a final grayscaled picture...
            picBuffer[newpos] = (unsigned char)grayValue;
            picBuffer[newpos + 1] = (unsigned char)grayValue;
            picBuffer[newpos + 2] = (unsigned char)grayValue;
            ...
        }
        ...
    }
    return picBuffer;
}
```

Listing 2.3: unsigned char* CImageProcessing::array2buffer(float** picArray, const int pHEIGHT, const int pWIDTH, const bool drawLine=0)

2.4.2 Abtasten

Abtasten eines Bildes kann in Analogie zur Abtastung eines analogen Signals gesehen werden. In jedem Fall geht bei der Abtastung Information über das abgetastete Objekt verloren, aber gerade in der Bildverarbeitung ist dieser Effekt so wirkungsvoll, weil damit der Rechenaufwand enorm abnimmt. Für ein Bild mit der Höhe 240 Pixel und einer Breite von 320 Pixeln ergibt das grob geschätzt 76800 Multiplikationen. Würde das Bild zuerst noch entsprechend kleiner gemacht, also abgetastet, so ergibt das mit einem Abtastfaktor von 4 einen 16-fach kleineren Multiplikationsaufwand (4800 Multiplikationen). Da die Rechenleistung eines Intel Pentium III limitiert ist, ist eine vorgängige Abtastung des Bildes zwingend notwendig um der Datenflut entgegenzuwirken.

Implementation:
*void CImageProcessing::downsampling(float** pSource, float** pResult, const int pHEIGHT, const int pWIDTH)*

2.4.3 Kantendetektion

Das Kantenbild wird zusammen mit dem Helligkeitsbild in der Polynomapproximation gebraucht, weil das Kantenbild gegenüber Helligkeitsänderungen nicht so empfindlich ist wie das Helligkeitsbild. Mit dem ersten Parameter kann der zu verwendende Kantendetektor (=Hochpassfilter) ausgewählt werden, dieses Feature kommt bei der Fahrt in der Verzweigung besonders gut zur Geltung.
Den zu wählenden Kantendetektor kann man mittels einer Enumeration auswählen, konkret sieht diese so aus: `enum edgeDetectorFilter { NORMAL, LEFT_BYPASS, RIGHT_BYPASS };`

- `NORMAL`: Alle Kanten werden erfasst.

$$G_x = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \quad G_y = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \quad \text{mit} \quad G = \sqrt{G_x^2 + G_y^2}$$

- `LEFT_BYPASS`: Kanten mit einem Richtungsvektor von unten links nach oben rechts werden nicht erfasst.

$$G = \begin{pmatrix} 0 & -1 & -1 \\ 1 & 0 & -1 \\ 1 & 1 & 0 \end{pmatrix}$$

- `RIGHT_BYPASS`: Kanten mit einem Richtungsvektor von unten rechts nach oben links werden nicht erfasst.

$$G = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -1 \end{pmatrix}$$

```
void CImageProcessing::edgeDetection( int type, ... )
{
    ...

    if( type==NORMAL ){
        ...
    }

    if( type==LEFT_BYPASS ){
        ...
    }

    if( type==RIGHT_BYPASS ){
        ...
    }
    ...
}
```

Listing 2.4: void CImageProcessing::edgeDetection(int type, float** source, float** destination, const int pHEIGHT, const int pWIDTH)

2.4.4 Normalisierung

Theoretisches zu diesem Thema in [1]. Die Implementation hat aber geändert!

Implementation:
*void CImageProcessing::normalize(float** pIntensityPicture, const float pMinIntensity, float** pEdgePicture, const float pMinEdge, const int pHEIGHT, const int pWIDTH)*

2.4.5 Polynomapproximation

In [1] wurde die Polynomapproximation schon beschrieben. Auch hier hat nur die Implementation geändert.

Implementation:
*void CImageProcessing::polynomApproximation(float** pIntensityPicture, float** pEdgePicture, float* result, const int pHEIGHT, const int pWIDTH)*

2.4.6 Radsteuerung

Es wurde der 3. Algorithmus gemäss [1] S. 19fff. implementiert.

Implementation:
int CImageProcessing::bezierApproximation(const float pLineParameters, const int pAnzahl = 20, const float pGewichtung = 20, const float pRadius1 = 15, const float pRadius2 = 20)*

2.4.7 Erkennen von Hindernissen und Verzweigungen

Template Matching

Die erste Variante zum Erkennen der Verzweigungen führte über Template Matching. In einem Grafikprogramm wurden anhand einiger aufgenommener Helligkeits- und Kantenbilder entsprechende Vorlagen (=Templates) für Verzweigungen aufgenommen. Die Implementation dieses Feature war fertig, bevor das Auto fahren konnte und daher war diese Implementation noch nicht ausgetestet worden. Als dann mal das Auto wieder fahren konnte, stellte sich heraus, dass das Auto zu schnell fuhr. Sogar mit der kleinstmöglichen Geschwindigkeit war das Auto für die Benutzung des Template Matching zu schnell; oder umgekehrt formuliert, die Bildverarbeitung mit eingeschaltetem Template-Matching war zu langsam. Messungen ergaben eine Bildverarbeitungsrate von etwa 10 Bildern/sec.
Das Erkennen von Verzweigungen mittels Template-Matching wurde dann fallen gelassen und man implementierte das Erkennen von Verzweigungen und Hindernissen mittels Linien-Detektoren.

Linien-Detektoren

Das Prinzip der Linien-Detektoren ist folgendes: Aufgrund der berechneten Linien-Parameter wie Steigung und Offset werden ähnliche Linien konstruiert. Danach werden mit den ähnlichen Linien einige Berechnungen durchgeführt und entschieden, ob eine Verzweigung oder ein Hindernis vorliegt. Auf dem

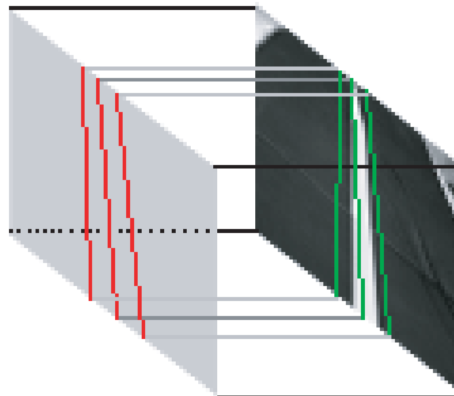


Abbildung 2.3: Linien-Detektoren

linken Teilbild sind 3 Linien ersichtlich, welche sich mit gleichen Linien im Helligkeits- und Kantenbild decken. Es werden nun Vektoren erstellt, welche die Werte unter den deckungsgleichen Linien aus dem Kanten- sowie Helligkeitsbild enthalten. Wir definieren die Helligkeitslinie als die Linie in der Mitte und die anderen als Seitenlinien.

Erkennen einer Verzweigung:
Die Vektoren für die Seitenlinien enthalten die Werte des Kantenbildes. Aus den obersten 22 Werten wird die Standardabweichung berechnet, übersteigt diese einen bestimmt festgelegten Wert, so liegt eine Verzweigung vor und die entsprechenden Stellen im Resultatvektor werden gesetzt.

Erkennen eines Hindernisses:
Per Definition ist ein Hindernis ein Linienunterbruch. Gleiches Vorgehen wie beim Erkennen der Verzweigung, nur werden hier nicht die obersten 22 Werte für die Berechnung berücksichtigt, sondern man fängt weiter unten an. Zusätzlich wird die Linie auf dem Helligkeitsbild überprüft: Ist unter der berechneten Linie wirklich eine weisse Linie vorhanden? Wenn auch hier die Standardabweichung einen bestimmen

Wert übersteigt, dann wurde das Hindernis erkannt und der Resultatvektor wird gesetzt.
Desweiteren werden im Kantenbild unter der Mittenlinie auch noch die Anzahl Übergänge Hell↔Dunkel gezählt. Sind es mehr als 1 Übergang, so liegt es nahe, dass ein Hindernis auf der Fahrstrecke liegt und der Resultatvektor wird wieder entsprechend gesetzt.
Die Standardabweichung eines Vektors x der Länge N wird wie folgt berechnet:

$$\text{stddev}(x) = \sqrt{\frac{1}{N-1}\sum_{k=0}^{N-1}(x_k - \bar{x})^2}$$

Implementation:
bool CImageProcessing::verifyLineContinuity()*

2.4.8 Bild speichern

Diese Member-Funktion wurde ähnlich schon in [1] implementiert.

Implementation:
void CImageProcessing::saveBitmap(char pFile, float** picArray, const int pHEIGHT, const int pWIDTH, const bool drawLine=0)*

2.4.9 Bild analysieren

Die Member-Funktion `analyzePicture( unsigned char* pBuffer, int typeOfEdgeDetection )` analysiert ein Bild auf eine Linie hin und gibt als Resultat den Winkel für die Steuerung der Räder zurück. Der erste Übergabeparameter ist das Bild in einem Vektor, der zweite Übergabeparameter ist der Typ des Kantendetektors.

Implementation:
int CImageProcessing::analyzePicture(unsigned char pBuffer, int typeOfEdgeDetection)*

3 Bestandteile des Hauptprogramms

3.1 Grafische Ein- und Ausgabe mit Device Context

Das Programmieren mit Device Context wurde im Praktikum der Digitalen Medien besprochen. Weitere Informationen findet man unter `http://medialab.ch/dm/praktikum/praktikum5/original/`, wo die Praktikumsanleitung zu finden ist.
Auch eine gute Hilfestellung bietet die Site `http://msdn.microsoft.com`.

3.2 Die Klasse *CMy80211CarDlg*

3.2.1 Initialisierung der Variablen

Diese Funktion initialisiert Variablen des Dialogfensters, wie etwa den Schwellwert für die Kantendetektion oder die Startgeschwindigkeit des Autos. Desweiteren werden noch das Socket für die Kommunikation mit dem Auto, Headers für die Bitmaps und das ActiveX-Element initialisiert. Die Funktion `initRoutine()` ist für die Initialisierung von Bildverarbeitungsvariablen verantwortlich.

```
BOOL CMy80211CarDlg::OnInitDialog()
{
    ...

    // Initialize some dialog-elements
    ...

    // Our dialog is always on top, why?
    // Because we extract the picture from the screen, and assume that there
    // is another window partly covering our picture from the camera, then
    // the edge picture will contain the edge of the covering window!
    SetWindowPos(&this->wndTopMost, 0, 0, 0, 0, SWP_NOMOVE|SWP_NOSIZE);

    // Initialize our socket
    ...

    // Initialize bitmap headers
        // One header for the source picture
        m_sourceBMI.bmiHeader.biSize = 40;
        ...

        // One header for the destination picture
        m_destinationBMI.bmiHeader.biSize = 40;
        ...

    // Initialize variables for image processing
    initRoutine();

    // Run the ActiveX component
    ...
    m_xplug.Play();
    ...
}
```

Listing 3.1: BOOL CMy80211CarDlg::OnInitDialog()

3.2.2 Starten der Bildverarbeitung

Ein Klick auf den Start-Button erstellt zwei neue Threads; der eine übernimmt die gesamte Bildverarbeitung, der andere beschäftigt sich mit der Batterie-Spannung. Mehrmaliges Klicken auf den Start-Button führt zu einem Programmabsturz, darum wird noch die Member-Variable `m_startButton` gesetzt, welche eine mehrmalige gleichzeitige Ausführung beider Threads nicht erlaubt.

```
void CMy80211CarDlg::OnStart()
{
    if( !m_startButton ){
        m_startButton = TRUE;
        m_speedControl.SetPos(7);
        speedCounter = 0;
        UpdateData( FALSE );

        m_hMSB = CreateMutex( NULL, 0, NULL );
        WaitForSingleObject( m_hMSB, INFINITE );
        m_SB = FALSE;
        ReleaseMutex( m_hMSB );

        DWORD dwThreadID1;
        m_hT1 = CreateThread(
            NULL,            // No special security attributes
            0,               // Default stack size
            workingThread,   // Thread function
            this,            // Thread argument
            0,               // Run this thread immediately after creation
            &dwThreadID1);   // Pointer that receives the thread identifier

        DWORD dwThreadID2;
        m_hT2 = CreateThread(
            NULL,
            0,
            processBatteryVoltage,
            this,
            0,
            &dwThreadID2);
    }
    else{
        Beep(500,25);
    }
}
```

Listing 3.2: void CMy80211CarDlg::OnStart()

3.2.3 Reset

Die Reset-Funktion ist dazu da, dass wenn das Auto die Linie verliert und das Line-Locking dann falsche Werte liefert, die Gewichtungsmatrix für das nächste Bild neu zu setzen, und zwar mit den Standardeinstellungen.

```
void CMy80211CarDlg::OnReset()
{
    m_imageProcessingObject.resetWeightingPicture();
}
```

Listing 3.3: void CMy80211CarDlg::OnReset()

3.2.4 Stoppen der Bildverarbeitung

Ein Klick auf den Stop-Button setzt die Member-Variable `m_SB` auf `TRUE`. Dies veranlasst den mit einem Klick auf den Start-Button erstellten Thread zu stoppen. Will die Bildverarbeitung wieder gestartet werden, so geschieht das wiederum mit einem Klick auf den Start-Button, da die entsprechende Member-Variable `m_StartButton` wieder gesetzt wurde.

```
void CMy80211CarDlg::OnStop()
{
    WaitForSingleObject( m_hMSB, INFINITE );
    m_SB = TRUE;
    ReleaseMutex( m_hMSB );

    // Stop the car
    stopCar();

    // Set start-button variable
    m_startButton = FALSE;

    // Set m_speedCounter
```

```
    m_speedCounter = 0;
}
```

Listing 3.4: void CMy80211CarDlg::OnStop()

3.2.5 Programm beenden

Das Verlassen des Programms wird mit einem Klick auf den Exit-Button veranlasst. In diesem Fall wird noch die Member-Funktion `OnStop()` ausgeführt und danach wird noch das Socket geschlossen und mittels `WSACleanup()` der Gebrauch von ws2_32.dll beendet.

```
void CMy80211CarDlg::OnExit()
{
    OnStop();
    closesocket(mySocket); WSACleanup();

    CDialog::OnOK();
}
```

Listing 3.5: void CMy80211CarDlg::OnExit()

3.2.6 Bilder von der Kamera holen

Die Bilder werden nicht direkt von der Kamera erhalten, wie das in der Überschrift erwähnt ist. Die Einbindung der Kamera erfolgt über die von D-Link mitgelieferte ActiveX-Komponente. Wird diese Komponente in den Dialog eingebunden und initialisiert (s. Member-Funktion 3.1), liefert sie Bilder, aber es war nicht möglich herauszufinden, wie man an die Bilder-Rohdaten gelangt. Das Problem wurde mit einem kleinen Turn-Around gelöst; man erstellt zwei Device Context (s. Abschnitt 3.1), einen für den Bildschirm und einen für den Speicher und macht die beiden zueinander kompatibel. Danach erstellt man ein Bitmap mit den entsprechenden Parametern und mittels den Befehlen `BitBlt(...)` und `GetDIBits(...)` schiebt man die rohen Bilddaten in den durch `picBuffer` bereitgestellten Speicher.

```
unsigned char* CMy80211CarDlg::extractPicture( const int pHEIGHT, const int pWIDTH )
{
    unsigned char* picBuffer = new unsigned char[3*pHEIGHT*pWIDTH];

    CClientDC cdc(this);
    CDC dc;
    dc.CreateCompatibleDC( &cdc );

    CBitmap bm;
    bm.CreateBitmap(pWIDTH,pHEIGHT,1,GetDeviceCaps(dc,BITSPIXEL),NULL);

    dc.SelectObject(&bm);
    BitBlt(dc,0,0,pWIDTH,pHEIGHT,cdc,11+15,11+17,SRCCOPY);
    GetDIBits(dc,bm,0,pHEIGHT,picBuffer,&m_sourceBMI,DIB_RGB_COLORS);

    return picBuffer;
}
```

Listing 3.6: unsigned char* CMy80211CarDlg::extractPicture(const int pHEIGHT, const int pWIDTH)

3.2.7 Bilder anzeigen

Bilder in einem MFC-basierten Programm anzuzeigen ist sehr einfach. Auch hier erstellt man zwei Device Context (s. Abschnitt 3.1), jeweils einen für den Bildschirm und einen für den Speicher und macht die beiden kompatibel zueinander. Anschliessend wird ein Bitmap erzeugt und mittels der Funktion `SetDIBits(...)` und den Bitmap-Daten (`unsigned char* pBuffer`) in den entsprechenden Device Context geladen. Die Funktion `StretchBlt(...)` streckt das Bitmap auf die vorgegebene Breite und Höhe. Zum Schluss wird das Array der Bitmap-Daten noch gelöscht.

```
void CMy80211CarDlg::displayBitmap( unsigned char* pBuffer, const int pHEIGHT, const int pWIDTH )
{
    m_destinationBMI.bmiHeader.biWidth = pWIDTH;
```

```
    m_destinationBMI.bmiHeader.biHeight = pHEIGHT;

    CClientDC cdc(this);
    CDC dc;
    dc.CreateCompatibleDC(&cdc);

    CBitmap bm;
    bm.CreateBitmap(pWIDTH,pHEIGHT,1,GetDeviceCaps(dc,BITSPIXEL),NULL);

    SetDIBits(dc,bm,0,pHEIGHT,pBuffer,&m_destinationBMI,DIB_RGB_COLORS);

    dc.SelectObject(&bm);
    cdc.StretchBlt(11+320+20+17,11+17,320,240,&dc,0,0,pWIDTH,pHEIGHT,SRCCOPY);

    delete [] pBuffer;
}
```

Listing 3.7: void CMy80211CarDlg::displayBitmap(unsigned char* pBuffer, const int pHEIGHT, const int pWIDTH)

3.2.8 Entscheidung zwischen Hindernis und Verzweigung

In dieser Funktion werden die wichtigsten Entscheidungen bezüglich Hindernis und Verzweigung getroffen. Entscheidungsgrundlage liefert die Funktion `bool* CImageProcessing::verifyLineContinuity()`. Die Entscheidungsfindung beginnt erst ab 10 Bildern, solange hat die Bildverarbeitung Zeit, eine vernünftige Linie zu finden und sich somit zu stabilisieren.

```
void CMy80211CarDlg::makeDecisions()
{
    if( ++m_counter>=10 ){
        bool* parameter;
        parameter = m_imageProcessingObject.verifyLineContinuity();

        if( (parameter[2] && parameter[3]) || (parameter[4] && parameter[5]) ){
            // interruption
            stopCar();
            handlingLineInterruption();
        }
        else if( parameter[0] && parameter[1] ){
            // bypass
            stopCar();
            handlingBypass( displayMessageOnDetectedBypass() );
        }
        UpdateData( FALSE );
    }
}
```

Listing 3.8: void CMy80211CarDlg::makeDecisions()

3.2.9 Benutzer-Eingabe bei einer Verzweigung

Wird eine Verzweigung erkannt, muss der Benutzer wählen, ob er Links oder Rechts weiterfahren möchte.

```
bool CMy80211CarDlg::displayMessageOnDetectedBypass()
{
    int answer;
    answer = AfxMessageBox("Verzweigung!␣\n\nYES=Links␣und␣NO=Rechts", MB_ICONWARNING|MB_YESNO);
    if( IDYES==answer ){
        return 1;
    }
    else{
        return 0;
    }
}
```

Listing 3.9: bool CMy80211CarDlg::displayMessageOnDetectedBypass()

3.2.10 Fahren in der Verzweigung

Hier wird die Fahrt in der Verzweigung geregelt. Das Wichtigste in dieser Funktion ist der Einsatz der Bildverarbeitung mit verschiedenen Kantendetektoren `analyzePicture(inBuffer,<Typ des Kantendetektors>)`. Je nachdem, was der Benutzer für eine Richtung wählt, wird ein richtungs-selektives Hochpassfilter (Kantendetektor) verwendet, welche die andere Seite der Verzweigung nicht detektieren kann oder zumindest nur sehr schwach.

```
void CMy80211CarDlg::handlingBypass( bool pDirection )
{
   stopCar();

    unsigned char* inBuffer;
    unsigned char* outBuffer;
    const int pWidth = m_sourceBMI.bmiHeader.biWidth;
    const int pHeight = m_sourceBMI.bmiHeader.biHeight;

    if( pDirection==1 ){
        // Go left
        for( int i=0; i<m_numberOfPicturesToAnalyze; i++ ){
            inBuffer = extractPicture(pHeight,pWidth);
            m_angle = m_imageProcessingObject.analyzePicture(inBuffer,LEFT_BYPASS);
            outBuffer = m_imageProcessingObject.array2buffer(
                                m_imageProcessingObject.m_edgePictureB,
                                m_imageProcessingObject.EDGE_PIC_HEIGHT,
                                m_imageProcessingObject.EDGE_PIC_WIDTH,1);
            displayBitmap(outBuffer,m_imageProcessingObject.EDGE_PIC_HEIGHT,
                                m_imageProcessingObject.EDGE_PIC_WIDTH);

            // ... send here udp packets to the car
            sendMessage( createMessage(m_angle) );
        }
    }
    if( pDirection==0 ){
        // Go right
        for( int i=0; i<m_numberOfPicturesToAnalyze; i++ ){
            inBuffer = extractPicture(pHeight,pWidth);
            m_angle = m_imageProcessingObject.analyzePicture(inBuffer,RIGHT_BYPASS);
            outBuffer = m_imageProcessingObject.array2buffer(
                                m_imageProcessingObject.m_edgePictureB,
                                m_imageProcessingObject.EDGE_PIC_HEIGHT,
                                m_imageProcessingObject.EDGE_PIC_WIDTH,1);
            displayBitmap(outBuffer,m_imageProcessingObject.EDGE_PIC_HEIGHT,
                                m_imageProcessingObject.EDGE_PIC_WIDTH);

            // ... send here udp packets to the car
            sendMessage( createMessage(m_angle) );
        }
    }
}
```

Listing 3.10: void CMy80211CarDlg::handlingBypass(bool pDirection)

3.2.11 Hindernis erkannt

Erkennt die Bildverarbeitung ein Hindernis, wird der Benutzer gefragt, ob er manuelle Steuerung will. Ist dies der Fall, dann startet die Member-Funktion `OnStartManualCarControl()` einen neuen Prozess mit der manuellen Steuerung.

```
void CMy80211CarDlg::handlingLineInterruption()
{
    stopCar();

    if( displayMessageOnLineInterruption() ){
         OnStartManualCarControl();
    }
    else{
         OnStop();
    }
}
```

Listing 3.11: void CMy80211CarDlg::handlingLineInterruption()

3.2.12 Benutzer-Eingabe bei einem Hindernis

Rückgabewert der Funktion ist der `int`-Wert des geklickten Buttons.

```
bool CMy80211CarDlg::displayMessageOnLineInterruption()
{
    int answer;
    answer = AfxMessageBox("Hindernis!␣\n\nWollen␣Sie␣manuelle␣Steuerung?", MB_ICONQUESTION|MB_YESNO);
    if( IDYES==answer ){
        return 1;
    }
    else{
        return 0;
    }
}
```

Listing 3.12: bool CMy80211CarDlg::displayMessageOnLineInterruption()

3.2.13 Manuelle Steuerung als neuer Prozess

Wünscht der Benutzer manuelle Steuerung, so wird diese Funktion aufgerufen. Die Struktur `STARTUPINFO` wird bei Erstellen eines neuen Prozesses benötigt und enthält Angaben wie die Erscheinung des Dialogfensters des neuen Prozesses. Die Struktur `PROCESS_INFORMATION` wird bei Aufruf von `CreateProcess()` mit prozess- und thread-spezifischen Variablen (Handle & ID) des neu erstellten Prozesses initialisiert. Kann der neue Prozess nicht erstellt werden, so wird eine Fehlermeldung angezeigt; mögliche Fehlerursachen sind zum Beispiel ein falscher Pfad zur ausführbaren Datei der manuellen Steuerung.

```
void CMy80211CarDlg::OnStartManualCarControl()
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    // Fill a block of memory with zeros
    // memset(...) could be also used
    ZeroMemory(&si,sizeof(si));

    // Size of structure STARTUPINFO in bytes
    si.cb = sizeof(si);

    ZeroMemory(&pi,sizeof(pi));

    if( !CreateProcess( ".\\ManualControl\\ManualCarControl.exe",
                                    // Which application to start
                        NULL,       // No command line
                        NULL,       // Process handle not inheritable
                        NULL,       // Thread handle not inheritable
                        FALSE,      // Set handle inheritance to FALSE
                        0,          // No creation flags
                        NULL,       // Use parent's environment block
                        NULL,       // Use parent's starting directory
                        &si,        // Pointer to STARTUPINFO structure
                        &pi )       // Pointer to PROCESS_INFORMATION structure
            ){
        AfxMessageBox("CreateProcess␣in␣handlingLineInterruption()␣failed",MB_ICONERROR);
    }
    // Wait until child process terminates
    WaitForSingleObject(pi.hProcess,INFINITE);

    // Close process and thread handles
   ...
}
```

Listing 3.13: void CMy80211CarDlg::OnStartManualCarControl()

Die manuelle Steuerung wurde als alleinstehendes Projekt in MS Visual C++ geschrieben und ist auch als alleinstehendes Programm brauchbar. Der Code der manuellen Steuerung wird hier nicht weiter besprochen, weil er so einfach ist. Desweiteren kann man selbst wiederum ein Programm schreiben, welches die manuelle Steuerung des Autos übernimmt, allerdings muss am Schluss die ausführbare Datei oder die Applikation in Listing 3.13 umbenannt werden.

3.2.14 Erstellen der Nachricht an das Auto

Die Member-Funktion `createMessage( int pAngle )` ist auch wieder selbsterklärend. Der Übergabeparameter ist der Winkel zur Radsteuerung, dieser wird in der Funktion noch in einen String gewandelt. Die Member-Variable `m_speedPacket`[1] ist vom Typ const int und wird mittels Initialisierungsliste im Klassenkonstruktor gesetzt. Der Rückgabewert ist die erstellte Nachricht.

```
char* CMy80211CarDlg::createMessage( int pAngle )
{
    char* myMessage = new char[m_sizeOfMessage];
    char temp[3];

    if( !(packetCounter%m_speedPacket) ){
        // prepare the speed information
        // Format: V+xx or V-xx
        ...
    }
    else{
        // prepare the direction information
        // Format: H+xx  or  H-xx
        ...
    }

    ++packetCounter;

    return myMessage;
}
```

Listing 3.14: char* CMy80211CarDlg::createMessage(int pAngle)

3.2.15 Senden einer Nachricht an das Auto

Die Member-Funktion `sendMessage( char* pMessage )` sendet eine Nachricht in Form eines UDP Packetes an das Auto. Die Member-Variable `m_sizeOfMessage` ist vom Typ const int und wird mittels Initialisierungsliste im Klassenkonstruktor gesetzt. Nach dem Senden wird die Nachricht noch gelöscht, da sie dann nicht mehr gebraucht wird.

```
void CMy80211CarDlg::sendMessage( char* pMessage )
{
    sendto( mySocket, pMessage, m_sizeOfMessage, 0, (sockaddr*)&addrSrv, sizeof(addrSrv) );
    delete [] pMessage;
}
```

Listing 3.15: void CMy80211CarDlg::sendMessage(char* pMessage)

3.2.16 Auto anhalten

Die Member-Funktion `stopCar()` stoppt das Auto, indem es ihm für die Geschwindigket `V+00` und für die Radstellung `H+00` sendet.

3.2.17 Geschwindigkeitsregelung beim Anfahren

Die Anfahrtgeschwindigkeit des Autos ist für die Bildverarbeitung ein wenig zu schnell, Verzweigungen und Hindernisse werden dann entsprechend später oder gar nicht mehr erkannt. Darum wird die Geschwindigkeit des Autos nach dem Anfahren sukzessive nach Unten korrigiert. Das alles geschieht automatisch ohne eine User-Eingabe, aber der User kann die Geschwindigkeit auch nach eigenem Ermessen selbst einstellen.

```
int CMy80211CarDlg::adjustSpeed( int actualSpeed )
{
    switch(m_speedCounter){
        case 0:
            break;
        case 1:
```

[1] Ein Wert von 3 für diese Variable heisst, das jede dritte Nachricht an das Auto die Geschwindigkeitsinformation enthält.

```
            break;
        case 2:
            break;
        case 3: --actualSpeed;
            break;
        case 4:
            break;
        case 5:
            break;
        case 6: --actualSpeed;
            break;
    }

    ++m_speedCounter;
    return actualSpeed;
}
```

Listing 3.16: int CMy80211CarDlg::adjustSpeed(int actualSpeed)

3.2.18 Aktualisieren der Dialogelemente

Die Member-Funktion `updateDialogVariables()` aktualisiert den Schwellwert für die Kantendetektion und die Geschwindigkeit des Autos.

3.2.19 Batterie-Spannung anzeigen

Der Stromverbrauch des Autos ist hoch, bedenkt man was die Digitalkamera verbraucht. Darum ist es nötig, die aktuelle Batteriespannung auszulesen und anzuzeigen. Somit weiss der Benutzer, wann das Auto bald nicht mehr fahren kann. Die Batteriespannung wird als Progress-Bar angezeigt, wobei zwischen drei Farben unterschieden wird. Grün für genügend Strom, Gelb für den normalen Bereich und Rot für ungenügende Batteriespannung.

```
void CMy80211CarDlg::getBatteryVoltage()
{
        char recvMessage[] = {'0'};

        sendto( mySocket, &m_getCmdBatteryVoltage, 1, 0, (sockaddr*)&addrSrv, sizeof(addrSrv) );
        recvfrom( mySocket, recvMessage, 1, 0, NULL, NULL );

        int batteryVoltage = (((unsigned char)recvMessage[0])*1000*10/255);

        ...

        m_ctrlBatteryVoltage.SetPos( batteryVoltage );
}
```

Listing 3.17: CMy80211CarDlg::getBatteryVoltage()

Das Auslesen der Batteriespannung geschieht in einem eigens dafür erstellten Thread, weil das Warten auf die Antwort des Autos den Programmablauf (Bildverarbeitung) blockieren würde (`recvfrom()` blockiert!).

```
DWORD WINAPI processBatteryVoltage( LPVOID lpParam )
{
    CMy80211CarDlg* pDlg = (CMy80211CarDlg*)lpParam;
    BOOLEAN localStopButton = FALSE;

    while( !localStopButton ){
        pDlg->getBatteryVoltage();
        Sleep(10000);
        ...
    }
    ...
}
```

Listing 3.18: DWORD WINAPI processBatteryVoltage(LPVOID lpParam)

4 PIC Board für die Cardbus-Wirelesskarte

Das Board ER21 ist eine komplette Lösung für eine Schnittstelle im Wireless- Netzwerk. Es besteht aus einer Cardbus-Wirelesskarte (nur 5V Typen), einem PIC18F451, einem RJ45 Netzwerkanschluss und einem seriellen Port. Auf dem PIC ist bereits eine funktionsfähige Software installiert, die verschiedene Funktionen beinhaltet:

- Webserver zur Überprüfung der Funktion und zur Konfiguration des Boardes
- Timeserver
- Echoserver
- etc.

Die Software des PIC's besteht aus der Hauptdatei `p18web.c` und mehreren Include-Dateien.
Anfangs der Arbeit versuchte man möglichst viel Strom zu sparen. Somit kam man auf die Idee, die Wirelesskamera durch ein CCD-Sensor zu ersetzen. Dann versuchte man möglichst viele Pins am PIC frei zu bekommen, damit man später genug Pins für den CCD-Sensor zur Verfügung hat. Zum Beispiel wurde anstelle der RS232 Schnittstelle der I^2C-Bus benutzt. Während der Arbeit sah man jedoch, dass man damit nicht ans Ziel kam und benutzte wieder die serielle Schnittstelle. Daher werden nachfolgend je zwei Programme dokumentiert. Eines benutzt die serielle Schnittstelle und das andere den I^2C-Bus.

4.1 Zusammenfassung der Änderungen

I^2C:

- In der Datei `p18_udp.c` wurde der Frog-Server eingefügt.
  In der Datei `p18_udp.c` wurden zusätzliche Funktionen definiert für die Kommunikation via I^2C-Bus.

RS232:

- In der Datei `p18web.c` wurde die Baudrate für die serielle Kommunikation von 9600 auf 4800 gesetzt.
- In der Datei `p18_udp.c` wurde der Frog-Server eingefügt.

4.2 Version mit I^2C-BUS

Beim CCS-Compiler wurden diverse Beispielprogramme mitgeliefert. Im Installationsordner findet man die Files (`ex_extee.c`, `2464.c`, `ex_slave.c`) für die Kommunikation via I^2C. Die Beispielprogramme simulieren ein externes EEPROM. Die Architektur wurde für unser Projekt übernommen, um die Daten zwischen dem ersten und zweiten PIC auszutauschen. Im File `p18_udp.c` wurden nun die Funktionen `write_pic()` und `get_pic()` implementiert.

```
void write_pic(BYTE faddress, BYTE fdata, BYTE fdata2) {
    i2c_start();
    i2c_write(PIC_ADDR);
    i2c_write(faddress);
    i2c_write(fdata);
    i2c_write(fdata2);
```

```
    i2c_stop();
}
```

Listing 4.1: void write_pic(BYTE faddress, BYTE fdata, BYTE fdata2)

```
BYTE get_pic() {
    BYTE data;
    i2c_start();
    i2c_write(PIC_ADDR);
    i2c_write(0x04);
    i2c_start();
    i2c_write(PIC_ADDR+1);
    data=i2c_read(0);
    i2c_stop();
    return(data);
}
```

Listing 4.2: BYTE get_pic()

Die Adresse des Slaves kann im Hauptfile `p18web.c` definiert werden. `write_pic` schreibt zwei Bytes in das Register `faddress` des zweiten PIC. `get_pic` setzt das Register des zweiten PIC auf `04` und liest dann diesen Wert. Dieser Ablauf ist analog eines externen EEPROM. Beim zweiten PIC wurden die Funktionen etwas abgeändert.

```
typedef enum {NOTHING, CONTROL_READ, ADDRESS_READ, ADDRESS_READ2, READ_COMMAND_READ} I2C_STATE;
I2C_STATE fState;
byte address, buffer[0x10];
#INT_SSP
void ssp_interupt ()    //I2C {
    byte incoming;
#if DEBUG               //debug
    output_high(PIN_B0);
    output_low(PIN_B0);
#endif
    if (i2c_poll() == FALSE)
    {
        if (fState == ADDRESS_READ)
        {  //i2c_poll() returns false on the
            i2c_write (buffer[address]);//interupt receiving the second
            fState = NOTHING;           //command byte for random read operation
            //DEBUG_PUTS("Write I2C");
        }
    }
    else
    {
        incoming = i2c_read();
        if (fState == NOTHING){
            fState = CONTROL_READ;
            //DEBUG_PUTS("I2C chip conect");
        }
        else if (fState == CONTROL_READ)
        {
            address = incoming;
            fState = ADDRESS_READ;
            //DEBUG_PUTS("I2C read address");
        }
        else if (fState == ADDRESS_READ)
        {
            buffer[0] = incoming;
            fState = ADDRESS_READ2;
            //DEBUG_PUTS("I2C read byte1");
        }
        else if (fState == ADDRESS_READ2)
        {
            buffer[1] = incoming;
            fState = NOTHING;
            DEBUG_PUTS("I2C␣read");
            set_pwm(address);
        }
    }
}
```

Listing 4.3: void ssp_interupt()

Die Interrupt Routine wird gestartet, sobald der I^2C Empfänger ein Byte empfangen hat. Nach jedem Byte befindet sich der Empfänger im nächsten Zustand. Sobald das letzte Byte empfangen wurde, wird die

Funktion `set_pwm()` aufgerufen. Im Gegensatz zum Beispielprogramm werden die empfangen Bytes nicht im adressiertem Register abgelegt, sondern nur in den ersten beiden. Der Grund ist die Geschwindigkeit. Die Daten werden möglichst schnell gespeichert und erst später ausgewertet, um zu verhindern, dass ein Byte nicht verarbeitet werden konnte. Darum wurde auch die Debug-Funktion ausgeklammert.
Wird etwas vom zweiten PIC gelesen, wird die oberste Verzweigung benutzt. Sobald die Adresse des zu lesenden Register gesetzt wurde, schreibt der PIC dieses auf den I^2C-Bus. Dies gelang jedoch bisher nicht, obwohl genau der Code aus den Beispielen übernommen wurde.

4.3 Version mit RS232

Während der Semesterarbeit funktionierte die serielle Schnittstelle nur bedingt. Der minimale Abstand der Befehle musste mindestens 100ms betragen, weil der PIC mit einem Uhrenquarz betrieben wurde. Idealerweise wird ein 1 MHz Oszillator verwendet und somit kann eine höhere Baudrate erreicht werden. Das wertvolle am CCS-Compiler ist, dass er reklamiert falls man eine nicht funktionierende Baudrate wählt. Für die Kommunikation wurde möglichst ein Standartwert benutzt. Dazu eignete sich für 1 MHz Taktfrequenz eine Baudrate von 4800. Die maximale Rate wäre ca. 7800 Baud.

$$\text{Baud Rate} = \frac{F_{osc}}{64(N+1)} \qquad \text{für } N = 1 \ldots 255$$

Die Einstellung der Baudrate erfolgt in der Datei `p18web.c` in Zeile 202: `#define SER_BAUD 4800` und im File für den zweiten PIC gleich am Anfang des Files `#use rs232(baud=4800, xmit=PIN_C6, rcv=PIN_C7)`. In der Semesterarbeit benutzte man noch eine Verzögerung um einzelne Bytes über die serielle Schnittstelle zu senden. Diese ist nun überflüssig, da beide PIC gut synchronisieren. Bevor `#use rs232()` benutzt werden kann, muss noch `#use delay` definiert werden:

```
#use delay(clock=1000000)
#use rs232(baud=4800, xmit=PIN_C6, rcv=PIN_C7)
```

Die Schnittstelle wird immer mit der Baudrate, dem Sendepin und dem Empfangspin eingestellt. Mit `PUTC()` kann man nun einen Character über den seriellen Bus senden. Damit beim zweiten PIC keine Daten verloren gehen, werden die ankommenden Bytes mittels einer Interruptroutine in einen Zwischenspeicher geschrieben:

```
//Daten von der seriellen Schnittstelle lesen
#int_rda void serial_isr()
{
    int t;
    buffer[next_in]=getc();
    t=next_in;
    next_in=(next_in+1)%BUFFER_SIZE;
    if(next_in==next_out) next_in=t; // Buffer full !!
}
```

Listing 4.4: int_rda void serial_isr()

Um es später einfach zu machen, wird ein Bit verknüpft um zu Prüfen, ob Daten im Speicher sind:
`#define bkbhit (next_in!=next_out)`
Die Daten müssen auch aus dem Speicher gelesen werden. Dies macht die folgende Funktion:

```
byte bgetc()
{
    byte c;
    while(!bkbhit) ;
    c=buffer[next_out];
    next_out=(next_out+1)%BUFFER_SIZE;
    return(c);
}
```

Listing 4.5: byte bgetc()

Jetzt muss nur noch der Interrupt eingeschaltet werden:

```
enable_interrupts(global);
enable_interrupts(int_rda);
```

4.4 Ansteuerung der Servos

Abbildung 4.1 zeigt das Signal zum Ansteuern der Servos. Die Spannung beträgt +5V und die Periodendauer der Pulsweitenmodulation ist 20ms. Der Servo ist in der Neutralstellung, wenn ein Puls der Länge 1.5ms ansteht. Weicht der Puls um ±0.9ms ab, fährt der Servo um 90° nach links oder rechts. Die Periodendauer ist nicht extrem wichtig, sie kann abweichen. Falls sie jedoch zu kurz ist, schaltet der Servo nicht mehr richtig und wenn sie zu lang ist, verliert der Servo das Haltemoment und kann von Hand locker bewegt werden, bis wieder ein Steuerpuls kommt. Im Versuch zeigte sich, dass die Servos noch ohne Problem bei einer Periode von 16ms funktionieren. Dies ist der Fall, wenn der PIC mit einem 1MHz Oszillator betrieben wird.

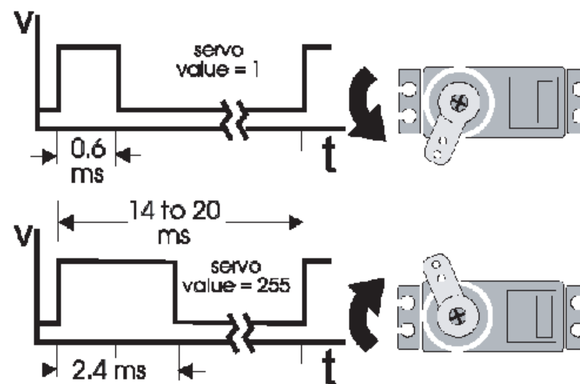


Abbildung 4.1: Ansteuerung der Servomotoren

4.5 Taktfrequenz und Pulsweitenmodulation bei der Version mit I^2C

Die Pulsweitenmodulation soll eine Frequenz von 50Hz haben und der I^2C-Bus verlangt einen schnellen Quarz. Die Wahl fiel auf einen 20Mhz Oszillator, weil dies die maximal mögliche Frequenz des PICs ist und weil dann beide PICs auf der gleichen Frequenz laufen. Doch die PWM Modulation verlangt eine Taktfrequenz von maximal 1Mhz, damit die 50 Hz ungefähr entstehen. Die Lösung dazu ist, dass der Timer2, der für die PWM Modulation verantwortlich ist, ausgeschaltet wird und der Timer1 benutzt wird um den Timer2 zu inkrementieren. Der Timer1 wird während des Interrupts neu gesetzt, damit die Periode der PWM 50Hz ergibt.

```
#INT_TIMER1      // This function is called every time
clock_isr() {
                 // the timer1 overflows (0xffff->0x0000).
    SET_TIMER1(0xff4f); //176
    TIMER_2++;
} main() {
    // *Initialisation*
    setup_timer_1( T1_INTERNAL | T1_DIV_BY_2);  // 1/20000000*4*176*2*256=70.4us
    enable_interrupts(INT_TIMER1);
    enable_interrupts(global);
    setup_ccp1(CCP_PWM);   // Configure CCP1 as a PWM
```

```
    // The timer2 (8bit) is disabled. Timer1 (16bit) count the timer2.
    // The overflow time ist about 20ms
    setup_timer_2(T2_DISABLED, 255, 1);
}
```

Listing 4.6: Taktfrequenz und Pulsweitenmodulation bei der Version mit I^2C

4.6 Taktfrequenz und Pulsweitenmodulation bei der Version mit RS232

Die Periode der Pulsweitenmodulation ist abhängig von der Taktfrequenz des PICs. Somit kann nicht jeder beliebige Quarz eingesetzt werden. Die folgende Rechnung zeigt, welche Quarze eingesetzt werden können:

$$\text{PWM Periode} = \frac{1}{f_{osc}}(\text{Periode} + 1) \cdot 4 \cdot \texttt{T2_DIV}$$

bzw.

$$f_{osc} = \frac{1}{\text{PWM Periode}} \cdot (\text{Periode} + 1) \cdot 4 \cdot \texttt{T2_DIV}$$

Periode: 0. . .255
`T2_DIV`: 1, 4, 16 (Prescaler)
Für die Periode von 20ms ergibt das folgende Resultate:

$$fosc = 1/0.02 \cdot (255 + 1) \cdot 4 \cdot 16 = 819.2kHz$$

Die Änderung der Pulsbreitenmodulation ist je nach Taktfrequenz unterschiedlich:

$$t = 1/f_{osc} \cdot \texttt{T2_DIV}$$

$$t = 1/800kHz \cdot 1 = 20\mu s$$

Bei 800kHz ist die Änderung 20μs. Dazwischen pendelt die Änderung hin und her. Der springende Punkt ist jeweils das Umschalten des Teilers `T2_DIV`. Anstelle des Uhrenquarzes, welcher in der Semesterarbeit benutzt wurde, wird ein Oszillator mit der Frequenz von 1MHz benutzt. Dies ergibt eine Pulsänderung von 16μs und eine schnellere Baudrate. Nachfolgend noch die Implementation:

```
setup_ccp2(CCP_PWM); // Configure CCP2 as a PWM setup_ccp1(CCP_PWM); // Configure CCP1 as a PWM
// The cycle time will be (1/clock)*4*t2div*(period+1)
// In this program clock=1000000  and period=255 (below)
// For the three possible selections the cycle time is:
// (1/1000000)*4*16*255 = 16.32 ms or 61 hz
setup_timer_2(T2_DIV_BY_16, 255, 1); value=77; // Neutralstellung set_pwm2_duty(value);
set_pwm1_duty(value);
```

Listing 4.7: Taktfrequenz und Pulsweitenmodulation bei der Version mit RS232

Zuerst wird der CPP Ausgang als PWM konfiguriert. Danach muss noch der TIMER2 eingestellt werden und `T2_DIV` ist der Teiler des Timers. Danach kommt der Wert, wo der Timer ein Overflow auslöst. Mit der letzten Zahl kann der Interrupt des Timers verzögert werden. Nun kann die Pulsdauer fortlaufend verändert werden.

4.7 Empfangen und weiterleiten der UDP Nachrichten

Die Steuerdaten werden via UDP-Packete dem ersten PIC gesendet. Dieser überprüft die Checksumme und leitet die Befehle weiter. Je nach Version ist dies der serielle Port oder der I^2C-Bus. In der Datei `p18_udp.c` wurde der Code so erweitert, dass alle Pakete, die auf den Port 3000 empfangen werden, direkt an die serielle Schnittstelle bzw. dem I^2C-Bus weitergeleitet werden.

```
/* Receive an incoming UDP datagram, return 0 if invalid */
BOOL udp_recv(void)
{
    WORD addr;
    int i=255;

    checkhi = checklo = 0;
    checkflag = udp_checkoff = 0;
    DEBUG_PUTC('u');
    DEBUG_PUTC('>');
    if(get_word(&remport) && get_word(&locport) && // Source & dest ports
        get_word(&udplen) && get_word(&ucsum) &&    // Dgram length, checksum
        udplen>=UDPHDR_LEN && udplen<=MAXUDP_DLEN){

        DEBUG_PUTC('>');
        if(ucsum){
            check_word(udplen);                  // Check pseudoheader
            check_dword(locip);
            check_dword(remip);
            check_byte(0);
            check_byte(PUDP);
            addr = rxout;
            skip_data(rxleft());
            setpos_rxout(addr);
        }                                        // If checksum OK
        if(ucsum==0 || (checkhi==0xff && checklo==0xff)){
                                                 // ..call UDP handler
            init_txbuff(0);
            checkhi = checklo = 0;               // Clear checksum
            checkflag = 0;
            DEBUG_PUTC('*');
            if(locport == ECHOPORT){            // Echo: return copy of data
                setpos_txin(UDPIPHDR_LEN);
                copy_rx_tx(udplen - UDPHDR_LEN);
                udp_xmit();
                DEBUG_PUTC('U');
            }
            else if(locport == DAYPORT){        // Daytime: return string
                print_lcd = print_serial = FALSE;
                print_net = TRUE;
                setpos_txin(UDPIPHDR_LEN);
                putstr(DAYMSG);
                udp_xmit();
            }
#if INCLUDE_DHCP
            else if(locport == DHCPCLIENT_PORT)    // DHCP client
                dhcp_handler();
#endif #if INCLUDE_TIME
            else if(locport == TIMECLIENT_PORT)    // Time client
                time_handler();
#endif

//new inclueded from r.guldener
            else if(locport == FROG_PORT){  // FROG client
                while(get_byte(&recived)){
                    serial_putch(recived);
                    if(recived=='G'){
                        print_lcd = print_serial = FALSE;
                        print_net = TRUE;
                        setpos_txin(UDPIPHDR_LEN);
                        while (!serial_kbhit() && (i-->=0)) {}
                        recived=serial_getch();
                        putch(recived);
                        udp_checkoff=TRUE;
                        udp_xmit();
                        udp_checkoff=FALSE;
                    }
                }
            }
            return(1);
        }
    }
    DEBUG_PUTC('!');
    return(0);
}
```

Listing 4.8: BOOL udp_recv(void)

Die empfangenen UDP Packete werden byteweise umgehend an den seriellen Port weitergeleitet. Jedes Paket wird überprüft, ob es ein G enthält. Ist dies der Fall, wird auf ein Byte des seriellen Ports gewartet, das den Spannungswert des Akkus beinhaltet. Dieses Byte wird sofort an den Sender des UDP Paketes zurückgesandt. Da die Checksumme nicht berechnet wird, muss diese natürlich ausgeschalten werden.
Falls der I^2C-Bus benutzt wird, sieht der Frog-Port folgendermassen aus:

```
        else if (locport == FROG_PORT)          // FROG client
        {
            BYTE recived;
            BYTE recived2;
            BYTE address=255;
            BYTE value=0;
            BOOL ok=TRUE;

            if (get_byte(&recived))
            {   if (recived=='V')
                {   address=0; puts("set address 0");
                }
                else if (recived=='H')
                {   address=2; puts("set address 2");
                }
                else if (recived=='G')
                {
                    print_lcd = print_serial = FALSE;
                    print_net = TRUE;
                    setpos_txin(UDPIPHDR_LEN);
                    putch(get_pic());
                    udp_checkoff=TRUE;
                    udp_xmit();
                    udp_checkoff=FALSE;
                }
            }
            if (get_byte(&recived))
            {   if (recived=='-')
                {   address++;
                    puts("set negative");
                }
                else if (recived=='+')
                    puts("set positive");
                else address=255;
            }

            if (get_byte(&recived))
            {   if ((recived<='9')&&(recived>='0'))
                {   puts("Z");
                    puts(recived);
                }
            }

            if (get_byte(&recived2))
            {   if ((recived2<='9')&&(recived2>='0'))
                {   puts("E");
                    puts(recived2);
                    write_pic(address, recived, recived2);
                }
            }


            while ( get_byte(&recived)) puts("loop");       //Eingangsbuffer leeren
        }
```

Listing 4.9: Frog-Port mit I^2C-Bus

Der PIC erwartet vier Bytes oder ein G. Die Daten werden bei dieser Funktion schon mal teilweise decodiert. Bei der ersten zwei Verzweigung wird die Adresse des Registers des zweiten PICs bestimmt. `0x00` steht für vorwärts, `0x01` für rückwärts, `0x02` für links und `0x03` für rechts. Das Register `0x04` ist für die Spannungsüberwachung. Bei den weiteren Verzweigungen wird der Wert überprüft und zwischengespeichert. Falls die letzte Verzweigung erreicht worden ist, werden die Werte über den I^2C-Bus gesendet.
In der ersten Verzweigung wird auch überprüft, ob ein G empfangen wurde. Falls dies der Fall ist, wird das Register `0x04` auf dem zweiten PIC gesetzt und der Wert dieses Registers ausgelesen. Danach wird der Wert ohne Checksumme via UDP zurückgesendet.

4.8 Hauptprogramm mittels I^2C-Bus

Im Hauptprogramm werden bei dieser Variante nur die Interrupts und die Timer gesetzt. Alle anderen Funktionen werden direkt aus den Interruptroutinen aufgerufen. Später versuchte man die Zuckungen der Servos zu eliminieren, was eine zusätzliche Verzweigung in der Endlosschleife ergab. Die I^2C Schnittstelle kann Fehler enthalten und läuft falsch. Darum wird ab und zu ein Softwarereset ausgelöst. Dass dabei nicht die Standartwerte bei der PWM gesetzt werden, wird beim Start des Prozessors unterschieden:

```
switch (restart_cause())        //after softwarereset, the speed and angel doesn't set
{
case WDT_TIMEOUT:
case NORMAL_POWER_UP:
    speed=255;
    value=54;
    set_pwm1_duty(value);       // This sets the time the pulse is
}                   // high each cycle.
                    // The high time will be:
                    //  if value is LONG INT:
                    //    value*(int_timer1)*t2div
                    // for example a value of 19 and t2div
                    // of 1 the high time is 0.57ms
                    // WARNING:  A value to high or low will
                    //           prevent the output from
                    //           changing.
```

In der *Application Note* steht, dass man die PWM nicht zwischen `0xef` und `0xff` setzen soll. Es könnte sein, dass der Puls nicht korrekt ist. Daher wurde der nachstehende Code eingeführt, welcher die PWM setzt.

```
while (true)
{
    if ((TIMER_2<0xf0)&&(TIMER_2>0xe0)&&update_pwm) // The PWM value must change
    {                                               // before the timer2 overflows
        set_pwm1_duty(duty);
        update_pwm=false;
    }
}
```

4.9 Hauptprogramm mittels RS232

Der PIC bekommt jeweils vier Bytes bzw. ein Byte, z.B. `H+12`, `V+18` oder `G` (für Get). Der erste Parameter ist für die Unterscheidung der Servos. `H` bedeutet Horizontal und meint den Steuerservo. `V` wäre für Vertikal und meint den Fahrtenregler. Der zweite Parameter bestimmt das Vorzeichen. Die letzten zwei Werte repräsentieren den Wert selbst. Somit können Werte von $\pm 90°$ für die Steuerung und ± 99

```
while(true)
{               //Die Empfangenen Daten werden auf das vorgegebene Format ueberprueft
                //und decodiert.
    if (bkbhit)         //Falls Daten im Empfangsbuffer
    {
        temp=bgetc();
        DEBUG_PUTC(temp);
        switch (recivecounter)
        {
        case 4:
            if ((temp=='H') || (temp=='V'))
            {
                axes=temp;
                //recivecounter=-1;
            }
            else if (temp=='G')     //Batteriespannung zurueckgeben
            {
                putc(adc8);
                recivecounter=0;
            }
            DEBUG_PUTC('4');
            break;
        case 3:
            if (temp=='+')
                oper = false;
```

```
                else if (temp=='-')
                    oper=true;
                else recivecounter=0;   //Empfangsfehler
                DEBUG_PUTC('3');
                break;
            case 2:
                if ((temp <= 57) && (temp >= 48))
                    value=(temp-48)*10;
                else recivecounter=0;   //Empfangsfehler
                DEBUG_PUTC('2');
                break;
            case 1:
                if ((temp <= 57) && (temp >= 48))
                {
                    value=value+(temp-48);
                        set_pwm(axes,value,oper);   //Empfang vollstaendig -> PWM setzen
                }
                else recivecounter=0;   //Empfangsfehler
                DEBUG_PUTC('1');
                break;
            }
            if (recivecounter <= 0) recivecounter=4;
            else if (--recivecounter==0) recivecounter=4;
        }
        else if (--adc_count<=0)    // Falls nichts empfangen wurde, wird nach
        {                           // einer kurzen Zeit der AD-Wandler ausgelesen
            ADC=READ_ADC();
            adc8=ADC>>2;        //von 10 bit auf 8 bit
            adc_count=0x1fff;
        }
    }
```

Das Programm läuft in einer Endlosschleife und sobald etwas neues im Speicher der seriellen Schnittstelle steht (`bkbhit=true`), wird das empfangene Byte untersucht. Damit der PIC weiss, welches Byte er als nächstes empfangen sollte, wird jeweils ein Zähler von vier herunter gezählt. Je nach Zählerstand wird das Byte nach dem Inhalt überprüft. Kommt mal etwas Unerwartetes an, wird der Zähler zurückgesetzt. Kommen alle Bytes an, wird die Funktion `set_pwm()` aufgerufen.

4.10 Die Funktion `set_pwm()`

Die Unterschiede der beiden Varianten sind nicht gross. Diese Funktion wird hauptsächlich benutzt um die empfangenen Daten zu decodieren und die Pulsweitenmodulation zu setzen. Bei der Variante mit dem I^2C-Bus wird nur die Richtung übergeben. Diese ist das Register, in das die Steuerwerte geschrieben werden. Als erstes wird der Watchdog-Timer zurückgesetzt (s. Abschnitt 4.11). Anschliessend wird überprüft, ob wahrheitsgetreue Werte in dem ersten zwei Bytes des Buffers stehen. Ist dies der Fall, werden die ASCII Werte umgewandelt. Die FOR-Schleife kann durch `wert=(bufer[0]-'0')*10+(buffer[1]-'0');` vereinfacht werden. Die Schleife ist entstanden, weil es Probleme gab bei der Weiterleitung im File `p18_udp.c`. Falls ein ungültiger Wert der Funktion übergeben wird, wird am Schluss ein Softwarereset ausgeführt, um den I^2C Empfänger wieder in einen gültigen Zustand zu setzen. Je nach Richtung wird in der Switch-Anweisung der decodierte Steuerwert in die PWM-Information umgewandelt. Dies geschieht über eine Lookup-Table. In diesem Programmteil ist noch die Steuerung für den Schrittmotor geschrieben. Für die Geschwindigkeit des Motors werden daher die Steuerdaten nur zwischengespeichert. Die Funktion für den Schrittmotor holt dann die Werte von diesem Speicherplatz. Für die Steuerung werden die Daten auch zwischengespeichert, nur im Unterschied, dass dem Hauptprogramm noch mitgeteilt wird, dass sich die Daten geändert haben (`update_pwm()`).

```
//setzen der PWM Pulsweite
void set_pwm(DIRECTION fdirection) {   int wert;
    int i,j;
    DEBUG_PUTS("set_pwm");
    restart_wdt();

    if ((buffer[0]<='9')&&(buffer[0]>='0')&&(buffer[1]<='9')&&(buffer[1]>='0'))
    {
        DEBUG_PUTC(buffer[0]);
        DEBUG_PUTC(buffer[1]);
        for (i=0;i<10;i++)      //Decoder (geht auch viel einfacher!!!)
```

```
            if (C[i]==buffer[0])
                for (j=0;j<10;j++)
                    if (C[j]==buffer[1])
                    wert=i*10+j;

        switch (fdirection) //set the the speed or angel
        {
            case FORWARD:
                speed = V[wert];
                ahead=true;
                DEBUG_PUTS("FORWARD");  //debug
                DEBUG_PUTC(speed);  //debug
                break;
            case BACKWARD:
                speed = V[wert];
                ahead=false;
                DEBUG_PUTS("BACKWARD"); //debug
                DEBUG_PUTC(speed);  //debug
                break;
            case LEFT:
                DEBUG_PUTC(wert);   //debug
                wert+=90;
                duty=H[wert];
                update_pwm=true;
                DEBUG_PUTS("LEFT"); //debug
                DEBUG_PUTC((byte)duty); //debug
                break;
            case RIGHT:
                putc(wert);
                wert=90-wert;
                duty=H[wert];
                update_pwm=true;
                DEBUG_PUTS("RIGHT");    //debug
                DEBUG_PUTC((byte)duty); //debug
                break;
        }
    }
    else
    {
#if DEBUG       //debug
        DEBUG_PUTS("ERROR␣I2C");
        DELAY_MS(10);
#endif      //reset cpu
        #asm
        goto 0x00
        #endasm
    }
}
```

Listing 4.10: void set_pwm(DIRECTION fdirection)

Bei der Variante mit der seriellen Schnittstelle wird in der Interruptroutine schon der grösste Teil decodiert. Sind die Daten komplett, wird die Funktion `set_pwm()` mit den drei Steuerbefehlen (Achse, Wert und Operationszeichen) aufgerufen. Als erstes wird der Watchdog-Timer zurückgesetzt (s. Abschnitt 4.11). Je nach Achse werden die Steuerwerte aus einer anderen Lookup-Table geholt. `H` ist für die Lenkung (Horizontal) und `V` ist für die Geschwindigkeit (Vertikal). Danach wird sofort die Pulsweitenmodulation gesetzt. Ein Zucken der Servos ist nicht aufgetaucht, daher wurde auch auf die Implementation der oben erwähnten *Application Note* verzichtet.

```
//setzen der PWM Pulsweite
void set_pwm(byte axes, long value, boolean oper) {
    long wert;
    restart_wdt();
    //if (value>30) value=30;   //Sicherheitsschaltung
    switch (axes)
    {
    case 'H': if (oper) value=value+91; else value=91-value;
        wert=H[value];
        set_pwm1_duty(wert);
        break;
    case 'V': if (!oper) value=value+100; else value=100-value;
        wert=V[value];
        set_pwm2_duty(wert);
        break;
    }
}
```

Listing 4.11: void set_pwm(byte axes, long value, boolean oper)

Die Servos arbeiten mit einem Dreharm, damit ist die Steuerung nicht linear. Die Bewegungen bei 0° Ausschwenkung sind grösser als bei 30°. Um dies wieder zu linearisieren, wurde die Lookup-Table mit einer Sinusfunktion angeglichen (s. Abbildung 4.2).

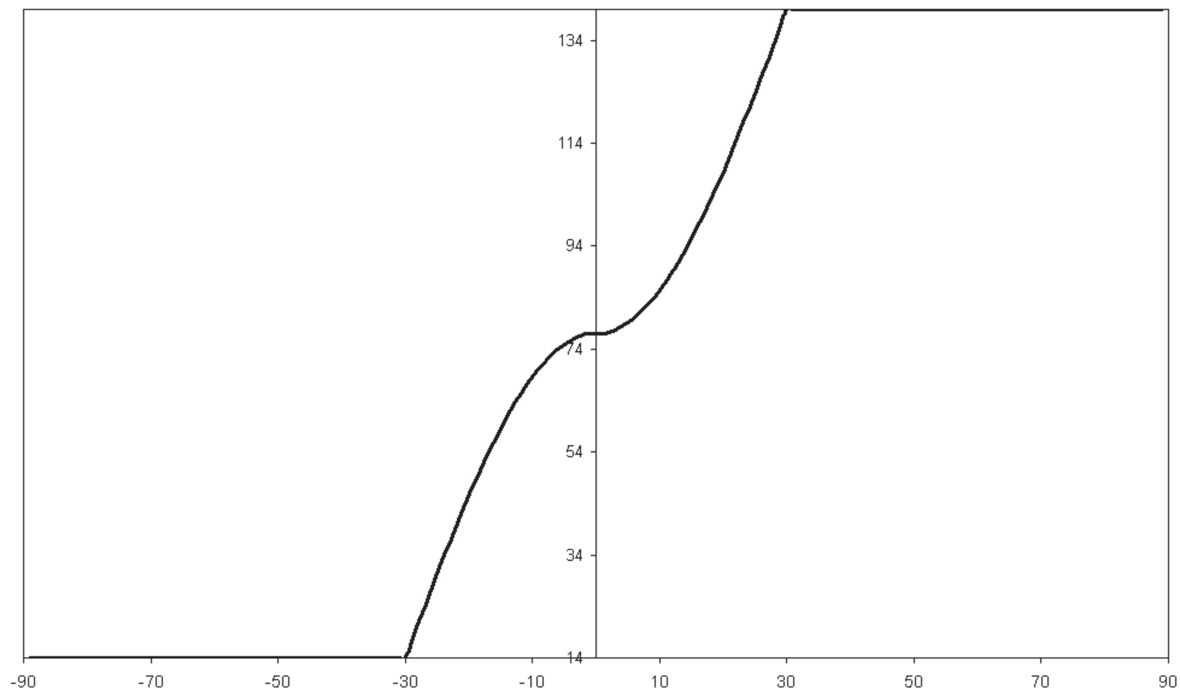


Abbildung 4.2: Linearisierung der Steuerung

4.11 Automatische Abschaltung mittels Watchdog-Timer

Da das Auto viel Elektronik mit sich führt, sind dementsprechend viele Störungsquellen vorhanden. Eine trickreiche Störungsquelle ist, wenn der Datenstrom abbricht und das Auto keine Befehle mehr erhält. Entweder ist der PC abgestürzt oder die Akkus sind leer. Der Watchdog-Timer sorgt in diesem Fall dafür, dass der PIC immer wieder zurückgesetzt wird und beim Neustart die Servos wieder in die Neutralstellung zurückgesetzt werden. Natürlich ist der Reset nach einem erfolgreichen Empfangen eines Datenpaketes nicht erwünscht. Darum wird in der Funktion `set_pwm()` der Watchdog-Timer immer wieder zurückgesetzt. Mit `setup_counters` kann der Watchdog-Timer initialisiert werden. Hier wird der Timer nach 1152ms einen Reset durchführen:

```
setup_counters(RTCC_INTERNAL,WDT_1152MS);
```

Mit `restart_wdt()` wird der Zähler des Watchdog-Timers zurückgesetzt:

```
restart_wdt();
```

Bei den Testfahrten wurde festgestellt, dass als erstes Modul die Kamera aussteigt und dann keine Bilder mehr sendet.

4.12 Die Verbindung der zwei PICs

Bei der Variante mit der seriellen Schnittstelle wird der zweite PIC für die Steuerung der Servos mit dem RS232 Stecker des Boards verbunden. Zu beachten ist, das der IC MAX232 auf dem Board des PIC18 entfernt und mit Drahtbrücken ersetzt wird. Die Drahtbrücken werden wie folgt gesetzt: P7-P10, P8-P9, P11-P14 und P12-P13. Bei der Version mit dem I^2C-Bus muss die Leitung SCL und SDA zwischen den beiden PICs verbunden werden.

4.13 Die Batteriespannungsüberwachung

Da der PIC auch analoge Eingänge besitzt, liegt es nahe, dass man die Batteriespannung überwacht. Bei unserem Auto werden Nickelmetallhybrid Akkus mit einer Spannung von 7,2 Volt benutzt. Das ist einiges höher, als die erlaubte Eingangsspannung des PICs. Daher wird die Spannung mit zwei 100kΩ Wiederständen halbiert. Wegen den grossen Einschaltströmen des Fahrtenreglers wird zusätzlich noch ein Kondensator dazugeschaltet, um diese Spannungsschwankungen herauszufiltern. Der AD-Wandler liefert ein 10bit Wert. Unsere Applikation läuft jedoch mit 8bit, was heisst, dass zwei Bytes über den seriellen Port gesendet werden sollten. Dieser höhere Transferaufwand war nicht wünschenswert und daher würde das Resultat des AD-Wandlers auf 8bit gekürzt.

4.14 Der Schrittmotor

Anfangs der Arbeit war man noch der Überzeugung, dass ein Schrittmotor die richtige Lösung für den Antrieb ist. Der Code dazu ist auch einfach. Das Beispielprogramm, das mit dem Compiler mitgeliefert wurde, musste nur in das bestehende Programm integriert werden:

```
//byte const POSITIONS[4] = {0b01010,   //Fullstep
//              0b10010,
//              0b10100,
//              0b01100};
byte const POSITIONS[8] =   {0b01010,       //Halfstep
                0b00010,
                0b10010,
                0b10000,
                0b10100,
                0b00100,
                0b01100,
                0b01000};

long int_count;     //Number of interrupts left before change step

#INT_TIMER1     // This function is called every time clock_isr() {           // the timer1
overflows (0xffff->0x0000).
    static byte stepper_state = 0;
    SET_TIMER1(0xff4f);
    if (speed==255)
        port_a =0x0;    //stop motor
    else
    {
        if(--int_count==0)
        {
            port_a = POSITIONS[ stepper_state ];
            if(ahead)
                stepper_state=(stepper_state+1)&(sizeof(POSITIONS)-1);  //left
            else
                stepper_state=(stepper_state-1)&(sizeof(POSITIONS)-1);  //right
            int_count=speed+20;
        }
    }
}
```

Nach jedem Interrupt des Timer1 wird der `int_counter` dekrementiert bis er Null ist. Dann wird der Ausgang des Schrittmotors je nach Drehrichtung geändert. Mit dem Setzen des Timer1 nach jedem Interrupt auf einen bestimmen Wert, kann man genau die maximale Frequenz des Schrittmotors einstellen. Damit bei kleinen Werten von `speed` nicht die Frequenz verdoppelt wird, ist die Addition von `20` notwendig. Ist der Wert von `speed 255`, so wird der Motor gestoppt, indem alle Ausgänge auf den gleichen Pegel gesetzt werden.

4.15 Endstufe für den Schrittmotor

Der Schrittmotor konsumiert pro Phase ca. 300mA. Somit musste ein Verstärker konstruiert werden. Das Prinzip der Endstufe wurde von einem Schrittmotortreiber abgeschaut. Die Dioden bewirken einen Einschaltoffset, so dass kein Kurzschluss entsteht. Im Betrieb zeigte sich, dass bei niedrigen Batteriestand der Strom abnimmt und bei einer Batteriespannung von über 8,2V ein leichter Kurzschluss entsteht. Daher ist sie nicht ideal für unsere Anwendung, weil die Batterie im frisch geladenen Zustand über 8,2V beträgt.

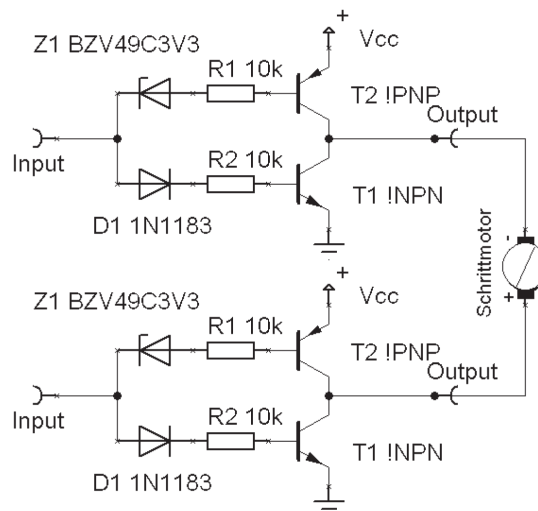


Abbildung 4.3: Ansteuerung einer Phase beim Schrittmotor

4.16 Konfiguration

4.16.1 Chipweb Board

Nachdem der PIC neu programmiert wurde, muss er auch wieder neu konfiguriert werden. Dazu muss ein PC mit einem Terminalprogramm an die serielle Schnittstelle des Boardes angeschlossen werden. Als Terminalprogramm eignet sich besonders gut *RealTerm*. Der Port wird folgendermassen konfiguriert:

- 4800 Baud
- none parity
- 8 data bits
- 1 stop bit
- non flow control

Wenn nun der PIC gestartet wird, fragt dieser, wie man diesen konfigurieren will:

P18Web V2.17
Config
Serial num?
Die Serienummer befindet sich auf dem Board auf dem Netzwerkchip. Bei unserem Board ist diese 79.

IP addr?
Jetzt kann die IP Adresse der Wirelesskarte gewählt werden. Bei uns ist diese 192.168.10.20.

Infra or Adhoc?
Wird das Auto über einen Accesspoint betrieben, muss hier Infra gewählt werden. Wir arbeiten jedoch nur mit Cardbus Karten und arbeiten peer-to-peer. Somit müssen wir Adhoc wählen.

Channel num (1-13)?
Nun muss noch der Kanal der Wirelesskarte konfiguriert werden. Bei unserem Projekt ist dieser 11.

Die Karte kann jederzeit neu konfiguriert werden, indem man beim Starten des PICs den Taster auf dem Board drückt.

4.16.2 Wirelesskamera

Die Kamera hat ein Webinterface, wo sämtliche Einstellungen geändert werden können. Allerdings benötigt man dazu ein Benutzername und ein Passwort. Der Benutzername lautet `Admin` und das Feld für diePassworteingabe wird leer gelassen. Bei unserem Projekt wird die Kamera folgendermassen konfiguriert:

- IP: 192.168.10.200
- Subnet: 255.255.255.0
- SSID: non-spec
- Channel: 11
- Mode: Adhoc

4.16.3 Wirelesskarte beim PC

Zu beachten ist, dass die Wirelesskarten des PC sowie die Wirelesskamera im gleichen Subnet wie die Wirelesskarte des Chipweb-Boardes ist. Es dürfen auch nicht mehrere gleiche IP-Adressen vorhanden sein. Für die PC's benutzen wir IP-Adressen zischen 192.168.10.1 . . . 10. Ansonsten sind es auch wieder die gleichen Parameter:

- Subnet: 255.255.255.0
- SSID: non-spec
- Channel: 11
- Mode: Adhoc

5 Bildsensoren anderer Hersteller

5.1 CCD-CMOS Sensor

Zu Beginn der Arbeit war es ein zwingender Punkt der Aufgabenstellung, dass nur eine Spannungsversorgung benutzt werden darf. Die erste Überlegung war natürlich *Strom sparen!*. Der grösste Verbraucher ist die Wirelesskamera, die verbraucht etwa ein Ampere. Es liegt nahe, dass man nun versucht diese Kamera zu ersetzen. Was kann in diesem Fall in Frage kommen? Die Schnittstelle zwischen Auto und PC ist ja eigentlich mit dem ChipWeb Board schon vorhanden. Die Wirelesskarte auf dem Board könnte noch dazu benutzt werden um Bilder zu versenden, hiermit käme ein Bildsensor in Frage. Nach einer intensiven Suche auf dem Internet wurde von *National Semiconductors* ein geeigneter CCD Image Sensor mit VGA Auflösung und 30 Frames pro Sekunde gefunden. Der Chip kann direkt auf einem Headboard (LM9617HEDBOARD) bezogen werden, so dass die Verdrahtung wesentlich vereinfacht wird.

5.2 CCD-Sensor von National

Der s/w Sensor kann via I^2C-Bus konfiguriert werden und besitzt ausserdem noch einen Snapshot Eingang. Damit keine Daten verloren gehen, wird zwischen Sensor und Wirelesskarte ein FIFO geschaltet. Der PIC steuert nur den Datenfluss und muss keine Werte zwischenspeichern.

Das Datenblatt des Sensors ist sehr ausführlich. Um den Sensor zu programmieren, muss die Adresse

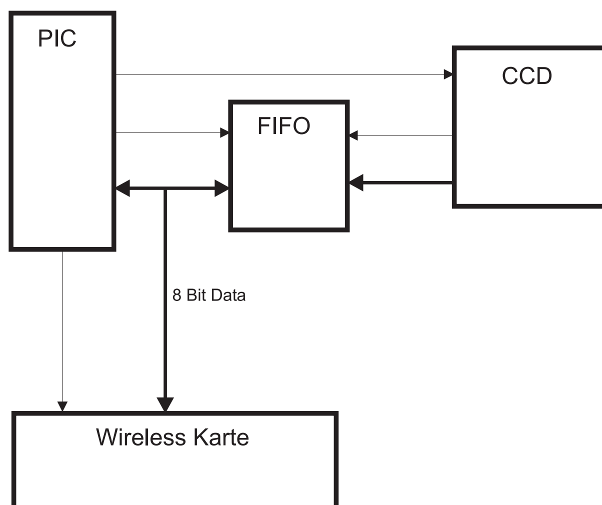


Abbildung 5.1: Einbindung des CCD-Sensors in das bestehende Projekt

des Chips bekannt sein. Der Sensor kann hardwaremässig zwei Adressen besitzen. Wenn der Pin `sadr` auf Masse ist, besitzt der Chip die Adresse 1010101 oder wenn `sadr` auf Vcc ist, 0110011. Ab Seite 20 des Datenblattes wird exakt beschrieben, wie die Synchronisationssignale eingestellt werden können. Für unsere Applikation ist es wichtig, dass das Signal `pclk` nur während den gültigen Daten wechselt. Somit muss `pclk` in *data ready mode* gesetzt werden. Je nach FIFO Eingang muss dieser Clock noch invertiert werden. Jetzt ist der Sensor eigentlich schon bereit, um den Zwischenspeicher zu füllen. Aber da der Sensor eine Auflösung von VGA (640x480) hat, fällt eine grosse Datenmenge pro Bild an. Der Sensor hat

die Möglichkeit, dass nur jeder zweite Pixel verarbeitet wird und somit eine Auflösung von 320x240 Pixel entsteht. Ab Seite 25 des Datenblattes sind sämtliche Register des Sensor aufgelistet. Nachfolgend eine Liste, welche Änderungen der Register vorgenommen werden muss:

- Register 02
  Bit 3 set (Snapshot enable)
- Register 03
  Bit 3 set (Horizontal subsampling)
  Bit 4 set (Vertical subsampling)
- Register 06
  Bit 7:6 10 set to 8bit mode
- Register 07
  Bit 7 set `pclk` in *data ready mode*
  Bit 4 set active edge of pixel the clock to negative
  Bit0 Assert to tri-state all output signals

Weiter wird das sogenannte Windowing in den Registern 0B bis 11 eingestellt. Mit diesen Optionen kann man den Bereich des Sensors verändern und die Pixelzahl verkleinern oder vergrössern.

5.3 Die Software

Die Software zur Steuerung des Datenflusses ist schon vom Herausgeber des Buches TCP/IP LEAN[1] schon geschrieben worden. Somit muss der Code nur noch auf den Sensor angepasst werden. Die Funktionen für das Senden der Bilder befinden sich in der Datei `p16_cap.c`. Interessant ist die Funktion `capnic_data()`. In dieser Funktion wird der Datenstrom so geschaltet, dass dieser direkt vom Sensor bzw. FIFO in die Wirelesskarte geschrieben wird. Da das FIFO kein Outputenable besitzt, muss hier darauf geachtet werden, dass am Schluss der Pin `CAP_RE=1` ist. So wird der Ausgang des FIFO in den Tri-State geschaltet. Am Anfang des gleichen Files sind alle Definitionen, welche je nach Pin und Steuerleitung angepasst werden müssen.
Zum Sourcecode des PICs ist auch ein PC Programm vorhanden, welches Bilder vom Sensor über den PIC holt. Wenn man mit Ethereal die Netztwerkkarte überwacht, sieht man, dass dieses Programm dem PIC ein UDP Packet auf den Port 1502 sendet. Dies ist der Port, welcher als Videoport deklariert ist und somit der PIC die Videofunktionen aufruft. Im File `p16_udp.c` ist der Videoport deklariert. Er kann eins zu eins in das Projektfile `p18_udp.c` übernommen werden. Im Videoport wird als erste die Funktion `vid_handler()` ausgeführt. Diese übernimmt den ganzen Prozess der Bildübertragung.
Ein Ethernetframe hat eine MTU[2] von 1500 Bytes. Das ergibt abzüglich der Headergrössen der Netzwerkschicht (20 Bytes) und der Transportschicht (in unserem Fall UDP 8 Bytes) noch 1472 Bytes zur freien Verfügung. Unser Bild hat jedoch 320x240 Bytes was 76800 Bytes ergibt. Somit müssten 53 Pakete versendet werden, bis ein komplettes Bild übertragen wurde. Die Aufsplittung des Frames ist schon in den Funktionen enthalten. Im Sourcecode ist auch eine Funktion enthalten, welche es ermöglicht nicht erhaltene Pakete nochmals zu senden. Dies erfordert jedoch ein Feldmemory. Da wir aber mit einem FIFO arbeiten sind die Daten nur einmal zugänglich und sind dann gelöscht. Erneutes Senden eines verloren Packetes macht bei unserer Applikation auch keinen Sinn, denn dies würde den ganzen Prozess nur noch mehr verzögern.

5.4 Die Geschwindigkeit

Die Geschwindigkeit ist auch sehr wichtig, damit die Bilder möglichst schnell ausgewertet werden können. Der Datentransfer liegt im Moment bei 5Mbit, falls ein Oszillator mit einer Frequenz von 20MHz benutzt

[1] s. [2]
[2] Maximum Transmission Unit, s. RFC 1191 oder [4] S. 29f.

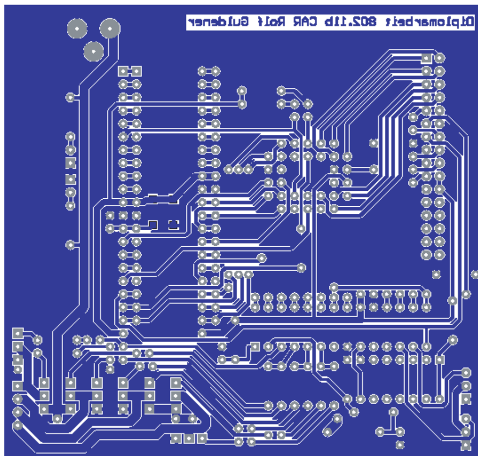
wird. Die Geschwindigkeit des PICs kann noch bis 40MHz erhöht werden. Zu beachten ist nur, gemäss Mail mit Herr Jeremy Bentham (Autor von TCP/IP LEAN), dass die Verzögerungen für die Netzwerkadapter angepasst werden. Somit könnte ein Durchsatz von 10Mbit erreicht werden. In der Praxis zeigt sich jedoch, dass mit einem 11Mbit Wirelessnetzwerk eine maximale Durchsatzrate von ca. 7Mbit erreicht werden kann. Somit müsste auch die Karte gegen eine schnellere ausgetauscht werden, damit der Frequenzwechsel Sinn macht.
Mit dem aktuellen Clock von 20MHz vergehen mindestens 1/5000000*76800=15,36ms plus Headerinformationen. Dazu kommt noch die Verzögerung, bis das erste Byte vom Sensor bereit steht und die Verarbeitungszeit im PIC.
Zu beachten ist auch, dass der CCD Sensor mit der gleichen Geschwindigkeit das FIFO füllt, wie der PIC die Daten herauslesen kann, damit es zu keinem Overflow kommt. Somit muss je nach dem der Quarz auf dem Headboard ersetzt werden.

5.5 CCD-Sensor von Omnivision

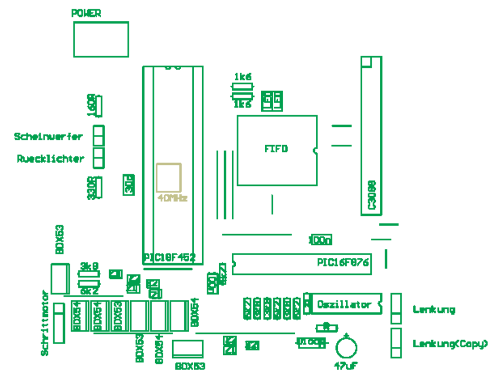
Der Lieferant bestätigte am Anfang der Arbeit, dass sie den Sensor von *National* innert zwei Wochen liefern kann. Nach zwei Wochen kam jedoch die Nachricht, dass der Sensor erst in Woche 50, also in der letzten Woche der Diplomarbeit, geliefert wird. Somit musste ein anderer Sensor gesucht werden um das Ziel zu erreichen. Durch Mitdiplomanden wurde in Erfahrung gebracht, dass in den Semesterarbeiten des Faches Embedded Systems solche Sensoren verwendet wurden. Prof. Brändle stellte seinerseits den Omnivision Sensor C3088A zur Verfügung. Leider war dieses Datenblatt nicht so übersichtlich wie dasjenige von National und daher konnte in diesem Fall keine befriedigende Lösung gefunden werden (Zeitdruck!). Nach mehrmaligen Durchlesen des Datenblattes konnte man sich vage ein Bild davon machen, wie der Sensor funktionierte. Der grosse Nachteil ist, dass der Chip keinen Snapshot hat. Somit läuft der Chip immer weiter und liefert Bilder. Eine Möglichkeit besteht darin, dass der Chip nach dem ersten Bild in den Power Save Mode gesetzt wird. Somit werden die Register nicht gelöscht und zurückgesetzt, was bei einem Reset der Fall wäre. Der Vorteil hingegen ist, dass der Chip mit 5V läuft und keine Spannungsanpassung gemacht werden muss. Ansonsten ist der Chip ähnlich dem von National. Zu Beginn müssen zuerst auch verschiedene Register gesetzt werden:

- Register 11
  Bit 0-5 clock prexcaler
- Register 15
  Bit 6 pclk polarity
- Register 16
  Bit 2-7 field interval selection
- Register 17-1A
  Windowing
- Register 39
  Bit 6 pclk output timing selection
  Bit 2 tristate

Zeitdruck. Daher wurde nur noch das Layout gezeichnet. Die Steuerleitungen des Chips wurden nicht verdrahtet, sondern nur mit einem Lötpunkt versehen, damit man später die Möglichkeit hat die Signalpfade zu ändern. In diesem Layout wurde der zweite PWM Ausgang des zweiten PICs wieder verdrahtet, da der Schrittmotor nicht wunschgemäss funktionierte und nun wieder der Fahrtenregler angesteuert wird. Die Treiberstufe des Schrittmotors wurde nicht entfernt, falls später trotzdem noch ein Schrittmotor eingesetzt wird. Zusätzlich ist noch die Spannungsüberwachung eingefügt. Jedoch wird die Diode nicht benötigt.



(a) Layout



(b) Bestückungsplan

Literaturverzeichnis

[1] Balczarczyk T., Guldener R.: Automatische Autofernsteuerung mittels 802.11b und Bildverarbeitung auf einem PC. Semesterarbeit SS03. Hochschule Rapperswil 11.Juli 2003.

[2] Bentham J.: TCP/IP Lean. Web Servers for Embedded Systems, Second Edition. CMP Books Lawrence, Kansas 66046.

[3] - : C Compiler Reference Manual. Custom Computer Services Inc. September 2002.

[4] Stevens W. Richard: TCP/IP Illustrated Volume 1, The Protocols. Addison-Wesley 2000.

A Installation der Programme

A.1 ActiveX Komponente und Treiber für die Kamera installieren

Im Verzeichnis *D-Link* auf der CD befinden sich zwei Dateien:

1. dcs1000W_ipview_app_346.exe
2. dcs1000W_xplugctrl_app_346.exe

Diese beiden Dateien müssen ausgeführt werden, damit werden Treiber für die Kamera installiert. Vorsicht ist geboten bei erneuter Installation der zweiten Datei. Bei erneutem Ausführen wird die ActiveX-Komponente deinstalliert, danach müsste man die zweite Datei nochmals ausführen, um die ActiveX-Komponente wieder zu installieren.

A.2 Bildverarbeitung

Das Programm muss vorgängig noch kompiliert, linked und erstellt werden. Dazu muss zuerst die Projektdatei dekomprimiert werden. Danach wird MS Visual C++ gestartet und das Projekt geladen. Nun kann man das ausführbare Programm erstellen.
No einmal zusammengefasst:

1. Im Verzeichnis *Bildverarbeitung Software* auf der CD die Datei *Version 1.0 80211 Car* dekomprimieren.
2. MS Visual C++ starten und den Workspace *80211Car.dsw* öffnen.
3. Lauffähiges Programm kann nun erstellt werden.

B Die Control-Unit des Autos

Dieses Programm entstand eigentlich zuerst nur zur Testzwecken. Es wurde so konzipiert, dass einzelne Werte sowie auch ein Stream dem Auto gesendet werden kann.
Oben links befinden sich die beiden Schaltflächen **Send**. Wird auf die Taste gedrückt, sendet das Pro-

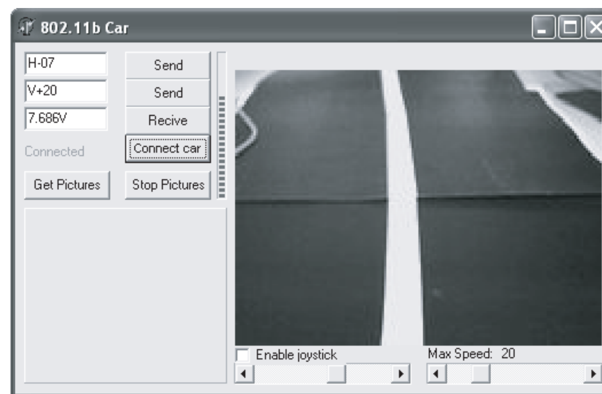


Abbildung B.1: Oberfläche der Control-Unit

gramm den zugehörigen Text aus dem Feld links der Schaltfläche.
Mit der Schaltfläche **Recive** holt das Programm die Batteriespannung von dem Auto und zeigt diese links in Volt an und rechts im Statusbalken.
Mit der Schaltfläche **Connect car** wird der Stream mit den Steuerbefehlen für das Auto eingeschalten.
Mit **Get Pictures** und **Stop Pictures** wird die Kamera eingeschaltet bzw. ausgeschaltet.
Falls ein Joystick in der Systemsteuerung des Betriebssystem konfiguriert wurde, kann mit **Enable joystick** dieser eingeschaltet werden.
Da das Auto etwas heikel ist mit der Geschwindigkeit, kann die **maximale Geschwindigkeit** mit dem Slider unten rechts begrenz werden.
Das Auto kann nicht nur mit dem Joystick **gesteuert** werden, sonder auch mit der Maus. Dies funktioniert mit dem Panel unten links. Fährt man mit dem Mauszeiger nach oben auf dem Panel, so wird das Auto nach vorne beschleunigt. Analog passiert dies mit links, rechts und unten.

Sämtliche Funktionen sind *event*-gesteuert. `Connect car` schaltet die beiden Timer ein und aus. Der Timer 1 löst alle 25ms die Funktion `timer1time()` aus, welche abwechslungsweise die Lenkungswerte und Geschwindigkeitswerte sendet. Der Timer2 löst ein Klicken der Schaltfläche Recive aus, welcher die aktuelle Batteriespannung holt.
Die Batteriespannung wird jedoch erst im Event des UDP-Recivers verarbeitet. Sobald der PC ein UDP Packet auf den Port 3000 bekommt, wandelt er das empfangene Byte in den Spannungswert um und zeigt diesen an. Der Joystick ist eine Freeware Komponente, welche die Daten des Joystick aus der Systemsteuerung des Betriebssystems holt. Diese muss jeweils aktiviert werden. Das geschieht beim Programmstart oder beim anklicken der Checkbox.