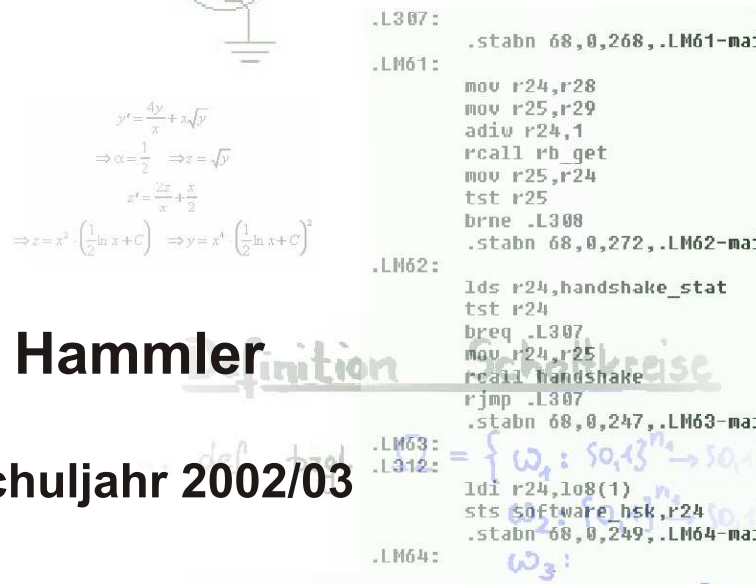
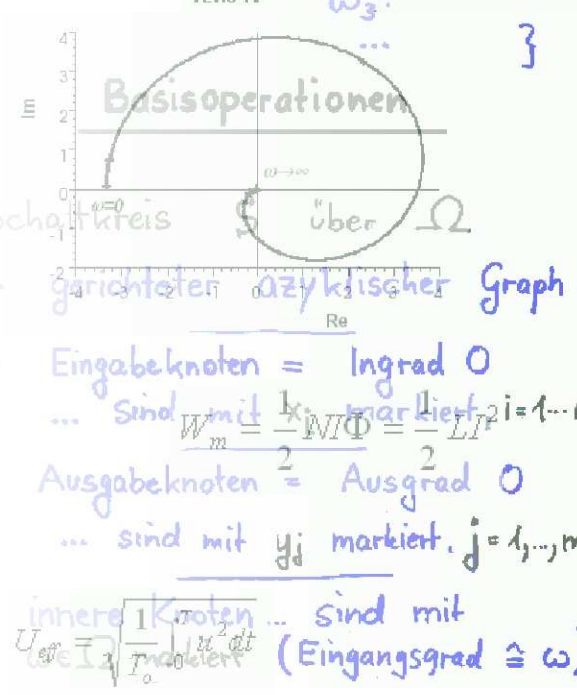
Palm PrintAdapter

Fachbereichsarbeit Informatik

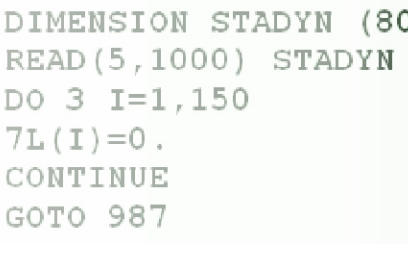


Nikolaus Hammler

8. B Klasse, Schuljahr 2002/03



Sacré Coeur Graz



Palm PrintAdapter

Fachbereichsarbeit Informatik

Nikolaus Hammler

8.B Klasse, Schuljahr 2002/03

Sacré Coeur Graz

Mein besonderer Dank gilt Herrn Prof. Mag. Franz Schmid für die freundliche Unterstützung und fachliche Begleitung bei der Erstellung der Fachbereichsarbeit.

Inhalt

1. Vorwort

Letztes Jahr zu Weihnachten bekam ich einen Palm Handheld (PDA) geschenkt. Bei diesem Gerät handelte es sich nicht um das leistungsfähigste Modell, schließlich sind diese kleinen Geräte sehr teuer.

Durch meinen Drang nach zusätzlicher Hardware, eigenen Basteleien und einfach neuen „Features“ auf einem Gerät kam ich auf so manch komplizierte Idee. Das war schon bei meinem TI92+ Taschenrechner so, dem ich über ein Ethernet Interface Zugang zum Internet gewähren wollte; das Projekt „TiSock“ steht zur Zeit noch in Entwicklung.

Warum ich draufgekommen bin, unbedingt so einen PrintAdapter bauen zu müssen, ist schnell erklärt: Es war weniger der praktische Nutzen den ich darauf zielen wollte, Texte und Bilder von meinem Palm aus ausdrucken zu können sondern einfach der Drang danach, dem Palm mehr (mobile) Funktionalität zu spendieren. Als ich irgendwann im Frühling letzten Jahres auf der quasi „Palm-Hauptseite“ www.palmgear.com durch die Gegend surfte um neue nützliche oder weniger nützliche Palm Software ausfindig zu machen, stieß ich auf das Programm „PalmPrint“. Dieses Programm suggerierte mir in seiner Beschreibung, es könne Dokumente auf dem Palm direkt an einen Drucker schicken und ausdrucken. Aufgeregt las ich natürlich sofort weiter, mit dem Gedanken wie das überhaupt funktionieren sollte, denn der Palm bot ja nur eine einfache serielle RS232 Schnittstelle; Drucker jedoch nutzen die parallele oder Centronics Schnittstelle.

Doch schon bald machte sich Ernüchterung breit: „PalmPrint“ unterstütze natürlich nur Drucker die über einen IrDA-Port verfügen, mit dem man

sozusagen per Infrarot „kabellos“ drucken kann. Doch Drucker, die über diese Schnittstelle verfügen sind in der Regel sehr teuer und selten zu finden. Doch auf der Homepage[1] des besagten Programmes wurde auf andere Möglichkeiten aufmerksam gemacht, den Drucker an den Palm anschließen zu können. Das wäre einerseits ein „IrDA zu Parallelkonverter“, andererseits ein „Seriell zu Parallelkonverter“. Die Preise von über 60 € ließen mich sofort zu dem Gedanken kommen, ob man so etwas nicht auch selbst bauen könne.

Graz, im Februar 2003 Nikolaus Hammler

[1] http://www.stevenscreek.com/palm/palmprint.shtml

2. Idee und Nutzen

Ich arbeitete schon seit längerer Zeit mit Atmel Mikrocontrollern und wollte die Erfahrung daraus für die Lösung dieses Problems nutzen. Doch im Internet waren keine Projekte zu finden, die meinen Anforderungen entsprachen. Es gab zwar einige Konverter, diese basierten aber auf anderen Mikrocontrollern, die nicht so „elegant“ programmiert werden konnten wie das bei Atmel der Fall war. Oft brauchte man sogar noch zusätzliche ICs wie Buskontroller und externen Speicher, die einen mobilen Einsatz unmöglich machten.

So beschloss ich, mit Hilfe eines Atmel Mikrocontrollers meinen eigenen Palm PrintAdapter zu bauen. Dieser Adapter sollte lediglich fähig sein, vom Palm aus drucken zu können. Er musste also nicht in der Lage sein, seine Baudrate umzustellen oder von parallel zu seriell zu konvertieren. Die Atmel Mikrocontroller schienen dafür wie geschaffen:

- Dank des eingebauten UART hatte ich sofort einen Anschluss an die RS232 Schnittstelle des Palm.
- Dank des eingebauten Programm und Arbeitsspeichers war nur der Mikrocontroller und nicht weitere externe Komponenten nötig
- Dank der „ISP“ Unterstützung der 90(L)Sxxxx-Reihe musste kein externes und teures Programmiergerät angeschafft werden
- Dank meiner bisherigen Erfahrung mit der 90er Atmel Reihe hatte ich am Anfang keine Schwierigkeiten, den Grund-Schaltkreis aufzubauen.
- Dank der TTL unterstützten I/O Pins und den integrierten Pullup-Widerständen konnte ich den Parallelport direkt mit dem Mikrocontroller verbinden.

So war der Palm PrintAdapter schnell auf eine Lochrasterkarte „gefädelt“. Nach ein paar anfänglichen Schwierigkeiten funktionierte der Schaltkreis dann. Ich hatte am Anfang einen „nur“ 4Mhz schnellen Quarzoszillator verwendet, der mir eine fehlerfreie Übertragung bei 9600 Baud ermöglichte. Damit ging das Drucken langsam. Aber es ging.

Um den Adapter dann schließlich auch verwenden zu können braucht man natürlich auch eine passende Software am Palm. Hierfür gibt es mehrere Programme. Dabei sollte jedenfalls darauf geachtet werden, ein Programm zu verwenden, welches auch Office Dokumente aus „DocumentsToGo“ drucken kann, denn dann macht der PrintAdapter erst wirklich Sinn.

Hier eine Liste bekannter Palm Druckerprogramme:

- TealPrint[2]
- PrintBoy[3]
- IRPrint[4]
- InstepPrint[5]
- PalmPrint[6]
- BtPrint[7]
- PrintCard[8]

[2] http://www.tealpoint.com/

[3] http://www.bachmannsoftware.com/

[4] http://www.iscomplete.com/, druckt nur über Infrarot

[5] http://www.instepgroup.com/

[6] http://www.stevenscreek.com/palm/palmprint.shtml

[7] http://www.iscomplete.com/, druckt nur über Bluetooth

[8] Von Sergey Udovenko, keine Homepage verhanden

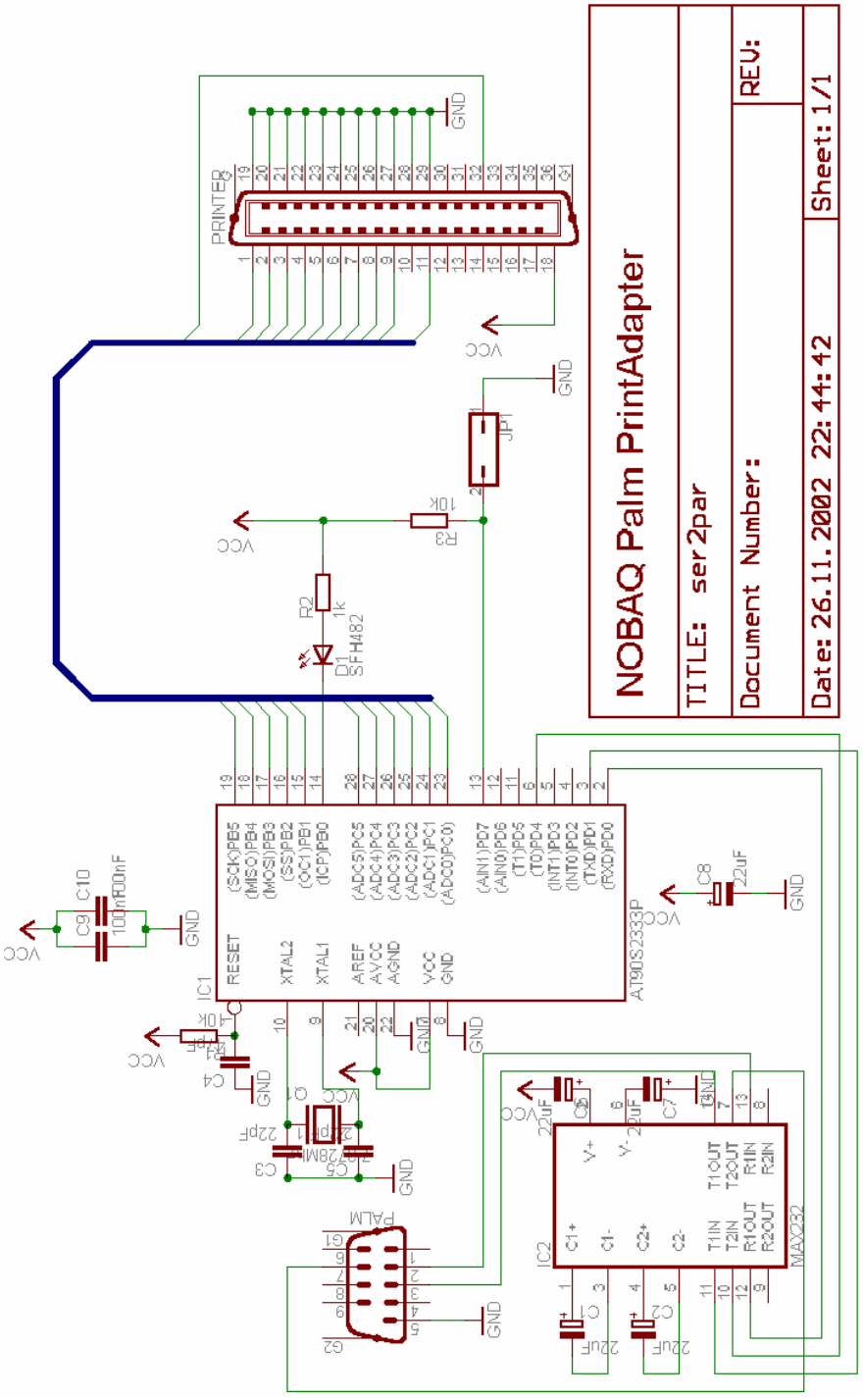
Danach ist die Palm Software entsprechend zu konfigurieren. Die Einstellungen sollten folgende sein:

Baudrate:	115.200
Handshake:	CTS-Hardware Handshake, unter Umständen kann auch das XON/XOFF Handshake Protokoll verwendet werden, sofern dies per Jumper ausgewählt wurde.
Datenbits:	8
Parität:	Keine
Stoppbits:	1

Normalerweise müssen die letzten drei Einstellungen nicht eingestellt werden.

3. Beschreibung des Schaltkreises (Hardware)

3.1. Der Schaltplan



(Abb. 1: Schaltplan des Palm PrintAdapters)

Das Herzstück der Schaltung bildet der Mikrocontroller AT90S2333 der Firma Atmel. An diesen Mikrochip sind alle weiteren Komponenten angeschlossen. An xtal1 und xtal2 ist der Quarz angeschlossen, der sozusagen das „Herz“ des Computers bildet. Der Quarz liefert den benötigten Systemtakt. In diesem Schaltkreis kommt ein Quarz von 7,3728 Mhz zum Einsatz.

Der Elektroniker hat zwei verschiedene Möglichkeiten, um dem Mikrocontroller mit Systemtakt zu versorgen. Die erste ist, wie beschrieben, einen normalen Quarz zu verwenden. Die zweite Möglichkeit besteht darin, einen Quarzoszillator zu verwenden. Quarzoszillatoren haben den Vorteil, dass nur der Ausgang des Quarzoszillators an xtal1 angeschlossen werden muß. Dafür benötigt der Oszillator jedoch eine Verbindung mit der Versorgungsspannung VCC und der Masse GND.

Da es sich bei dem Palm PrintAdapter um einen tragbaren Schaltkreis handelt, wurde beim Design gesteigerter Wert darauf gelegt, die Stromkosten zu senken; aus diesem Grund kommt im Schaltkreis ein normaler Quarz zum Einsatz, auch wenn der zusätzlich benötigte Strom für den Quarzoszillator lediglich bei 1 mA liegen würde. Die zwei Kondensatoren C5 und C3 mit je 22 pF bilden die Pufferkondensatoren für den Quarz, die bei einem Quarzoszillator nicht nötig wären.

Der Pin1 des Controllers ist RESET Pin, der Active HIGH geschaltet ist. Dieser hat die Funktion – wie der Name vermuten läßt – den Prozessor zurückzusetzen. Am RESET Pin ist nun die Resetschaltung angeschlossen, welche aus einem 47 pF großen Kondensator, der nach GND geschaltet ist, und einem 10 kΩ Pullup-Widerstand, der nach VCC geschaltet ist, besteht. Bei der Programmierung zieht der ISP-Programmer die Resetleitung auf GND, wodurch der Eingang LOW wird. Der Mikrocontroller kann nun über das SPI Interface programmiert werden, das sind die Pins SCK, MISO und MOSI am Controller. Da der PrintAdapter ein tragbares Gerät ist, wurde nicht nur, wie

bereits oben erwähnt, gesteigerten Wert auf einen niedrigen Stromverbrauch gelegt, sondern auch auf die Größe der Platine. Aus diesem Grund wurde beim Design des Schaltkreises der ISP Stecker für die Programmierung weggelassen; der Prozessor kann also nur in einem externen Programmiergerät programmiert werden.

Der Pin VCC am Controller erwartet die Versorgungsspannung VCC von 5 Volt und ist mit VCC, also der Stromversorgung verbunden. GND ist mit der Masse GND verbunden. Die Pins AREF, AVCC und AGND werden für den AD-Wandler benötigt, aber auch, wenn man den Port C als normalen I/O Port verwenden will. Deswegen wird in diesem Fall auch AREF mit VCC verbunden und AGND mit GND.

Der Pin PD7 ist als Eingang konfiguriert, an ihn ist der Konfigurationsjumper angeschlossen. Dieser ist nach GND geschaltet, erhält aber einen 10 kΩ Pullup-Widerstand nach VCC.

An Pin PB0 ist die LED verdrahtet, die mit einem Vorwiderstand gegen VCC geschaltet ist. Das LED leuchtet dann, wenn der Pin auf LOW (also Masse) geschaltet wird, dann kann ein Strom durch die gegen VCC geschaltete LED fließen. Das LED ist aus, wenn der Pin auf HIGH geschaltet ist.

Die Pins PB1 bis PB5 und PC0 bis PC5 sind über einen Bus mit dem Parallelport verbunden. Ich habe die Verdrahtung so gewählt, dass sich die verschiedenen Leiterbahnen nicht überkreuzen und habe nicht darauf geachtet, dass der gesamte PORTB die ersten 6 Datenbits darstellen könnte. PC5 ist mit dem STROBE Signal (ein Active HIGH Signal), PC4 bis PC0 mit den Datenbits 1 bis 5, PB5 bis PB3 mit den Datenbits 6 bis 8, PB2 mit dem ACK Signal und PB1 mit dem Error-Signal verbunden. Über den Centronics Stecker (36 Pin, männlich) „PRINTER“ sind die Leitungen mit dem Drucker verbunden. Wiederum aus Gründen der Gesamtgröße der Platine wurde auf externe Pullup-Widerstände verzichtet, die sowieso nicht nötig sind, da der AVR

integrierte Pullup-Widerstände besitzt, ebenso auf einen HEX-Buffer in Form eines großen IC.

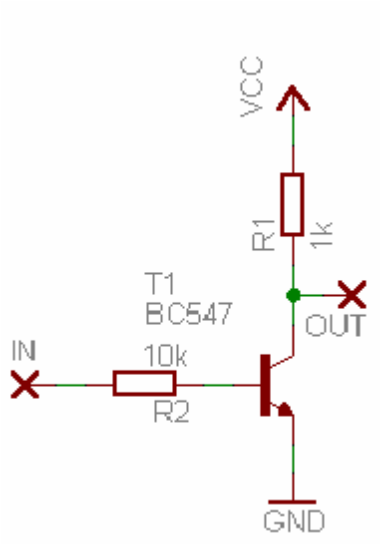
Die Pins 19 bis 29 sind mit der Masse GND verbunden. Laut Centronics Standard ist jeder dieser Pins die Masse für ein bestimmtes Signal, doch heute findet man nur mehr Geräte und Kabel, bei denen diese Pins ohnehin zusammen gelötet sind.

Pin18 liefert die Versorgungsspannung VCC für die Schaltung, die jeder streng Centronics konforme Drucker liefern sollte; jedoch ist dieser Pin bei sehr vielen neueren Druckern nicht mehr verbunden. Dieser Pin18 am Drucker ist direkt mit der 5 V Versorgungsspannung des Druckers verbunden. Laut Centronics Spezifikationen dürfen diesem Pin höchstens 50 mA entnommen werden. Der Palm PrintAdapter benötigt jedoch lediglich 26 mA.

Die Pins RxD und TxD am AVR Prozessor werden vom integriertem UART verwendet. Der Pin PD4 wird für die Hardware Handshake-Leitung CTS verwendet (RTS wird in diesem Fall nicht benötigt, da die Übertragung der Daten bis auf die Xon/Xoff Kontrollbytes nur half-duplex abläuft).

RxD, TxD und CTS führen nun zum zweiten IC im Schaltkreis. Es handelt sich dabei um einen Pegelwandler, nämlich den Standard Pegelwandler MAX232. Dieser macht die +12/-12 Volt der RS232 Schnittstelle mit dem 5 Volt TTL Pegel des Controllers kompatibel. Wie im Kapitel über das RS232 Protokoll noch erläutert werden wird, bedeutet bei RS232 –12 Volt eine logische Eins und +12 Volt eine logische Null. Der Pegelwandler im Palm ist jedoch so tolerant, dass er auch noch Null Volt (also die logische Null laut TTL-Pegel) als –12 Volt interpretiert und +5 Volt als +12 Volt. Aus diesem Grund würde ein normaler Inverter, bestehend aus einem Transistor und 2 Widerständen Abhilfe schaffen.

Dieser Inverter[9] ist in Abbildung 2 dargestellt, aus einer logischen 1 wird eine logische 0 und aus einer logischen 0 eine logische 1. Aus Kompatibilitätsgünden wurde aber dennoch nicht auf einen Pegelwandler verzichtet, um einen RS232 Pegel sicherzustellen. Damit werden Störungen ausgeschlossen und der PrintAdapter kann auch auf anderen Gebieten Einsatz finden, zum Beispiel um von der seriellen Schnittstelle des PC aus zu drucken.



(Abb 2: Inverter aus NPN Transistor aufgebaut)

Die Verdrahtung des MAX232[10] geschieht wie folgt: C1+, C1- und C2+, C2- werden je mit einem 22 µF Elko verbunden, V+ über einen 22 µF Elko an VCC und V- über einen 22 µF Elko an GND. Zwischen VCC und GND kommt möglichst nahe am MAX232 noch ein 22uF Elko zwischen VCC und GND; VCC und GND bilden die Stromversorgung des MAX232. Über T1OUT, R1IN und T2OUT ist der MAX232 mit dem 9-poligen Sub-D Stecker für die serielle Schnittstelle verbunden. Dabei ist zu beachten, dass die Pins 2 und 3 (RxD und TxD) vertauscht sind, und CTS mit Pin7 anstatt mit Pin8 verbunden ist. Der

[9] Basic Bipolar Transistor Logic
(http://www2.ele.ufes.br/~ailson/digital2/cld/AppendixB/appendixB.doc3.html)

[10] Die Seite „RS232 to TTL cable“ gibt zum Beispiel Auskunft über den Anschluß des Pegelwandlers: http://www.seattlerobotics.org/encoder/aug97/cable.html

Palm stellt ja normalerweise das „Slave-Gerät“ dar und man benötigt ein Null Modem Kabel, um den Palm und ein „Slave-Gerät“ korrekt miteinander verbinden zu können. Dieses Nullmodem Kabel wurde damit sozusagen gleich mit eingebaut.

3.1.1. Anschlußschwierigkeiten

Um den PrintAdapter möglichst klein und kompakt zu gestalten, wurde schon bald nach einer Möglichkeit gesucht, die anfangs verwendeten 3 AAA Batterien durch eine andere Spannungsquelle zu ersetzen. Die Kontrollpins der seriellen Schnittstelle des Palm bieten gerade einmal genug Strom, um eine LED zu betreiben. Eine Stromversorgung über die Steuer und Datenleitungen des parallelen Ports ist ebenfalls nicht möglich. Deswegen wurde auf den Pin 18 der Centronics Schnittstelle, welcher eine 5 Volt Spannung bei einer maximalen Stromabgabe von 50 mA liefern sollte, zurückgegriffen.

Da dieser Pin in Verbindung mit der Masse GND die Versorgungsspannung des Schaltkreises darstellt, kann der PrintAdapter nicht in Verbindung mit Druckern verwendet werden, bei denen dieser Pin intern nicht verbunden ist. In so einem Fall kann der Benutzer seinen Drucker „modifizieren“, indem er den Pin manuell verbindet.

Diese Möglichkeit ist jedoch für den Drucker nicht ganz ungefährlich. Es kann nämlich vorkommen, dass der Schaltkreis mehr Strom entnimmt, als das Netzteil liefern kann. Deshalb wird empfohlen, einen Zwischenadapter zu verwenden, bei dem alle Datendrähte verbunden sind, der jedoch 3 AAA Batterien oder Akkus enthält, dessen negative Pole mit GND und positive Pole mit Pin18 verbunden sind.

3.2. Die Platine

Um den Schaltplan zu zeichnen, habe ich das Eagle Programmpaket[11] benutzt, das als Lite Version gratis zum Download[12] bereitsteht. Eine Einschränkung der Lite-Version ist unter anderem das Größenlimit der Platine. Es können höchstens Karten in Euro-Größe bearbeitet werden.

Im Programmpaket dabei ist auch ein „Boarddesigner“. Mit diesem kann der Hardwaredesigner die Elemente auf der Platine positionieren und „routen“, das heißt die Drähte virtuell verlegen. Das fertige Layout kann dann ausgedruckt werden und daraus eine Platine erstellt werden.

Da für den mobilen Einsatz des Schaltkreises ein stabiler Aufbau von großer Bedeutung ist, wird die Platine von einer Fachfirma[13] hergestellt. Da PCBPool die Platinen sehr präzise herstellen kann, verwende ich doppelseitige Platinen, um den Schaltkreis so klein wie möglich zu gestalten. Natürlich wäre es auch möglich, den PrintAdapter mit SMD Bauteilen herzustellen, jedoch ist das Löten mit SMD Bauteilen sehr schwierig.

[11] http://www.cadsoft.de

[12] ftp://ftp.cadsoft.de/pub/program/4.09r2/eagle-4.09r2g.exe

[13] PCBPool, http://www.pcbpool.de

4. Beschreibung der Firmware (Software)

Für die Programmierung der Firmware wurde C verwendet, da für diese Sprache ein leistungsfähiger Compiler angeboten wird, und der vom Compiler erzeugte Code sehr gut optimiert werden kann. Als Compiler wurde der GNU-Compiler avr-gcc gewählt, der ein Derivat des bekannten gcc Compilers unter Linux ist.

Das Source-Verzeichnis der Firmware enthält 2 Quelltext Dateien (ser2par.c und ringbuffer.c), eine Header Datei (ringbuffer.h) und eine Makefile. Die restlichen Dateien werden im Laufe des Compilierens vom Compiler erzeugt.

4.1. Die Datei „Makefile“

Die Datei „Makefile“ wird für die Automatisierung des Compilierens verwendet – wie unter Unix üblich – um dem Programmierer eine schnelle Möglichkeit des Compilierens zu bieten.

Die Erstellung wird einfach mit dem „make“ Befehl gestartet. Zuerst werden einige Variablen definiert, zum Beispiel „MCU“, die den Prozessortyp definiert, damit die passenden Include Dateien automatisch eingefügt werden können. „TRG“ definiert die Hauptdatei des Projekts (ohne Dateiendung) und „SRC“ enthält eine Liste aller Quelldateien des Projekts inklusive Endung.

Die Variable ASFLAGS wird dem Assembler als Parameter hinzugefügt, CPFLAGS dem C Compiler und LDFLAGS dem Linker.

4.2. Die Datei „ser2par.c“

Die Datei „ser2par.c“ ist die Hauptdatei des Projekts, wobei am Anfang der Datei die Standard Headerdateien des avrgcc eingebunden werden:

```
#include <io.h>
#include <interrupt.h>
#include <sig-avr.h>
#include "ringbuffer.h"
```

Die Datei *io.h* enthält Definitionen zu den Adressen der I/O Ports und Makros zum Ansprechen derselben.
Die Datei *interrupt.h* enthält Definitionen zur Verwendung von Interrupts, die für das UART benötigt werden.
Die Datei *sig-avr.h* enthält das Makro SIGNAL() zur Definition eines Interrupt Handlers.
Die Datei *ringbuffer.h* gehört zum lokalen Projekt und beinhaltet Prototypen für die Verwendung des Ringpuffers und auch eine Konstante für die Größe des Ringpuffers.

Die nächsten Zeilen

```
#define ON             1
#define OFF            0
#define X_ON          17
#define X_OFF         19
```

definieren die Makros ON, OFF, X_ON und X_OFF. ON und OFF sind einfache boolesche Konstanten. ON bedeutet eine logische „1“; OFF bedeutet eine logische „0“. X_ON und X_OFF definieren die ASCII Zeichen als dezimale Zahlen, die für XON und XOFF gesendet werden müssen.

Die nachfolgenden Zeilen

```
#define STROBE          5
#define BUSY            2
#define BIT5            0
#define BIT4            1
#define BIT3            2
#define BIT2            3
#define BIT1            4
#define BIT6            5
#define BIT7            4
#define BIT8            3
#define ERROR           1

#define CTS             4
#define LED             0
#define XONXOFF         7
```

definieren die Bits (nicht die Ports), an die die diversen Leitungen für die parallele Schnittstelle angeschlossen sind, gefolgt von Definitionen für die CTS Leitung, das LED und den XON/XOFF Konfigurationsjumper.

Dann werden zwei Variablen definiert, handshake stat und software hsk, die nur boolesche Werte enthalten:

```
unsigned char handshake stat = 0;
unsigned char software hsk = 0;
```

Da der Mikrocontroller ein 8-Bit Prozessor ist, definiere ich solche Variablen gleich als „unsigned char“. Die Variablen werden gleich mit „0“ initialisiert.
Die Variable `handshake stat` hat den Wert „1“ wenn der Empfang durch Verwendung des CTS Handshakes blockiert ist (das heißt der Ringpuffer voll ist).
Die Variable `software hsk` hat den Wert „1“ wenn der Benutzer per Jumper das Software-Handshake aktiviert hat.

Als nächstes wird das Makro rwait() definiert, das mit Hilfe von nop Befehlen den Prozessor ein wenig warten lässt. Eine Ansammlung von nop Befehlen wie hier, vor allem wenn sie als Makro verwendet werden, können den Programmcode erheblich vergrößern. Dafür lässt sich die Zeit aber auch wirklich um jeden Zyklus regulieren, da nicht zusätzliche Zeit in Anspruch genommen wird, die zum Beispiel beim Aufruf einer Funktion entstehen. Da der Prozessor mit 2 Kilobytes genug Speicher für den Maschinencode bietet, wurde diese Möglichkeit vorgezogen:

```
//für 7,3728Mhz
#define rwait() asm volatile("nop" "\n\t" "nop"
    "\n\t" "nop" "\n\t" "nop" "\n\t" "nop" "\n\t"
    "nop" "\n\t" "nop" "\n\t" "nop" "\n\t" "nop"
    "\n\t" "nop" "\n\t" "nop" "\n\t" "nop" "\n\t"
    "nop" "\n\t" "nop" "\n\t" "nop" "\n\t" "nop"
    "\n\t" "nop" "\n\t" "nop" "\n\t" "nop" "\n\t"
    "nop" "\n\t" "nop");
```

Das Makro baud() in der nächsten Zeile schreibt die zu verwendende Baudrate in das Register UBRR:

```
// 115200baud @ 7,3728Mhz UBRR=3
#define baud()      asm   volatile("ldi   r16,3"
    "\n\t" "out 0x09,r16")
```

Der Wert, den man in das Register UBRR schreiben muss, berechnet sich mit der Formel:

$$UBRR \Leftarrow \frac{Taktfrequenz}{Baudrate * 16} - 1$$

Die Taktfrequenz wird hierbei in Hertz angegeben. Wenn dieser Wert eine natürliche Zahl ist, gibt es 0% Fehler; sofern es eine Kommazahl ist, gibt es einen gewissen Prozentsatz Fehler. Ich habe den 7,3728 Mhz Quarz gewählt,

da er bei allen Baudraten, vor allem aber bei 115.200 Baud, eine Fehlerrate von 0% aufweist. Der erhöhte Stromverbrauch beträgt lediglich ein paar Milliampère und ist zu vernachlässigen. Eine Tabelle für alle Baudraten und Taktfrequenzen befindet sich im Anhang.

Die nächsten Zeilen definieren die Funktion eine sekunde(), die in etwa eine Sekunde wartet und mit dem Programm AVRDelayLoop[14] erstellt wurde:

```
void eine sekunde(void)
{
  asm volatile(
  "ldi  R17, 0x24"            "\n\t"
  "WGLOOP0:  ldi  R18, 0xBC"   "\n\t"
  "WGLOOP1:  ldi  R19, 0xC4"   "\n\t"
  "WGLOOP2:  dec  R19"         "\n\t"
  "brne WGLOOP2"               "\n\t"
  "dec  R18"                   "\n\t"
  "brne WGLOOP1"               "\n\t"
  "dec  R17"                   "\n\t"
  "brne WGLOOP0"               "\n\t"
  "ldi  R17, 0x01"             "\n\t"
  "WGLOOP3:  dec  R17"         "\n\t"
  "brne WGLOOP3"               "\n\t"
  "nop"                        "\n\t");
}
```

In diesem Programm gibt man die Taktfrequenz des Prozessors und die zu wartende Zeit ein und das Programm gibt den passenden Assembler-Code aus.

[14] http://www.home.unix-ag.org/tjabo/avr/AVRdelayloop.html

Die nächste Funktion ist send par(). Diese sendet ein Byte nach der Centronics Spezifikation an die parallele Schnittstelle.

```
void send par(unsigned char in)
{
  while(bit is set(PINB, BUSY));
```

Die erste while-Schleife wartet, bis das BUSY LOW ist, erst dann kann gesendet werden.
Als nächstes werden die einzelnen Bits über die I/O Ports des AVR ausgegeben:

```
  if((in & 0x01) == 0x01)
    sbi(PORTC, BIT1);
  else
    cbi(PORTC, BIT1);
  if((in & 0x02) == 0x02)
    sbi(PORTC, BIT2);
  else
    cbi(PORTC, BIT2);
  if((in & 0x04) == 0x04)
    sbi(PORTC, BIT3);
  else
    cbi(PORTC, BIT3);
  if((in & 0x08) == 0x08)
    sbi(PORTC, BIT4);
  else
    cbi(PORTC, BIT4);
  if((in & 0x10) == 0x10)
    sbi(PORTC, BIT5);
  else
    cbi(PORTC, BIT5);
  if((in & 0x20) == 0x20)
    sbi(PORTB, BIT6);
  else
```

```
    cbi(PORTB, BIT6);
  if((in & 0x40) == 0x40)
    sbi(PORTB, BIT7);
  else
    cbi(PORTB, BIT7);
  if((in & 0x80) == 0x80)
    sbi(PORTB, BIT8);
  else
    cbi(PORTB, BIT8);
```

Dieser lange Block ist deswegen nötig, da die Bits 0 bis 5 nicht vollständig an einen 6-Bit Port angeschlossen sind, damit sich auf der Platine die Leitbahnen nicht kreuzen.
Das folgende `rwait();` wartet ca. 10 µs, danach wird mit `cbi(PORTC, STROBE);` die STROBE Leitung auf LOW gelegt (da diese ja Active-High ist), nach 10us wieder mittels `rwait();sbi(PORTC, STROBE);` auf HIGH, und danach mit 3-fachem Aufruf von `rwait();` ca. 30 µs gewartet.

```
}
```

Die nächste Funktion `void handshake(unsigned char state)` schaltet das Handshake ein oder aus. Diese Funktion wird immer dann aufgerufen, wenn der Ringpuffer voll ist, oder der Ringpuffer wieder leer ist. `handshake(ON)` schaltet das Handshake ein; es können keine Daten mehr empfangen werden. `handshake(OFF)` schaltet das Handshake wieder aus, so dass wieder Daten empfangen werden können.

Die erste if-Anweisung überprüft, ob das Handshake eingeschaltet oder ausgeschaltet werden soll. Wenn das Handshake eingeschaltet werden soll, dann wird mit dem „sbi“ Befehl der Pin „CTS“ an PORTD auf HIGH gesetzt. CTS ist nun aktiviert. Die nächste if-Anweisung prüft, ob das Software Handshake verwendet werden soll. Wenn dies der Fall ist, wartet das

Programm mit Hilfe der while() Schleife, bis das UART frei zum Senden ist. Danach wird das Byte X ON in das Register UDR geschrieben; das UART sendet das Zeichen.
„handshake stat = ON“ setzt die Variable auf ON, was bedeutet, dass das Handshake gerade aktiviert ist.

```
if(state==ON)
{
  sbi(PORTD, CTS);
  if(software hsk)
  {
    while(!bit is set(UCSRA, UDRE));
    outp(X ON, UDR);
  }
  handshake stat = ON;
}
```

Der „else“ Zweig soll das Handshake wieder abschalten. Dazu setzt die „cbi“ Anweisung die CTS Leitung wieder auf LOW. Wenn das Software Handshake aktiviert ist, wird das X OFF Byte gesendet.
Schließlich wird die Variable „handshake stat“ wieder auf OFF gesetzt.

```
  else
{
  cbi(PORTD, CTS);
  if(software hsk)
  {
    while(!bit is set(UCSRA, UDRE));
    outp(X OFF, UDR);
  }
  handshake stat = OFF;
}
```

Die nächsten Zeilen definieren die Interrupt Handling Routine für den UART Empfangs Interrupt, der immer aufgerufen wird, wenn ein Zeichen empfangen wurde. Das empfangene Zeichen befindet sich dann im Register UDR (das UART Datenregister).

```
SIGNAL(SIG UART RECV)
{
  register unsigned char got;
```

Diese Anweisung definiert eine temporäre 8-Bit Varibale „got“ für das empfangene Zeichen. Das Schlüsselwort „register“ bedeutet, dass für die Variable direkt ein Register verwendet werden soll, was den Zugriff auf die Variable sehr beschleunigt.

```
  got = inp(UDR);
```

Diese Anweisung holt das Byte aus dem Datenregister UDR und schreibt es in die Variable „got“.

```
  if(!rb put(got))
    handshake(ON);
}
```

Mit dieser Anweisung wird das empfangene Byte im Ringpuffer gespeichert. Falls der Ringpuffer schon voll ist, gibt `rb put()` den Wert FALSE zurück. Das empfangene Byte wird noch im Ringpuffer gespeichert und mit Hilfe der Anweisung `handshake(ON)` das Handshake aktiviert, damit keine Daten mehr empfangen werden.

Die letzte Funktion in dieser Datei ist die main() Funktion: Zuerst wird dabei eine Variable „rec“ definiert, die immer das gerade zu bearbeitende Byte enthält. Die nächste Anweisung schaltet den Empfang des UARTS ein, die darauf folgende das Senden. Mit „baud()“ wird das oben definierte Makro zum Setzen der Baudrate aufgerufen.

```
int main(void)
{
  unsigned char rec;
  sbi(UCSRB, RXEN);
  sbi(UCSRB, TXEN);
  sbi(UCSRB, RXCIE);
  baud();
```

Die folgenden Anweisungen konfigurieren alle Ausgänge als Ausgänge. Das sind die Datenbits 1 bis 8 und das STROBE Signal. Weiters ist die CTS Leitung und das LED Pin ein Ausgang.

```
  sbi(DDRC, BIT5);      // Bit 5
  sbi(DDRC, BIT4);      // Bit 4
  sbi(DDRC, BIT3);      // Bit 3
  sbi(DDRC, BIT2);      // Bit 2
  sbi(DDRC, BIT1);      // Bit 1
  sbi(DDRC, STROBE);    // STROBE
  sbi(DDRB, BIT6);      // Bit 6
  sbi(DDRB, BIT7);      // Bit 7
  sbi(DDRB, BIT8);      // Bit 8
  sbi(DDRD, CTS);       // CTS
  sbi(DDRB, LED);       // LED
```

Nun werden alle Eingänge konfiguriert. Das ist die BUSY Leitung, die ERROR Leitung und der XON/XOFF Jumper zum Konfigurieren des Software Handshakes.

```
cbi(DDRB, BUSY);      // BUSY
cbi(DDRB, ERROR);     // Error
cbi(DDRD, XONXOFF);   // Xon/Xoff Jumper
```

Alle Ausgänge werden nun auf ihre Standardwerte gesetzt. Alle Datenbits erhalten „0“ als Initialisierung, STROBE wird auf HIGH gesetzt, da es eine Active HIGH Leitung ist und die CTS Leitung wird ausgeschaltet.

```
cbi(PORTC, BIT5);     // Bit 5
cbi(PORTC, BIT4);     // Bit 4
cbi(PORTC, BIT3);     // Bit 3
cbi(PORTC, BIT2);     // Bit 2
cbi(PORTC, BIT1);     // Bit 1
cbi(PORTB, BIT6);     // Bit 6
cbi(PORTB, BIT7);     // Bit 7
cbi(PORTB, BIT8);     // Bit 8
sbi(PORTC, STROBE);   // STROBE = HIGH
cbi(PORTD, CTS);      // CTS
```

Mit `handshake stat = OFF;` wird die Variable für die Verwendung des Handshakes auf OFF gesetzt.

Die nächsten Anweisungen lassen die LED als Zeichen der Bereitschaft kurz blinken. Zuerst wird das LED durch die „cbi“ Anweisung eingeschaltet.

Danach wird der Zustand des Software Handshake Jumpers überprüft. Wenn dieser gesetzt ist, wird die Variable „software hsk“ auf 1 gesetzt. Wenn

das Software Handshake aktiviert ist, blinkt die LED einmal kurz. Ist der Jumper nicht gesetzt, wird die Variable „software hsk“ auf 0 gesetzt.

```
cbi(PORTB, LED);
if(!bit is set(PIND, XONXOFF))
{
  software hsk = 1;
  eine sekunde();
  sbi(PORTB, LED);
  eine sekunde();
  cbi(PORTB, LED);
}
  else
{
  software hsk = 0;
}
```

Der Befehl `sei();` wird direkt in die Assembler Anweisung „sei“ umgesetzt, welche alle Interrupts aktiviert und damit erst den Empfang über das UART aktiviert.
Die letzten Zeilen der Funktion main bilden die Hauptschleife, die niemals verlassen wird. Das untere „return“ wird also nie erreicht.

```
while(1)
{
```

Die folgende if Anweisung holt ein Byte aus dem Ringpuffer und schreibt es nach rec. Die Funktion gibt FALSE zurück wenn der Puffer leer ist.

```
  if(!rb get(&rec))
  {
```

Zu diesem Zeitpunkt ist der Puffer leer. Nun wird überprüft, ob das Handshake aktiviert ist. Falls es aktiviert ist, wird es abgeschaltet, da nun wieder freier Platz im Ringpuffer verfügbar ist.

```
    if(handshake stat)
```

```
            handshake(OFF);
        }
          else
        {
```

Im Ringpuffer war ein Byte, dieses ist jetzt in der Variable „rec“. Mit send par() wird dieses Byte nach Centronics Spezifikation am parallelen Port ausgegeben

```
            send par(rec);
        }
      }
      return 0;
    }
```

4.3. Die Datei „ringbuffer.c“

Diese Datei enthält die Funktionen für die Verwendung des Ringpuffers. Die erste Zeile

```
#include "ringbuffer.h"
```

inkludiert die zugehörige Header Datei.

Mit

```
static unsigned char    iput = 0;
static unsigned char    iget = 0;
static unsigned char    n = 0;
static unsigned char    buffer[BUFFER];
```

werden einige globale Variablen definiert: „n“ enthält die Anzahl der im Ringpuffer enthaltenen Zeichen. „iput“ ist ein Zeiger auf einen Speicherplatz im Ringpuffer, in den das nächste Byte geschrieben werden soll, „iget“ ein Zeiger auf das Byte, das gelesen werden soll. Da der Atmel ein 8-Bit Prozessor ist, habe ich gleich überall 8-Bit Variablen verwendet. „buffer“ ist der Speicherplatz des Ringpuffers. Es wird Speicher statisch mit der Größe, die in BUFFER

angegeben ist, reserviert. Alle Variablen werden mit dem Schlüsselwort „static“ deklariert. Das bedeutet, dass die Variablen interne Variablen dieser Datei sind und sonst nirgends verwendet werden können außer in dieser Datei. Der Zugriff wäre nur über Pointer möglich.

Die Funktion

```
void init buffer(void)
{
  iput = 0;
  iget = 0;
  n = 0;
}
```

ist die Initialisierungsfunktion für den Ringpuffer und setzt alle Variablen auf „0“.

Die Funktion

```
unsigned char addring(unsigned char i)
{
  return (i+1) == BUFFER ? 0 : i + 1;
}
```

gibt die Werte für `iput` und `iget` zurück. Der Parameter `i` beschreibt die aktuelle Position im Ringpuffer. Wenn nun diese Position + 1 im Ringpuffer gleich der Puffergröße ist, gibt die Funktion „0“ zurück. Damit kann anhand des Rückgabewertes festgestellt werden, ob der Ringpuffer schon voll ist, falls geschrieben werden soll. Wenn gelesen werden soll, kann festgestellt werden, ob das Ende des Puffers erreicht ist. Falls der Parameter + 1 nicht die Puffergröße ergibt, wird Parameter + 1 zurückgegeben. Das hier verwendete Konstrukt ist eine vereinfachte if-Anweisung in C, deren Syntax folgendermaßen lautet:

„`Bedingung ? Anweisung : else-Anweisung`“.

Diese Anweisung kann nur für einfache Ausdrücke verwendet werden, Anweisungsblöcke sind nicht möglich.

Die Funktion

```
unsigned char rb get(unsigned char *tosave)
{
  unsigned char pos;
```

holt das nächste Zeichen aus dem Ringpuffer und schreibt es nach `tosave`. Die Variable `pos` definiert die aktuelle Zeichenposition.

```
  if(n > 0)
  {
```

Wenn sich Zeichen im Ringpuffer befinden, wird zunächst die alte Position nach `pos` geschrieben, um nachher dieses Zeichen zurückzuliefern. Danach wird `iget` der neue Wert zugewiesen und der Wert von `n` um eins vermindert. Das gelesene Zeichen wird in den Speicher geschrieben, der vom Benutzer über den Parameter angegeben wurde.

```
    pos = iget;
    iget = addring(iget);
    n--;
    *tosave = buffer[pos];
```

Die Funktion beendet mit TRUE (1) wenn noch Zeichen im Puffer sind oder FALSE (0) wenn der Puffer leer ist.

```
    return 1;
  }
  return 0;
}
```

Die Funktion `rb put` schreibt das Zeichen, das als Parameter der Funktion übergeben wurde, in den Ringpuffer.

```
unsigned char rb put(unsigned char z)
{
```

Es wird überprüft, ob die Anzahl der Zeichen den Puffer schon überschritten hat. Normalerweise sollte das nie der Fall sein und diese Anweisung ist als Sicherheitsmaßnahme zu sehen.

```
    if(!(n < BUFFER))
      return 0;
```

Danach wird das übergebene Zeichen in die aktuelle Position im Ringpuffer geschrieben, `iput` auf den nächsten freien Platz gesetzt und der Wert von `n` um eins erhöht. Wenn die Anzahl der Zeichen im Puffer (n) den Platz im Puffer noch nicht übersteigt, wird TRUE (1) zurückgegeben; falls der Puffer bereits voll ist, FALSE (0).

```
    buffer[iput] = z;
    iput = addring(iput);
    n++;
    if(n < BUFFER)
      return 1;
    return 0;
  }
```

4.4. Die Datei „ringbuffer.h“

Diese Datei enthält nur

```
#ifndef   RINGBUFFER
#define   RINGBUFFER

#define BUFFER        40

void init buffer(void);
unsigned char addring(unsigned char i);
unsigned char rb get(unsigned char *tosave);
unsigned char rb put(unsigned char z);

#endif
```

Die ersten Zeilen sind Präprozessor Direktiven. Wenn das Makro RINGBUFFER noch nicht definiert wurde, wird es definiert und der Inhalt der Datei eingebunden. Wenn es schon definiert ist, wird nichts in der if-Abfrage eingebunden. Diese Technik wird verwendet, um das doppelte Einbinden einer Header Datei und damit Compiler Fehler zu vermeiden.

Das Makro BUFFER definiert die Größe des Ringpuffers, in diesem Fall 40 Byte.
Die restlichen 4 Zeilen sind Prototypen für die in ringbuffer.c definierten Funktionen.

4.5. Vom Compiler erzeugte Dateien

Zuerst bearbeitet der Präprozessor die Source Dateien und löst die Direktiven auf. Die erstellten Dateien mit der Endung „i“ sind die vom Präprozessor erstellten Dateien. Danach wird der Sourcecode durch den Compiler in Assembler Code übersetzt. Die Ausgabedateien tragen die Endung „s“. Diese werden dann vom Assembler verarbeitet. Der Assembler erstellt die Dateilistings mit der Endung „lst“. Diese werden in Objektdateien assembliert; weiters wird noch eine „map“ Datei erstellt, die Informationen zum Speicher des AVR bieten. Der Linker fügt diese Objektdateien mit der Endung „o“ zu einem ganzen Programm zusammen, das dann in die Ausgabedatei „elf“ geschrieben wird. Das ist jetzt eine ausführbare Linux ELF-Datei. Natürlich kann diese Datei nicht wirklich unter einem Linux System ausgeführt werden, da der darin enthaltene Code ja AVR Code ist. Diese ELF Datei wird dann in reinen Maschinencode konvertiert, der sich in der Datei „ser2par.rom“ befindet. Weiters wird die Datei „ser2par.eep“ erstellt, welche die Daten für den EEPROM enthalten, der EEPROM wird jedoch vom Palm PrintAdapter nicht in Anspruch genommen. Sowohl die *.eep als auch die *.rom Dateien sind reine ASCII Dateien, die HEX Codes der binären OP Codes enthalten. Damit die Datei „ser2par.rom“ mit isp avr[15] kompatibel ist, wird sie mit dem Programm „hexbin2.exe“ in reinen binären Maschinencode konvertiert. Die endgültige Ausgabedatei heißt nun „ser2par.bin“ und kann mit Hilfe eines Programmers in den Speicher des AVR geschrieben werden.

[15] http://people.freenet.de/buss/avr.html

4.6. Laden der Firmware auf den AVR

Wie bereits erläutert, kommt der ISP Programmer isp avr.exe zum Einsatz, weil er sehr leicht zu verwenden ist und man zum Programmieren keinen eigenen IC benötigt, sondern nur einige Pullup-Widerstände.

Um den AVR zu programmieren[16], verbindet man diesen über „xtal1“ und „xtal2“ mit einem Quarz. Die zwei Kondensatoren (ca. 22 pF) dürfen nicht vergessen werden. Über VCC und GND wird der AVR an eine stabile 5 Volt Stromversorgung angeschlossen.

Dann wird der AVR mit dem Parallelport des PC verbunden:

Centronics Pin6 → RESET
Centronics Pin7 → MOSI
Centronics Pin8 → SCK
Centronics Pin10 → MISO
Centronics Pin24 und Pin25 → GND

Die Leitungen MOSI, SCK und MISO erhalten noch einen 10 kΩ Pullup-Widerstand gegen VCC. RESET wird zusätzlich mit einem 10 kΩ Widerstand gegen VCC geschaltet und zwischen RESET und GND noch ein 46 pF Kondensator. Um Störungen zu vermeiden, sollte möglichst nahe am AVR noch ein kleiner 100 nF Kondensator zwischen VCC und GND geschaltet werden. Wenn alles verbunden ist und das parallele Kabel an LPT1 angeschlossen ist, kann der AVR mit dem folgenden Kommando programmiert werden:

```
ISP AVR.EXE /LPT1 /ERASE SER2PAR.BIN
```

Danach ist der AVR sofort einsatzbereit.

[16] Die Anleitung zur Verwendung und Verdrahtung des ISP-Adapter stammt von der isp avr Homepage (http://people.freenet.de/buss/avr.html)

5. Das RS232 Protokoll

Das RS232 Protokoll ist ein sehr simples und sehr altes Protokoll. Es stellt einen kompletten Standard dar, der nicht nur die elektronischen Eigenschaften vorgibt, sondern auch die Signal und Spannungspegel, die Verdrahtung, Pinbelegung und Stecker und die Kontrollinformationen.

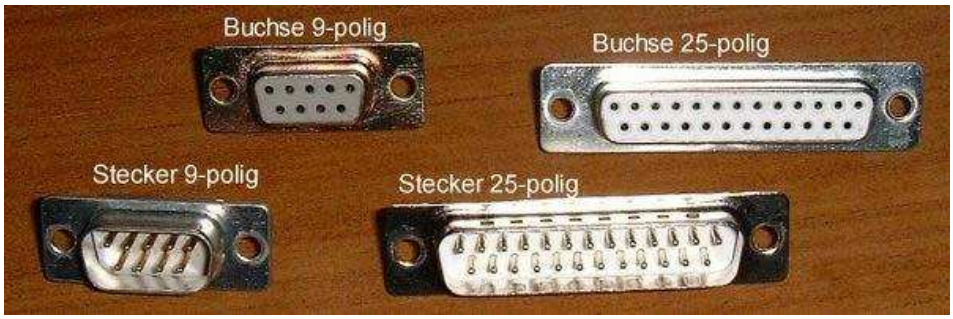
RS232 stammt aus dem Jahre 1962. Da zu diesem Zeitpunkt der TTL Standard noch nicht definiert war, verwendet RS232 nicht +5 Volt und 0 Volt, sondern +12 Volt und -12 Volt. Der tatsächliche Pegel darf auch etwas abweichen, nach dem ursprünglichen Standard darf die Spannung zwischen +5 und +15 Volt bzw. –5 und –15 Volt liegen. Im Gegensatz zum TTL Standard bedeuten –12 Volt ein logisches HIGH, seinerzeit auch als „marking“ bezeichnet und +12 Volt ein logisches LOW oder „spacing“. Im Ruhezustand liegt die Leitung auf HIGH, also auf –12 Volt.

Die Kommunikation mittels RS232 läuft immer zwischen einem „Host“ (Data Terminal Equipment – DTE) und einem Peripherie Gerät (Data Circuit-Terminating Equipment – DCE) ab. Normalerweise ist der Computer immer das DTE Gerät und die Peripherie das DCE Gerät, also Modems, Mäuse oder auch der Palm. Wenn man jetzt zwei Peripheriegeräte verbinden will muss also eines davon das DTE werden. Deshalb muss man die RxD und die TxD Leitung vertauschen. Dies passiert beim PrintAdapter direkt auf der Platine.

Außer der Spannung ist auch die elektrische Impedanz vorgegeben. Die Belastung sollte zwischen 3 kΩ und 7 kΩ liegen. Ursprünglich war auch die Länge des Kabels mit 15 Meter definiert. Doch später definierte man die kapazitative Belastung des Kabels mit 2.500 pF. Die maximale Kabellänge wurde dann mit der Kapazität pro Kabeleinheit berechnet.

Neben den elektrotechnischen Eigenschaften sind auch alle Signale und Pinbelegungen definiert. Die Signale lassen sich in verschiedene Gruppen einteilen: Common, data, control, timing. RS232 definiert in der Urform 23 Signale, doch in den seltensten Fällen werden alle diese Signale gebraucht. Sogar ein Modem, das schon sehr viele Leitungen benötigt, braucht höchstens 8 dieser Leitungen. Im PC sind überhaupt nur mehr neun Leitungen vorhanden. Einfache Peripherie benötigt nur vier Signale (zwei für Daten, zwei für Handshake). Die einfachste Form von RS232 besteht einfach aus zwei Leitungen (RxD und TxD), die kein Hardware Handshake verwendet. Wenn nur Kommunikation in eine Richtung gewünscht ist, reicht sogar ein einziges Signal. Im Anhang ist eine Tabelle mit den heute gebräuchlichen Signalen abgebildet.

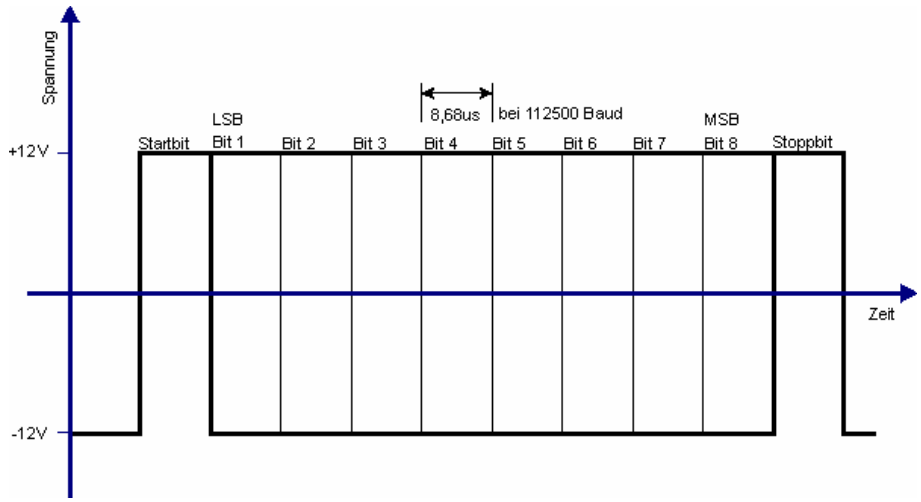
Ebenfalls definiert sind die mechanischen Eigenschaften. RS232 verwendet für gewöhnlich einen 25-Pin oder einen 9-Pin Sub-D Stecker. Der DTE Teil hat fest eingebaut einen männlichen Stecker und das Kabel einen weiblichen. Der DCE Teil hat im Gerät einen weiblichen Stecker, das Kabel ist männlich.



(Abb. 3: Verschiedene RS232 Stecker)

Diese Abbildung zeigt verschiedene RS232 kompatible Stecker. Wie bereits im Hardware Kapitel beschrieben, arbeiten moderne Elektronikgeräte nach dem TTL Standard; dieser arbeitet mit 0 Volt als logisches LOW und 5 Volt als logisches HIGH. Um die verschiedenen Pegel zu konvertieren, braucht man einen Pegelwandler, wie zum Beispiel den MAX232.

Die tatsächliche Datenübertragung findet auf den zwei Signalen RxD und TxD statt; eine für das Senden und die andere für das Empfangen von Daten. Die Geschwindigkeit der Übertragung hängt von der Baudrate ab, die zuvor vom Benutzer eingestellt werden muss und sowohl auf dem DCE als auch auf dem DTE Gerät gleich sein muss.



(Abb. 4: Signaldiagramm für RxD oder TxD)

Parameter, die der Benutzer für die Übertragung mittels RS232 einstellen muss, sind folgende:

- Übertragungsrate (Bits pro Sekunde in Baud)
- Datenbits (4, 5, 6, 7, 8)
- Parität (gerade, ungerade, keine, Markierung, Leerzeichen)
- Stoppbits (1, 1.5 oder 2)
- Protokoll (Keines, Hardware oder Software)

Natürlich müssen diese Parameter ebenso wie die Baudrate auf beiden Geräten gleich eingestellt sein.

Abbildung 4 zeigt den Signalfluss für die Datensignale RxD und TxD. Die Datenleitung befindet sich im Ruhezustand auf -12 Volt, also auf HIGH. Zuerst wird das Startbit übertragen. Durch die fallende Flanke des Startbits wird der Beginn einer neuen Übertragung gekennzeichnet. Danach werden die Datenbits übertragen; und zwar so viele wie in der Einstellung „Datenbits“

angegeben. Allerdings ist dieser Wert praktisch immer 8, also ein Byte. Von diesem Byte wird das LSB (Least Significant Bit) als erstes übertragen und das MSB (Most Significant Bit) als letztes. Die Länge einer Bitzelle errechnet sich aus der Baudrate. Je größer die Baudrate, desto schneller die Übertragung. Allerdings ist die Übertragung bei hoher Baudrate auch sehr fehleranfällig. Der PrintAdapter empfängt Daten mit 115.200 Baud, also mit 115.200 Bits pro Sekunde.

$$\frac{1}{115200} = 0{,}868 * 10^{-5} s = 8{,}6806 \mu s$$

Ein Bit braucht also 8,68 µs oder ein ganzes Byte 86,806 µs, sofern die Standardeinstellung von einem Stoppbit beibehalten wird.

Da der PrintAdapter nur Daten empfängt und selbst keine sendet, wird nur eine Handshake Leitung benötigt, nämlich die für das Empfangen von Daten: CTS (Clear To Send). Diese Kontrollleitung liegt im Ruhezustand auf +12 Volt, also auf 0 Volt am Mikrocontroller. Erst wenn der Mikrocontroller zu viele Daten auf einmal empfängt, der Ringpuffer voll ist und somit keine Daten mehr empfangen kann, legt der Mikrocontroller die Leitung auf HIGH. Der Palm wartet nun mit dem Senden der Daten solange, bis die Signalleitung wieder auf HIGH geschaltet ist. Ein vollständiger Hardware Handshake würde die Leitungen CTS, RTS und DSR und DTR beanspruchen.

Alternativ stellt RS232 auch einen Software Handshake zu Verfügung, nämlich das XON/XOFF Protokoll. Falls der Mikrocontroller keine Daten mehr empfangen kann, schickt er einfach das XON Zeichen (0x11). Sobald er wieder fähig ist, Daten zu empfangen, schickt er XOFF (0x13). Dieser Handshake sollte allerdings nur bei reiner Textübertragung verwendet werden und nicht bei binären Daten.

6. Das Centronics Protokoll

Die Centronics Schnittstelle (parallele Schnittstelle) wurde im Jahre 1981 entwickelt. Sie wurde ursprünglich nur für den Anschluss eines Druckers an den IBM-PC verwendet. Mittlerweile dient die parallele Schnittstelle auch zum Anschluss anderer Geräte wie z.B. des ZIP-Drive. Ebenso wie das RS232 Protokoll definiert auch die Centronics Spezifikation die Stecker, Kabel, elektrische Eigenschaften und das Protokoll selbst.

Centronics verwendet einen 25-Pin Sub-D Stecker für den „Sender“ der Daten. Der Stecker am Gerät ist weiblich, der am Kabel männlich. Das andere Ende, welches meist am Drucker angeschlossen ist, besitzt einen 36-Pin Centronics Stecker. Der Stecker am Drucker ist weiblich, der am Kabel männlich. Im Gegensatz zum RS232 Protokoll überträgt Centronics die 8 Bits parallel und ist damit in der Regel schneller als RS232. Allerdings können die Daten nur in eine Richtung übertragen werden.

Da aber schon bald andere Geräte, wie externe CD-Rom Laufwerke oder ZIP-Drives, für den Parallelport angeboten wurden, wurde 1991 von IBM, Lexmark, Texas Instruments und IEEE ein neuer Standard geschaffen. Dieser heißt IEEE 1284 und unterstützt fünf verschiedene Modi:

- **Compatibility Modus:** Der eigentliche alte „Centronics“ Standard der weiter unten genauer beschrieben wird. Daten können nur in eine Richtung übertragen werden, außer in Verbindung mit dem „nibble mode“.
- **Nibble Modus:** Mit Hilfe des Nibble mode kann der Drucker auch Daten an den Computer zurückschicken. Dazu werden zwei Statusleitungen für die Übertragung eines Nibbles verwendet. Da die zwei Nibbles hintereinander übertragen werden müssen, braucht die Übertragung doppelt so lange wie die von Computer zu Drucker.

- **Byte oder PS/2 Modus:** Im Gegensatz zum Nibble mode wird gleich ein ganzes Byte übertragen. Die Geschwindigkeit von Drucker zu Computer ist gleich schnell wie von Computer zu Drucker.
- **ECP (Enhanced Capability Port):** Dieser Modus überträgt die Daten bidirektionnell. Der Modus wurde vor allem für Drucker und Scanner entwickelt. Die Übertragungsrate liegt bei ca. drei Megabyte pro Sekunde. Im ECP Modus kann der Anwender mehrere Endgeräte an den Parallelport anschließen und gemeinsam verwenden. Das wird durch „Kanaladressierung“ möglich.
- **EPP (Enhanced Parallel Port):** EPP ist ebenso wie ECP bidirektional und erlaubt Übertragungsraten bis zu zwei Megabyte pro Sekunde. EPP verwendet Datenzyklen, die Daten vom Computer zur Peripherie und umgekehrt übertragen und Adresszyklen, die Adressen, Befehle oder sonstige Informationen übertragen.

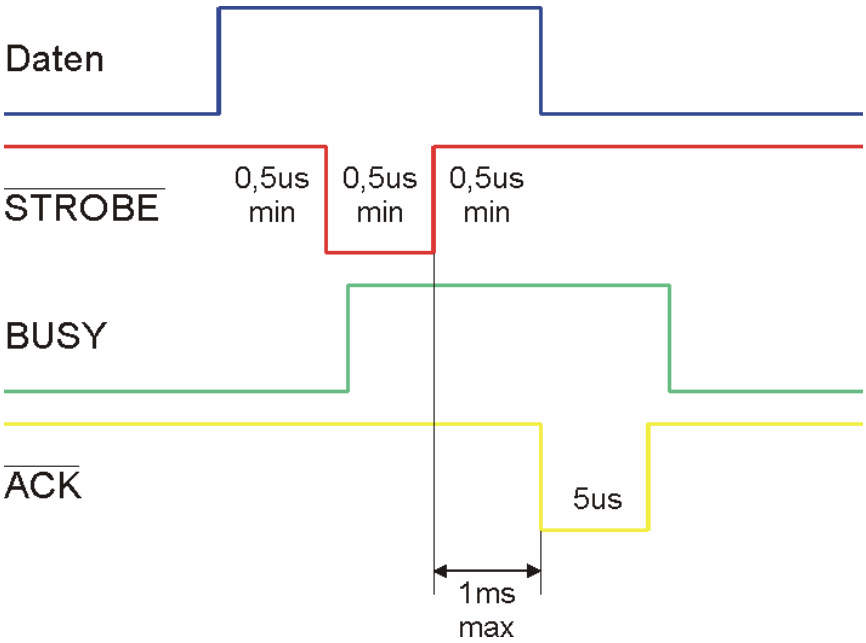
Die ersten beiden Modi werden auch häufig mit SPP (Standard Parallel Port) zusammengefasst.

Im Gegensatz zu RS232 verwendet der Parallelport den TTL Pegel und kann somit ohne weitere Bauteile (außer Pullup-Widerständen) an einen Mikrocontroller angeschlossen werden.

Damit der PrintAdapter mit einem normalen Drucker drucken kann, muss er also den SPP Modus verwenden. Bei diesem Modus werden Daten nur vom PrintAdapter zum Drucker übertragen, der Drucker kann verschiedene Statusinformationen auf den Kontrollleitungen zurückliefen. Im Anhang ist eine Tabelle mit den Pinbelegungen abgedruckt.

Die Leitungen lassen sich – ähnlich wie bei RS232 – in drei Kategorien einteilen:

- **Datenleitungen:** Diese Leitungen übertragen die eigentlichen Daten (Pin 2 bis 9)
- **Handshakeleitungen:** Diese dienen zur korrekten Übertragung der Daten. Zu diesen Kategorie gehört das STROBE Signal, das ACK Signal und das BUSY Signal.
- **Status- und Steuerleitungen:** Das Select In Signal teilt dem Drucker mit, dass er „Online“ ist. Die RESET Leitung veranlasst ein Zurücksetzen des Druckers und AUTOFEED nach jedem CR auch ein LF zu drucken. Der Drucker kann auch dem Computer seinen Status mitteilen: PE bedeutet „Kein Papier mehr“, Select Out zeigt den Status des Druckers an und Error einen Fehler.



(Abb. 5: Kommunikation zwischen PrintAdapter und Drucker)

Diese Abbildung zeigt die Kommunikation zwischen dem PrintAdapter und dem Drucker. Bevor der PrintAdapter Daten senden kann, muss er die Bereitschaft des Druckers abwarten. Die BUSY Leitung ist normalerweise auf LOW; wenn der Drucker keine Daten empfangen kann geht diese auf HIGH.

Danach wird das zu übertragende Byte auf die Datenleitungen gelegt. Danach legt der PrintAdapter die STROBE Leitung für minimal 0,5 µs auf LOW. Das STROBE Signal muss mindestens 0,5 µs nach Anfang der Datensignale anfangen und mindestens 0,5 µs vor Ende der Datensignale wieder aufhören. Damit wird dem Drucker mitgeteilt, dass sich gültige Daten auf den Datenleitungen befinden. Während der Drucker die Daten verarbeitet, legt er die BUSY Leitung auf HIGH. Das muss spätestens 0,5 µs nach dem STROBE Signal geschehen. Wenn der Drucker das Byte erfolgreich empfangen hat, aktiviert er für 5 µs bis 10 µs das ACK Signal, um den Empfang zu bestätigen. Das sollte ca. 7 µs vor Ende des BUSY Signals geschehen und spätestens 7 µs nach dem BUSY Signal wieder deaktiviert werden. Ebenso sollte zwischen Ende des STROBE Signals und dem Anfang des ACK Signals nicht mehr als eine Sekunde vergehen, um eine schnelle Übertragung garantieren zu können. Die in diesem Text verwendeten Zahlen sind keine exakten Werte, da diese ohnehin nicht genau eingehalten werden müssen: Sowohl nach vorne, als auch nach hinten gibt es eine gewisse Toleranzgrenze. Der Palm PrintAdapter wartet zuerst das LOW auf der BUSY Leitung ab, legt dann die Daten auf die Datenleitungen, wartet etwa 10 µs, schaltet STROBE auf LOW, nach weiteren 10 µs wieder auf HIGH und wartet 30 µs. Die Zeiten wurden also auf Kosten der Übertragungsgeschwindigkeit großzügig gewählt, um eine sichere Datenübertragung zu gewährleisten.

Da das ACK Signal nur eine zusätzliche Hilfe und keine Notwendigkeit für das Handshake darstellt, arbeitet der PrintAdapter nur mit dem BUSY und dem STROBE Signal.

7. Aufbau, Funktionsweise und Zweck eines Mikrocontrollers

Ein Mikrocontroller ist im Prinzip ein normaler Prozessor wie er auch im PC vorkommt. Der Unterschied zwischen „normalem“ Prozessor und Mikrocontroller besteht darin, dass im Chip des Mikrocontrollers alles vorhanden ist, was man für eine kleine Applikation benötigt. Dazu zählen zum Beispiel der Speicher, häufig aber auch UART AD Wandler und integrierte EEPROMS. Es gibt sogar Mikrocontroller, die den Quarz schon integriert haben.

Für gewöhnlich haben Mikrocontroller folgende Eigenschaften:

- Geringer Preis
- Robust
- Meist viel geringere Leistung als ein „richtiger“ Prozessor
- Wichtige Hardware direkt auf dem Chip integriert
- Kleine Bauform
- Geringer Stromverbrauch

Der Zweck eines Mikrocontrollers besteht vor allem darin, kleine Steuerungsaufgaben leicht und schnell erledigen zu können. Der Entwickler braucht nur ein paar I/O zu Ports verbinden und den Mikrocontroller zu programmieren. Es ist nicht nötig, externen Speicher oder andere Peripherie über einen großen, komplexen Bus anzuschließen, damit er überhaupt funktioniert.

Zunächst findet man im Mikrocontroller den Prozessor, den RAM und eventuell noch andere integrierte Peripherie.

Der Prozessor besteht aus fünf wichtigen Komponenten: der ALU, den Registern, dem Befehlszähler und Befehlsregister, dem Flag Register und dem Adress- und Datenbus.

8. Eigenschaften und Programmierung des Atmel-Controllers

Die Mikrocontroller der Atmel Reihe AT90S besitzen eine RISC (Reduced Instruction Set Computer) Architektur. Im Gegensatz zur CISC (Complex Instruction Set Computer) Architektur hat ein Prozessor mit RISC Architektur viel weniger Befehle. Das hat Vor- und Nachteile. Im Jahre 1974 hat John Cocke von IBM herausgefunden, dass nur 20% der Befehle eines Prozessors häufig genützt werden und die anderen 80% überflüssig sind. Die Vorteile von RISC Prozessoren liegen in der Geschwindigkeit beim Abarbeiten der Befehle (bis zu 4x schneller). Ein anderer Vorteil ist, das RISC Prozessoren viel einfacher zu bauen sind und damit auch viel billiger. Ein gewaltiger Nachteil ist jedoch – vor allem bei Mikrocontrollern, wo der Speicherplatz begrenzt ist – dass der Code viel größer wird.

Die meisten Befehle des AT90S2333 werden in einem Taktzyklus ausgeführt. Dadurch schafft er eine Rechenleistung von 1 MIPS (Million Instruction Per Second) pro Megahertz. Bei sehr vielen Prozessoren wird die Taktfrequenz intern noch einmal geteilt, so dass viele Prozessoren trotz eines 20 Mhz Quarzes nur mit z.B. 10 Mhz arbeiten. Das ist bei den Atmel Prozessoren nicht der Fall.

Der Atmel hat 32 8-Bit Register von denen keiner eine besondere Funktion hat. Das bedeutet, dass der Programmierer alle 32 Register nach Belieben verwenden kann. Die einzige Einschränkung besteht darin, dass die Register 1 bis 16 nur für gewisse Befehle verwendet werden können. Ein weiterer großer Vorteil besteht darin, dass alle Register direkt mit der ALU verbunden sind, dadurch können zwei verschiedene Register in einem Befehl (der nur einen Taktzyklus dauert) verwendet werden.

Die Vorteile des AT90S2333 sind beträchtlich:

- Architektur: RISC mit 118 Befehlen, davon sind die meisten in nur einem Taktzyklus ausführbar; 32 8-Bit Register
- Reichlich Speicher: 2 kb Flash für das Programm, 128 Bytes SRAM als Arbeitsspeicher und 128 Bytes EEPROM
- Viel nützliche Peripherie: SPI Interface, UART, ADC, Watchdog Timer, Analog Comperator, 16 Bit Timer und Counter

Ein weiterer sehr wichtiger Vorteil ist die Fähigkeit, den Prozessor ohne externen Programmieradapter zu programmieren. Damit entfällt ein teures Programmiergerät, wie es zum Beispiel bei PIC Mikrocontrollern nötig ist. Die Atmel Mikrocontroller verwenden zum Programmieren die ISP (In System Programming) Technik. Mit dieser Technik ist es möglich, den Mikrocontroller zu programmieren, ohne ihn aus der Schaltung zu nehmen.

Der Entwickler sollte aber dennoch einen Schaltkreis wie den „Programmieradapter für Atmel AVR Prozessoren“[17] verwenden. Der 74HC244 auf dem Schaltkreis sorgt dafür, dass man auch andere Bauteile an die ISP Pins schließen kann. Es gibt aber auch Programmieradapter für den seriellen Port.

Eine einfache Alternative ist isp avr[18] von Bolger Buss. Dieser Programmieradapter kommt ohne IC aus und verbindet ganz simpel die Datenleitungen des ISP Interfaces mit der parallelen Schnittstelle. Der Nachteil an diesem Programmieradapter ist, dass nicht problemlos Peripherie an die Pins der ISP Schnittstelle angeschlossen werden können.

[17] http://rumil.de/hardware/avrisp.html

[18] http://people.freenet.de/buss/avr.html

9. Source Code

9.1. „makefile“

```
# part of Palm PrintAdapter
# (C) 2002 by Niki Hammler <nhammler@scc.co.at>
MCU = at90s2333
TRG = ser2par
SRC = $(TRG).c ringbuffer.c
ASRC =
LIB =
INC =
ASFLAGS = -Wa, -gstabs
CPFLAGS = -g  -Os -Wall -Wstrict-prototypes -Wa,-ahlms=$(<:.c=.lst) --save-
temps
LDFLAGS = -Wl,-Map=$(TRG).map,--cref
CONVERT = hexbin2.exe

include $(AVR)/avrfreaks/avr make
$(TRG).o : $(TRG).c
ringbuffer.o : ringbuffer.c
```

9.2. “ser2par.c”

```
/* main file of Palm PrintAdapter firmware
   Copyright (C) 2002 by Niki Hammler <nhammler@scc.co.at>
   This file is part of the "Palm PrintAdapter".

   The "Palm PrintAdapter" is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   The "Palm PrintAdapter" is distributed in the hope that it will be
   useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library
   General Public License for more details.

   You should have received a copy of the GNU Library General Public
   License along with the GNU C Library; see the file COPYING.LIB. If not,
   write to the Free Software Foundation, Inc., 59 Temple Place - Suite
   330, Boston, MA 02111-1307, USA.

   For more information on copyright see file LICENSE.txt
*/
```

```
/*
UART ist ausgelegt auf:
- 115200 Baud
- 7,3728 Mhz Quarz

Der Parallelport und der Rest ist verbunden wie in der Skizze:
*/
/* ser2par.c
 *
 *
 * Beschreibung der Schaltung
 * ==========================
 *
 *         SW <-- RESET\ |           | PC5  --> STROBE
 *     MAX232 <--    RxD |           | PC4  --> BIT1
 *     MAX232 <--    TxD |           | PC3  --> BIT2
 *                   PD2 |           | PC2  --> BIT3
 *                   PD3 |           | PC1  --> BIT4
 *        CTS <--    PD4 |           | PC0  --> BIT5
 *        VCC <--    VCC |   Atmel   | AGND --> GND
 *        GND <--    GND |           | AREF
 *      4 Mhz <--  XTAL1 |           | AVCC --> VCC
 *                 XTAL2 |           | SCK  --> BIT6
 *                   PD5 |           | MISO --> BIT7
 *                   PD6 |           | MOSI --> BIT8
 *    Jumper  <--    PD7 |           | PB2  --> BUSY
 *       LED  <--    PB0 |           | PB1  --> ERROR
 *                           AT90S2333
 *
 *
 */

#include <io.h>                       // Zugriff auf I/O Ports
#include <interrupt.h>                // Interrupt Unterstützung (UART)
#include <sig-avr.h>                  // SIGNAL() Makro
#include "ringbuffer.h"               // Ringpuffer Definitionen

#define ON                      1     // Status "ein" für Handshake
#define OFF                     0     // Status "aus" für Handshake
#define X ON                   17     // Software HSK ein
#define X OFF                  19     // Software HSK aus

#define STROBE                  5
#define BUSY                    2
#define BIT5                    0
#define BIT4                    1
#define BIT3                    2
#define BIT2                    3
#define BIT1                    4
#define BIT6                    5
#define BIT7                    4
#define BIT8                    3
#define ERROR                   1

#define CTS                     4
#define LED                     0
#define XONXOFF                 7
```

```
unsigned char handshake stat = 0;     // Ist Empfang durch Handshake
                                      // blockiert?
unsigned char software hsk = 0;       // Software Handshake aktiviert?

#if 0
/*
 * Alte Version von rwait(). Das naked-Attrebut läßt die Funktion durch
 * einfachen jmp ausführen, wodurch keine Zeit vergeudet wird.
 * Wegen besserer Präzesion jetzt als Makro
 */
  attribute  ((naked)) void rwait(void)
{
  asm volatile(
  "ldi  R31, 0xff" "\n" "WGLOOPx:  dec  R31" "\n\r"
  "brne WGLOOPx");
}
#endif

/*
 * Dieses Makro wartet ein wenig (in etwa 10us).
 * Wichtig für die timings des \STROBE Signals
 */
//für 4Mhz
//#define rwait() asm volatile("nop" "\n\t" "nop"   \
"\n\t" "nop" "\n\t" "nop" "\n\t" "nop" "\n\t" "nop" \
"\n\t" "nop" "\n\t" "nop" "\n\t" "nop" "\n\t" "nop" \
"\n\t" "nop");

//für 7,3728Mhz (das doppelte wie oben-1 ca)
#define rwait() asm volatile("nop" "\n\t" "nop"     \
"\n\t" "nop" "\n\t" "nop" "\n\t" "nop" "\n\t" "nop" \
"\n\t" "nop" "\n\t" "nop" "\n\t" "nop" "\n\t" "nop" \
"\n\t" "nop" "\n\t" "nop" "\n\t" "nop" "\n\t" "nop" \
"\n\t" "nop" "\n\t" "nop" "\n\t" "nop" "\n\t" "nop" \
"\n\t" "nop" "\n\t" "nop" "\n\t" "nop");

/*
 * Stellt die Baudrate ein. 0x09 ist die Adresse des Registers UBRR
 */

// 9600baud @ 4Mhz UBRR=25
// #define baud()                                   \
asm volatile("ldi r16,4000000/(9600*16)-1" "\n\t" "out 0x09,r16")

// 115200baud @ 7,3728Mhz UBRR=3
#define baud() asm volatile("ldi r16,3" "\n\t" "out 0x09,r16")


/*
 * Wartet genau eine Sekunde, durch die doppelte Taktfrequenz etwas
 * mehr als 0,5 Sekunden
 */
void eine sekunde(void)
{
  asm volatile(
  "ldi  R17, 0x24"               "\n\t"
  "WGLOOP0:  ldi  R18, 0xBC"     "\n\t"
```

```
  "WGLOOP1:  ldi  R19, 0xC4"     "\n\t"
  "WGLOOP2:  dec  R19"           "\n\t"
  "brne WGLOOP2"                 "\n\t"
  "dec  R18"                     "\n\t"
  "brne WGLOOP1"                 "\n\t"
  "dec  R17"                     "\n\t"
  "brne WGLOOP0"                 "\n\t"
  "ldi  R17, 0x01"               "\n\t"
  "WGLOOP3:  dec  R17"           "\n\t"
  "brne WGLOOP3"                 "\n\t"
  "nop"                          "\n\t");
}

/*
 * Sendet ein Byte nach Centronics Spezifikation
 * Wartet bis BUSY LOW ist, damit gesendet werden kann, setzt
 * die I/O Ports richtig, wartet ein wenig, erzeugt ein
 * LOW Signal an STROBE und wartet wieder ein wenig.
 * 1. Argument: Das zu sendende Zeichen
 * return-Wert: -
 */
void send par(unsigned char in)
{
  while(bit is set(PORTB, BUSY));     // Warten, bis BUSY LOW

  if((in & 0x01) == 0x01) sbi(PORTC, BIT1); else cbi(PORTC, BIT1);
  if((in & 0x02) == 0x02) sbi(PORTC, BIT2); else cbi(PORTC, BIT2);
  if((in & 0x04) == 0x04) sbi(PORTC, BIT3); else cbi(PORTC, BIT3);
  if((in & 0x08) == 0x08) sbi(PORTC, BIT4); else cbi(PORTC, BIT4);
  if((in & 0x10) == 0x10) sbi(PORTC, BIT5); else cbi(PORTC, BIT5);
  if((in & 0x20) == 0x20) sbi(PORTB, BIT6); else cbi(PORTB, BIT6);
  if((in & 0x40) == 0x40) sbi(PORTB, BIT7); else cbi(PORTB, BIT7);
  if((in & 0x80) == 0x80) sbi(PORTB, BIT8); else cbi(PORTB, BIT8);

  rwait();                            // 10us warten
  cbi(PORTC, STROBE);                 // STROBE von HIGH auf LOW schalten
  rwait();                            // 10us warten
  sbi(PORTC, STROBE);                 // STROBE von LOW auf HIGH schalten
  rwait();                            // 10us warten
  rwait();                            // 10us warten
  rwait();                            // 10us warten
}

/*
 * Schaltet den Handshake ein oder aus. Argumente sind ON oder OFF
 * 1. Argument: ON oder OFF
 * return-Wert: -
 */
void handshake(unsigned char state)
{
  if(state==ON)                       // Wenn eingeschaltet werden soll
  {
    sbi(PORTD, CTS);                  // ->HSK auf HIGH
    if(software hsk)                  // Falls Software Handshake gewählt
    {
      while(!bit is set(UCSRA, UDRE));// Warte bis gesendenet werden kann
      outp(X ON, UDR);                // Und sende ON Sequenz
    }
```

```
    handshake stat = ON;              // Ab jetzt kein Empfang mehr...
  }
    else                              // Wenn ausgeschaltet werden soll
  {
    cbi(PORTD, CTS);                  // CTS wieder LOW legen
    if(software hsk)                  // Falls Software HSK eingestellt
    {
      while(!bit is set(UCSRA, UDRE));// Warten bis UART bereit
      outp(X OFF, UDR);               // schicke eine OFF Sequenz
    }
    handshake stat = OFF;             // Es kann wieder empfangen werden
  }
}

/*
 * Die Interrupt Handling Routine
 * Speichert empfangenes Byte im 33Byte großen Ring Puffer
 */
SIGNAL(SIG UART RECV)
{
  register unsigned char got;         // Speicherplatz im uC (Register)
  got = inp(UDR);                     // Hole das Byte ab...
  if(!rb put(got)) handshake(ON);     // Und speichere es im Ringpuffer.
                                      // Falls der Puffer voll ist,
                                      // speichert er dieses byte noch und
                                      // schaltet dann den Handshake ein
}

/*
 * Die main() Routine
 * Hier wird am Anfang oder nach einem RESET begonnen
 */
int main(void)
{
  unsigned char rec;                  // Temporärer Speicherplatz
  // UART aktivieren
  sbi(UCSRB, RXEN);                   // Empfang aktivieren
  sbi(UCSRB, TXEN);                   // Senden aktivieren
  sbi(UCSRB, RXCIE);                  // Erweiterten Empfang einschalten
  baud();                             // Baudrate setzen

  // Den Datenport als Ausgang konfigurieren
  // die gesamte rechte seite an datenports == ausgabe
  sbi(DDRC, BIT5);                    // Bit 5
  sbi(DDRC, BIT4);                    // Bit 4
  sbi(DDRC, BIT3);                    // Bit 3
  sbi(DDRC, BIT2);                    // Bit 2
  sbi(DDRC, BIT1);                    // Bit 1
  sbi(DDRC, STROBE);                  // STROBE
  sbi(DDRB, BIT6);                    // Bit 6
  sbi(DDRB, BIT7);                    // Bit 7
  sbi(DDRB, BIT8);                    // Bit 8

  // Restliche Ausgänge konfigurieren
  sbi(DDRD, CTS);                     // CTS als Ausgang
  sbi(DDRB, LED);                     // LED als Ausgang

  // Alle Eingänge konfigurieren
```

```
cbi(DDRB, BUSY);                    // BUSY als Eingang
cbi(DDRB, ERROR);                   // Error als Eingang
cbi(DDRD, XONXOFF);                 // Xon/Xoff Jumper als Eingang


// Alle Ausgangsports auf Standard setzen
cbi(PORTC, BIT5);                   // Bit 5
cbi(PORTC, BIT4);                   // Bit 4
cbi(PORTC, BIT3);                   // Bit 3
cbi(PORTC, BIT2);                   // Bit 2
cbi(PORTC, BIT1);                   // Bit 1
cbi(PORTB, BIT6);                   // Bit 6
cbi(PORTB, BIT7);                   // Bit 7
cbi(PORTB, BIT8);                   // Bit 8

sbi(PORTC, STROBE);                 // STROBE auf HIGH (Standard) setzen
cbi(PORTD, CTS);                    // CTS 0 setzen

// Zur Bereitschaft LED einmal kurz blinken lassen
cbi(PORTB, LED);                    // LED einschalten

handshake stat = OFF;               // Am Anfang können wir empfangen...

if(!bit is set(PIND, XONXOFF))      // Der Jumper ist GESETZT
{
  software hsk = 1;                 // Software Handshake AN
  // Bei Software Handshake blinkt das LED kurz (insg. ~1.5 Sek.)
  eine sekunde();
  sbi(PORTB, LED);
  eine sekunde();
  cbi(PORTB, LED);
}
  else
{
  software hsk = 0;                 // Software Handshake AUS
}

sei();                              // Das UART/Interrupts aktivieren

/*
 * Die Hauptschleife. Endlosschleife.
 * Sie holt die Daten aus dem Ringpuffer und gibt sie
 * am Parallelport aus. Managed auch Handshake
 */
while(1)
{
  if(!rb get(&rec))
  {
    // Der Puffer ist leer --> continue
    // Transfer durch Handshake blockiert?
    if(handshake stat) handshake(OFF);
  }
    else
  {
    // Im Puffer ist ein Byte!
    send par(rec);
  }
}
```

```
  return 0;                             // wird nie erreicht
}
```

9.3. “ringbuffer.c”

```
/* Implementation of a simple ringbuffer
   Copyright (C) 2002 by Niki Hammler <nhammler@scc.co.at>
   This file is part of the "Palm PrintAdapter".

   The "Palm PrintAdapter" is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   The "Palm PrintAdapter" is distributed in the hope that it will be
   useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Library General Public License for more details.

   You should have received a copy of the GNU Library General Public
   License along with the GNU C Library; see the file COPYING.LIB.  If not,
   write to the Free Software Foundation, Inc.,
   59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

   For more information on copyright see file LICENSE.txt
*/

#include "ringbuffer.h"

static unsigned char iput = 0;        // Schreibzeiger
static unsigned char iget = 0;        // Lesezeiger
static unsigned char n = 0;           // Anzahl Bytes im Puffer
static unsigned char buffer[BUFFER];  // Ringpuffer

/*
 * Initialisiert alle globalen Variablen.
 * 1. Argument: -
 * return-Wert: -
 */
void init buffer(void)
{
  iput = 0;
  iget = 0;
  n = 0;
}
/*
 * Gibt den neuen Lese/Schreibzeiger zurück.
 * 1. Argument: Derzeitiger Wert von iget/iput
 * return-Wert: Neuer Wert von iget/iput
 */
unsigned char addring(unsigned char i)
{
```

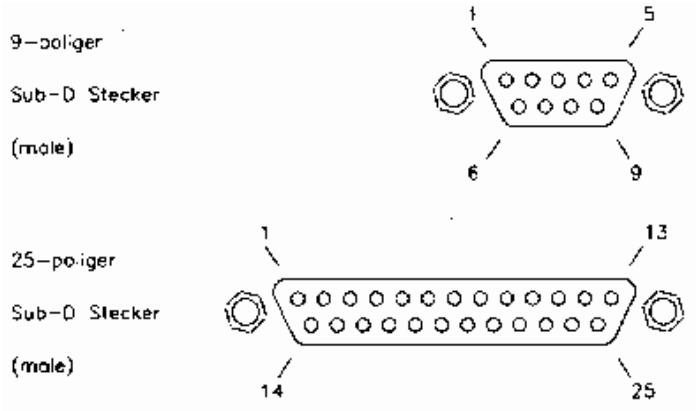
```
  return (i+1) == BUFFER ? 0 : i + 1;
}

/*
 * Liest ein Byte vom Ringpuffer
 * 1. Argument: Zeiger auf Speicher, in die das Byte geschrieben werden
 *              soll
 * return-Wert: TRUE wenn noch Zeicher im Puffer, FALSE wenn Puffer leer
 */
unsigned char rb get(unsigned char *tosave)
{
  unsigned char pos;
  if(n > 0)
  {
    pos = iget;
    iget = addring(iget);
    n--;
    *tosave = buffer[pos];
    return 1;
  }
  return 0;
}

/*
 * Schreibt ein Byte in den Ringpuffer
 * 1. Argument: Das zu speichernde Byte
 * return-Wert: TRUE wenn noch Platz, FALSE wenn Puffer voll
 */
unsigned char rb put(unsigned char z)
{
  if(!(n < BUFFER)) return 0;
  buffer[iput] = z;
  iput = addring(iput);
  n++;
  if(n < BUFFER) return 1;
  return 0;
}
```

9.4. “ringbuffer.h”

```
/* Header file for the ringbuffer implementation
   Copyright (C) 2002 by Niki Hammler <nhammler@scc.co.at>
   This file is part of the "Palm PrintAdapter".

   The "Palm PrintAdapter" is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   The "Palm PrintAdapter" is distributed in the hope that it will be
   useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Library General Public License for more details.

   You should have received a copy of the GNU Library General Public
   License along with the GNU C Library; see the file COPYING.LIB.  If not,
   write to the Free Software Foundation, Inc.,
   59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

   For more information on copyright see file LICENSE.txt
*/

#ifndef   RINGBUFFER
#define   RINGBUFFER

#define BUFFER                40

void init buffer(void);
unsigned char addring(unsigned char i);
unsigned char rb get(unsigned char *tosave);
unsigned char rb put(unsigned char z);


#endif
```

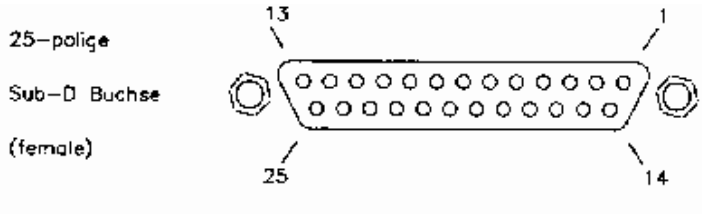
10. Anhänge

10.1 RS232 Pinbelegungen



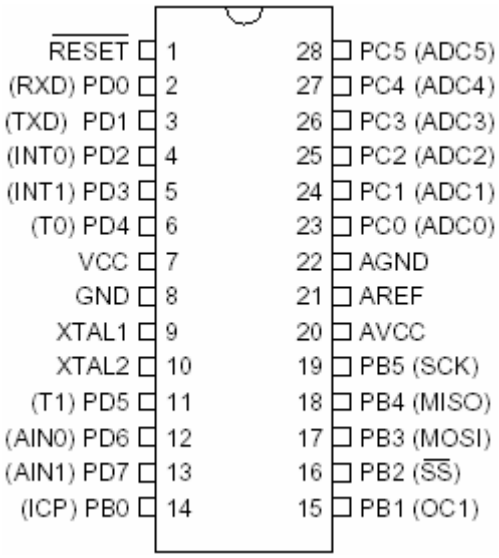
Funktion	Pin	
	9polig	25polig
Ground	5	1,7
Transmit Data	3	2
Receive Data	2	3
Request to send	7	4
Clear to send	8	5
Data set ready	6	6
Carrier detect	1	8
Check modem	-	9
Data terminal ready	4	20
Ring indication	9	22

10.2 Centronics Pinbelegungen



Funktion	Pin
Strobe	1
Bit 1	2
Bit 2	3
Bit 3	4
Bit 4	5
Bit 5	6
Bit 6	7
Bit 7	8
Bit 8	9
Acknowledge	10
Busy	11
Paper out	12
Select out	13
Autofeed	14
Error	15
Drucker Reset	16
Select in	17
Masse	18-25

10.3 Pinbelegung des AT90S23333



Signal	Beschreibung
RESET	Wenn LOW dann Reset
PD0	Datenempfang über UART
PD1	Senden von Daten mit UART
PD2	NC; Externer Interrupt 1
PD3	NC; Externer Interrupt 2
PD4	CTS Handshake
VCC	+5 Volt Stromversorgung
GND	Signal Ground
XTAL1	Quarz Eingangssignal
XTAL2	Quarz Ausgangssignal
PD5-6	NC
PD7	Konfigurationsjumper
PB0	Kontroll- LED
PC5	STROBE
PC4-0	Datenbits 1-5
AGND	Signal Ground
AREF	NC
AVCC	+5V Stromversorgung
PB5-3	Datenbits 6-8
PB2	BUSY
PB1	ERROR

10.4 Tabelle für das UBRR Register

For standard crystal frequencies, the most commonly used baud rates can be generated by using the UBRR settings in Table 15. UBRR values which yield an actual baud rate differing less than 2% from the target baud rate, are bold in the table. However, using baud rates that have more than 1% error is not recommended. High error ratings give less noise resistance.

Table 15. UBRR Settings at Various Crystal Frequencies

Baud Rate	1 MHz		% Error	1.8432 MHz		% Error	2 MHz		% Error	2.4576 MHz		% Error
2400	UBRR=	**25**	**0.2**	UBRR=	**47**	**0.0**	UBRR=	**51**	**0.2**	UBRR=	**63**	**0.0**
4800	UBRR=	**12**	**0.2**	UBRR=	**23**	**0.0**	UBRR=	**25**	**0.2**	UBRR=	**31**	**0.0**
9600	UBRR=	6	7.5	UBRR=	**11**	**0.0**	UBRR=	**12**	**0.2**	UBRR=	**15**	**0.0**
14400	UBRR=	3	7.8	UBRR=	**7**	**0.0**	UBRR=	8	3.7	UBRR=	10	3.1
19200	UBRR=	2	7.8	UBRR=	**5**	**0.0**	UBRR=	6	7.5	UBRR=	**7**	**0.0**
28800	UBRR=	1	7.8	UBRR=	**3**	**0.0**	UBRR=	3	7.8	UBRR=	4	6.3
38400	UBRR=	1	22.9	UBRR=	**2**	**0.0**	UBRR=	2	7.8	UBRR=	**3**	**0.0**
57600	UBRR=	0	7.8	UBRR=	**1**	**0.0**	UBRR=	1	7.8	UBRR=	2	12.5
76800	UBRR=	0	22.9	UBRR=	1	33.3	UBRR=	1	22.9	UBRR=	**1**	**0.0**
115200	UBRR=	0	84.3	UBRR=	**0**	**0.0**	UBRR=	0	7.8	UBRR=	0	25.0

Baud Rate	3.2768 MHz		% Error	3.6864 MHz		% Error	4 MHz		% Error	4.608 MHz		% Error
2400	UBRR=	**84**	**0.4**	UBRR=	**95**	**0.0**	UBRR=	**103**	**0.2**	UBRR=	**119**	**0.0**
4800	UBRR=	**42**	**0.8**	UBRR=	**47**	**0.0**	UBRR=	**51**	**0.2**	UBRR=	**59**	**0.0**
9600	UBRR=	**20**	**1.6**	UBRR=	**23**	**0.0**	UBRR=	**25**	**0.2**	UBRR=	**29**	**0.0**
14400	UBRR=	**13**	**1.6**	UBRR=	**15**	**0.0**	UBRR=	16	2.1	UBRR=	**19**	**0.0**
19200	UBRR=	10	3.1	UBRR=	**11**	**0.0**	UBRR=	**12**	**0.2**	UBRR=	**14**	**0.0**
28800	UBRR=	**6**	**1.6**	UBRR=	**7**	**0.0**	UBRR=	8	3.7	UBRR=	**9**	**0.0**
38400	UBRR=	4	6.3	UBRR=	**5**	**0.0**	UBRR=	6	7.5	UBRR=	7	6.7
57600	UBRR=	3	12.5	UBRR=	**3**	**0.0**	UBRR=	3	7.8	UBRR=	**4**	**0.0**
76800	UBRR=	2	12.5	UBRR=	**2**	**0.0**	UBRR=	2	7.8	UBRR=	3	6.7
115200	UBRR=	1	12.5	UBRR=	**1**	**0.0**	UBRR=	1	7.8	UBRR=	2	20.0

Baud Rate	7.3728 MHz		% Error	8 MHz		% Error	9.216 MHz		% Error	11.059 MHz		% Error
2400	UBRR=	**191**	**0.0**	UBRR=	**207**	**0.2**	UBRR=	**239**	**0.0**	UBRR=	287	-
4800	UBRR=	**95**	**0.0**	UBRR=	**103**	**0.2**	UBRR=	**119**	**0.0**	UBRR=	**143**	**0.0**
9600	UBRR=	**47**	**0.0**	UBRR=	**51**	**0.2**	UBRR=	**59**	**0.0**	UBRR=	**71**	**0.0**
14400	UBRR=	**31**	**0.0**	UBRR=	**34**	**0.8**	UBRR=	**39**	**0.0**	UBRR=	**47**	**0.0**
19200	UBRR=	**23**	**0.0**	UBRR=	**25**	**0.2**	UBRR=	**29**	**0.0**	UBRR=	**35**	**0.0**
28800	UBRR=	**15**	**0.0**	UBRR=	16	2.1	UBRR=	**19**	**0.0**	UBRR=	**23**	**0.0**
38400	UBRR=	**11**	**0.0**	UBRR=	**12**	**0.2**	UBRR=	**14**	**0.0**	UBRR=	**17**	**0.0**
57600	UBRR=	**7**	**0.0**	UBRR=	8	3.7	UBRR=	**9**	**0.0**	UBRR=	**11**	**0.0**
76800	UBRR=	**5**	**0.0**	UBRR=	6	7.5	UBRR=	7	6.7	UBRR=	**8**	**0.0**
115200	UBRR=	**3**	**0.0**	UBRR=	3	7.8	UBRR=	**4**	**0.0**	UBRR=	**5**	**0.0**

Literaturverzeichnis

[1] Linux/UNIX Systemprogrammierung
Helmut Herold
ISBN 3-8273-1512-3
Verlag Addison-Wesley

[2] C/C++ Kompendium
Dirk Louis
ISBN 3-8272-5669-0
Verlag Markt&Technik

[3] Datenblatt des AT90S2333

[4] Datenblatt des MAX232

[5] DALLAS Semiconductor
Application Note 83
Fundamentals of RS-232 Serial Communications

[6] PC Aufrüsten und Reparieren
Schüller, Veddeler
ISBN 3-89011-562-4
Verlag Data Becker

Bildnachweis

[Umschlag]	Nikolaus Hammler
[Abb. 1]	Schaltplan des PrintAdapter, Nikolaus Hammler
[Abb. 2]	Inverter aus einem NPN Transistor, Nikolaus Hammler
[Abb. 3]	Verschiedene RS232 Stecker, http://www.ats-vienna.com/rs232 2.php
[Abb. 4]	Signaldiagramm RS232, Nikolaus Hammler
[Abb. 5]	Signaldiagramm Centronics, Nikolaus Hammler