Programmieren des AVR Microcontrollers mit GCC, libc 1.0.4

von Guido Socher (homepage)

Über den Autor:

Guido mag Linux, weil es ein wirklich gutes System ist, um seine eigene Hardware zu entwickeln.

Zusammenfassung:

Der AVR 8−Bit RISC Microcontroller von Atmel ist ein sehr verbreiteter Microcontroller. Es ist nur ein Chip mit EEPROM, Ram, Analog zu Digital Umwandler, einer Menge digitaler Ein− und Ausgabezeilen, Timern, UART für RS 232 communication und vielen anderen Dingen.

Das beste ist jedoch, dass es dafür unter Linux eine vollständige Programmierumgebung dafür gibt: Man kann diesen Microcontroller in C programmieren, mit dem GCC Compiler.

Ich habe schon im März 2002 einen Artikel über dasselbe Thema geschrieben. Eine Menge Dinge haben sich seitdem in der avr−libc Entwicklung geändert und der AT90S4433 Microcontroller, den ich 2002 benutzt habe, wird nicht länger von Atmel hergestellt. Dies ist deshalb ein Update des März 2002 Artikels. Ich benutze libc−1.0.4 und den ATmega8 Microcontroller.

Dieser Artikel soll nur eine Einführung sein und in späteren Artikel werden wir wieder interessante Hardware bauen, die aber dies Mal auf dem ATmega8 basierend.

Einleitung

Viele Leute waren am Programmieren des Atmel Microcontrollers interessiert, als ich 2002 den Artikel schrieb. Der erste Schritt, die Entwicklungsumgebung zum Laufen zu kriegen, ist jedoch der schwerste. Wenn etwas nicht funktioniert, hat man absolut keinen Hinweis, wo der Fehler liegen könnte. Liegt es am Programmierkabel?? Ein Fehler in der Schaltung? Fehlerhafte Installation? Ist der Parallelport im Bios abgeschaltet? Sind die Kernelmodule für ppdev falsch kompiliert? Es gibt eine Menge Gründe, warum es nicht funktioniert.

Um den Einstieg in die aufregende Welt der Microcontroller einfacher zu machen, gibt es bei shop.tuxgraphics.org jetzt eine bootbare CD mit einem kleinen Handbuch und der Programmierhardware. Alles, was du dann machen mußt, ist, von der CD zu booten und alles läuft. Es ist keine Softwareinstallation erforderlich und auf deinem Computer wird nichts geändert.

Selbst ich benutze seit einer Weile so eine CD, da die Hardware, die ich baue, oft einige Generationen von Kerneln und Softwareinstallationenen auf meinem PC überlebt. Wenn ich dann später die Microcontrollersoftware updaten will, brauche ich mir keine Gedanken darüber zu machen, ob meine Entwicklungsumgebung auf meinem Linux−PC noch läuft. Ich boote einfach von der CD und alles läuft.

Unabhängig von der CD erkläre ich in den nächsten Abschnitten die Installation der GCC avr Entwicklungsumgebung. Wenn du die CD von tuxgraphics hast, dann lies ab "Ein kleines Testprojekt" weiter.

Softwareinstallation: Was man dazu braucht

Um die GNU C Entwicklungsumgebung benutzen zu können, braucht man die folgende Software:

binutils−2.15.tar.bz2	Verfügbar unter: ftp://ftp.gnu.org/gnu/binutils/ oder einem der Mirrors. Z.B. ftp://gatekeeper.dec.com/pub/GNU/binutils/
gcc−core−3.4.2.tar.bz2	Verfügbar unter: ftp://ftp.gnu.org/gnu/gcc/ oder einem der Mirrors. Z.B. ftp://gatekeeper.dec.com/pub/GNU/gcc/
avr−libc−1.0.4.tar.bz2.tar	Die AVR C−library ist verfügbar unter: http://savannah.nongnu.org/projects/avr−libc/
uisp−20040311.tar.bz2	Die AVR Programmiersoftware gibt es auf: http://savannah.nongnu.org/projects/uisp

Wir installieren alle Programme nach /usr/local/avr. Dadurch halten wir das Programm vom normalem Linux C Compiler getrennt. Erzeuge dieses Verzeichnis mit dem Befehl:

```
    mkdir /usr/local/avr

Du kannst es jetzt schon zu deinem PATH hinzufügen:
    mkdir /usr/local/avr/bin
    export PATH=/usr/local/avr/bin:${PATH}
```

Softwareinstallation: GNU binutils

Das binutils Paket enthält alle nötigen low−level Utilities, um Objektdateien zu bauen. Es beinhaltet einen AVR assembler (avr−as), Linker (avr−ld), library handling tools (avr−ranlib, avr−ar), Programme, zum Erzeugen von Objektdateien, die auf das EEPROM des Microcontrollers (avr−objcopy) geladen werden können, disassembler (avr−objdump) und utilities wie avr−strip und avr−size.

Laß die folgenden Befehle laufen, um die binutils zu bilden und zu installieren:

```
tar jxvf binutils-2.15.tar.bz2
cd binutils-2.15/
mkdir obj-avr
cd obj-avr
../configure --target=avr --prefix=/usr/local/avr --disable-nls
make

# as root:
make install
```

Füge die Zeile /usr/local/avr/lib zu der Datei /etc/ld.so.conf hinzu und laß den Befehl /sbin/ldconfig laufen, um den Linker−cache erneut zu bilden.

Softwareinstallation: AVR gcc

avr−gcc ist unser C Compiler.

Laß den folgenden Befehl zum Bilden und Installieren laufen:

```
tar jxvf gcc-core-3.4.2.tar.bz2
cd gcc-3.4.2

mkdir obj-avr
cd obj-avr
../configure --target=avr --prefix=/usr/local/avr --disable-nls --enable-language=c

make

# as root:
make install
```

Softwareinstallation: Die AVR C−library

Die C−library ist inzwischen ganz stabil, verglichen mit der, die ich im März 2002 vorgestellt habe. Laß den folgenden Befehl zum Bilden und Installieren laufen:

```
tar jxvf avr-libc-1.0.4.tar.bz2.tar
cd avr-libc-1.0.4
PREFIX=/usr/local/avr
export PREFIX
sh -x ./doconf
./domake
```

```
cd build
#as root:
make install
```

Softwareinstallation: Der Programmierer

Die Programmiersoftware lädt den speziell preparierten Objektcode in das EEPROM unseres Microcontrollers.

Der uisp Programmierer für Linux ist ein sehr guter Programmierer. Er kann direkt von einem Makefile aus benutzt werden. Man fügt einfach eine "make load" Regel hinzu und schon kann man die Software in einem kompilieren und laden.

uisp wird wie folgt installiert:

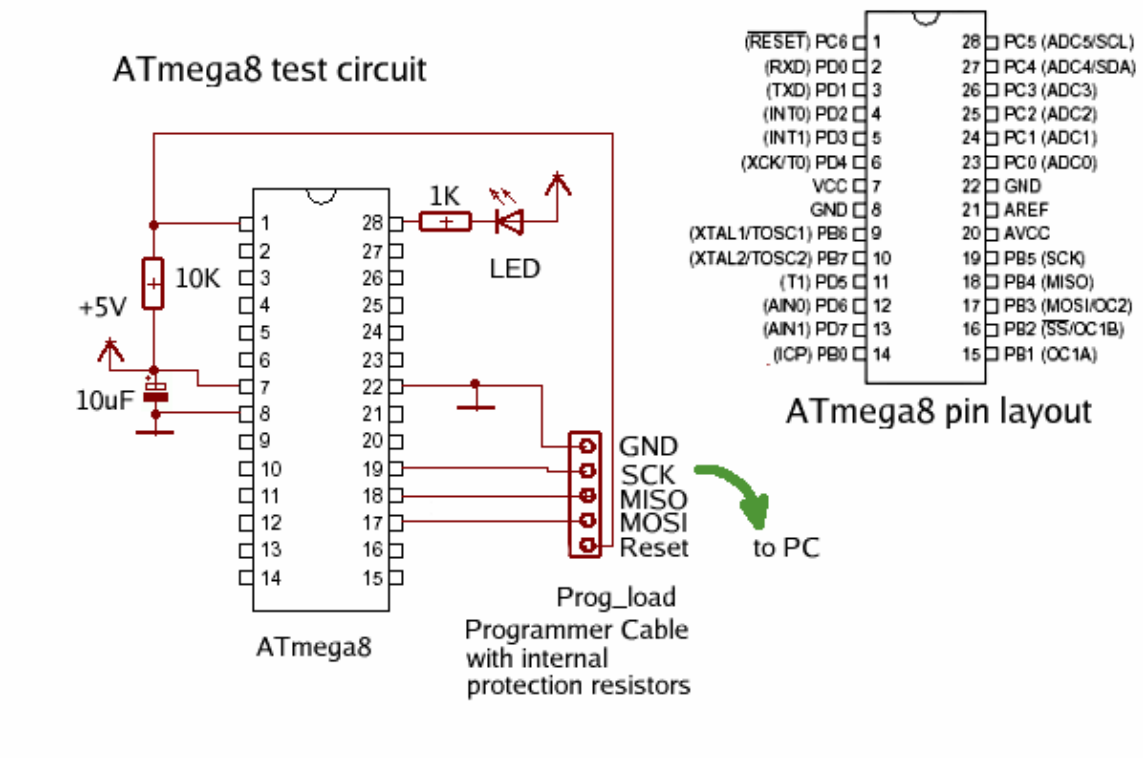
```
tar jxvf uisp-20040311.tar.bz2.tar
cd uisp-20040311
./configure --prefix=/usr/local/avr
make

# as root:
make install
```

Ein kleines Testprojekt

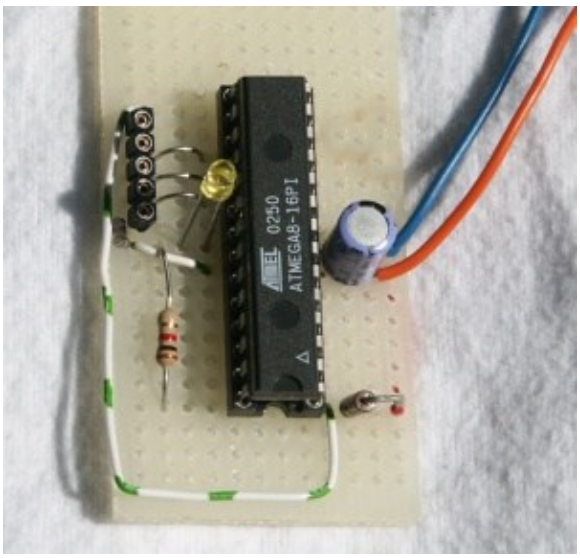
Wir beginnen mit einer kleinen Testschaltung, die du dann später erweitern kannst.

Dieses Schaltung kann auch als einfache Testumgebung für komplexere Hardware benutzt werden. Man kann einfach Software zum Testen laden und dann Sensoren oder Meßinstrumente anschließen.

Unser Testprogramm, so wie es hier dargestellt ist, bringt einfach eine LED zum Blinken.



Benötigte Hardware



Du brauchst die unten aufgeführten Teile. Obwohl es ein ganz gewöhnlicher Microcontroller ist, ist er eventuell nicht in jedem Elektroladen vor Ort zu haben, aber größere Distributoren für elektronische Komponenten wie (www.conrad.de (Deutschland), www.selectronic.fr (Frankreich), digikey.com (US, Canada), etc... haben sie vorrätig.
Du kannst sowohl den gesamten Bausatz als auch nur den Microcontroller bei shop.tuxgraphics.org bestellen.

1 x ATmega8 DIP version, Atmel 8 bit Avr risc processor.
1 x 28 pin 7.5mm IC Sockel Der 28 pin Sockel ist ein bißchen schwieriger zu bekommen. Normalerweise sind die 28 Sockel 14mm breit, aber wir brauchen einen 7.5mm Sockel.
1 x 10K Widerstand (Farbcode: braun, schwarz, orange) 1 x 1K Widerstand (Farbcode: braun, schwarz, rot) 1 x 10uF Elektrolytkondensator einigen Draht

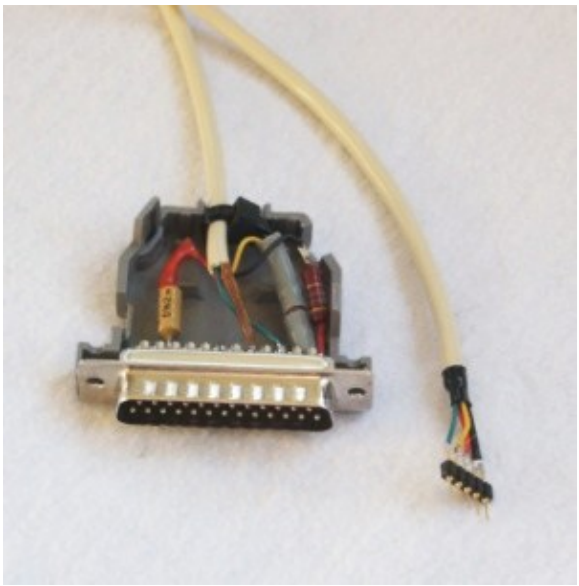
1 x LED matrix board
Das folgende wird für den Programmierer benötigt (es wird nicht gebraucht, wenn du dir das "Linux AVR programming kit" von tuxgraphics besorgst): 1 x DB25 Stecker, der in den Parallelport paßt. Irgendeinen 5 pin Stecker/Sockel für den Programmierer. Ich empfehle, Prezisonsstreifenstcker (ähnlich der IC Sockel) zu benutzen und 5 Pins abzubrechen. 1 x 220 Ohm Widerstand (Farbcode: rot, rot, braun) 3 x 470 Ohm Widerstand (Farbcode: gelb, lila. braun)

Zusätzlich zu den oben genannten Teilen brauchst du eine 5V elektronisch stabilisierte Spannungsversorgung oder man kann eine 4.5V Batterie als Stromversorgung benutzen.

Du hast wahrscheinlich bemerkt, daß wir keinen Kristall brauchen. Das ist so, weil der ATmega8 jetzt einen eingebauten Oszillator hat. Dieser Oszillator kann benutzt werden, wenn die genaue Taktfrequenz keine Rolle spielt. Wenn du jedoch präzise Meßinstrumente bauen willst oder die UART/RS232 Schnittstelle benutzen willst, dann brauchst du einen Kristall. Welcher Typ von Oszillator benutzt wird, kann durch die "fuse bits" definiert werden, die man mit dem Programmierer verändern kann. Ab Werk der interne 1Mhz Oszilator aktiv.

Bauen der Programmiererhardware

Die AVR Microcontroller erlauben in circuit programming (ISP).
D.h. man braucht den Microcontroller nicht von der Platine abmachen, um ihn zu programmieren. Es gibt verschiedene Programmierhardware zwischen 50−150 Euro. Mit Linux ist es jedoch möglich, einen sehr einfachen Programmierer zu bauen, der dasselbe macht. Man braucht einen freien Parallelport und das folgende Kabel.



Beachte, dass das hier ein verbesserter Programmierer ist im Vergleich zu dem im März 2002 Artikel vorgestellten. Wir bauen die Schutzwiderstände in den Programmierer ein. Das spart dann einiges an Platz und Teilen auf der Schaltungsplatine. Die Verdrahtung für das Programmiererkabel muß wie folgt sein:

Pin auf pcb	Pin auf AVR	Schutzwiderstand	Pin am Parallel Port
5	Reset (1)	−−	Init (16)
4	MOSI (17)	470 Ohm	D0 (2)
3	MISO (18)	220 Ohm	Busy (11)
2	SCK (19)	470 Ohm	Strobe (1)
1	GND	−−	GND (18)

Das Kabel sollte nicht länger as 70cm lang sein.

Die Schutzwiderstände können in den Stecker wie in dem Bild rechts eingebaut werden.

Software schreiben

Der Atmeag 8 kann in normalem C mit der Hilfe von gcc programmiert werden. Ein bißchen AVR assembler zu können, kann nützlich sein, ist aber nicht notwendig.

Die AVR libc kommt mit einem avr−libc−user−manual−1.0.4.pdf (1139921 bytes), das alle in C verfügbaren Funktionen dokumentiert. Auf Atmels Website, (www.atmel.com, und dann: avr products −> 8 bit risc−> Datasheets), kann man das gesamte Datenblatt herunterladen. Es beschreibt alle Register und wie man die CPU benutzt.

Eine Sache, die man beachten muß, wenn man einen Microcontroller benutzt, ist, dass er nur ein paar Bytes an Ram besitzt. D.h., man kann keine großen Datenstrukturen oder Strings deklarieren. Dein Programm sollte auch keine tief verschachtelten Funktionsaufrufe oder Rekursion benutzen.

Viel besser als alle Theorie ist ein richtiges Beispiel. Wir schreiben ein kleines Programm, das unsere LED in 0.5 Sekundenintervallen blinken läßt. Nicht sehr nützlich, aber sehr gut für den Anfang.

Die avr−libc hat sich sehr geändert. Früher hat man ein Bit auf einem Port mit sbi gesetzt und es mit cbi wieder gelöscht. Jetzt werden solche Funktionen abgelehnt. Zuerst stelle ich den "guten alten Weg" dar:

```
/* defines for future compatibility */
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif

void main(void)
{
      /* INITIALIZE */
      /* enable PC5 as output */
      sbi(DDRC,PC5);


      /* BLINK, BLINK ... */
      while (1) {
            /* led on, pin=0 */
            cbi(PORTC,PC5);
            delay_ms(500);
            /* set output to 5V, LED off */
            sbi(PORTC,PC5);
            delay_ms(500);
      }
}
```

Das folgende Beispiel macht genau dasselbe, benutzt aber die neue Syntax:

```
void main(void)
{
      /* INITIALIZE */
      /* enable PC5 as output */
      DDRC|= _BV(PC5);


      /* BLINK, BLINK ... */
```

```
        /* PC5 is 5 (see file include/avr/iom8.h) and _BV(PC5) is 00100000 */
        while (1) {
              /* led on, pin=0 */
              PORTC&= ~_BV(PC5);
              delay_ms(500);
              /* set output to 5V, LED off */
              PORTC|= _BV(PC5);
              delay_ms(500);
        }
}
```

Das obige Codestück zeigt, wie einfach es ist, ein Programm zu schreiben. Du siehst nur das Hauptprogramm, die delay_ms Funktion ist in dem kompletten Programm (avrm8ledtest.c) enthalten. Um Pin PC5 als Output zu benutzen, mußt du das PC5 Bit in dem Data Direction Register für port C (DDRC) setzen. Danach kannst du PC5 auf 0V setzen, mit der Funktion cbi(PORTC,PC5) (clear bit PC5) oder auf 5V mit sbi(PORTC,PC5) (set bit PC5). Der Wert von "PC5" ist in iom8.h definiert, das in io.h enthalten ist. Du mußt dir darüber keine Gedanken machen. Wenn du schon mal Programme für multi user / multi tasking Systeme wie Linux geschrieben hast, weißt du, dass man niemals eine nicht blockierende Endlosschleife schreiben darf. Dies wäre eine Verschwengung von CPU−Zeit und würde das System sehr verlangsamen. Im Falle des AVR ist das anders. Es gibt nicht mehrere Aufgaben und es läuft auch kein weiteres Programm. Es gibt nicht mal ein Betriebssystem. Es ist deshalb ganz normal, Schleifen für immer laufen zu lassen.

Kompilieren und Laden

Bevor du anfängst, stell sicher, dass du /usr/local/avr/bin im PATH stehen hast. Wenn nötig, ändere dein .bash_profile oder .tcshrc und füge folgendes hinzu:

export PATH=/usr/local/avr/bin:${PATH} (for bash)
setenv PATH /usr/local/atmel/bin:${PATH} (for tcsh)

Wir benutzen den Parallelport und uisp zum Programmieren des AVRs. Uisp benutzt das ppdev Interface des Kernels. Deshalb mußt du die folgenden Kernelmodule geladen haben:

```
# /sbin/lsmod
parport_pc
ppdev
parport
```

Überprüfe mit dem Befehl /sbin/lsmod, dass sie geladen sind, ansonsten lade sie (als root) mit:

```
modprobe parport
modprobe parport_pc
modprobe ppdev
```

Es ist eine gute Idee, diese Befehle automatisch beim Starten auszuführen. Du kannst sie zu einem rc script (z.B. für RedHat /etc/rc.d/rc.local) hinzufügen.
Um das ppdev Interface als normaler Benutzer zu benutzen, muß root dir Schreibrechte geben durch einmaliges Laufenlassen des Befehls

chmod 666 /dev/parport0

Stell auch sicher, dass kein Druckerdaemon auf dem Parallelport läuft. Wenn du einen laufen hast, dann stoppe ihn, bevor du das Programmierkabel anschließt. Jetzt ist alles fertig, um unseren Microcontroller zu

kompilieren und zu programmieren.

Das Paket für unser Testprogramm (avrm8ledtest−0.1.tar.gz) enthält ein Make−file. Alles, was du tun mußt, ist, das folgende zu tippen:

make
make load

Dies kompiliert und lädt deine Software. Ich werde die Befehle nicht im Detail beschreiben. Du kannst sie im Makefile sehen und sie sind immer gleich. Ich erinnere mich selber nicht mehr an alle. Ich weiß nur, daß ich "make load" benutzen muß. Wenn du ein anderes Programm schreiben willst, mußt du nur alle avrm8ledtest in dem Makefile durch den Namen deines Programms ersetzen.

Einige interessante binutils

Interessanter als der tatsächliche Komilationsprozess sind einige binutils.
Diese utilities haben sich jedoch seit März 2002 nicht geändert. Schau deshalb im Abschnitt "Einige interessante binutils" im Artikel 231, März 2002 nach.

Ideen und Anregungen

Der ATmega8 ist für die meisten Anwendungen zu dem AT90S4433 kompatibel. Du mußt die "fuse bits" programmieren, um den externen Oszillator zu benutzen und die früher dargestellte Hardware arbeitet möglicherweise mit nur kleinen Änderungen. Leider hatte ich noch nicht genügend Zeit, um alle Schaltungen nochmal für den ATmega8 zu testen. Wenn du auf der sicheren Seite sein willst, dann benutze den AT90S4433 für die alten Artikel. Wenn es dir nichts ausmacht, Fehler zu suchen und Probleme zu lösen, dann probier den ATmega8 mit den alten Artikeln/Schaltungen aus.

Hier ist eine Liste der früheren Hardwareartikel:

- Eine LCD Anzeige und Steuertasten für den Linux Server
- Ein Microcontroller gesteuertes Labornetzteil
- Frequenzzähler 1Hz−100Mhz mit LCD Display und RS232 Interface
- Linux USB LCD Display mit Watchdog und Tasten
- Bau eines autonomen Lichtfinder−Robots

Beachte, daß der hier dargestellte Programmierer schon die Schutzwiderstände enthält, die in den älteren Hardwareartikeln auf die Platine eingebaut wurden. Um den neuen Programmierer mit den alten Platinen zu benutzen, mußt du nur die Schutzwiderstände auf der Platine durch die Drähte ersetzen.

Referenzen

- Pascal Stangs AVRlib: http://www.procyonengineering.com/avr/avrlib/index.html
- Der tavrasm assembler für Linux: www.tavrasm.org
- **Alle Software und Dokumente , die in diesem Artikel erwähnt wurden**
- Die Atmel Website: www.atmel.com
- Elektronikseiten im tuxgraphics shop: shop.tuxgraphics.org
  (Hier kannst du die Linux AVR programming CD, kits und Microcontroller bekommen)

Der LinuxFocus Redaktion schreiben http://www.LinuxFocus.org	Autoren und Übersetzer: en −−> −− : Guido Socher (homepage)

2005−02−24, generated by lfparser_pdf version 2.51