

Einrichten einer Entwicklungsumgebung für die AVR-C-Programmierung unter Ubuntu-Linux

Ein Erfahrungsbericht

Ich habe schon mehrere Anläufe unternommen, von Windows auf Linux umzusteigen. Zunächst mit beiden Betriebssystemen, in der Hoffnung, irgendwann nur noch mit Linux zu arbeiten. Aber es kam immer umgekehrt. Das lag wohl daran, dass nach der Installation von Linux manches nicht so lief wie es sollte. Auch ist der Einstieg in so etwas wie die AVR-C-Programmierung mit WinAVR und Windows einfacher als mit Linux.

Nun habe ich mit Ubuntu 10.10 und der gerade erschienenen Version 11.04 wieder einen Umsteigeversuch unternommen. Und ich sehe gute Chancen, in Zukunft bei Linux zu bleiben, weil ich die AVR-C-Programmierung erfolgreich einrichten konnte und Ubuntu meiner Vorstellung, wie ein Betriebssystem sein müsste, sehr nahe kommt.

Toolchain

Die Programme, die man braucht, um Software, z.B. für einen Mikrocontroller (MC) zu erstellen und ihn zu programmieren, werden zusammengefasst als Toolchain bezeichnet. Da diese Programme wie Werkzeuge erscheinen, die nacheinander zum Einsatz kommen, haben sie den Charakter einer Kette, daher der Name. Ich arbeite mit den MCs der AVR-Familie von Atmel und der Programmiersprache C.

Mit dem Editor werden die Quelltexte der Programme geschrieben. Jeder Editor ist prinzipiell geeignet, kann aber mehr oder weniger vorteilhaft für den beabsichtigten Zweck sein.

Der C-Compiler *avr-gcc* ist der Kern der Toolchain für die Programmierung der Mikrocontroller der AVR-Familie. Compiler, die Programmcode erzeugen, der nicht für den PC gedacht ist, werden als Cross-Compiler bezeichnet. *avr-gcc* wird in der Regel auch als Linker aufgerufen, der aus Objekt-Code-Dateien (*.o), die in der Bibliothek *avr-libc* vorhanden oder durch das Compilieren entstanden sind, als Endergebnis eine Hex-Datei *.hex macht. *avr-libc* gehört auch zur Toolchain.

Nicht zu vergessen Debugger, von denen ich aber keinen Gebrauch mache.

Zur Programmierung des MC - das ist das Übertragen und Schreiben von Programmen in den MC-Speicher - dient das Programm *avrdude*, das wie *avr-gcc* ein Kommandozeilen-Programm ist. Es kommuniziert mit dem Mikrocontroller mit Hilfe eines Programmieradapters (Programmer), der an eine Schnittstelle des PCs angeschlossen ist. *avrdude* verwendet zum Programmieren die von *avr-gcc* erzeugte Hex-Datei.

Editor, *avr-gcc* mit der Bibliothek *avr-libc* und *avrdude* bilden meine Toolchain, aber *avr-gcc* ist ohne das Hilfsprogramm *make* für die praktische Arbeit nicht benutzbar. Das liegt an den unglaublich vielfältigen Optionen, mit denen *avr-gcc* aufzurufen ist. Es würde viel zu viel Arbeit machen, diese Optionen immer richtig einzugeben.

Statt dessen wird *make* aufgerufen, ein Programm, das zum Linux-„Inventar“ gehört und nicht installiert werden muss. Es entnimmt aus der Textdatei *makefile*, die im aktuellen Verzeichnis liegen muss, was für ein Programm mit welchen Optionen aufgerufen werden soll. Dabei kann *make* selbst auch mit verschiedenen Option aufgerufen werden, was z.B. bewirkt, dass nur *avrdude* aufgerufen wird, obwohl dessen Optionen vergleichsweise übersichtlich sind.

makefile selbst zu schreiben, sollte man sich nicht antun. Ein Muster bekommt man, wenn man das Programm *mfile* installiert, mit dem die wichtigsten Eintragungen in *makefile* gemacht werden können, ohne *makefile* zu editieren, was natürlich im Sonderfall auch möglich ist. Manchmal wird *mfile* deshalb auch als „Ausfüllhilfe“ bezeichnet.

Auch wenn man eine integrierte Entwicklungsumgebung verwendet, läuft darunter eine Toolchain, deren zentraler Bestandteil, sowohl unter Linux als auch unter Windows, der Compiler *avr-gcc* ist.

Wie es unter Windows war

Der Blick zurück ist nicht unwichtig, denn ich bin mir ziemlich sicher, dass mir das Einrichten der AVR-C-Programmierung unter Linux nur wegen der Erfahrungen, die ich unter Windows sammeln konnte, gelungen ist.

Zur AVR-C-Programmierung kam ich durch den kleinen Fahr-Roboter Asuro und die Entwicklungsumgebung WinAVR unter Windows. Ich habe dann auch C-Programme für andere Zwecke geschrieben und so wurde der zu verwendende Programmieradapter ein Thema. Lange arbeitete ich mit einem selbst gebauten für die Parallelschnittstelle, bestehend aus einigen Widerständen.

Man bekommt von der Toolchain unter WinAVR nicht viel mit. Aber es gibt sie, und zwar die oben beschriebene. Mit WinAVR wird auch gleich noch *mfile* installiert.

Aber ich bin nicht nur bei WinAVR geblieben. Atmel bietet AVRstudio4 für Windows, eine komfortable integrierte Entwicklungsumgebung, zum kostenlosen Download an, und die USB-Schnittstelle für den Anschluss eines Programmers ist natürlich viel zeitgemäßer. Also habe ich mir den USB-Programmieradapter AVRISP mk II von Atmel zugelegt und AVRstudio4 installiert.

Die Verbindung zum Programmer wollte zunächst nicht funktionieren. Dem Impuls, dann den Adapter unter WinAVR auszuprobieren und den entsprechenden Treiber aus LibUSB32 zu installieren, sollte man aber nicht nachgeben, wie ich es getan habe. Weder unter WinAVR noch unter AVRstudio4 lief der Adapter.

Im Gerätemanager der Systemsteuerung wurde der USB-Treiber von AVRstudio4 als defekt markiert. Er outete sich nach seiner Neuinstallation als Jungo-Treiber.

avrdude unter WinAVR kann mit dem AVRISP und der USB-Schnittstelle arbeiten, wenn der Treiber aus LibUSB32 installiert wird. Es hat ein bisschen gedauert bis ich aus den Foren entnehmen konnte, dass der Jungo-Treiber von AVRstudio4 und der aus LibUSB32 sich nicht vertragen. Schließlich verstand ich, dass es den Leuten nicht darum ging, überhaupt den AVRISP unter WinAVR zum Laufen zu kriegen, sondern wie man Filter-Treiber installiert, mit denen dieser gegenseitige Ausschluss beseitigt wird, was auch nicht problemlos zu sein scheint.

Ich finde dieses Nebeneinander bedeutungslos, zumal mein Interesse auf Linux zielt. Etwas Wichtiges habe ich aber daraus gelernt, nämlich wie man bei WinAVR, also für avrdude, die USB-Schnittstelle als Port für den Programmieradapter im makefile einträgt. Der Eintrag lautet: `usb:<seriennummer>`, wobei die Seriennummer auf dem Typenschild des Programmieradapters zu finden ist. mfile trägt im makefile bei Auswahl des Ports nur usb ein, ohne Hinweis, dass noch etwas fehlt.

Interessant ist auch, dass AVRstudio4 für C-Programme den Compiler avr-gcc verwendet. Er wird problemlos gefunden, wenn bereits WinAVR installiert ist. Da wundert es nicht, dass Programmtexte, die unter Windows erstellt wurden, ohne Änderung in Linux weiter verwendet werden können. Für makefile gilt das auch.

So wenig man mit der Toolchain unter Windows in Berührung kommt, umso mehr unter Linux, wo es durchaus möglich ist, auf der Kommandozeile ein Programm zu entwickeln und den MC zu programmieren. Das ist eher ein Vorteil, da man die Toolchain sehr gut kennenlernt.

Arbeiten auf der Kommandozeile

Sie wird von der Shell bereitgestellt. Das Programm dahinter heißt unter Ubuntu *bash* (es gibt auch andere). Mit *Anwendungen/Zubehör/Terminal* wird ein Terminal-Fenster geöffnet, das auch Konsole genannt wird. Es zeigt die Kommandozeile in Form des Prompts und des Schreibcursors.

Man braucht für die Installation der Programme root-Rechte, was normal ist. Leider aber auch für die Anwendung von avrdude. Das zu vermeiden, ist mir nicht gelungen. Darüber später mehr.

Bei Ubuntu ist es nicht möglich, sich als User root anzumelden. Ein entsprechendes Passwort gibt es deshalb nicht, und der Befehl `su` ist blockiert. Jedoch können User mit dem Befehl `sudo -s -H` Super-User root werden, wenn sie der Gruppe admin angehören und nach dem Aufruf ihr Passwort eingeben. Der User, der Linux installiert hat, ist automatisch Mitglied bei admin.

Es lohnt sich für den Umgang mit Dateien und Verzeichnissen auf der Kommandozeile den Midnight-Commander *mc* zu installieren. Er ist dem legendären Norton-Commader aus den Zeiten von DOS nachempfunden. Insbesondere kann mc auch den Kommandozeilen-Editor *nano* aufrufen.

Ein Blick auf die Paketverwaltung

Mit einer Linux-Distribution werden eine Reihe von alltäglichen Programmen installiert. Und man wird darüber informiert, dass so gut wie alles, was noch gebraucht werden könnte, von einer CD und aus dem Internet in Form von Paketen zu beziehen ist. Gemeint sind Paketquellen (auch repositories), die für die Distribution typisch sind und von denen das installierte Linux die wichtigsten auch kennt. In einem Paket ist alles zusammengefasst, was für die Installation gebraucht wird bis hin zur Dokumentation.

Tatsächlich ist es ganz leicht, mit einem Paketverwaltungsprogramm, die gewünschten Programme zu installieren. Meistens sind die Programme davon abhängig, dass auch zusätzliche Pakete installiert werden oder schon installiert sind. Diese Abhängigkeit kann ziemlich kompliziert sein, aber die Paketverwaltung kümmert sich um alles. Deshalb ist auch ein sauberes Deinstallieren möglich, weil von jeder Datei jederzeit festgestellt werden kann, zu welchem Paket sie gehört. „Rückstände“ wie bei Windows gibt es nicht.

Für eine Debian-Distribution oder eine, die darauf basiert, wie Ubuntu, heißt die grafische Paketverwaltung *synaptic*. Sie arbeitet auf der Basis aller Pakete, der bereits installierten ebenso wie die der bekannten lokalen und externen Quellen. Pakete können gefiltert, gesucht, vorgemerkt, installiert und deinstalliert werden.

Aber auch für die Kommandozeile gibt es eine debian-typische Paketverwaltung, und zwar den Befehl *apt-get*. Ruft man einfach ein Programm auf, das nicht installiert ist, z.B. *avrdude*, erhält man einen Hinweis, wie *apt-get* zwecks Installation aufzurufen ist.

Komplizierter wird die Sache, wenn man Paketquellen verwenden möchte, die zwar zur Distribution passen, aber dem installierten Linux unbekannt sind. Oder, wenn das Paket Quelltexte enthält, die kompiliert werden müssen.

Toolchain unter Linux installieren

Die Programme *avr-gcc* mit *avr-libc*, *avrdude* und *mfile* müssen installiert werden, den textorientierten Editor *nano* und den grafischen *gedit* sowie das Programm *make* gibt es bereits.

Nach dem Herunterladen und Entpacken installiert man als root auf der Kommandozeile.

avr-gcc und avrdude

Bei welchen Distributionen diese Pakete vorhanden sind, darüber erhält man Auskunft bei http://www.mikrocontroller.net/articles/AVR_und_Linux Ich habe beide mit *synaptic* bei den Ubuntu-Paketquellen gefunden und installiert. Die Warnung im ersten Abschnitt der Seite habe ich nicht ernst genommen und wurde prompt eines Besseren belehrt. Dateien, die *avr-gcc* benötigte, waren nicht vorhanden. Nach einigen vergeblichen Reparaturversuchen, gab ich auf.

Die Lösung steht aber auch im ersten Abschnitt der Seite. Unter <http://www.wrightflyer.co.uk/avr-gcc/> gibt es fertige AVR-Debian-Pakete, die funktionieren und als Datei herunter geladen werden können, in der Regel die neueste Version für 32-Bit-Rechner (i386). Die Datei [avr-gcc-4.3.4-avrfreaks-30-jun-2010-u10.04.i386.deb](#) ist ein Paket, das *avr-gcc* mit *avr-libc* und *avrdude* installiert. Außerdem gibt es einen Begleittext mit Hinweisen.

Wie mit einer solchen Paketdatei *.deb umzugehen ist, ist eine weitere Installationsvariante, die im Vorwort der Seite beschrieben ist: `dpkg -i <chosen.deb>` Dieser debian-spezifische Befehl ist der eigentliche Linux-Paketverwaltungsbefehl. Die Programme *synaptic* und *apt-get* benutzen ihn. Es lohnt sich <http://www.debiananwenderhandbuch.de/> aufzurufen, um mehr über *dpkg* zu erfahren.

Auf der Kommandozeile geht man als root in das Verzeichnis, in dem sich die herunter geladene Paketdatei befindet (in der Regel Downloads), und ruft `dpkg -i ...` auf. Ich habe den langen Dateinamen in den Befehl hinein kopiert. Die Installation läuft ziemlich schnell ab.

Noch müssen *avr-gcc* und *avrdude* mit ihren Pfad aufgerufen werden:

`/usr/local/avr/bin/avr-gcc` bzw. `/usr/local/avr/bin/avrdude`, was man ohne Optionen zu Testzwecken tun sollte.

mfile

Es ist zwar keine große Sache, ein Muster-makefile mit einem Editor abzuändern. Besser geht das jedoch mit *mfile*. Die Seite mit den AVR-Debian-Paketen stellt dafür die Datei [mfile-ubuntu.tar.gz](#) zum Download bereit.

Wenn man sie entpackt, entsteht ein Verzeichnis `mfile-ubuntu`, das in `mfile` umbenannt wird. Dieses Verzeichnis wird nach `/usr/local/share/` verschoben, wofür man root sein muss. Man kann dazu durchaus auf der Kommandozeile den Dateimanager *nautilus* aufrufen. Wenn er beendet wird, steht die Kommandozeile wieder zur Verfügung.

Der vollständige Name des Programms ist `mfile.tcl`, was zeigt, dass es sich um ein Script der Sprache Tcl handelt. Falls noch nicht geschehen, muss mit *synaptic* das Paket *tk* installiert werden, das Tcl verfügbar macht. Das Programm wird mit vollem Pfad aufgerufen, und zwar mit `/usr/local/share/mfile/mfile.tcl`. Für einen ersten Test muss dieser Aufruf auf der Kommandozeile eingegeben werden, später wird er in einem Shell-Script stehen.

Ein Test auf der Kommandozeile

Mit dem Dateimanager erzeugt man als User ein neues Verzeichnis, z.B. `avrttest`, kopiert ein fertiges C-Quellprogramm `main.c` hinein, sofern man hat, oder schreibt es mit einem Editor. Ersteres ist besser, denn es gibt keine Fehler.

Dann öffnet man das Terminal und startet, wieder als User, auf der Kommandozeile `/usr/local/share/mfile/mfile.tcl`. Das Fenster zeigt ein Muster-`makefile`, das abgeändert wird, und zwar bei folgenden Punkten:

MCU type	z.B. <code>atmega32</code>	verwendeter MC
Main file name	z.B. <code>main.c</code>	Name des Hauptprogramms
Programmer	z.B. <code>avrisp2</code>	Programmieradapter
Port	z.B. <code>usb:0200040133</code>	Um die Seriennummer einzugeben, die anzupassen ist, muss das Editieren von <code>makefile</code> freigeschaltet werden.

Bei größeren Projekten werden weitere Quelldateien zugefügt unter:

C/C++ source file(s)	z.B. <code>test.c set.c</code>	In der Regel ist nur dieser Eintrag bei verschiedenen Projekten unterschiedlich.
----------------------	--------------------------------	--

Mit Datei/Save As wird das `makefile` im aktuellen Verzeichnis gespeichert und das `mfile`-Fenster wird geschlossen.

Was leicht übersehen wird: `mfile` öffnet immer das Muster-`makefile`. Meistens will man aber das `makefile` im aktuellen Verzeichnis anpassen, das deshalb explizit geöffnet und auch gespeichert werden muss, weil sonst die Änderungen nicht wirken.

Damit `avr-gcc` und `avrdude` gefunden werden, wenn `make` sie aufruft, muss die Umgebungsvariable `PATH` mit dem Pfad, der zu diesen Programmen führt, erweitert werden, und zwar durch `export PATH=$PATH:/usr/local/avr/bin`.

Man wird aber feststellen, dass diese Erweiterung sehr flüchtig ist. Sie gilt nur für den User, der den Befehl eingegeben hat. Und auch nur solange wie er User auf der Kommandozeile bleibt.

Mit `make all` sollte nun der Compiler den Quelltext verarbeiten und die Datei `main.hex` neben anderen produzieren. Beim zweiten Versuch macht er sich nur diese Mühe, wenn sich an den Dateien etwas geändert hat. Soll er dennoch compilieren, muss vorher `make clean` aufgerufen werden.

Wenn nun mit `make program` avrdude zum Programmieren des Bausteins aufgerufen wird, entsteht eine Fehlermeldung. Die USB-Schnittstelle kann nicht geöffnet werden, weil das wohl nur root darf. Das zu ändern, ist mir nicht gelungen. Die Ausgestaltung der Entwicklungsumgebung, die ich zusammengebaut habe, wird wesentlich von diesem Umstand bestimmt.

Also wird man mit `sudo -s -H` auf der Kommandozeile root. Für root muss nun der Pfad mit `export PATH=$PATH:/usr/local/avr/bin` noch einmal gesetzt werden und dann sollte mit `make program` der Baustein programmiert werden.

Der folgende Aufruf, der später noch eine Rolle spielen wird, programmiert den Baustein genauso wie `make program` (die Seriennummer ist natürlich anzupassen):

```
avrdude -p atmega32 -P usb:0200040133 -c avrisp2 -U flash:w:main.hex
```

Auf jeden Fall sollte nur avrdude von root aufgerufen werden. Mit exit kann man auf der Kommandozeile vom Super-User wieder zum User werden. Den Umgang mit Dateien und die Aufrufe von avr-gcc sowie mfile muss der User machen, ansonsten würde root Eigentümer der erzeugten Dateien, und man müsste root bleiben, um damit zu arbeiten.

Entwicklungsumgebung mit gedit

Das Problem, dass avrdude von root aufgerufen werden muss, könnte man dadurch lösen, dass root die gesamte Programmentwicklung durchführt oder dadurch, dass der User vor jedem Aufruf von avrdude root wird, wie oben gezeigt. Beides ist inakzeptabel, was ich nicht weiter ausführen muss.

Bei meiner Lösung verzichte ich auf den Aufruf von `make program`, wodurch auch die Eintragungen zu Port und Programmer in makefile gegenstandslos werden. Während der Programmentwicklung gibt es neben dem Editor-Fenster ein Terminal von root, das nur für den direkten Aufruf von avrdude verwendet wird, wobei Shell-Scripte helfen. Weil direkte Aufrufe, z.B. für die Behandlung der Fuse-Bits ohnehin erforderlich sind, wenn man nicht makefile umschreiben will, kann man auch gleich die Programmierung auf diese Weise durchführen.

Der Gnome-Editor *gedit* lässt sich ganz einfach zur Mini-Entwicklungsumgebung ausbauen. Zu beachten ist, dass Konfigurationen nur für den User gelten, der sie vorgenommen hat! *gedit* wird auf dem Desktop, also vom User, aufgerufen.

Unter *Bearbeiten/Einstellungen/Plugins* wird das Plugin *Externe Werkzeuge* eingebunden und konfiguriert. Später geht Konfigurieren auch unter *Werkzeuge/Externe Werkzeuge verwalten*. Externe Werkzeuge sind Shell-Scripts. Einige sind vorhanden. Man kann neue erzeugen und vorhandene löschen.

Ich habe erzeugt:

`avrbuild` Zum Compilieren, Linken

```
#!/bin/sh
export PATH=$PATH:/usr/local/avr/bin
make all
```

`avrclean` Zum Entfernen temporärer Dateien

```
#!/bin/sh
export PATH=$PATH:/usr/local/avr/bin
make clean
```

avrmfile

Aufruf von mfile zum Bearbeiten von makefile

```
#!/bin/sh
/usr/local/share/mfile/mfile.tcl
```

Den export-Befehl für den Pfad zu avr-gcc habe ich auch in den Shell-Scripts untergebracht.

Beim Konfigurieren von avrbuild ist anzugeben, dass die geänderten Dateien vor dem Compilieren gespeichert werden sollen. Nur dann werden sie auch verwendet.

Die Namen für Werkzeuge können beliebig gewählt werden, auch Tastenkürzel lassen sich definieren.

Das vorhandene Werkzeug Terminal öffnen wird später noch eine Rolle spielen. Das Werkzeug erscheint aber nur, wenn in gedit bereits eine Datei geöffnet oder gespeichert wurde. Das Terminal hat den Vorteil, dass beim Öffnen bereits das Verzeichnis der geöffneten Datei aktuell ist, man sich also darum nicht mehr kümmern muss.

Für den Aufruf von avrdude werden ebenfalls Shell-Skripte erzeugt, und zwar außerhalb von gedit. Für das Shell-Script *pg* zum Programmieren des MC geht das so:

Terminal aufrufen, root werden und Verzeichnis wechseln mit `cd /usr/local/bin`

Der Editor gedit wird auf der Kommandozeile gestartet.

Shell-Script schreiben und im aktuellen Verzeichnis mit *pg* speichern (Seriennummer anpassen)

```
#!/bin/sh
/usr/local/avr/bin/avrdude -p atmega32 -P usb:0200040133 -c avrisp2 -U
flash:w:main.hex
```

gedit beenden und `chmod +x pg` aufrufen. Dadurch wandelt sich *pg* von einer einfachen Textdatei zu einem Shell-Script (Voraussetzung dafür ist auch, dass die erste Zeile

`#!/bin/sh` lautet).

Wenn das Verzeichnis, in dem sich main.hex befindet, aktuell ist, kann root *pg* aufrufen, um den MC zu programmieren.

Auf die gleiche Weise werden die anderen Shell-Skripte erzeugt, die man so braucht. Dafür sollte man sich auch mit den Optionen von avrdude beschäftigen, die bei mir in der Datei

`/usr/local/avr/share/man/man1/avrdude.1` dokumentiert sind. Der Midnight-Commander *mc* kann bei mir die Datei lesbar öffnen, nicht jedoch ein Editor oder Aufrufe wie `man avrdude` oder `man avrdude.1` Konfigurieren lässt sich avrdude mit Hilfe der Datei `/usr/local/avr/etc/avrdude.conf`

readflash

```
#!/bin/sh
/usr/local/avr/bin/avrdude -p atmega32 -P usb:0200040133 -c avrisp2 -U flash:r:temp:i
```

writeflash

```
#!/bin/sh
/usr/local/avr/bin/avrdude -p atmega32 -P usb:0200040133 -c avrisp2 -U flash:w:temp:i
```

readeprom

```
#!/bin/sh
/usr/local/avr/bin/avrdude -p atmega32 -P usb:0200040133 -c avrisp2 -U eeprom:r:temp:i
```

writeeprom

```
#!/bin/sh
/usr/local/avr/bin/avrdude -p atmega32 -P usb:0200040133 -c avrisp2 -U eeprom:w:temp:i
```

readfuse

```
#!/bin/sh
/usr/local/avr/bin/avrdude -p atmega32 -P usb:0200040133 -c avrisp2 -U hfuse:r:temp:b
```

readlfuse

```
#!/bin/sh
/usr/local/avr/bin/avrdude -p atmega32 -P usb:0200040133 -c avrisp2 -U lfuse:r:temp:b
```

writefuseshex

```
#!/bin/sh
/usr/local/avr/bin/avrdude -p atmega32 -P usb:0200040133 -c avrisp2 -u -U
hfuse:w:0xC1:m -U lfuse:w:0xFF:m
```

Anmerkungen

Nichts spricht dagegen, noch weitere Shell-Skripte zu erzeugen.

pg ist der häufigste Aufruf. Weil sie nur selten gebraucht werden, sind die Namen der anderen Aufrufe länger und aussagekräftiger. Zu merken braucht man sich nur pg. Bei den anderen Aufrufen kann man auch mal nachschauen, um sich zu vergewissern, wie sie heißen und wie sie wirken.

temp ist eine Datei im aktuellen Verzeichnis, die angelegt wird (und dann root gehört) oder überschrieben wird, wenn sie schon existiert. Ein Verzeichnis temp darf es im aktuellen Verzeichnis nicht geben.

Der Zusatz :i bedeutet, dass die Datei im Hex-Format beschrieben und dass beim Lesen das Hex-Format erwartet wird. Deshalb tut writeflash das gleiche wie pg, nur die Dateien sind unterschiedlich.

Die beiden Fuse-Bytes (siehe auch Fuse-Bits) werden einzeln gelesen. Weil das fast immer zu Kontrollzwecken geschieht, ist das Binärformat (:b) am zweckmäßigsten.

Mit writefuseshex werden beide Fuse-Bytes geschrieben, wobei die beiden Hex-Werte direkt im Aufruf angegeben werden und gegebenenfalls anzupassen sind. Mit diesem Aufruf sollte man sehr sorgfältig umgehen. Falsche Werte können den MC unbrauchbar machen.

Shell-Skripte können ganz einfach durch Editieren geändert werden. Nach dem Speichern wirken die Änderungen sofort.

Arbeiten mit der Entwicklungsumgebung

In den meisten Fällen werden Programme geschrieben, indem vorhandene Quelldateien geändert und weiter entwickelt oder auch einfach nur eingebunden werden.

Ich ziehe es vor, in einem Verzeichnis *AVR* Arbeitsverzeichnisse anzulegen, die jeweils für ein Projekt stehen. Ein Projektverzeichnis enthält alle Quell- und Header-Dateien sowie makefile.

In Programmentexten werden Header-Dateien inkludiert mit:

```
#include "xxx.h"
#include "yyy.h"
..
```

Quelldateien, außer *main.c*, werden in *makefile* aufgenommen mit:

```
..xxx.c  yyy.c  ...
```

Das Hauptprogramm heißt bei allen Projekten *main.c*. Das hat den Vorteil, dass *makefile* und Aufrufe von *avrdude* in dieser Hinsicht nicht angepasst werden müssen. Unterscheiden lassen sich Projekte durch die Verzeichnisnamen.

Vorzubereiten ist folgendes :

- Falls nötig mit dem Dateimanager Verzeichnis und Dateien vorbereiten

- Editor *gedit* starten und eine Datei aus einem der Projektverzeichnisse öffnen.

- Mit externem Werkzeug von *gedit* Terminal öffnen und in diesem Terminal mit `sudo -s -H root` werden. Das Terminal bleibt geöffnet. Sein Vorteil ist, dass, wenn es erscheint, das Projektverzeichnis bereits aktuell ist.

- Mit dem externem Werkzeug *avrmfile* notwendige Eintragungen in *makefile* machen, um z.B. Quelldateien hinzuzufügen.

- Zweckmäßig ist, das externe Werkzeug *avrclean* aufzurufen, um überflüssige Dateien zu entfernen. Achtung: *main.hex* ist dann auch weg.

Nun folgt der jedem Programmierer bekannte Zyklus: Editieren, Compilieren/Linken, Programmieren, Testen und wieder von vorn.

Unterstützung beim Schreiben von Programmen findet man in den Header-Dateien der Bibliothek unter `/usr/local/avr/avr/include` und Beispiele gibt es unter `/usr/local/avr/share/doc/avrlibc-1.7.0/examples` Ich benutze auch die Dokumentation, die WinAVR mitbringt.

Zum Compilieren/Linken wird das externe Werkzeug *avrbuild* aufgerufen.

Zum Programmieren des MC wechselt man in das geöffnete Terminal und ruft *pg* auf (oder ein anderes der definierten Shell-Scripte). Mit einem Mausklick ist man wieder im Editor-Fenster.

Das war's.

Fuse-Bits

Fuse-Bits stehen für grundlegende Funktionen und können nur außerhalb von MC-Programmen verändert werden. Wenn dabei Fehler entstehen, kann es sein, dass der Baustein nicht mehr arbeitet.

Beim fabrikneuen MC ATmega32 sind die Fuse-Bits in Gestalt von 2 Bytes so gesetzt, wie die folgende Tabelle zeigt:

High-Fuse-Byte ATmega32, fabrikneu

Hex	OCDEN	JTAGEN		CKOPT	EESAVE	BOOTSZ1	BOOTSZ0	BOOTRST
0x99	1	0	0	1	1	0	0	1

Low-Fuse-Byte ATmega32, fabrikneu

Hex	BODLEVEL	BODEN	SU1	SUT0	CKSEL3	CKSEL2	CKSEL1	CKSEL0
0xE1	1	1	1	0	0	0	0	1

Ganz wichtig: Eine Null bedeutet, dass die mit dem Bit verbundene Funktion, aktiviert ist. Das Bit wird auch als *programmiert* bezeichnet. Die Eins bedeutet: *Funktion ist deaktiviert*. Das ist entgegen der Gewohnheit und hat wohl damit zu tun, dass Bits im gelöschten Flash-Speicher im Zustand 1 sind.

Die Datei temp enthält beispielsweise nach dem Aufruf von readfuse beim fabrikneuen ATmega32 die folgenden Zeichen, die sich in der vorstehenden Tabelle wiederfinden:

```
0b10011001
```

Ich konfiguriere die Fuse-Bits des ATmega32 für den Betrieb mit einem 16 MHz-Quarz, mit einem deaktivierten JTAG-Interface und mit einem gegen Löschen geschützten EEPROM. Die Tabelle sieht dann so aus:

High-Fuse-Byte ATmega32, Praxis

Hex	OCDEN	JTAGEN		CKOPT	EESAVE	BOOTSZ1	BOOTSZ0	BOOTRST
0xC1	1	1	0	0	0	0	0	1

Low-Fuse-Byte ATmega32, Praxis

Hex	BODLEVEL	BODEN	SU1	SUT0	CKSEL3	CKSEL2	CKSEL1	CKSEL0
0xFF	1	1	1	1	1	1	1	1

Zuletzt

Ich denke, dass ich mit der beschriebenen Entwicklungsumgebung mindestens so gut wie mit WinAVR arbeiten kann, was die Fuse-Bits betrifft, sogar besser. Es bleibt nur ein Schönheitsfehler: Um avrdude anzuwenden, muss man root sein. Sicher werden mir meine fachkundigen Leser Hinweise geben, wie das auszuräumen ist.

