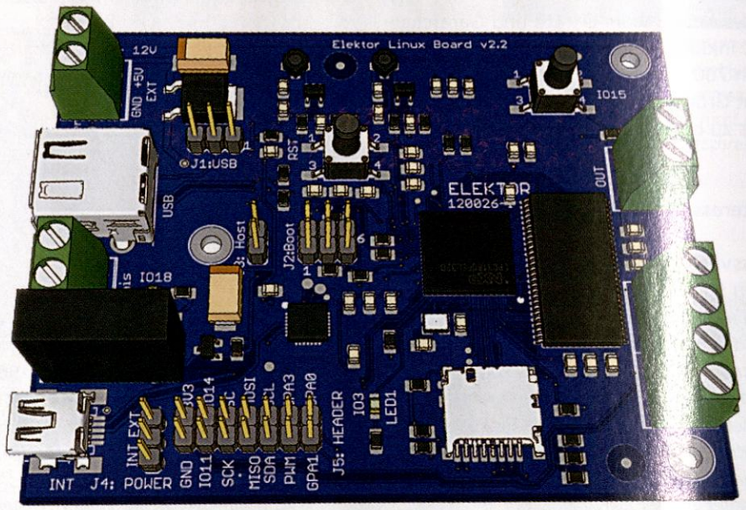
Embedded Linux leicht gemacht (7)

I²C, UART, RS485

Von
Benedikt Sauter [1]

In der vorangegangenen Ausgabe haben wir unsere Leser dazu aufgerufen, den Inhalt des letzten Teiles dieser Artikelserie mitzubestimmen. Vielen Dank an die weit über 100 Leser, die an der Umfrage teilgenommen haben! Auf vielfachen Wunsch werden wir diesmal noch etwas genauer auf die Entwicklung eigener Anwendungen eingehen. Auch serielle Schnittstellen wie I²C und UART, die man in eigenen Projekten häufig benötigt, sind ein Thema.



3D-Modell des Elektor-Linux-Boards, erstellt mit dem Tool EagleUp [14].

Open-Source-Software lebt von der offenen Entwicklung in der Community. Eine Quelle für Updates, Patches oder Erweiterungen sind daher die Webseiten des jeweiligen Projektes, auf denen man wichtige Hinweise und meist auch ein User-Forum findet.
Basis des Elektor-Linux-Boards ist das Projekt Gnublin [2], das der Autor gemeinsam mit der Hochschule Augsburg 2009 als Plattform für die Embedded-Linux-Entwicklung gestartet hat. In der Zwischenzeit sind neben verschiedenen Board-Versionen, passenden Ergänzungs-Schaltungen (Schrittmotorsteuerung, CAN, ...) auch diverse Erweiterungen der Software dazugekommen. Unter anderem existieren inzwischen Updates des Kernels und der Toolchain samt C/C++-Compiler. Diese wollen wir sogleich installieren!

Updates

Um Programme nicht immer direkt auf dem Elektor-Linux-Board übersetzen zu müssen, haben wir im dritten Teil der Serie [3] einen C/C++-Compiler auf dem PC installiert. Wir können die neueste Version dieses Compilers parallel mit der übrigen Toolchain installieren.

Am besten öffnet man dafür in unserem Ubuntu-Linux mit der Tastenkombination „Strg-Alt-t“ eine neue Konsole und gibt dort den folgenden Befehl ein:

```
wget http://gnublin.org/downloads/eldk-
eglibc-i686-arm-toolchain-qte-5.2.1.tar.bz2
```

Ist das Archiv fertig heruntergeladen, dann kann man es einfach in das eigene Dateisystem entpacken:

```
sudo tar xjf eldk-eglibc-i686-arm-
toolchain-qte-5.2.1.tar.bz2 -C /
```

Um den neuen Compiler jetzt nutzen zu können, muss man ein kleines Skript schreiben, das die Umgebungsvariablen zu den Verzeichnissen der neuen Installation automatisch setzt. Dafür legt man eine Datei „set.sh“ im Ordner „/opt/eldk-5.2.1“ an:

```
sudo gedit /opt/eldk-5.2.1/set.sh
```

In **Bild 1** sieht man, wie die Verzeichnisse zu der Umgebungsvariablen PATH hinzugefügt werden. Nach dem Speichern kann man die Datei auf der Konsole aufrufen (auf Syntax achten: Punkt Leerstelle Schrägstrich):

```
. /opt/eldk-5.2.1/set.sh
```

Dieses Kommando muss ab jetzt immer wieder aufgerufen werden, wenn man ein Programm auf dem Entwicklungs-PC für das Elektor-Linux-Board übersetzen möchte.
Der neue Compiler der Toolchain kann dann wie folgt aufgerufen werden:

```
arm-linux-gnueabi-gcc -v
```

C/C++-Entwicklungsumgebung

Viele Leser haben nach einer Möglichkeit gefragt, C/C++-Software für das Board mit einer komfortablen IDE wie zum Beispiel Eclipse entwickeln zu können. Wir beschreiben daher in diesem Abschnitt kurz, wie man auf eine einfache Art und Weise C/C++-Programme auf dem Entwicklungsrechner schreiben und direkt auf dem Board testen kann.

Was man dazu braucht ist:

- Konsole des Linux-Boards (Zugriff per UART oder Netzwerk)
- Netzwerk-Dateisystem (für den Zugriff vom Entwicklungs-PC aus)
- Optional natürlich die Entwicklungsumgebung

Da in einem Linux-Verzeichnisbaum Geräte abgebildet werden wie Dateien, ist es naheliegend, mit diesem Betriebssystem auch entfernte Speichermedien genauso in den Verzeichnisbaum einhängen zu können wie einen lokalen Ordner bzw. Unterordner. Hierfür gibt es unter Linux (natürlich) wieder viele verschiedene Möglichkeiten wie z.B. NFS, FTP oder Samba.

Im vorangegangenen Teil der Serie [4] wurde eine Möglichkeit zur Fernwartung des Linux-Boards per SSH vorgestellt. SSH ist ein Server-Programm, mit dem man Zugriff auf die Konsole des Boards per Netzwerk bekommt. SSH ist vergleichbar mit Telnet, nur bietet es eine erhöhte Sicherheit, da es die Verbindung vom Server bis zum Nutzer verschlüsselt. Außerdem lassen sich auf einfache Weise Dateien zwischen Nutzer und Server austauschen; auf Nutzerseite kann man dafür das Programm *scp* verwenden.

```
P1=/opt/eldk-5.2.1/armv5te/sysroots/i686-eldk-linux/usr/bin/armv5te-linux-gnueabi/
P2=/opt/eldk-5.2.1/armv5te/sysroots/i686-eldk-linux/bin/armv5te-linux-gnueabi/
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-
export PATH=$P1:$P2:$PATH
```

Bild 1.
Umgebungsvariablen für die neue Toolchain.

Hat das Linux-Board beispielsweise die IP-Adresse 192.168.0.190, so kann man vom Entwicklungs-PC eine Datei direkt in das Dateisystem des Linux-Boards übertragen:

```
scp test.c root@192.168.0.190:/root
```

Soll ein ganzer Ordner übertragen werden, dann schreibt man:

```
scp -r myproject root@192.168.0.190:/root
```

Als erstes gibt man die Datei oder den Ordner an. Im Falle des Ordners muss man ein zusätzliches -r angeben, damit alle weiteren Ordner und Dateien darunter rekursiv mit übertragen werden. Danach muss man noch den Benutzer-

namen angeben, gefolgt von einem @-Zeichen und der IP-Adresse. Nach dem Doppelpunkt gibt man schließlich den Zielpfad im Verzeichnisbaum des Linux-Boards an.
Jetzt könnte man auf diese Art und Weise in regelmäßigen Abständen die übersetzten Programme auf das Board übertragen. Aber es geht noch eleganter!
Ein Profi installiert sich unter Linux das Programm *sshfs*. Auf unserem Entwicklungs-PC kann man dies machen mit:

```
sudo apt-get install sshfs
```

Mit dem Tool können wir das komplette Dateisystem des Linux-Boards live in unserem Entwicklungs-PC freigeben. So müssen wir nie wieder Dateien von Hand auf die SD-Karte des Linux-Boards kopieren. Voraussetzung hierfür ist (wie bereits bei SSH oder scp) natürlich eine bestehende Netzwerkverbindung.
Auf dem Dateisystem des Entwicklungs-PCs muss man zuerst einen sogenannten „mount-Punkt" anlegen. An dieser Stelle kann man das entfernte Dateisystem einhängen. Der mount-Punkt ist ein gewöhnliches leeres Verzeichnis, das man mit dem Programm *mkdir* über die Konsole oder auch mit einem grafischen Dateiexplorer anlegen kann.

Zuerst wechseln wir in den Home-Ordner...

```
cd ~
```

... und legen den mount-Punkt an:

```
mkdir elektor-linux-board
```

Jetzt kann man mit dem nächsten Befehl das gesamte Dateisystem des Linux-Boards auf dem Entwicklungs-PC verfügbar machen:

```
sshfs root@192.168.0.190:/
elektor-linux-board
```

Nach dem Aufruf wird man wieder einmalig nach dem Passwort gefragt. Ist dies erfolgreich eingegeben, kann man mit `cd` und `ls` im Dateisystem herumnavigieren. Testet man ein paar Verzeichnisse durch, wird man ab und zu aber ein „Permisson denied" gemeldet bekommen. Denn beim Elektor-Linux-Board gehören die Systemprogramme natürlich dem Benutzer *Root*, doch man arbeitet aktuell mit der Benutzer-Identität des Entwicklungs-PCs. Im Wesentlichen wird ein Benutzer durch seine User-ID bestimmt. Um sich komplett frei in dem eingehängten Dateisystem bewegen zu können, muss man sich als *Root* anmelden. Am einfachsten startet man dafür eine Konsole, die mit Root-Rechten ausgeführt wird:

```
sudo /bin/bash
```

In dieser Konsole kann man sich ganz frei in dem eingehängten Dateisystem bewegen, Dateien einfügen und löschen.
Möchte man jetzt Programme für das Board auf dem Entwicklungs-PC schreiben, dann muss man den Projektordner einfach in den eingehängten Dateisystem ablegen. Wie man Eclipse für die Entwicklung von C/C++ verwendet, ist im Wiki [5] beschrieben.
Idealerweise legt man sich vor dem Start eines neuen Projektes einen eigenen Benutzer auf dem Elektor-Linux-Board an. Dies macht man mit:

```
adduser elektor
```

Dann kann man auch nur das Home-Verzeichnis des Benutzers in das eigene Dateisystem einhängen:

```
sshfs elektor@192.168.0.190:/home/elektor
elektor-linux-board
```

Arbeitet man nur immer alleine am PC und mit dem Board, so erhalten beide Nutzer automatisch die User-ID 1000 und man hat in diesem Ordner nie Probleme mit den Rechten. Trotzdem kann man natürlich zwischendurch ein Programm auch als *Root* ausführen.

Kernel-Update

Der Kernel dient den Anwendungen als Schnittstelle zur Hardware. Möchte man zusätzliche oder neuere Hardware ansprechen, dann kann es immer wieder notwendig sein, sich eine neue Kernel-Version herunterzuladen und manuell zu übersetzen.
Für das Elektor-Linux-Board ist das Kernel-Archiv des Gnublin-Projektes die erste Anlaufstelle. Seit dem Beginn der Artikelreihe wurde der Kernel erweitert, so kann man beispielsweise jetzt auch CAN-, SPI- oder I2C-Geräte ansprechen.
Zentral wird der Quelltext in einer Versionsverwaltung gepflegt. Entwickler können hier Änderungen einfach und nachvollziehbar anderen Entwicklern zur Verfügung stellen. Als Software zur Verwaltung des Softwarearchivs wird GIT verwendet. Um auf ein GIT-Repository zugreifen zu können, benötigt man das GIT-Softwarepaket. Hierzu gibt man am Entwicklungsrechner ein:

```
sudo apt-get install git-core
```

Anschließend kann man in einen beliebigen Ordner wechseln und dort das Softwarearchiv aus dem Repository „klonen".

```
git clone http://code.google.com/p/
gnublin-develop-kernel/
```

Dies kann etwas dauern. Sobald das Herunterladen erfolgt ist, blinkt der Cursor wieder und ist bereit für neue Eingaben. Um den neuesten stabilen Kernel für das Board zu „bauen", wechselt man in das Verzeichnis für den 2.6.33er Kernel (Experimentierfreudige können gerne mal die Version 3.3.0 des Kernels testen):

```
cd linux-2.6.33-lpc313x
```

In diesem Ordner findet man die passende Standardkonfiguration:

```
make elektor_de_config
```

Möchte man manuelle Änderungen an der Konfiguration vornehmen, kann man dies wieder erledigen mit:

```
make menuconfig
```

Anschließend wird der Kernel übersetzt (Aufruf von „set.sh" nicht vergessen):

```
make zImage
```

Ist der Kernel fehlerfrei erzeugt worden, kann dieser wieder mit einem SD-Kartenleser auf die SD-Karte übertragen werden (für „LABEL_SD_CARD" den entsprechenden Ordner einsetzen):

```
sudo cp arch/arm/boot/zImage /media/
LABEL_SD_CARD/
```

Die Module übersetzen wir mit folgendem Befehl:

```
make modules
```

Danach installieren wir die Module in ein Zwischenverzeichnis:

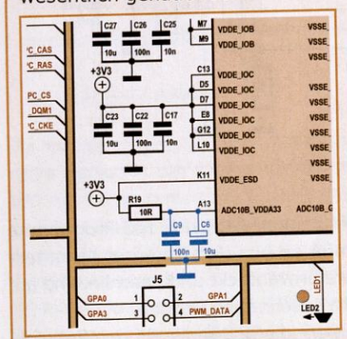
```
make modules_install INSTALL_MOD_PATH=/
tmp
```

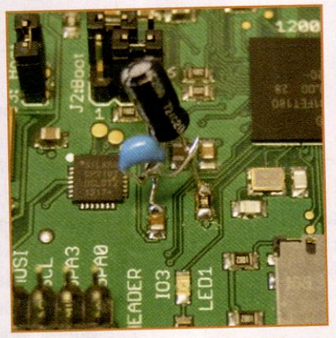
Vom Zwischenverzeichnis aus werden die fertigen Module schließlich auf die SD-Karte kopiert:

```
sudo cp -r /tmp/lib /media/LABEL_SD_CARD/
```

Genauere Spannungs-Messungen

Im fünften Teil der Serie [15] hatten wir gezeigt, wie man den Analog/Digital-Wandler des Controllers für Spannungsmessungen verwendet, und natürlich haben das viele Leser gleich ausprobiert. Dabei gab es vereinzelt Rückmeldungen, dass die 3,3-V-Board-Spannung, die vom ADC als Referenz genutzt wird, nicht stabil genug für präzise Messungen ist. Doch es gibt hierfür eine einfache Lösung: Um zu einer stabileren Spannung zu kommen, muss man bei R19 einfach zwei Kondensatoren mit ca. 10 µF und 100 nF nach Masse schalten (**Bild 18** und **Bild 19**). Schon sollten die Messungen wesentlich genauer sein.





I2C-Tools

I2C ist seit Jahrzehnten ein sehr beliebter Bus im Mikrocontroller-Bereich. Es gibt viele verschiedene Bausteine, vom A/D-Wandler über den I/O-Expander bis hin zum Temperatursensor reicht die Palette. Um I2C unter Linux verwenden zu können, müssen die entsprechenden Treiber im Kernel aktiviert sein.

```
Device Drivers → I2C support → I2C
Hardware Bus support → I2C bus support
for Philips PNX targets
```

In der Standardkonfiguration ist der Treiber bereits dabei.

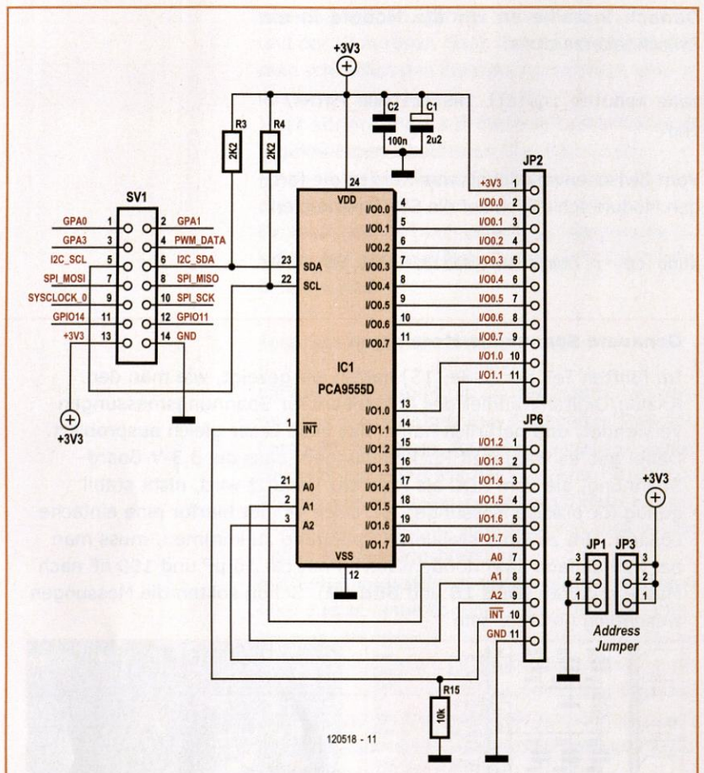


Bild 2.
Schaltung mit dem I2C-Portexpander PCA9555.

Für einfache Tests bietet Linux I2C-Programme an. Diese müssen aus dem Internet heruntergeladen, kurz umverpackt und anschließend auf die SD-Karte geschrieben werden.
Zuerst laden wir die I2C-Tools auf den Entwicklungs-PC herunter:

```
wget http://ftp.de.debian.org/debian/
pool/main/i/i2c-tools/i2c-tools_3.0.2-5_
armel.deb
```

Dann wandeln wir das Debian-Paket in ein Standard-Archiv um:

```
alien -t i2c-tools_3.0.2-5_armel.deb
```

Jetzt findet man im gleichen Ordner eine Datei mit dem Namen „i2c-tools-3.0.2.tgz“. Diese lässt sich per SD-Kartenleser auf die SD-Karte kopieren:

```
cp i2c-tools-3.0.2.tgz /media/
LABEL_SD_CARD/
```

Nach dem Neustart des Linux-Boards muss man einmalig das eben umgewandelte Paket entpacken und zusätzlich einmalig die Gerätedateien für I2C anlegen.
Zuerst ist also das Entpacken der I2C-Linux-Tools an der Reihe:

```
cd /

tar xvzf i2c-tools-3.0.2.tgz
```

Die Gerätedateien legen wir wieder mit dem Befehl `mknod` an [6]. Sicherheitshalber kann man die Major-Nummer ermitteln mit:

```
cat /proc/devices
```

Beim entsprechenden Eintrag sollte eine „89“ stehen. Steht hier eine andere Nummer, müssen die nächsten Befehle entsprechend angepasst werden.

```
mknod /dev/i2c-0 c 89 0
mknod /dev/i2c-1 c 89 1
```

Sind die Gerätedateien angelegt und ist die Software installiert, kann das erste I2C-Gerät angeschlossen und angesprochen werden. Für einfache Tests bietet sich der Baustein PCA9555 an, ein Portexpander, den wir wie in **Bild 2** an den 14-poligen Erweiterungsstecker des Boards anschließen.
Für unsere Tests benötigen wir eine Stromversorgung von 3,3 V, die am Erweiterungsstecker abgegriffen werden kann. An den Eingängen A0 bis A2 des Portexpanders kann man direkt Jumper anbringen, um die Leitungen entweder auf GND oder 3,3 V legen zu können. Hiermit stellt man die Adresse des Bausteins ein.
Ist der Chip entsprechend verschaltet und über SDA und SCL an das Elektor-Linux-Board angeschlossen, dann kann mit dem Tool „i2cdetect“ der komplette I2C-Adressraum am Bus gescannt werden:

```
i2cdetect 1
```

Als Ausgabe erhält man **Bild 3**. Die Adresse 0x6e stammt vom LPC3131-Interface. Da alle Adressleitungen des PCA9555 in unserer Schaltung auf High gelegt worden sind, haben wir für den Baustein die Adresse 0x27 gefunden.
Für einen einfachen Test haben wir drei LEDs an den Port 0 des Bausteins gehängt (siehe **Bild 4**). Mit diesen LEDs kann man nun einen einfachen Funktionstest des I2C-Busses durchführen. Um kurze I2C-Nachrichten per Kommandozeile versenden zu können, verwendet man das Program „i2cset“. Alle I/O-Pins als Ausgang definieren:

```
i2cset 1 0x27 0x06 0x00
```

Setzen der Ausgänge auf „High“:

```
i2cset 1 0x27 0x02 0xff
```

Jetzt sollten die angeschlossenen LEDs leuchten. Nachdem wir jetzt schon einfache I2C-Tests direkt auf der Konsole durchführen können, wollen wir uns nun einem einfachen C-Programm für eine I2C-Steuerung widmen.

```
root@gnublin:~# i2cdetect 1
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-1.
I will probe address range 0x03-0x77.
Continue? [Y/n]
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- 27 -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- 6e --
70: -- -- -- -- -- -- -- --
root@gnublin:~#
```

Bild 3.
Ausgabe mit dem Programm „i2cdetect“.

I2C in C

Der LPC3131 bietet zwei physikalisch getrennte I2C-Schnittstellen an, wobei am 14-poligen Stecker nur der zweite Bus verfügbar ist.
Die Gerätedateien heißen „/dev/i2c-0“ bzw. „/dev/i2c-1“. Der Zugriff auf die Hardware erfolgt wie immer; man öffnet die Gerätedatei und greift anschließend lesend oder schreibend auf diese zu. In **Listing 1** sieht man ein kleines C-Programm, mit dem wir wieder unsere LEDs leuchten lassen. Das Programm kann man auf dem Entwicklungs-PC übersetzen mit:

```
arm-linux-gnueabi-gcc -o i2ctest
i2ctest.c
```

Anschliessend nutzt man die SD-Karte (oder die Tools *scp* bzw. *sshfs*), um die übersetzte Datei auf das Board zu übertragen. Optional kann man die Datei auch direkt auf dem Elektor-Linux-Board mit dem integrierten C-Compiler *gcc* übersetzen. Idealerweise kopiert man das Testprogramm nach „/usr/bin“, dann kann es von jedem Verzeichnis aus ohne explizite Pfadangabe aufgerufen werden:

```
cp i2ctest /usr/bin
```

Das Programm startet man auf dem Board, mit der Gerätedatei für den I2C-Bus als Parameter:

```
i2ctest /dev/i2c-1
```

Jetzt sieht man die angeschlossenen LEDs an- und ausgehen.

Im oberen Teil des Quellcodes (Listing 1) sieht man, wie mit `open()` ein File-Handle für die Gerätedatei angelegt wird. Mittels `ioctl()` (dem Befehl für die Konfiguration von Gerätetreibern) wird dem Treiber die aktuelle Geräteadresse des gewünschten Bausteins mitgeteilt. Ist die Adresse eingestellt, dann kann mit `write()` einfach eine I2C-Nachricht an das Gerät gesendet werden. Das I2C-Kommando für das Setzen der Portexpander-I/O-Pins als Ausgang kennen wir bereits:

```
buffer[0] = 0x06;

buffer[1] = 0x00;
```

Die Kommandos für das Ein- und Ausschalten der LEDs stehen in der Endlosschleife (mit jeweils 100 ms Pause zwischen den Aufrufen).
Auf die gleiche Art und Weise können weitere Bausteine am I2C-Bus angesprochen werden. Für C-Profis ist eventuell auch die Bibliothek *libi2c* [7] interessant, mit der sehr flexibel auf den I2C-Bus zugegriffen werden kann. Aber auch diese

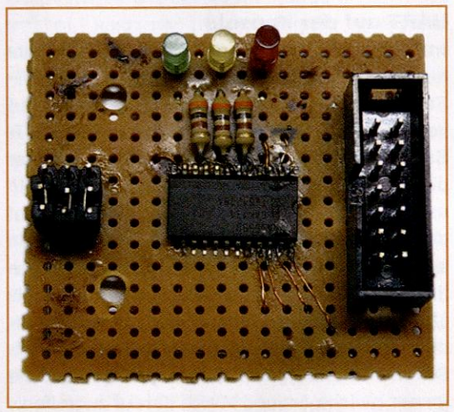


Bild 4.
Drei LEDs dienen zum Testen.

Listing 1: Ansteuerung eines I2C-Portexpanders

```
#include <stdio.h>
#include <fcntl.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>

//Slave Address
#define ADDR 0x27

int main (int argc, char **argv)
{
  int fd;
  char filename[32];
  char buffer[128];
  int n, err;

  if (argc == 0) {
      printf(„usage: %s <device>\n“, argv[0]);
      exit(1);
  }
  sprintf(filename, argv[1]);
  printf(„device = %s\n“, filename);

  int slave_address = ADDR;

  if ((fd = open(filename, O_RDWR)) < 0) {
    printf(„i2c open error“);
    return -1;
  }
  printf(„i2c device = %d\n“, fd);

  //prepare communication
  if (ioctl(fd, I2C_SLAVE, slave_address) < 0) {
    printf(„ioctl I2C_SLAVE error“);
    return -1;
  }

  write(fd, buffer, 2);

  //Port-0 GPIO as Output
  buffer[0] = 0x06;
  buffer[1] = 0x00;
  write(fd, buffer, 2);

  n = 0;
  while (1)
  {
    buffer[0] = 0x02;  /* command byte: write
output regs */
    buffer[1] = 0x00;  /* port1 data  */
    write(fd, buffer, 2);

    usleep(100000);
    buffer[0] = 0x02;  /* command byte: write
output regs */
    buffer[1] = 0xff;  /* port1 data  */
    write(fd, buffer, 2);

    printf(„%d\n“, n++);
    usleep(100000);
  }
}
```

Bibliothek verwendet die üblichen ioctl-, write- und read-Funktionen!

UART auf der Konsole

Im vierten Teil der Serie [6] hatten wir bereits einen USB/RS232-Wandler installiert. UART-Schnittstellen können in ein Linux-System auf verschiedenste Wege integriert werden. Die UART-Schnittstellen des Controllers werden meist über die Gerätedateien „/dev/ttyS0“, „/dev/ttyS1“ usw. angesprochen. Die Standardkonsole, auf der man alle Systemmeldungen und den Login sieht, wird zum Beispiel über „/dev/ttyS0“ realisiert. USB/UART-Schnittstellen werden als „/dev/ttyUSB0“, „/dev/ttyUSB1“ etc. eingebunden. Doch unabhängig davon, wie die UART-Schnittstelle integriert wird, man kann auf die immer gleiche Art und Weise darauf zugreifen.

Mit den Linux-Gerätedateien bekommt man auch zu einer gewissen Plattformunabhängigkeit: So ist es kein Problem, eine USB/UART-Anwendung komplett am PC zu entwickeln bzw. zu simulieren, denn auch dort werden die USB/UART-Konverter als „/dev/ttyUSB0“ usw. am System angemeldet. Steht die Software auf dem PC, so kann man diese einfach auf das Linux-Board umziehen, dort einmal cross-compilieren und dann mit den richtigen Parametern aufrufen und ausführen lassen. Sollte eine Gerätedatei fehlen, kann man diese wieder mit `mknod` anlegen, wobei immer zuvor der entsprechende Treiber geladen werden muss:

```
mknod /dev/usb/ttyUSB0 c 188 0
mknod /dev/usb/ttyUSB1 c 188 1
mknod /dev/usb/ttyUSB2 c 188 2
```

In vierten Teil der Serie haben wir *microcom* zur Kommunikation genutzt, ein kleines Terminalprogramm für serielle Schnittstellen. Als Parameter kann die gewünschte Baudrate und Gerätedatei angegeben werden. Ist das Programm gestartet, dann werden empfangene Zeichen direkt angezeigt. Gibt man Buchstaben oder Zeichen mit der Tastatur ein, werden diese automatisch über die serielle Schnittstelle ausgegeben.
Klar, dass wir jetzt auch einmal ohne solch ein fertiges Programm auskommen wollen. Das Empfangen und Senden von Daten ist nicht schwer, denn auf die Gerätedatei „/dev/ttyUSB0“ kann man ebenfalls wieder mit unseren einfachen Lese- und Schreibbefehlen zugreifen. Möchte man aber eine bestimmte Baudrate einstellen, dann muss man dies über eine gesonderte Methode machen. Unter Linux kann man dafür das Programm „stty“ verwenden. Um die Baudrate auf z.B. 38400 Baud zu setzen, ruft man folgenden Befehl auf:

```
stty -F /dev/ttyUSB0 38400
```

Jetzt kann man einfach Zeichen versenden mit:

```
echo “Hallo Welt” > /dev/ttyUSB0
```

oder auf Zeichen für den Empfang wie folgt warten:

```
cat < /dev/ttyUSB0
```

Am besten spielt man einfach einmal ein wenig mit diesen Befehlen herum. Wenn man nur einfache Protokolle verwendet, die mit darstellbaren ASCII-Zeichen arbeiten, so kann man manuell schon ganz gut testen und arbeiten. Möchte man aber auch binäre Werte übertragen, muss man zu einem kleinen C-Programm greifen.

UART in C

Der Schritt zu einem eigenen C-Programm ist aber nun nicht mehr besonders groß. Wie auch bereits auf der Konsole muss man zuerst die serielle Schnittstelle konfigurieren. Das wichtigste dabei ist die Baudrate. Im Download zu diesem Artikel [8] findet man ein C-Programm (bezeichnet mit „Listing 2“), dort sieht man das typische Vorgehen.
Als Vorlage für dieses Programm wurden die Beispiele aus [9] verwendet. Im oberen Teil des Programms wird mit Hilfe der Datenstruktur `struct termios options` die Grundkonfiguration vorgenommen. Mit `tcgetattr` wird zuerst die aktuelle Konfiguration in die Datenstruktur übernommen, wo sie dann modifiziert werden kann. Die Datenrate stellt man mit `cfsetispeed` (für eingehende Zeichen) und `cfsetospeed` (für ausgehende Zeichen) ein. Die Anzahl der Daten-, Start- und Stoppbits lässt sich hier ebenfalls einstellen. Anschließend wird die neue Konfiguration dem Treiber mit `tcsetattr` übergeben.
Das Versenden und Empfangen von Daten funktioniert wieder wie bei jedem Standard-Gerät. Mit `write()` kann man über das Handle – das zuvor mit `open()` geöffnet wurde – Daten senden und mit `read()` entsprechend Daten lesen. Im Beispiel erwartet unsere Gegenstelle als Kommando einen 6 Byte langen Wert. Man sieht beispielsweise, dass an die Variable `buffer[0]` eine 0 übergeben wird.

Als Antwort erwartet unser Programm von dem angeschlossenen Gerät ebenfalls wieder 6 Bytes. Das Empfangen ist hier mit einer Schleife gelöst, die insgesamt sechs Zeichen einliest. An der Stelle des read-Befehls innerhalb der Schleife wartet das Programm, bis das nächste Zeichen kommt (blockierender Aufruf). Das ist keine optimale Lösung,

Autor Benedikt Sauter im Elektor-Interview. Am Farnell/element14-Stand auf der Electronica 2012 waren gleich mehrere Kameraleute im Einsatz. Das Video kann man ansehen unter [13].

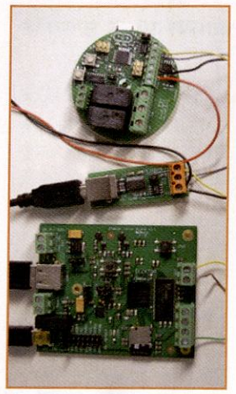


Bild 5.
RS485 mit dem ElektorBus.

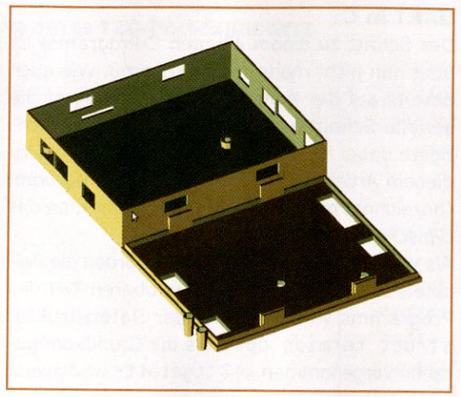


Bild 6.
Gehäuse für das Elektor-Linux-Board.

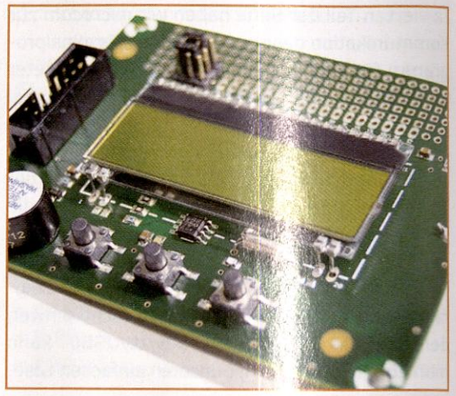


Bild 7.
Das Erweiterungsboard stellen wir im nächsten Heft vor.

da die Datenübertragung nur mit 19200 Baud erfolgt. Unser schneller Controller macht in der Schleife dann fast nichts anderes, als untätig auf Zeichen zu warten, doch trotzdem nimmt das kostbare Rechenzeit in Anspruch.

Hier wünscht man sich typischerweise einen Interrupt. Wenn Daten vorliegen, wird man benachrichtigt und greift auf den Empfangsspeicher zu. Mit einem Betriebssystem wie Linux lässt sich dann ein ressourcensparender Programmablauf erreichen. Sind keine Daten vorhanden, kann das empfangende Programm etwas anderes machen oder auch durch Linux schlafengelegt werden. Erst wenn wieder Daten eingehen, wird das Programm aufgeweckt, um die Daten zu verarbeiten. Im dritten Programm im Downloadordner [8] ist so ein Interrupt-Verhalten verwirklicht. Im Wesentlichen handelt es sich um das gleiche Beispiel wie eben, nur gibt es jetzt eine eigene Funktion `signal_handler_IO`, die aufgerufen wird, wenn Daten empfangen wurden.

RS485 mit dem ElektorBus

Wenn wir mit dem UART arbeiten können, dann sollten wir nun auch in der Lage sein, über eine RS485-Schnittstelle Bytes zu senden und zu empfangen. Nur dass wir jetzt statt eines USB/RS232-Konverters einen USB/RS485-Wandler an die USB-Schnittstelle unseres Boards anschließen müssen. Bei Elektor kann so ein Konverter unter der Nummer 110258-91 bestellt werden [8]. Er enthält einen FTDI-Chip; der entsprechende Treiber ist im Kernel bereits vorhanden und muss nur noch (so wie im vierten Teil beschrieben) im Kernel aktiviert werden.
Als RS485-Gegenstelle benutzen wir eine Elektor-Bus-Installationsplatine [8][10], die zwei Relais mitbringt (**Bild 5**). Für das ElektorBus-Protokoll muss man einfach Listing 2 etwas anpassen, zum Beispiel müssen die Baudrate auf 9600 Baud eingestellt und die Variable LENGTH auf 16 gesetzt werden. Das Array `Buffer` wird dann vor dem Versenden der RS485-Nachricht mit den entsprechenden 16 Bytes befüllt. Um das Relais 1 auf der Installationsplatine zu schalten, muss man folgendes Kommando senden:

```
170,0, 0,5,0,10, 96,1,0,0, 0,0,0,0, 0,0
(anziehen)
170,0, 0,5,0,10, 96,0,0,0, 0,0,0,0, 0,0
(abfallen)
```

Für Relais 2 lauten die Bytes:

```
170,0, 0,5,0,10, 0,0,96,1, 0,0,0,0, 0,0
(anziehen)
170,0, 0,5,0,10, 0,0,96,0, 0,0,0,0, 0,0
(abfallen)
```

Mehr zum ElektorBus findet man unter [11].

Ausblick

Wer auf der Suche nach einem passenden Gehäuse für das Elektor-Linux-Board (**Bild 6**) ist, wird mittlerweile im Internet fündig [12]. Unter dem Link findet man ein passendes 3D-Modell für einen 3D-Drucker. Es ist so konstruiert, dass man keine Schrauben benötigt, sondern einfach mit einem Schnappmechanismus Deckel und Grundkörper zusammenstecken kann.
Dies war der letzte Teil unseres Embedded-Linux-Kurses, doch damit sind wir noch längst nicht am Ende des Projekts angekommen! Mit einer neuen Platine (**Bild 7**) erweitern wir das Elektor-Linux-Board um viele sehr nützliche Funktionen:

Display (2 x 16 Zeichen)
drei Taster als Bedieneinheit bzw. Steuerung für Menüs
16 zusätzliche digitale Ein- und Ausgänge
RTC (Echtzeituhr) mit Batterie zur Versorgung des Systems mit der Uhrzeit
Summer zur akustischen Ausgabe von Meldungen
Experimentierfeld für weitere Schaltungen

In der nächsten Ausgabe stellen wir dieses Board ausführlich vor!

(120518)

Weblinks

[1] sauter@embedded-projects.net
[2] www.gnublin.org
[3] www.elektor.de/120180
[4] www.elektor.de/120578
[5] http://en.gnublin.org/index.php/Eclipse
[6] www.elektor.de/120181
[7] http://open-source.katalix.com/libi2c/
[8] www.elektor-magazine.de/120518
[9] www.tldp.org/HOWTO/Serial-Programming-HOWTO/
[10] www.elektor.de/110727
[11] www.elektor.com/elektorbus
[12] www.thingiverse.com/thing:29314
[13] www.element14.com/community/community/events/electronica
[14] http://eagleup.wordpress.com
[15] www.elektor.de/120182