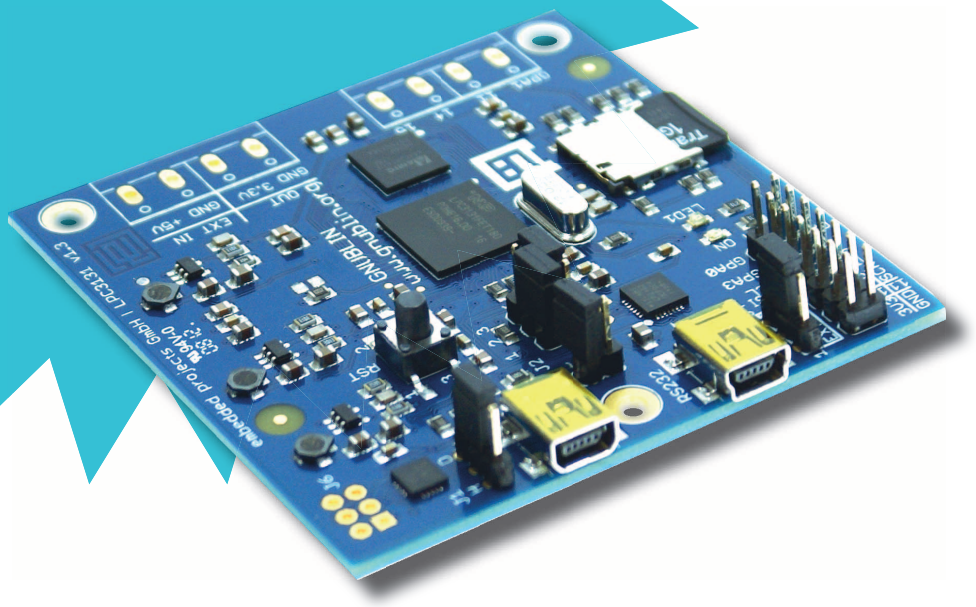
embedded-projects.net
JOURNAL
OPEN SOURCE SOFT-AND HARDWARE PROJECTS

Eine Open-Source Zeitschrift
zum Mitmachen!

GNU/LINUX

GNUBLIN



[PROJECTS]

- Embedded Linux mit GNUBLIN
- Ajax für Home Control
- Mit GNUBLIN im Netzwerk arbeiten
- Reif für den Hive?
- Das Infinity Racing Team
- Audiotrenner

Einleitung

GNUBLIN Start 2012

Embedded Projects Journal - Ausgabe No. 12

Startschuss zum Basteln

Angesteckt wurde ich mit dem Virus „Computer“ als keiner Junge, als ich zum ersten Mal an den schon damals uralten C64 meines Vaters durfte. Ich entdeckte schnell die Vorzüge, wie ich mit dem integrierten Editor meinen Punktestand im Spiel zu Beginn auf kurz vor Sieg setzen konnte. Nach und nach schrieb ich nur noch die verschiedensten Programme in Basic. Durch diesen längeren Prozess reifte - wie bei jedem, der etwas gerne und oft macht - der Erfahrungsschatz immer mehr aus. Man kann sich dann einfach treiben lassen, auf dem Weg zu neuen Ideen, Visionen, Projekten, etc. Das ist in vielen Disziplinen ähnlich: Man braucht immer einen gewissen Schatz an Zeit und Erfahrung, bis man an den Punkt gelangt, etwas wirklich neues, kreatives oder eigenes zu entwickeln.

Als kleiner Junge wollte ich nach und nach verstehen, wie ein Computer von Anfang bis Ende funktioniert. Das war meine persönliche Motivation. Der Weg bedeutete viel ausprobieren, studieren, versuchen und ab und zu auch etwas zu zerstören. Aber so hatte ich immer mehr das Wissen und Verständnis, was das Innere eines Computers ist.

In der Thematik Embedded Systemeist diese Einarbeitung ein wesentlicher Teil der Grundausbildung und ist vor allem eines: Die Grundlage Spass daran zu haben, die Technik zu berherschen und mit den vielen kleinen einfachen Chips spannende Anwendungen zu kreieren.

Es braucht viel Zeit zur Entwicklung einer Idee und einer späteren Leidenschaft für bestimmte Bereiche und für die Suche nach mehr. Toll ist es, wenn man langsam an die Thematik herngeführt wird, dann reifen viele Ideen nach und nach von selbst. Langsam einen passenden Mikrocontroller auswählen, die Schaltung samt Quarz und all den weiteren Bauteilen mal eben aufbauen... Typische Fehler bei der ersten Inbetiebnahme machen, einen Chip auch einmal abrauchen lassen. Nach und nach verstehen wie das alles so funktioniert.

Daher freue ich mich in dieser Ausgabe besonders auf einen Artikel wie den „Reif für den Hive“ oder auch unser neues Open-Source Projekt „Gnublin“ vorstellen zu können.

Technik für Bastler oder die es werden wollen. Technik für Menschen, die sich die Zeit nehmen wollen und hinter die Kulissen schauen möchten und vielleicht schon länger immer mal wieder das Bedürfnis erspüren mal wieder etwas ergründen zu wollen :)

Viel Spaß beim Lesen, Nachbasteln und kreativ sein wünscht

Benedikt Sauter
sauter@embedded-projects.net

und das embedded projects Team.

wawision.embedded-projects.net

waWision -
die Steuerzentrale für Ihre Firma

Verwal-tung	Plug & Play	Waren-eingang	Marke-ting
FiBu	Produk-tion	automa-tisches Lager	Online-Shops

EIN SYSTEM AUS EINER HAND

- keine Installation
- Betriebssystem unabhängig
- Standardhardware Plug & Play
- mitwachsend

DEMOVERSION

weitere Infos finden Sie auf unserer Internetseite

Holzbachstraße 4, D-86152 Augsburg
Tel +49 (0) 821 279599-0, Fax +49 (0) 821 279599-20
info@embedded-projects.net

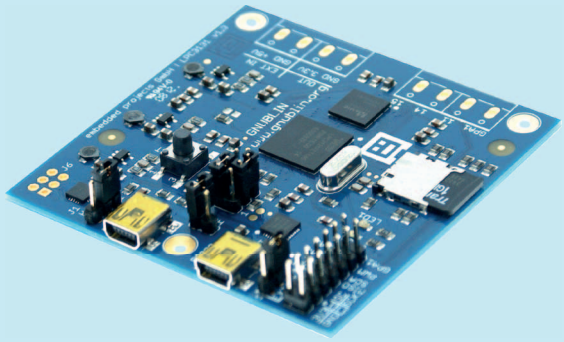
embedded projects GmbH

HARDWARE FOR PROJECTS

GNUBLIN

Embedded Linux mit Gnublin

Hubert Högl <Hubert.Hoegl@hs-augsburg.de>



In diesem Artikel stelle ich den aktuellen Stand des Gnublin LPC3131 Projektes vor. Im gleich folgenden Abschnitt zeige ich zunächst für Neugierige eine beispielhafte Sitzung mit Gnublin.

Eine kurze Sitzung

Um mit dem Gnublin Board zu arbeiten, muss man zusätzlich nur zwei Sachen haben:

- Eine Micro SD Karte mit vorinstalliertem GNU/Linux für Gnublin.
- Ein USB Kabel zwischen der USB Konsolenbuchse auf dem Gnublin Board und einem Entwicklungsrechner (PC) unter GNU/Linux.

Auf dem PC startet man kurze Zeit nach dem Anstecken des USB Kabel ein Terminal Emulationsprogramm, z.B. picocom:

```
sudo picocom -b 115200 /dev/ttyUSB0
```

Man beendet ``picocom`` mit der Tastenkombination ``Strg-A Strg-X``. Sollte picocom noch nicht installiert sein, findet man es als Paket mit gleichem Namen in jeder GNU/Linux Distribution. Damit wird der PC zur „Konsole“ des Gnublin Boards, das heisst, man sieht die Meldungen des Boards auf dem Bildschirm und Tastatureingaben am PC werden an das Board weitergeleitet. Nachdem das Board über das Konsolen-USB-Kabel angesteckt wurde, lädt es von der SD Karte zunächst den Apex Bootloader und führt ihn aus. Der Bootloader ist so konfiguriert, dass er nach einer kurzen Wartepause den Linux Kern von der SD Karte in den Speicher kopiert und dann ausführt. Linux bootet nun und mountet am Schluss das Root Filesystem von der SD Karte. An der Konsole sollte man nun sehen (die drei Punkte stehen für viele Ausgaben, die uns jetzt nicht interessieren):

```
...
EDLK (Built by Poky 5.0) 5.0 armv5te ttyS0
armv5te login:
```

Jetzt kann man sich mit dem Namen root anmelden (ohne Passwort) und sieht dann diesen Shell Prompt:

```
root@armv5te:~#
```

Nun kann man mit Gnublin (fast) wie auf einem gewöhnlichen GNU/Linux Rechner arbeiten. Das Root Filesystem enthält mit über 600 MByte Grösse eine sehr grosse Auswahl an Programmen, allerdings auch Programme, z.B. das X Window System, das die Gnublin LPC3131 Hardware nicht unterstützt.

Nun gibt man auf dem Shell Prompt ein:

```
root@armv5te:~# echo 3 > /sys/class/gpio/
export

root@armv5te:~# echo out > /sys/class/gpio/
gpio3/direction
```

```
root@armv5te:~# echo 1 > /sys/class/gpio/
gpio3/value
```

Die rote LED auf dem Board sollte nun leuchten, da der value (Wert) des GPIO Pins 1 geworden ist (das entspricht 3,3 Volt Spannung). Sie ist zwischen SD-Karte und Erweiterungstecker direkt neben der grünen Betriebs-LED angebracht. Die LED ist an den GPIO Pin Nummer 3 angeschlossen, deswegen taucht die Zahl 3 im vorherigen Beispiel auf. Durch die Ausgabe einer Null (logisch 0 bzw. 0 Volt) auf den Pin erlischt die LED wieder:

```
root@armv5te:~# echo 0 > /sys/class/gpio/
gpio3/value
```

Danach kann man den GPIO Pin 3 wieder „aufräumen“ mit:

```
root@armv5te:~# echo 3 > /sys/class/gpio/
unexport
```

Am Schluss schalten wir das Board mit dem Kommando halt aus:

```
root@armv5te:~# halt
```

Nach ein paar Sekunden gibt der Rechner die Meldung System halted. aus, danach kann man das USB Kabel wieder abstecken. Viel mehr Informationen zum Ansteuern der Peripherie findet man auf der Gnublin Homepage http://www.gnublin.org (Abb. 1)

Insbesondere für Einsteiger gibt es dort folgende Informationen:

- **Schnelleinstieg**
  http://www.gnublin.org/index.php?title=LPC3131-Schnelleinstieg

Abb. 1: Projektseite GNUBLIN: http://www.gnublin.org

- **Für Anfänger**
  http://www.gnublin.org/index.php?title=LPC3131-Anfaenger
- **Videos**
  http://www.gnublin.org/index.php?title=LPC3131-Videos

Wer schon etwas mehr Erfahrung mit Gnublin hat, wird sich in den Gnublin Application Notes umsehen und dort Anregungen für neue Experimente finden:

http://www.gnublin.org/index.php?title=ApplicationNotes

Ursprung

Das Gnublin Projekt [1] läuft nun schon eine ganze Weile (seit 2010) und hat in dieser Zeit etliche Haken geschlagen. Ganz am Anfang stand die Idee, ein Board selber zu bauen, das für Embedded Linux tauglich ist und das wir in der Ausbildung an der Hochschule Augsburg in der Technischen Informatik den Studenten zu einem günstigen Preis anbieten können. Als Erfahrungswert für die Obergrenze des Preises haben sich etwa 50 Euro herausgestellt. Wenn der Preis diesen Betrag nicht übersteigt, kaufen sich erfahrungsgemäß viele Leute das Board und denken sich eigene Experimente damit aus. Früher haben wir das „NGW100“ zu einem ähnlichen Preis bekommen, leider hat Atmel den AVR32 Controller eingestellt.

Das Board sollte von Anfang an einen Controller mit ARM Kern haben, gerne einen ARM926, der sehr weit verbreitet ist. Ich kann mich noch an ein selbstgemachtes Blockschaltbild mit dem Atmel SAM9G45 erinnern, der allerdings wegen des relativ hohen Preises und der schlechten Verfügbarkeit gegen einen Freescale i.MX287 eingetauscht wurde. Das war dann der Startschuss für die erste Phase des Gnublin-Projektes [3], eine sehr fruchtbare Zeitspanne, in der viel geplant und entwickelt wurde, in der Neues gelernt wurde und die zum Schluss dann auch die Pläne für ein 7x7 cm großes Board mit dem i.MX287 hervorgebracht hat. Auf der Embedded World 2011 haben wir uns zum Beispiel deswegen über Löt-Verfahren informiert. Hergestellt wurde dieses Board bis jetzt nicht, da die rohen Platinen mit 8 Lagen bei kleinen Stückzahlen sehr teuer sind.

Die nächste Wende kam mit dem „Gnublin LPC3131“ Board, bei dem Benedikt Sauter in einer Blitzaktion die Pläne radikal vereinfacht hat und eine Platine mit nur 2 Lagen daraus gemacht hat. Auf dem Board war nun ein preiswerter NXP LPC3131, bei dem viele der BGA-Bälle wegen der geringen Routing-Möglichkeiten der 2 Lagen nicht angeschlossen wurden. Ansonsten ist noch ein 8 MByte SDRAM Baustein und eine Micro-SD Karte als Massenspeicher drauf. Das entstandene Board hat aber trotz seines minimalistischen Ansatzes viele Einsatzmöglichkeiten, einige werde ich auch im Folgenden ansprechen. Da die rohen Platinen bei 2 Lagen nur einige Euro kosten kann man nun auch viel unverkrampfter eigene Lötversuche mit BGA Gehäusen unternehmen. Die Materialkosten liegen bei dem Board mit etwa 20 bis 25 Euro nur unwesentlich höher als bei einem Board mit 8-Bit Controller.

Ziele

- Günstig im Preis (<= 50 Euro)
- Geeignet für Anfänger:
  - Einfache Verwendbarkeit über 1 x USB
  - Natives Arbeiten auf dem Board mit Interpretersprachen
  - gnublin-installer verwaltet Micro-SD Karte
  - Anleitungen auf www.gnublin.org
- Transparenz der Bestandteile Bootloader, Kernel, Root-Filesystem
- Hostrechner mit Linux oder Windows
- Alle Teile (auch Hardware) unter freier Lizenz

Ein ganz wichtiger Punkt war neben dem geringen Preis auch immer die einfache Verwendbarkeit des Boards, so dass Anfänger nicht schon bei den Vorbereitungsarbeiten wie der Verkabelung und der Stromversorgung abgeschüttelt werden. Als ultimative Lösung erschien mir immer eine einzige Verbindung mit einem USB Kabel zum Hostrechner, über das die Stromversorgung und auch die serielle Schnittstelle für die Konsole bereitgestellt wird. Deswegen gibt es auf Gnublin eine USB-zu-UART Bridge.

Als mögliche Interessenten für Gnublin stelle ich mir auch Leute vor, die bisher gar keine Erfahrung mit Linux gesammelt haben. Für sie soll es möglich sein, mit dem Board nativ in einer interpretierten Sprache zu arbeiten, zum Beispiel in der Shell, in Lua oder in Python, so dass Programme direkt auf dem Board entwickelt und getestet werden können. Je nach den Vorkenntnissen des Anwenders können aber auch Programme in C direkt auf dem Board kompiliert werden.

Eine wichtige Zutat zum Board ist die Micro-SD Karte, die den Bootloader, den Linux Kernel und das Root-Filesystem enthält. Da es für Anfänger nicht so leicht ist, eine neue Karte mit den frei herunterladbaren Bestandteilen zu bauen, gibt es das grafische Werkzeug gnublin-installer. Es übernimmt die Partitionierung, die Formatierung, den Download und das Kopieren der Dateien auf die Karte. Das Programm wird weiter unten kurz vorgestellt.

Ich könnte mir die Verwendung von Gnublin in Zukunft so vorstellen: Nach dem man sich das Board besorgt hat, steckt man es an den Linux Hostrechner, installiert das in allen Distributionen verfügbare Paket gnublin und kann dann mit dem Board arbeiten, das heißt man kann auf der Konsole arbeiten, oder die Micro-SD Karte beschreiben, oder Dateien zum/vom Board übertragen. Um mehr Windows-Anwender zu gewinnen, sollte es dieses Programm auch für Windows geben. Wie gesagt, das ist noch Zukunftsmusik, aber gar nicht in so weiter Ferne. Mit gnublin-installer ist bereits ein erster Schritt gemacht. Zusätzlich hat das Gnublin Board eine Homepage [1], mit deren Hilfe man als Anfänger die wichtigsten einfachen Anwendungsfälle nachvollziehen kann. Neben Material für Anfänger gibt es aber auch Bereiche für fortgeschrittene Anwender, die sich eher in die Bereiche vorwagen, die mit Systemprogrammierung, mit Echtzeit und mit dem Kernel (Konfigurieren, Kompilieren, Treiber) zu tun haben, um nur ein paar Beispiele zu nennen. Ganz gerne würden wir auf gnublin.org auch Projekte vorstellen, die mit Gnublin gemacht wurden.

Klar ist, dass die Quellen für die Bestandteile Bootloader, Kernel und Root Filesystem (auch Schaltplan und Platinenpläne) mit den für Gnublin notwendigen Änderungen einfach zugänglich

sind, so dass sich auch Aussenstehende daran bei der Weiterentwicklung beteiligen können - das fordert ja schon die Lizenz GPL. Bei allen Bestandteilen besteht außerdem eine (zumindest einseitige) Verbindung zu den jeweiligen Upstream Projekten, das heißt, Gnublin sollte an den Änderungen, die in Upstream im Laufe der Zeit gemacht werden, teilhaben können. Das betrifft vor allem den Kernel und das Root Filesystem. Dazu weiter unten mehr.

Die Gnublin Hardware

Der LPC3131 ist ein Controller mit ARM926 Kern, der mit maximal 180 MHz Taktfrequenz läuft. Die Peripherie kommt nicht im Überfluss daher, reicht aber ideal für das, wofür Gnublin konzipiert wurde. Es gibt alle üblichen Schnittstellen wie GPIO, I2C, SPI, PWM und ADC, allerdings nur einen UART (Konsole). Vor allem hat der Controller jedoch eine high-speed USB OTG Schnittstelle, die sich als ideale Erweiterungsschnittstelle für Gnublin erweist. Da der Controller kein Ethernet MAC Modul hat, bietet es sich an, USB-zu-Ethernet Adapter oder USB-zu-WLAN Adapter anschliessen, um Gnublin mit einem Netzwerk zu verbinden (siehe Abb. 2).

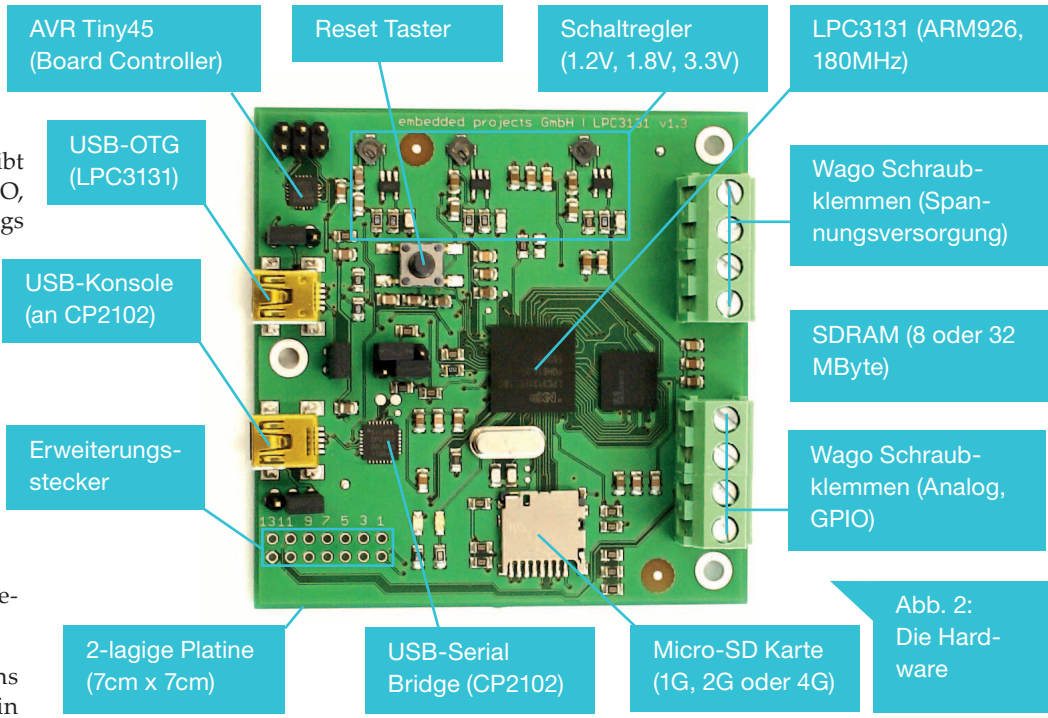


Abb. 2: Die Hardware

Abbildung 3 zeigt die wesentlichen Peripherieblöcke des LPC3131.

Das BGA-Gehäuse des LPC hat 180 Pins in einer 14 x 14 Matrix mit 0,8mm Pin Abstand, in der Mitte fehlen 4 x 4 Pins (TFBGA180). Da die Platine nur zwei Lagen hat, mussten viele Pins offen gelassen werden.

Die Spannungsversorgung und die Konsole werden über eine Mini-USB Buchse auf das Board gebracht. Die USB Verbindung wird über eine CP2102 (Silabs) USB-zu-UART Bridge mit dem einzigen UART des LPC verbunden.

Es sind 8 MByte Arbeitsspeicher auf dem Board. Das Mobile SDRAM ist als 4M * 16 organisiert. Diese Speichergrösse ist zwar nicht gerade üppig, reicht aber für viele Anwendungen aus. An die Grenzen des Speichers stösst man jedoch, wenn man zum Beispiel mit gcc nativ auf dem Board kompilieren möchte (kleinere Programme dauern schon etwa 30 Sekunden und benötigen eine Swap-Datei auf der SD-Karte) Man kann das Board aber auch mit einem 32 MByte SDRAM Baustein bestücken.

Auf dem Board ist ausserdem ein Board Controller mit einem AVR Tiny45, der sich um die Sequenzierung der drei Spannungsversorgungen kümmert.

Die Stromaufnahme liegt bei etwa 85 bis 100 mA, was einer Leistungsaufnahme von etwa einem halben Watt entspricht.

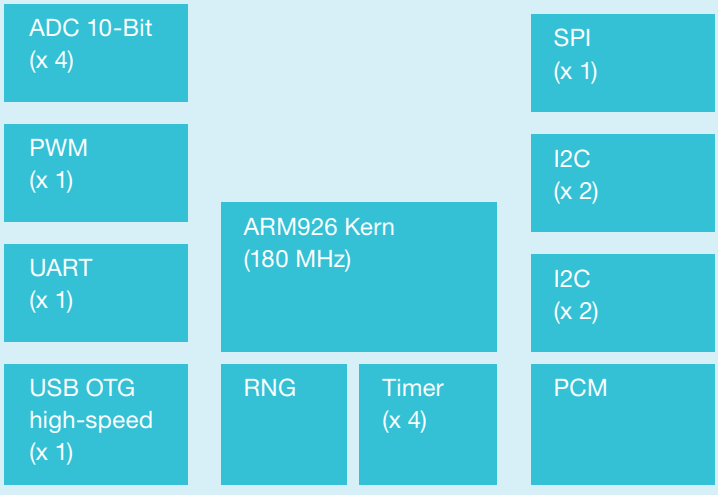


Abb. 3: LPC3131

Bedeutung der Jumper und I/O Steckverbinder

Das Gnublin Board verzichtet aus Gründen der Einfachheit auf technischen Schnickschnack und setzt vielmehr auf explizit gesetzte Jumper (siehe Abb. 4).

Wenn man über die USB-OTG Schnittstelle Geräte als Host ansteuern möchte, muss man den USB OTG PWR Jumper nach links stecken. Zusätzlich sollte man den USB OTG ID Jumper setzen.

Wenn Gnublin über einen externen USB Host angesteuert wird, zum Beispiel einem PC, muss man den USB OTG PWR Jumper nach rechts stecken und der USB OTG ID Jumper darf nicht ge-

setzt sein. Je nachdem woher die Stromversorgung des Board kommt, muss der Power Supply Jumper entweder rechts (USB Console) oder links (extern) gesteckt sein.

Die Betriebssoftware

Alle Teile der Betriebssoftware können von http://www.gnublin.org/downloads heruntergeladen werden.

Der Apex Bootloader

Wir verwenden den Apex Bootloader. Der Bootloader gibt den Prompt apex> aus, danach kann man Kommandos eingeben. Um den Kernel aus der zweiten Partition (ext2) in den Speicher an die Stelle 0x30008000 zu kopieren und anschliessend zu starten laufen die folgenden Kommandos bei jedem Bootvorgang automatisch ab:

```
apex> copy ext2://2/
zImage 0x30008000

apex> boot
```

Mit dem Kommando apex> help bekommt man eine Liste aller verfügbaren Kommandos.

Eine Anleitung zum Kompilieren findet man auf:

http://www.lpclinux.com/LPC313x/LPC313xGettingstartedEL-DK

Die Anpassungen für Gnublin findet man hier:

http://elk.informatik.fh-augsburg.de/pub/eplpc3131/work_eplpc3131/apex/

Es gibt ausserdem auch den U-Boot Bootloader, allerdings ist dieser noch nicht für das Gnublin Board angepasst worden.

Der Bootvorgang: Im LPC3131 gibt es ein Boot-Programm im ROM (first level bootloader), das nach dem Starten verschiedene Quellen nach ausführbaren Daten abtastet. Die Schnittstellen UART, USB, SPI und SD/MMC können als Boot-Quellen verwendet werden. Nachdem im Normalfall die SD/MMC Karte gefunden wurde, wird der Bootloader von der Karte geladen. Die ausführbare apex.bin Datei ist etwa 50 KByte gross. Sie wird in das 192 KByte grosse interne RAM des LPC3131 geladen und dort ausgeführt (second level bootloader). Apex kann dann entweder interaktiv auf der Konsole bedient werden oder es kann ein Boot-Skript automatisch ablaufen. Um auf den Apex Prompt zu gelangen, muss man sofort nach dem Drücken des Reset

Das Board bootet im Normalfall von der SD Karte, deshalb wähle die Einstellung SD/MMC.

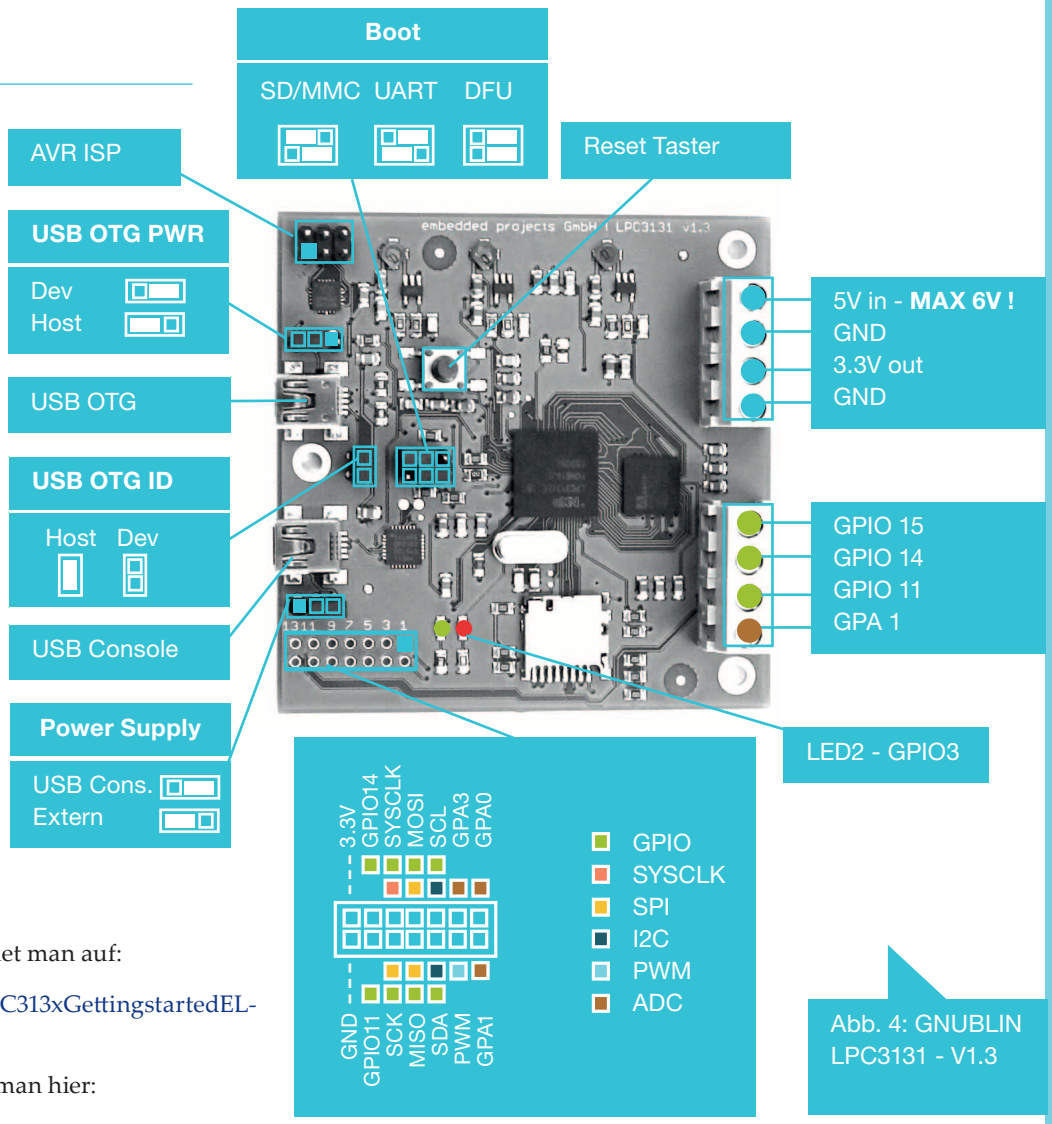


Abb. 4: GNUBLIN LPC3131 - V1.3

Tasters die Tastenkombination Strg-C im Terminal-Programm eingeben.

Die Kernel Kommandozeile wird vom Bootloader wie folgt gesetzt:

```
console=ttyS0,115200n8 root=/dev/mmcblk0p2 ro
rootwait
```

Das Apex Projekt findet man hier:

https://gitorious.org/apex

Eine kurze Beschreibung ist hier:

http://elinux.org/APEX_Bootloader

Der Kernel

Mitarbeiter von NXP und freie Entwickler haben den Linux Kernel 2.6.33 für den LPC3131 angepasst. Man findet die Quellen in folgendem Git Repository:

git://git.lpclinux.com/linux-2.6.33-lpc313x

Die für das Gnublin Board angepasste Version findet man hier:

ssh://git@elk.informatik.fh-augsburg.de:2222/gnublin-linux-2.6.33.git

Das Web Interface ist:

http://elk.informatik.fh-augsburg.de/cgi-bin/gitweb.cgi

Die Gnublin Anpassungen umfassen Änderungen wegen leichter Unterschiede in der Hardware und zusätzliche Treiber für bisher nicht unterstützte Schnittstellen, zur Zeit sind das ADC und PWM. Die Treiber wurden von Michael Schwarz und Nils Stek geschrieben, schönen Dank dafür (siehe [3]).

Auch wurde die sysfs Unterstützung für die GPIO Pins eingebaut (Patch von lpclinux.com), so dass man die GPIO Pins in der üblichen Weise durch Zugriffe auf /sys/class/gpio/ verwenden kann. Zusätzlich gibt es von Michael Schwarz das gpio-module ([3]), das den Zugriff auf die GPIO Pins auf pragmatische Weise erlaubt.

Die Kernel Konfiguration: Die aktuelle Kernel Konfiguration .config findet man im Gnublin Downloadbereich. Die wichtigsten Treiber sind als Module vorhanden. Wichtige Treiber, um Gnublin über USB als gadget anzusprechen oder um andere USB Geräte von Gnublin aus anzusprechen sind:

```
drivers/lpc313x_io/adc/
drivers/lpc313x_io/adc/
lpc313x_adc.ko
drivers/lpc313x_io/pwm/
lpc313x_pwm.ko
drivers/usb/class/cdc-acm.ko
drivers/usb/gadget/
    g_cdc.ko
    g_file_storage.ko
    g_printer.ko
    g_zero.ko
    g_ether.ko
    g_mass_storage.ko
    g_serial.ko
    gadgetfs.ko
drivers/usb/serial/cp210x.ko
    ftdi_sio.ko
    usbserial.ko
drivers/net/usb/
    asix.ko
    cdc_subset.ko
    pegasus.ko
    zaurus.ko
    cdc_ether.ko
    net1080.ko
    usbnet.ko
```

In den Kernel einkompiliert ist die Unterstützung für die seriellen Busse SPI und I2C und für den GPIO Zugriff mittels sysfs. Eine Anleitung zum Kompilieren des Gnublin Linux findet man auf der Gnublin Homepage.

Root Filesystem

Das Root Filesystem stammt vom Embedded Linux Development Kit (ELDK) in der Version 5.0 [8]. Vom Aufbau her ist es mit einer „richtigen“ Distribution für einen grossen Rechner vergleichbar, zum Beispiel ist auch die Unterstützung für grafische Oberflächen dabei und auch der GNU Compiler gcc kann verwendet werden. Dadurch ist das Root Filesystem auch ziemlich gross (550 MByte). Alle grafischen Programme können bei Gnublin nicht verwendet werden, da man kein Grafikdisplay (zumindest keines der üblichen TFT Displays) anschliessen kann. Alle anderen textuellen Programme können verwendet werden.

Seit dem ELDK 5.1 gibt es auch kleinere Varianten des Root Filesystems. Das ELDK ab der Version 5 wird mit einem mächtigen Distributionsbaukasten erzeugt, der auf OpenEmbedded und Poky Linux baut. Zusammengefasst werden diese Werkzeuge im Yocto Project [9]. Auf der ELDK Homepage findet man die „Baudaten“ für die jeweiligen ELDK Distributionen, aus denen man sich die konkreten Distributionen selber kompilieren kann. Man braucht dazu aber einen schnellen Rechner mit viel Plattenplatz.

Neben dem ELDK lassen sich auch andere Root Filesysteme für Gnublin verwenden. Fast alle grossen Distributionen wie Debian, Gentoo, Fedora und andere bieten für ARM angepasste Varianten an, die bereits über riesige Paket-Archive mit vorkompilierten Programmen verfügen. Es wäre zum Beispiel relativ einfach möglich ein Debian für ARM (Architektur armel) auf die Gnublin SD Karte zu schreiben. Damit könnte man dann im Betrieb wie bei einem „grossen“ Rechner mit dpkg und verwandten Werkzeugen zusätzliche Pakete aus dem Netz installieren.

So ganz einfach darf man sich den Umstieg auf ein neues Root Filesystems allerdings auch nicht vorstellen. Wenn man nur das „rohe“ Dateisystem auf die SD Karte kopiert, bootet der Rechner mit Sicherheit nicht. Es müssen vorher einige Konfigurationsdateien angepasst werden, vor allem im Bereich des init Systems. Auch die Konfiguration von udev ist eine sensible Sache. Da ich bei beiden Sachen keine grosse Erfahrung mitbringe, habe ich beim ELDK mehr oder weniger durch Weglassen von Diensten irgendwann die Bootfähigkeit geschafft. Beim aktuellen Root Filesystem mit dem ELDK 5 ist an dieser Stelle also noch viel Platz für Verbesserungen (udev ist zum Beispiel jetzt noch abgeschaltet).

Im Laufe der Zeit wurde das Root Filesystem ergänzt um wichtige Programme (Tabelle 1):

Root Filesystem	
Editoren	jove, zile (einfache Emacs-Klone) nano (für Anfänger bedienbar)
Drahtlose Netzwerke	iwconfig und andere iw* Utilities, wpa_ supplicant
Web	lighthttpd CGI lynx
Tools	screen tree wget curl netcat gnuplot rrdtool lrzsz alsa-utils vorbis-tools (u.a. ogg123)
Interpretersprachen	Python 2.7.2 Lua 5.1.4 Jim (Tcl) Squirrel Hedgehog Lisp gforth

Tabelle 1: Root Filesystem

Gnublin Installer

Der gnublin-installer ist ein GUI Werkzeug zum Partitionieren und Beschreiben der Micro-SD Karte. Das Programm hat Michael Schwarz [3] geschrieben. Der folgende Screenshot zeigt das Hauptfenster (Abb. 5):

Das Programm wird auf dem Hostrechner unter Linux gestartet. Die Micro-SD Karte muss über einen Kartenleser in den PC eingesteckt sein. Der Installer zeigt alle Karten kleiner einer bestimmten Speicherkapazität an (z.B. 16G), man wählt die Karte aus und kann diese dann durch Knopfdruck, ohne die technischen Details zu kennen, so aufbereiten, dass sie in Gnublin eingesteckt sofort funktioniert. Dazu werden folgende Aktionen ausgeführt:

- Löschen von vorhandenen Partition, erneute Partitionierung und anlegen eines Filesystems mit libparted.
- Download von Bootloader, Kernel und Root Filesystem vom Gnublin Downloadbereich.
- Kopieren der Daten auf die Karte.

Das Programm ist in der Sprache C geschrieben und verwendet WxWidgets, libparted, licurl und libarchive.

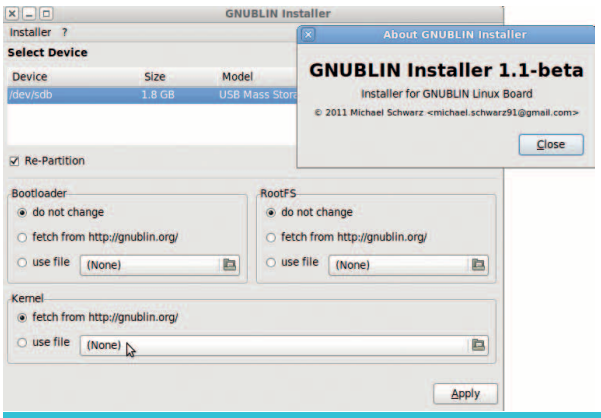
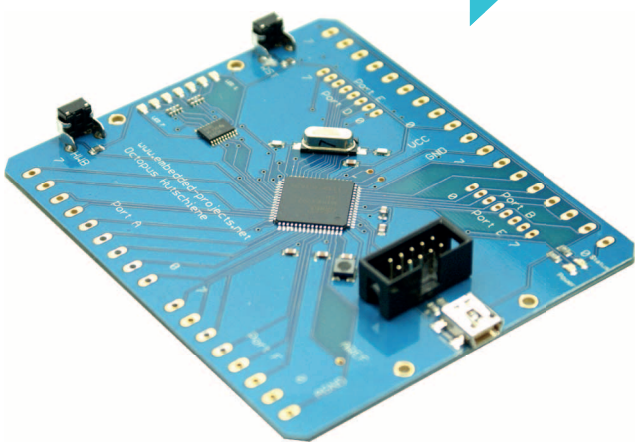


Abb. 5: GNUBIN Installer

Abb. 6: Octopus-Hut



Was läuft bisher alles?

Im Laufe der letzten Monate haben sich bei mir wie bei einem Baukasten viele Hardware Erweiterungen und Werkzeuge um das Gnublin Board angesammelt. Die meisten habe ich in der folgenden Abbildung zusammengestellt (Abb. 7):

Hier ist eine kurze Beschreibung zu den Teilen:

1) Gnublin LPC3131 Board
2) Micro SD Karte (1G, 2G, 4G)
3) Adapter von Mini USB auf USB-A Buchse zum Anschliessen von USB Geräten an den USB OTG Host.
4) USB zu Ethernet Adapter (mit „Pegasus“ Chip)
5) USB WiFi Adapter Asus WL-167G (ohne Abb.)
6) USB Bluetooth Adapter D-Link DBT-120
7) USB zu Seriell Adapter mit Ftdichip FT2232
8) USB zu Seriell Adapter mit Silabs CP2102: http://ic-board.de
9) USB Flash Dongle
10) USB Audio Adapter Speedlink VIGO
11) USB Hub zur allgemeinen Verwendung
12) USB Hub zur allgemeinen Verwendung
13) Diverse integrierte Schaltungen mit I2C oder SPI Schnittstelle (ohne Abb.)
14) USB Kartenleser für MicroSD Speicherkarten
15) LEDs zum Anschliessen an GPIO Pins (ohne Abb.)
16) Schraubenzieher
17) DOG Display 2x16 Zeichen mit SPI Anschluss
18) Batterie/Akku (**Vorsicht**: Die extern eingespeiste Spannung darf nicht grösser als 6V sein! Bei der gezeigten 9V Blockbatterie muss also noch ein 5V Regler dazwischengeschaltet werden.)
19) USB Kabel USB-A auf Mini-B

Ausserdem kann man die Octopus Erweiterung verwenden (siehe [6] und Abb. 6). Octopus besteht nur aus einem Atmel AVR AT90USB1287, der mit einer Firmware ausgestattet ist, die es erlaubt, fast alle Schnittstellen des AVR ferngesteuert über USB zu nutzen. Eingesteckt in die Gnublin USB Host Schnittstelle kann man mit der liboctopus auf die Schnittstellen des Octopus zugreifen.

Der Sourcecode liegt hier:

http://embeddedprojects.googlecode.com/svn/trunk/OPEN-SOURCE/octopus_at90/

Den Schaltplan und weitere Informationen dazu findet man auf:

http://shop.embedded-projects.net.

Im Folgenden (Tab. 2) ist eine Zusammenstellung der Experimente, die ich bisher mit Gnublin gemacht habe, ein paar stammen auch von [3]:

Wie soll man Anwendungen schreiben? Mein Vorschlag wäre, dass man möglichst alle Eigenschaften des Boards aus Skriptsprachen nutzen kann. Es gibt einige interpretierte Sprachen, die man auf Gnublin nutzen kann, z.B. Python, Perl, Lua oder Squirrel. Dann gibt es auch noch die bash Shell. Ganz exotisch dürften für die meisten GNU Forth (gforth) oder Hedgehog Lisp sein. Alle diese Sprachen kann man auf Gnublin verwenden.

Die aus meiner Sicht „beste“ Sprache für eine komplette interpretierte Programmierumgebung ist Python [10]. Volker Thoms hat sich schon in [4] mit dem Thema auseinandergesetzt, ob man die gängigen Schnittstellen für die Mikrocontrollerprogrammierung wie GPIO, UART, SPI, I2C, CAN aus Python heraus verwenden kann. Es geht sogar sehr gut, wie sich herausstellte. Die dabei entstandenen Python Module kann man sehr einfach auf Gnublin übertragen. Die Programmierung in Python ist für den Anfänger wesentlich einfacher erlernbar als andere Sprachen, vor allem C und Shell Skripte.

Experimente	
UART	USB-zu-Seriell Adapter - Mit Silabs CP210x - Mit Ftdichip FT232/FT2232 Programmierung der seriellen Schnittstelle in C (librs232) und Python (pyserial)
Netzwerk	USB-zu-Ethernet Adapter - Noname Produkt mit pegasus Chip - D-Link DUB-E100 (asix) USB-zu-WiFi Adapter - Asus WL-167G (rt2500) - Asus WL-167G V3 (RTL8192SU) - Patriot 802.11n USB Adapter (RTL8192SU) - D-Link WiFi Stick (rt73)
Webcam	gspca Treiber
GPIO	Rote LED ein-/ausschalten Tasten lesen
SPI	DOG Display 2x16 Zeichen mit SPI Schnittstelle
I2C	PCA9555 I/O Expander PCF8574 I/O Expander
PWM	PWM Signal mit bestimmten Puls/Pausen-Verhältnis ausgeben
ADC	Test des ADC Treibers.
Web/CGI	Einfaches CGI Skript in C, Perl und Python, das durch lighttpd aufgerufen wird
Audio	Alsa und ogg-vorbis Tools installiert. Mit einem USB Audio Adapter ogg Dateien wiedergegeben.

Tabelle 2: GNUBIN Experimente

Das Gnublin Home-Verzeichnis enthält viele Code-Schnipsel in unterschiedlichen Sprachen. Zur Zeit sieht es etwas unaufgeräumt aus:

```
TERATERM.INI  demo.py          hello.py        octopus
adc           dmesg.txt        i2c-device.sh   prozesse.txt
blink-two.sh  gnuplot          i2c-pca9555     pwm.c
blink.lua     gnuplot-nox-4.4.0.tgz  input.nut  spi
blink.sh      go               input.sh        usb-eth.sh
blink2.lua    gpio-int-test    libusb-0.1.12-demo
cyclic.sh     gpio14.sh        lighttpd-init.sh
demo.pl       hello.fs         mount-usbfs.sh
```

In Zukunft findet man diese Schnipsel auch auf der Gnublin Homepage. Die Homepage enthält ausserdem Application Notes, wie man die aufgeführte Hardware an Gnublin in Betrieb nehmen kann.

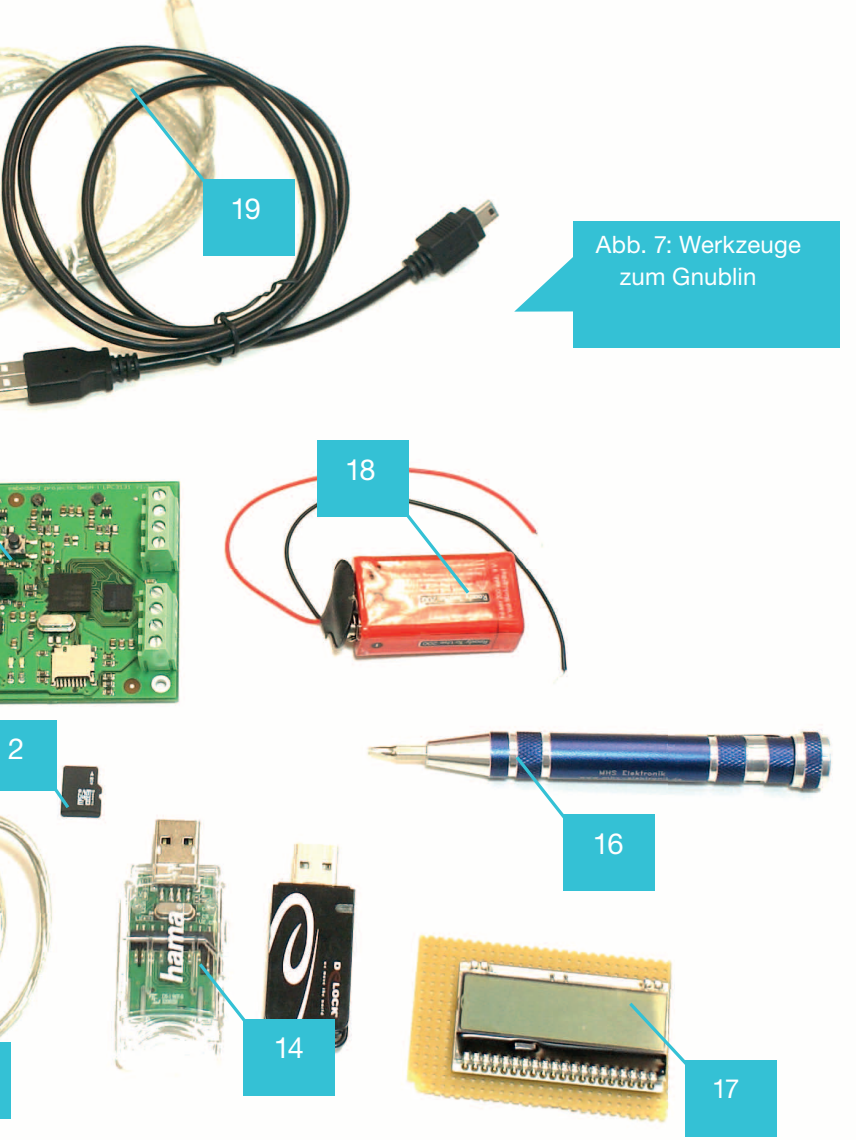


Abb. 7: Werkzeuge zum Gnublin

Ausblick

Mit Gnublin ist es möglich, preiswerte, kleine und tragbare Geräte selber zu bauen, die eine oder mehrere der folgenden Anforderungen haben:

- Einlesen von Sensoren für
  - Temperatur
  - Luftfeuchtigkeit
  - Luftdruck
  - Windgeschwindigkeit
  - Beschleunigung (ich teste gerade 3-Achsen Accelerometer an Gnublin)
  - viele andere ...
- Ansteuern von
  - LEDs
  - Kleinen Displays
  - Tastern
  - Motoren
  - Servos
  - beliebige Erweiterungen über SPI und I2C
- Funkkommunikation über low-power RF Module, z.B. Nordic nRF24L01 oder TI CC2420 oder ähnliche.
- GSM/GPS Module

Ein paar beispielhafte Anwendungen, die man realisieren könnte:

- Autonome Datenlogger die über längere Zeit Umgebungsdaten auf die SD Karte aufzeichnen, z.B. Temperaturen in einem Passivhaus.
- GPS Logger
- Fernwartung über GSM Mobilfunk
- Solargespeiste Wetterstation mit Funkanbindung (WLAN, 802.15.4)
- Heizungssteuerung
- Vernetzte Sensorknoten (drahtlos mit IEEE 802.15.4)
- Uhren bzw. Wecker
- Steuerung der LED Beleuchtung für ein Puppenhaus (ist bald meine Aufgabe)
- Jonglier-Keulen mit eingebauten Beschleunigungssensoren und Beleuchtung (interessanter Vorschlag eines ehemaligen Studenten, der in seiner Freizeit jongliert)

Bei allen diesen Beispielen steht die Anbindung an ein Netzwerk im regulären Betrieb nicht im Vordergrund, sondern eher die kompakte Bauweise und der stromsparende Betrieb aus Batterien oder Akkus.

Literatur

[1] Gnublin Homepage: Http://www.gnublin.org

[2] Gnublin i.MX287: http://elk.informatik.fh-augsburg.de/hhwiki/Gnublin

[3] Gnublin auf mikrocontroller.net: http://www.mikrocontroller.net/topic/237277

[4] Volker Thoms, „Mikrocontrollerprogrammierung mit Python“, 2010: http://elk.informatik.fh-augsburg.de/da/da-49/

[5] Community für den LPC3131 (Kernel, Root-Filesystem und anderes): http://www.lpclinux.com

[6] Octopus an Gnublin: http://www.gnublin.org/index.php?title=AppNote-AP0002

[7] embedded projects Journal: http://journal.embedded-projects.net

[8] Embedded Linux Development Kit (ELDK): http://www.denx.de/wiki/ELDK-5/WebHome

[9] Yocto Project: http://yoctoproject.org

[10] Die Sprache Python: http://www.python.org

Ajax für Home Control

Jan-Hendrik Caspers <mail@jh-caspers.de>

Ich möchte mit diesem Artikel darstellen, wie einfach es sein kann, eine dynamische Web-Anwendung für Home Control-Aufgaben zu erstellen. Als Beispiel nutze ich meinen 16Kanal-Dimmer, über den ich die Beleuchtung in meiner Wohnung steuere. Der Dimmer ist am Home Server angeschlossen, der Web-Server läuft also nicht auf dem Dimmer selbst. Die Technik funktioniert aber auch mit embedded Web-Servern.

Der Dimmer

Mein Dimmer ist eine recht einfache Konstruktion. Er besteht aus einem Atmel Mega16 mit 12MHz, auf dem ein Software-PWM läuft. Über einen USB-Seriell-Konverter ist der Dimmer am Server angeschlossen. Die Steuerung des Dimmers erfolgt über 5-Byte-Datenpakete.

Aufbau der Datenpakete 

```
KKWW!
   KK = Kanal
   WW = Wert
   ! = Befehlsabschluss

z.B.  00FF! = Kanal 0 auf
0xFF
   0110! = Kanal 1 auf
   0x10
```

Einführung in Ajax

Bei dynamischen Web-Anwendung wird nicht jedes Mal die komplette Seite neu geladen; es werden nur einzelne Teile aktualisiert. Dies reduziert unter anderem die Datenmenge, die übertragen wird. Hierdurch wird diese Technik auch für embedded Web-Server interessant. Für den Dimmer heißt dies, dass der aktuelle Zustand jedes Kanals regelmäßig aktualisiert wird. Um Daten abzufragen, wird JavaScript benötigt. Es wird das XMLHttpRequest-Objekt verwendet. Dieses ist in allen aktuellen Browsern vorhanden, heißt aber beim Internet Explorer anders. Hier gibt es je nach Version drei verschiedene Objekte. Über eine Methode dieses Objekts wird die Anfrage gesendet, die Antwort löst über einen Event die Verarbeitung der Daten auf und diese können dann auf der Seite dargestellt werden.

Daten abrufen

Da es zurzeit vier verschiedene Namen für das XMLHttpRequest-Objekt gibt, muss das JavaScript zunächst feststellen, welches im verwendeten Browser funktioniert. Hierfür gibt es verschiedene Wege. Ich verwende die folgende Lösung:

```
XMLReq = [
   function() { return new XMLHttpRequest(); },
   function() { return new ActiveXObject(„Microsoft.XMLHTTP“); },
   function() { return new ActiveXObject(„MSXML2.XMLHTTP.3.0“); },
   function() { return new ActiveXObject(„MSXML2.XMLHTTP“); }
   ];
   var req = null;
   for(var i=0; i<XMLReq.length; i++) {
     try {
            if (req == null) {
                   req = XMLReq[i]();
            }
     }
            catch(e) {
            continue;
     }
   }
   if (req == null) throw new Error(„Dieser Browser unterstützt Ajax nicht!“);
```

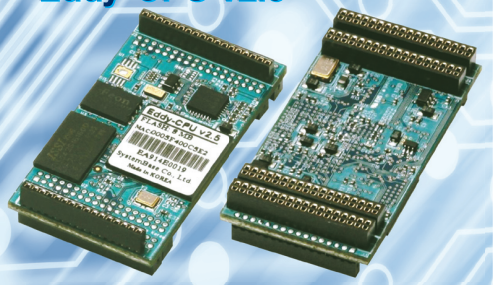
Es werden nacheinander alle Objekte aus dem Array geprüft, bis eines funktioniert. Sollte kein passendes gefunden werden, kommt eine Fehlermeldung. Hier könnte man z.B. auch auf eine Seite ohne Ajax umleiten. Nachdem das Objekt initialisiert ist, kann eine Anfrage gesendet werden. Hierfür wird über die open-Methode die Adresse angegeben und die Funktion zum Behandeln der Antwort über onreadystatechange definiert:

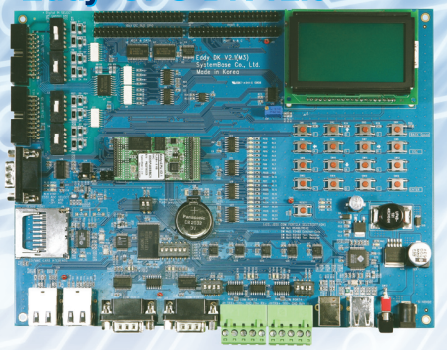
```
    req.open(,get', ,test.txt', true);
    req.onreadystatechange = AjaxEvent;
    req.send(null);
```

Verarbeitung der Antwort

Wenn sich der Status der Anfrage ändert, wird die angegebene Funktion aufgerufen. Sobald der Status den Wert 4 erreicht hat, ist die Anfrage beendet. Wenn der HTTP-Status 200 ist, war die Anfrage erfolgreich. Bei einem anderen Status (z.B. 404) sollte eine Fehlermeldung ausgegeben werden. Der Nachteil dieser Funktion ist, dass kein Timeout eingebaut ist. Sollte der Server nicht antworten, bleibt das Programm hängen. Hier muss zusätzlich ein Timeout definiert werden, dazu später mehr. Die empfangenen Daten können jetzt ausgewertet und im Browser angezeigt werden. Wenn im HTML-Code z.B. der Tag steht, dann kann dieser mit Document.GetElementById(„test").InnerHTML = req.responseText; mit der Antwort überschrieben werden:

```
test.html
<html>
<head>
   <title>Ajax Test</title>
   <script>
   XMLReq = [
     function() { return new XMLHttpRequest(); },
     function() { return new ActiveXObject(„Microsoft.XMLHT
     TP"); },
     function() { return new ActiveXObject(„MSXML2.XMLHT
     TP.3.0"); },
     function() { return new ActiveXObject(„MSXML2.XMLHTTP");
     }
     ];
   var req = null;
   for(var i=0; i<XMLReq.length; i++) {
            try {
            if (req == null) {
                  req = XMLReq[i]();
            }
     }
     catch(e) {
            continue;
     }
   }
   if (req == null) throw new Error(„Dieser Browser unter
   stützt Ajax nicht!");
   function AjaxGet() {
     req.open(,get', ,test.txt', true);
     req.onreadystatechange = AjaxEvent;
     req.send(null);
   }
   function AjaxEvent() {
     if (req.readyState == 4) {
            if (req.status == 200) {
            document.getElementById(,test').innerHTML = req.
responseText;
            }
     }
   }
   window.setTimeout(„AjaxGet()", 5000);
   </script>
</head>
```



```
<body>
<div><b>Ajax Test</b></div>
<span id="test">Dieser Text wird 5Sekunden nach dem Laden
ersetzt!</span>
</body>
</html>
test.txt
```

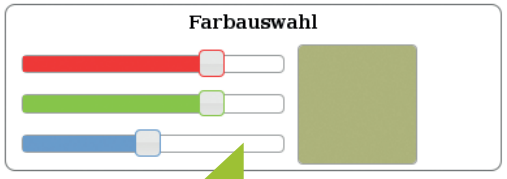


Abb. 1: Farbauswahl

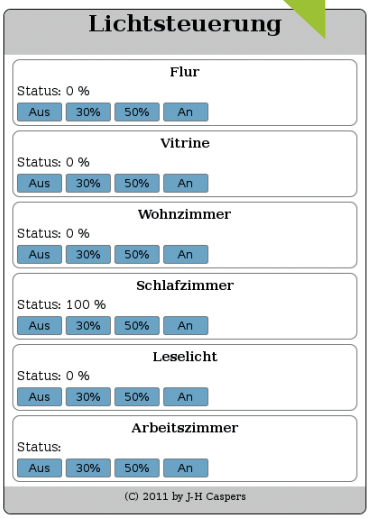


Abb. 2: Lichtsteuerung

Dieser Text wurde per JavaScript nachgeladen. Im Beispiel werden die zuvor gezeigten Funktionen angewendet. Zuerst wird das Objekt erstellt, dann wird 5 Sekunden gewartet, bevor die Anfrage gesendet wird. Die Antwort vom Server wird in den geschrieben.

Als Datenquelle dient eine einfache Textdatei, hier wird später die dynamische Antwort vom Server eingesetzt.

Der Nachteil am Code ist, dass keine Sicherung eingebaut ist, falls die Antwort ausbleibt. Wie zuvor erwähnt, wird dies nicht vom XMLHttpRequest-Objekt erledigt.

Es lässt sich jedoch leicht über JavaScript ein Timeout hinzufügen. Hierfür wird die Funktion AjaxGet wie folgt geändert:

```
function AjaxGet() {
   reqTimer = setTimeout(function() { req.abort();
                  // Hier die Fehlermeldung etc. einbauen
            }, 15000);
   ...
```

Jetzt ist ein Timeout von 15 Sekunden eingebaut. Der Timer muss nach erfolgreichem Empfang jedoch wieder gelöscht werden. Dies funktioniert so:

```
...
   if (req.State == 4) {
     if (reqTimer.) clearTimeout(reqTimer);
     ...
```

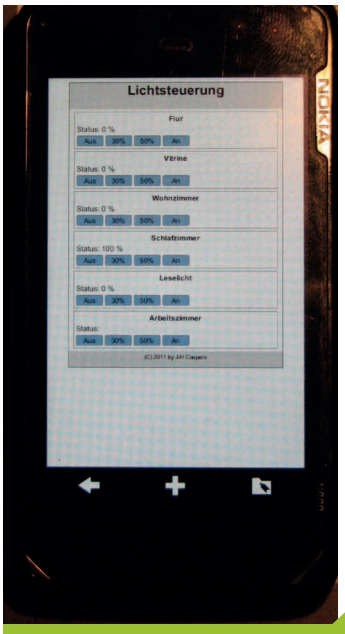


Abb. 3: Steuerung auf dem Smartphone

Komplexere Datenstrukturen

Wenn größere Datenmengen (z.B. die Werte von 16 Dimmer-Kanälen) übertragen werden, ist es sinnvoll, diese per XML oder JSON zu kodieren. Ich verwende JSON. Dieses kann von modernen Browsern „out of the Box" geparsed werden und auch bei älteren Browsern ist es leicht zu parsen.

Modern

```
data = JSON.parse(req.responseText);
```

Für ältere Browser

```
data = eval(,(, + req.responseText + ,)');
```

Achtung! Diese Methode birgt Gefahren, wenn die Datenquelle nicht verlässlich ist. Alles was per eval verarbeitet wird, wird als JavaScript Code ausgeführt. Hier besteht die Gefahr von Code-Injection. Es sollte also, wenn JSON.parse nicht verfügbar ist, ein alternativer Parser verwendet werden.

Beispiel

```
[{„kanal":"1","status":"0"},{„kanal":"2","status":"55"}]
```

Das Beispiel zeigt ein Array aus 2 Dimmer-Objekten mit den Eigenschaften „Kanal" und „Status". Den Aufbau von JSON hier zu erklären, würde den Rahmen des Artikels sprengen. Ich empfehle hier die Links zu Wikipedia[1] oder json.org[2].

Dimmer Frontend

Ich habe für den Dimmer ein Frontend entworfen (Abb. 2), auf dem die Kanäle untereinander dargestellt werden. Zu jedem Kanal gibt es mehrere Knöpfe für verschiedene Lichteinstellungen. Die Größe ist so gewählt, dass die Liste genau auf das Display meines Handys passt (Abb. 3). Auf dem PC ist es dennoch gut zu lesen.

Jeder der Kanäle sieht im HTML-Code so aus:

```
<div class="kanal">
   <div class="kanalname">
   Flur
   </div>
   <div class="kanalstatus">
   Status: <span id="status1"></span>
   </div>
   <div class="kanalcontrol">
   <input type="button" value="Aus"
   onclick="return SetChannel(1, 0);"/>
   <input type="button" value="30%"
onclick="return SetChannel(1, 30);" />
   <input type="button" value="50%"
onclick="return SetChannel(1, 50);" />
   <input type="button" value="An"
onclick="return SetChannel(1, 100);" />
     </div>
           </div>
```

Das mit bezeichnete Feld wird mit der Antwort des Dimmers überschrieben. Hierfür werden die per JSON codierten Daten nacheinander abgearbeitet und alle Kanäle aktualisiert:

```
for (var i=0; i<data.length; i++) {
   if (document.
getElementById(„status"+data[i].kanal) {
   document.getElementById(„status"+
data[i].kanal).innerHTML = data[i].status+"%";
   }
}
```

Die neuen Werte für die Kanäle werden auch per JavaScript über Ajax versendet. So muss auch hier die Seite nicht komplett neu geladen werden. Hierfür ruft die „onclick"-Methode eine Funktion auf und übergibt Kanal und neuen Wert. Die Parameter werden per GET übermittelt. Hierfür werden diese einfach an die Adresse angehängt:

```
req.open(,get', ,control.php?chan='+kanal+'&val
='+wert, true);
```

Die Anzeige wird regelmäßig aktualisiert, über die Timeout-Funktion wird einmal pro Sekunde eine Anfrage an den Server gesendet. Dieser Wert kann beliebig angepasst werden. Ich habe festgestellt, dass bei einer schlechten Mobilfunkverbindung eine Sekunde zu kurz ist.

Datenmenge

Da bei jeder Anfrage nur die Änderungen übertragen werden, der Rest der Seite aber stehen bleibt, werden weniger Daten vom Server zum Browser gesendet. Für den Dimmer mit den abgebildeten 6 Kanälen werden (incl. Header) 2422Bytes übertragen, die JSON-Antwort (inkl. Header) benötigt 432Bytes. Hier werden also schon 2KB eingespart.

Diese Werte stammen von meinem Apache-Server. Durch einen kleineren Header (ohne Servername, PHP Version, etc.) lässt sich noch mehr einsparen.

Für einen embedded Webserver auf einem kleinen Mikrocontroller ist dies eine deutliche Entlastung und lässt mehr Zeit für andere Aufgaben.

Ausblick

Ich verwende zum Teil farbige LEDs (RGB). Hierfür habe ich über die jQuery-Bibliothek ein Kontrollfeld mit Schiebereglern erstellt. Ich kann so die Lichtfarbe stufenlos einstellen (siehe Abb. 1).

Für den PC habe ich eine Bedienoberfläche, die einen Grundriss meiner Wohnung zeigt. Hier sind dann alle Bedienelemente den Räumen zugeordnet.

Ich hoffe, ich konnte mit diesem Artikel zeigen, mit wie wenig Aufwand sich moderne Web2.0 Techniken in die eigenen Projekte integrieren lassen. Die hier gezeigten Techniken können natürlich für viele verschiedene Anwendungen genutzt werden. Es geht vom einfachen Anzeigen von Messwerten bis zu komplexen Lösungen, wie der vollständigen Hausautomatisierung mit Licht, Rolläden und Heizung.

Links

Die Dateien zum Artikel, inklusive der PHP-Dateien zur Ansteuerung des Dimmers stehen auf meiner Website[3] zum Download bereit.

[1] http://de.wikipedia.org/wiki/JavaScript_Object_Notation

[2] http://http://json.org/json-de.html

[3] http://www.cascom.de

Mit GNUBLIN im Netzwerk arbeiten

Philipp Behrendt <phil@beph.de>

Hat man die ersten Schritte auf dem GNUBLIN Board gemacht, möchte man sich schnell außerhalb der seriellen Konsole bewegen und das eingebettete System für den Zweck nutzen, für den es geschaffen wurde: zum Überwachen, Steuern oder Regeln - und das möglichst aus der Ferne. Schließlich ist Linux nicht umsonst als Netzwerkbetriebssystem bekannt.
Hier wird eine Lösung vorgestellt, mit der man mit wenigen Zeilen Quellcode eigene Anwendungen zur Fernsteuerung entwickeln kann.

Socket

Diese Anwendung nutzt die Kommunikation über sogenannte Sockets. Diese sind bei Programmen (neudeutsch „Apps“) die auf Internetinhalte zurückgreifen, vor allem auch bei Smartphones sehr weit verbreitet. Bei Socket unterscheidet man zwischen Client und Server. Ein Client baut eine Verbindung zu einem Server auf, welcher dann Daten an den Client schickt oder umgekehrt. Welche Daten übertragen werden bleibt dem Nutzer freigestellt. Die Kommunikation verläuft also nicht einseitig. Die Übertragung verläuft über das Internetprotokoll. Um eine Verbindung herzustellen benötigt man lediglich die IP-Adresse des Servers und einen freien Port im Netzwerk, auf dem man im Daten austauschen möchte. Jeder Webserver, der eine Webseite zur Verfügung stellt ist nichts anderes als ein Socket, der auf Port 80 agiert, nur das hier die Datenübertragung nach dem HTTP Protokoll abläuft.

Abb. 1: Java Socket App

C und Java im Zusammenspiel

Um den Socket Server auf dem GNUBLIN Board zu realisieren ist C die Programmiersprache der Wahl. In C lässt sich ohne weiteres ein Socket Server erstellen und für Anfänger ist der Quelltext nachvollziehbar, zudem werden kaum Ressourcen benötigt. Für den Client wurde Java auserkoren. Java deshalb, weil die Stärken von Java ebenfalls im Netzwerk liegen und sich schnell eine grafische Oberfläche erstellen lässt, die Betriebssystemunabhängig funktioniert. Lediglich die JavaRuntimeEnvironment wird benötigt. Java auch deshalb, weil man für eine vollständige Anwendung auf nicht viel mehr als hundert Zeilen Quelltext kommt, selbiges gilt für das C-Programm. So bleibt zunächst alles überschaubar.

Der Server und die Kommunikation

Für der Programmierung eines Sockets in C findet man auf der Seite von Martin Pollakowski[1] viele Beispielanwendungen und Erklärungen sowohl für Server als auch Clients. Hieran lehnt sich auch der Quelltext für den Socket an. Auf eine umfangreiche Fehlerbehandlung wurde verzichtet. Im Programm werden zunächst die entsprechenden Bibliotheken für eine Socket Verbindung und den Netzwerkzugriff hinzugefügt. Danach werden die GPIO Ports initialisiert. Dies erfolgt über das Erzeugen von Shell Befehlen mithilfe des Kommandos

```
system();
```

Welche Befehle ausgeführt werden müssen ist auf der GNUBLIN Homepage erklärt. Eleganter ließe sich dies lösen, indem man die GPIOS wie eine Datei behandelt, dies hätte aber wesentlich mehr Quelltext erfordert und der Übersichtlichkeit geschadet. Nun werden noch die Variablen für den Socket Server angelegt. Der Port wird in der Variable port abgelegt, hier ist es Port 5000.

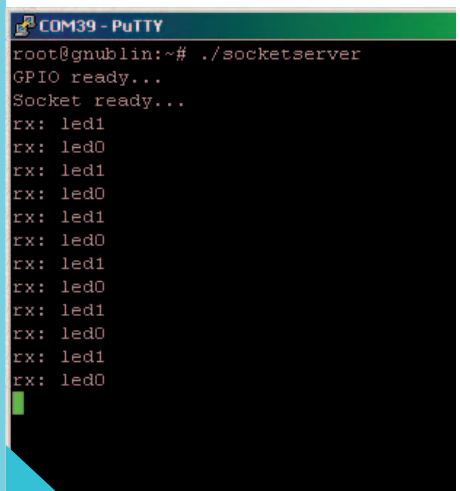


Abb. 2: C-Socket Server

Mit

```
htonl(INADDR _ ANY);
```

wird festgelegt, dass sich ein Client mit beliebiger IP mit dem Server verbinden kann.

Möchte man dies unterbinden, ist der Ausdruck durch

```
inet _ addr(„192.x.x.x“);
```

zu ersetzen. Dort kann dann eine beliebige IP eingetragen werden. Nun wird der Socket Server noch aktiviert. Dies wird über eine kurze Meldung in der Konsole angezeigt. Die Systematik und Funktionsweise der einzelnen Socketfunktionen wird unter [2] sehr gut erklärt.

Nun geht das Programm in eine Endlosschleife, welche Verbindungen von Clients annimmt und wartet, ob eine Zeichenkette gesendet wird. Wird eine Zeichenkette empfangen, wird diese auch noch auf der Konsole ausgegeben.

Im Folgenden wird dann mit der StringCompare Funktion überprüft, ob ihr Inhalt entweder „led0“ oder „led1“ ist. Dies sind frei gewählte Kürzel anhand derer festgelegt wird, ob die LED ein oder ausgeschaltet wird. Wird eine passende Zeichenkette empfangen, so wird dann der Konsolenbefehl zum Ein- oder Ausschalten der LED verschickt. Dort kann natürlich auch ein beliebiger anderer Code ausgeführt werden. Verwendet man mehrere Komponenten ist statt Kürzeln ein System - zum Beispiel angelehnt an das XML-Protokoll - empfehlenswert (Beispiel: <output=“led“ value=“1“>) und ein entsprechender Parser sollte verwendet werden. Sonst dürfte das Programm aufgrund vieler Vergleichsbedingungen träge werden.

Danach wird die Verbindung wieder geschlossen. Das Öffnen und Schließen der Verbindung ist üblich, aber auch notwendig, da der Server, sobald der Client weg fällt, ständig in der Endlosschleife leere Zeichen empfangen würde und somit letztendlich funktionsunfähig ist. Der Server selbst bleibt aber existent. Dies kann bei häufigem Aufruf des Programms ohne Neustart zu Problemen führen.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main(void){
   system(„echo 3 > /sys/class/gpio/ex
   port“);
   system(„echo \“high\“ > /sys/class/gpio/
   gpio3/direction“);
   system(„echo 0 > /sys/class/gpio/gpio3/
   value“);
   printf(„GPIO ready...\n“);
   int server _ socket, new _ socket, length,
   number;
   struct sockaddr _ in serverinfo,
   clientinfo;
   char rx _ chars[1000];
   unsigned short int port = 5000;
   server _ socket =
   socket(AF _ INET, SOCK _ STREAM, 0);
   serverinfo.sin _ family = AF _ INET;
   serverinfo.sin _ addr.s _ addr =
   htonl(INADDR _ ANY);
   serverinfo.sin _ port = htons(port);
   length = sizeof(serverinfo);
   bind(server _ socket,
   (struct sockaddr*)&serverinfo, length);
   listen(server _ socket, 3);
   printf(„Socket ready...\n“);
while(1){
   new _ socket = accept(server _ socket,
      (struct sockaddr *)&clientinfo,
      &length);
   number = read(new _ socket,rx _ chars,
      sizeof(rx _ chars));
   rx _ chars[number]=0;
   printf(„rx: %s \n“,rx _ chars);
   if(strcmp(rx _ chars, „led1“))
      system(„echo 0 > /sys/class/gpio/gpio3/
            value“);
   if(strcmp(rx _ chars, „led0“))
      system(„echo 1 > /sys/class/gpio/gpio3/
            value“);
   close(new _ socket);
   }
   //never reached
   //system(„echo 3 > /sys/class/gpio/unex
   port“);
   return 0;
}
```

Der Client

Der Java Client enthält im Gegensatz zum C Programm einige Methoden. Zum einen die Methode

```
Bool socket_connect(String ip);
```

welche die IP Adresse des Servers übergibt und welche versucht, eine Verbindung aufzubauen und im Idealfall True zurückgibt. Dies geschieht indem ein neuer Socket Client aus der Java eigenen Socket-Klasse erzeugt wird und dabei IP und Port übergeben werden. Hierbei wird weder Threading noch ein Timeout angewendet. Das Programm kann also durchaus beim Versuch sich zu verbinden hängen bleiben. Allerdings wird in der Regel - sofern kein Socket gefunden wurde oder die Verbindung abbricht - eine sogenenannte Exception (Ausnahme) ausgelöst, welche dann eine entsprechende Warnung über eine Nachrichtenbox ausgibt. Gleichzeitig wird ein Schreibpuffer (PrinterWriter) angelegt, welchem Strings übergeben werden können, die beim ausführen des Kommandos

```
flush();
```

an den Socket Server geschickt werden.

Zum anderen gibt es eine Methode um die Verbindung zum Server zu trennen. Eine ausführliche Anleitung zu Netzwerkverbindungen und Netzwerkprogrammierung unter Java gibt es bei Wikibooks[3]. Der Rest des Java Programms erzeugt ein Fenster (Panel genannt), auf dem die drei Buttons angelegt werden. Gleichzeitig wird ein sogenannter EventListener angelegt, welcher auslöst, sobald ein Button betätigt wird. Dort wird dann, je nach dem welcher Button gedrückt wurde, entweder über eine einfache Eingabemaske die IP abgefragt und als String abgelegt oder der Befehl zum ein oder ausschalten der LED verschickt (siehe oben), oder aber das Programm beendet. Der Programmablauf ist ebenfalls denkbar einfach. Die grafische Oberfläche wird erzeugt und zunächst auf die Eingabe der IP Adresse gewartet. Wenn man dann den Umschaltbutton betätigt, versucht das Programm eine Socketverbindung herzustellen, das entsprechende Kommando zu verschicken und die Verbindung wieder zu schließen. Schlägt das Herstellen der Verbindung fehl, wird zunächst der Umschaltbutton deaktiviert und wieder gewartet bis eine neue (korrigierte) IP-Adresse eingegeben wurde.

How-to

Vor der Benutzung sollte man sich vergewissern, dass eine Netzwerkverbindung zwischen dem späteren Client und dem Server existiert (gegebenenfalls ping durchführen).

Die Benutzung des Programms ist hingegen denkbar einfach:

- Man startet die Anwendung auf dem GNUBLIN Board.
- Danach ist das Java Programm zu starten.
- Anschließend gibt man die IP (oder den Hostnamen an) unter dem das GNUBLIN Board im Netzwerk aktiv ist.
- Danach ist es möglich die LED ein und aus zu schalten.

Falls dabei die Herstellung der Verbindung fehl schlägt gibt es eine Fehlermeldung. Beendet man den Server oder das Programm und die LED ist eingeschaltet, bleibt diese an. Die Java Anwendung warnt beim Verlassen, wenn die LED noch aktiv ist. Beim Server wird derzeit nichts abgefangen. Wenn man sein GNUBLIN Board per Kabel oder WiFi an einen Router angeschlossen hat und das Programm auch außerhalb des lokalen Netzwerks nutzen möchte, so kann man bei den meisten Routern heutzutage einen sogenannten PortForward anlegen. Dabei gibt man an, dass ein Zugriff auf einen frei wählbaren Port und die IP des Internetanschlusses auf ein anderes Gerät umgeleitet wird.

Java Listing

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.event.*;
public class Socket_host_application extends
JFrame implements ActionListener {
   private static final long serialVersionUID = 1L;
   private final JPanel mainPanel;
   private final JButton set_ip, toggle, exit;
   private static PrintWriter printWriter = null;
   private static Socket s;
   private String ip;
   private int led_state = 0;
   private static Boolean socket_connect(String ip) {
     try {
            Socket_host_application.s = new Socket(ip, 5000);
     } catch (UnknownHostException e) {
            JOptionPane.showMessageDialog(null, „Host not found“);
            return false;
     } catch (IOException e) {
            JOptionPane.showMessageDialog(null, „Server not found“);
            return false;
     }
     try {
            printWriter =
            new PrintWriter(new OutputStreamWriter(
                  s.getOutputStream()));
     } catch (IOException e) {
            e.printStackTrace();
     }
     return true;
   }
   private void socket_disconnect() {
     try {
            s.close();
     } catch (IOException e) {
            e.printStackTrace();
```

```
    }
    }
  private Socket _ host _ application(final String str1) {
    super(str1);
    mainPanel = new JPanel();
    set _ ip = new JButton(„Enter IP“);
    toggle = new JButton(„Turn LED on „);
    exit = new JButton(„Exit“);
    this.setDefaultCloseOperation(JFrame.EXIT _ ON _ CLOSE);
    this.setResizable(false);
    this.setSize(300, 80);
    this.setVisible(true);
    this.setContentPane(mainPanel);
    this.setLocationRelativeTo(getParent());
    mainPanel.add(set _ ip);
    mainPanel.add(toggle);
    mainPanel.add(exit);
    set _ ip.addActionListener(this);
    toggle.setEnabled(false);
    toggle.addActionListener(this);
    exit.addActionListener(this);
  }
  public void actionPerformed(final ActionEvent x) {
    final Object source = x.getSource();
    int temp = 0;
    if (source == set _ ip) {
          ip = JOptionPane.showInputDialog(„Enter Server IP:“);
          if (ip != null)
                toggle.setEnabled(true);
    }
    if (source == toggle) {
          if (socket _ connect(ip)) {
                if (led _ state == 0) {
                      led _ state = 1;
                      toggle.setText(„Turn LED off“);
                } else if (led _ state == 1) {
                      led _ state = 0;
                      toggle.setText(„Turn LED on „);
                }
                printWriter.print(„led“ +
                String.valueOf(led _ state));
                printWriter.flush();
                // only needed if you want to be serious
                socket _ disconnect();
          } else
                toggle.setEnabled(false);
    }
    if (source == exit) {
          if (led _ state == 1) temp = JOptionPane.
showOptionDialog(null,“You are ought to turn off LED before
Exit.\n Continue anyway?“,“Warning“,
JOptionPane.YES _ NO _ OPTION,JOptionPane.WARNING _ MESSAGE, null,
null, null);
          if (temp == 0)System.exit(0);
    }
  }
  public static void main(String[] args) {
    @SuppressWarnings(„unused“)
    Socket _ host _ application mainPanel =
    new Socket _ host _ application(„GNUBLIN Socket Host App“);
  }
}
```

Anwendungsbeispiele

Die Möglichkeiten einen solchen Socket zu verwenden sind fast grenzenlos. Das Schalten von Ausgängen am GNUBLIN ist zwar primitiv, bietet aber etwa die Möglichkeit mithilfe von Funksteckdosen oder ähnlichem sein Heim zu automatisieren. Aber auch Werte können vom Server übertragen und vom Client eingelesen werden, wie etwa die eines Temperatursensors oder Spannungen vom ADC-Converter. Wer Größeres automatisieren möchte und mehr als drei Buttons administrieren möchte kann auf Basis des vorliegenden Codes mithilfe der grafischen Editoren von NetBeans oder Eclipse in VisualStudio Manier eine grafische Oberfläche zusammenklicken. Eine Portierung in eine andere Hochsprache oder auf ein anderes System, wie etwa ein Handy mit Java Unterstützung ist ebenfalls denkbar.

Erweiterungen im Sicherheitsaspekt

Aufpassen sollte man bei solch einer Socket Verbindung allerdings:

- Die Daten werden unverschlüsselt verschickt und die Verbindung kann zunächst jeder beliebige Client zum Server aufbauen. Abhilfe lässt sich aber schaffen, indem man nur noch eine Client IP Adresse am Server zulässt.
- Weiterhin könnte der Client auch ein Passwort verschlüsseln (als md5-Hash) oder eine andere wechselnde Information an den Server schicken.
- Es ist auch möglich eine OpenSSL basierte verschlüsselte Verbindung zu erstellen. Entsprechende Bibliotheken existieren für Java ebenso wie für C.

Links / Download

[1] http://www.fh-gelsenkirchen.de/fb01/homepages/pollakowski/socket/index.html

[2] http://www.willemer.de/informatik/unix/unprsock.htm

[3] http://de.wikibooks.org/wiki/Java_Standard:_Socket_ServerSocket_%28java.net%29_UDP_und_TCP_IP

Reif für den Hive?

Ingo Kripahle <drohne235@gmail.com>

Einleitung

Wer kennt das nicht: Man findet einen interessanten Chip und ist spontan begeistert, hat aber keine Idee, was man damit sinnvolles anstellen könnte? So bei mir geschehen mit dem Mikrocontroller P8X32 von Parallax, welcher mit seinen acht RISC-Subsystemen, keinen Interrupts, keinen Devices und integrierter Hochsprachunterstützung ein ausgesprochen exotischer Aussenseiter ist. Im folgenden nun möchte ich ein kleines spaßiges Experiment mit diesem interessanten Chip und seine Geschichte vorstellen - den Hive Retro Style Computer: Drei Mikrocontroller mit insgesamt 24 RISC-Cores, VGA und Video-Ausgang, Stereosound, 1 MB RAM, SD-Card-Laufwerk, LAN-Interface, PS/2-Maus und Tastatur, freie Software, freie Hardware, einfach und unkompliziert in verschiedenen Sprachen programmierbar - und das alles auf einer kleinen Leiterplatte im Europaformat, ein minimalistischer Computerbausatz ohne SMD-Bauelemente, von jedem Bastler in wenigen Stunden aufzubauen und zu programmieren.

Am Anfang eine Frage

November 2008, „Strafarbeit“ im Bastelkeller: Aufräumen! :) Jeder kennt das, und die meisten wissen was dann passiert: Man findet die alten verstaubten Kisten und anstatt sie umzustellen, ergötzt man sich an den Schätzen darin. Alte Platinen, Überreste längst vergangener Projekte, Eigenbaujoysticks, vergrabene und lange vergessene Wunschträume eines Bastlers... Und dann findet man diesen „verwundeten“ alten Atari800XL, den man schon immer mal reparieren wollte, setzt sich hin (das ist dann der Punkt an welchem man es erfolgreich geschafft hat die „Strafarbeit“ zu verdrängen), wirft ganz nebenbei den Lötkolben an und grübelt was dem Schmuckstück fehlt. RAM defekt, also ein altes SIM-Modul mangels passender Bauteile eingebaut und wahrhaftig - der kleine Kerl beginnt wieder zu leben! Ein paar alte Kassetten ein paar Zeilen Basic und schon musiziert das Gerät lustig vor sich hin...

Und nach all diesen Jahren, die man nun mit Gigahertz- und Terabyte-Monstern verbracht hat, ist man wieder von neuem wie vor vielen Jahren fasziniert, erstaunt über die Leichtigkeit und Einfachheit, mit welcher sich diese Retros damals präsentierten. Es wurde mir dabei sehr bewusst, dass es genau diese Direktheit und Verstehbarkeit der alten Systeme war, der ich persönlich eine ganze Menge Urvertrauen und grundlegendes Wissen über diese Technik verdanke. Und dieser Lernprozess hat sich damals sehr gut gepaart mit Spaß und Neugierde. Mit den Jahren ist irgendwo diese Faszination mit der PC-Technik in dem undurchdringlichen Dickicht von Megas und Gigas in den meisten Aspekten verloren gegangen, oder wie jemand es treffend im Hive-Forum geschrieben hat: „PCs sind so spannend wie ein Schraubenzieher.“ Ich finde das bringt es gut auf den Punkt und bei so viel kommerziellem (Anti-)Charme verschwindet auch viel von der Kreativität und alle damit zusammenhängenden Programmier- und Bastelprojekte. Nun, ich schweife wohl ein wenig ab. Da stand ich nun also mit diesem phantastischen 8-Bit-Atari und es keimte eine faszinierende Frage:

Was wäre wenn...

Was wäre wenn man mal wieder einen kompletten Computer selbst entwirft und aufbaut? So wie ich das zu damaligen Zeiten mangels fertiger Geräte gemacht hatte. Natürlich alles selbst entwickelt, selbst aufgebaut, einfach aus Spaß an der Freude! Eine handvoll Chips und ein Lötkolben sollte reichen, keine fertige Hardware oder Software!

Was wäre wohl heute möglich? Könnte man mit einfachen Mitteln ein Gerät mit der Leistung zwischen einem Atari 800 und einem Atari ST, zwischen einem C64 und einem Amiga entwickeln? Würde es ebenso viel Spaß bringen, sich mit diesem Gerät zu beschäftigen, würde es die Neugier neu beflügeln? Schließlich waren jetzt seit der goldenen Zeit der Homecomputer viele Jahre vergangen. Aber es sollte einfach und unkompliziert, für jeden Interessierten nachbaubar und keine riesige Materialschlacht sein. Und es sollte auch kein einfacher Retro-Clone, sondern sollte „mehr“ sein, sollte an den guten Eigenschaften dieser alten Homecomputer anknüpfen und diese Bastelkultur weiterentwickeln. Kurz gesagt, sollte es ein System werden, welches sich so anfühlt wie die alten Retros, aber dabei sollte es moderner sein und darüber hinaus gehen.

Nach einigen schlaflosen Nächten hatte ich dann ein Konzept im Kopf, kein fertiger Schaltplan, sondern nur eine Art Blockschaltbild mit einigen nötigen Detailfragmenten. Also auf zur Tat: Nachdem die nötigen Bauteillieferungen mich erreichten, brauchte ich eine Woche um den ersten Prototypen auf

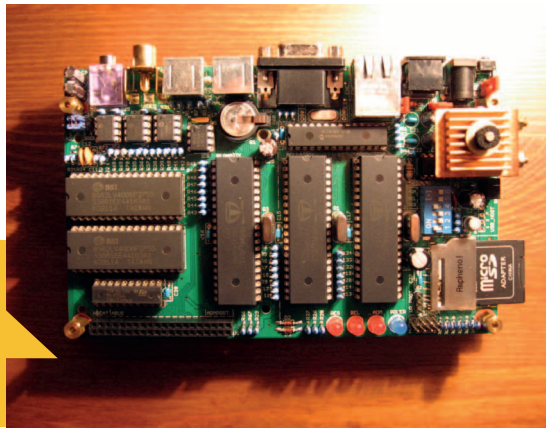


Abb. 1: Das aktuelle R14-Board in der Übersicht

einer Lochrasterplatine zum Leben zu erwecken. Nach einigen weiteren Wochen gab es eine minimalistische Kommandozeile [3], erste Tools die vom Laufwerk gestartet werden konnten und gegen Weihnachten ein erstes zaghaftes Programm: Der StarTracker [4] - eine Modulplayer in alter Tradition mit einer LCARS-Benutzeroberfläche wie auf der Enterprise.

Zu diesem Zeitpunkt hatte ich eigentlich nicht vor das System publik zu machen - ich hatte riesigen Spaß mit der Sache und das genügte mir völlig. Es war wie in alten Zeiten: Die Hardware war exotisch und neu, es gab und gibt so viel dabei zu entdecken, dass ich mich wirklich in die Zeit meiner ersten Computerexperimente zurück versetzt fühlte.

Und so nahm die Geschichte ihren Lauf: Bei einem Retrotreffen fand digger gefallen am Projekt, organisierte eine größere Sammelbestellung, zusätzlich eine ordentliche Infrastruktur im Web, welche wir mit Informationen rund um das Projekt füllten, inklusive einem Forum. Nebenher entwickelte ich die erste richtige Platine für den Hive. Und nun haben ihn schon viele Mitstreiter auf dem Tisch stehen: Den weltweit einzigen Multicore-Homecomputer-Bausatz, den wirklich jeder Interessierte selbst zusammen löten und programmieren kann. ;)

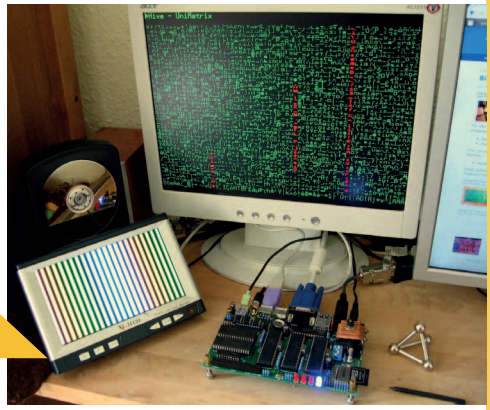


Abb. 2: Das R14-Board in Aktion mit einem Videotreiber, der gleichzeitig VGA und ein PAL-Videosignal erzeugt

Der Parallax Propeller

Herzstück des Hive sind drei Mikrocontroller P8X32D40 der Firma Parallax mit dem seltsamen Beinamen „Propeller“. So wie der Hive scheint auch dieser Propellerchip ein Experiment zu sein, denn bei diesem Mikrocontroller ist vieles anders als im Mainstream.

Kernstück des Propellers sind acht gleichartige RISC-Subsysteme, die sogenannten COGs. Jede COG besteht aus einer RISC-CPU mit 32 Bit Architektur, zwei 32 Bit Timern, einem Video Generator und einem lokalen RAM. Zusätzlich haben alle COGs einen gemeinsamen Zugriff auf die 32 I/O-Ports. Der lokale Speicher der COG ist privat, keine andere COG kann also auf diesen zugreifen, und ist in Form einer 512 x 32 Bit Registerbank organisiert. In diesem Speicher befinden sich sowohl die Steuerregister für die lokale Hardware als auch das Maschinenprogramm für die CPU in dieser COG.

Alle COGs gemeinsam haben Zugriff auf einen 64 KB Speicherbereich, ein Systemtimer und die 32 I/O-Ports im Propellerchip. Der Speicher ist unterteilt in 32 KB RAM und 32 KB ROM. Im ROM befindet sich ein Zeichensatz, drei Tabellen (Log, Anti-Log, Sin), sowie der Spin-Interpreter und der Code für die Boot Routine.

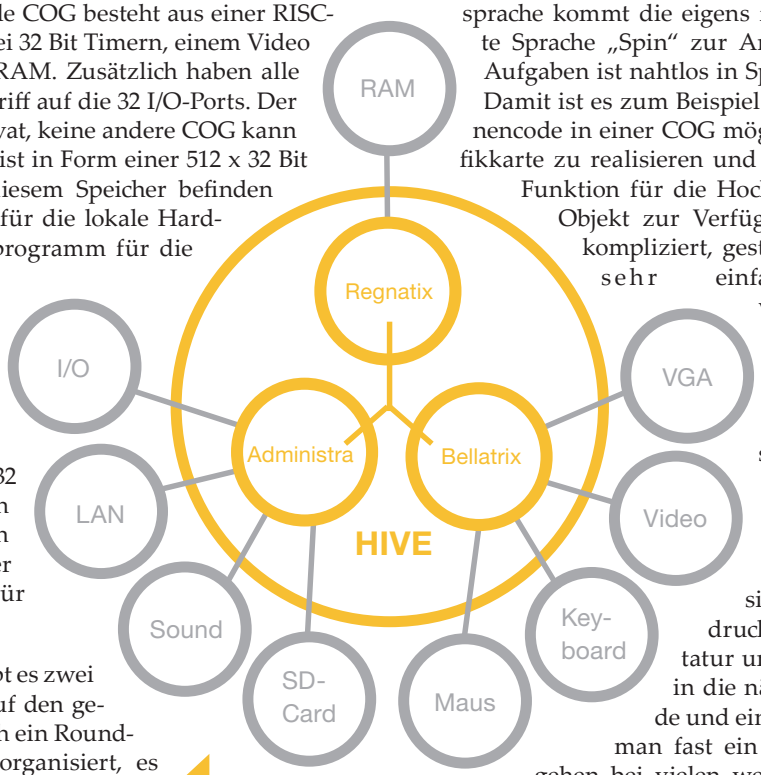


Abb. 3: Übersicht der Funktionskomplexe im Hive

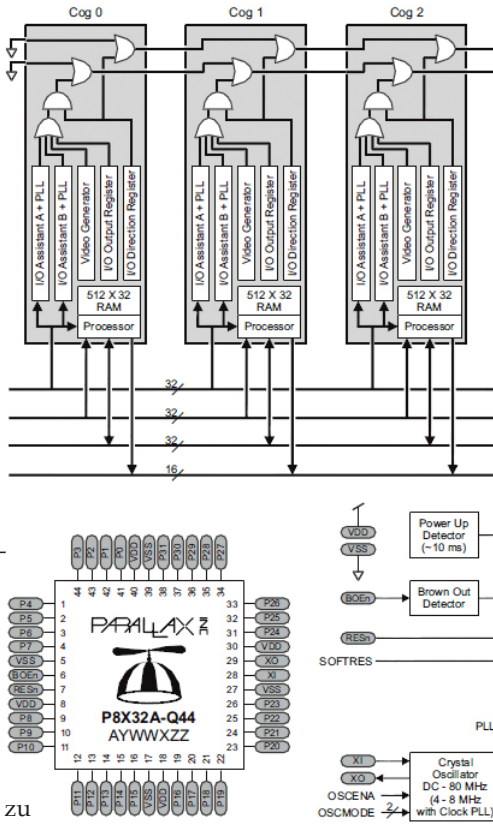
Bei den geteilten Ressourcen gibt es zwei Zugriffsmodelle: Der Zugriff auf den gemeinsamen Speicher wird durch ein Round-Rubin-Verfahren (Hub-RAM) organisiert, es kann also immer nur eine COG gleichzeitig zugreifen, was zu einer variablen Zugriffszeit von bis zu 15 Zyklen führt. Anders der Systemtimer und die I/O-Ports: Hier können alle COGs gleichzeitig und unabhängig voneinander zugreifen.

Der Propellerchip besitzt 32 allgemeine I/O-Ports. Von diesen sind 28 Ports universell nutzbar, die obersten vier Ports (P28..31) realisieren eine serielle Hostschnittstelle zur Programmierung und Kommunikation und eine I2C Schnittstelle, an welchem mindestens ein externer EEPROM angeschlossen wird, um das Programm dauerhaft zu speichern. Auf alle Ports hat jede COG über ein I/O Direction Register und ein Output Register Zugriff. Die Ports aller COGs sind mit einem logischen Oder verknüpft.

Die Grundidee der Propeller Architektur besteht darin, spezifische Hardware durch Software zu ersetzen. Als Programmiersprache kommt die eigens für den Propeller entwickelte Sprache „Spin“ zur Anwendung. Für zeitkritische Aufgaben ist nahtlos in Spin ein Assembler integriert. Damit ist es zum Beispiel mit wenigen Zeilen Maschinencode in einer COG möglich, die Funktion eine Grafikkarte zu realisieren und begleitend die nötigen Spin Funktion für die Hochsprachanbindung in einem Objekt zur Verfügung zu stellen. Das klingt kompliziert, gestaltet sich in der Praxis aber sehr einfach, auch dank vieler fertig verfügbarer Objekte: Man braucht eine VGA-Karte? Gut, dann füttert man eine oder mehrere COGs mit dem entsprechenden Code, schließt acht Widerstände und eine VGA-Buchse irgendwo an den Ports an und fertig ist die interne Grafikkarte und noch weitere sieben COGs warten unbeeindruckt auf eine Aufgabe. PS/2-Tastatur und Maus? Ein bisschen Code in die nächste COG, vier Widerstände und eine PS/2-Buchse und schon hat man fast ein Terminal. Analog das Vorgehen bei vielen weiteren Problemen: SD-Card, Soundkarte, serielle Schnittstelle - alles Softwaremodule und ein wenig minimalistische Hardware. Das ist fast so einfach wie Mikrocontroller-Lego.

Dabei bietet dieses Konzept auch eine sehr große Flexibilität, denn man kann alles in Spin ändern und anpassen. Selbst außergewöhnlichste Konfigurationen sind so realisierbar. Ein Projekt benötigt etwa zwei Grafikkarten oder sechzehn serielle Schnittstellen? Kein Problem, passenden Code laden, mit einigen Zeilen Spin auf der Metaebene verbinden und fertig. Es werden nur zwei Cores benötigt? Kein Problem, dann bleiben die anderen sechs Cores inaktiv und verbrauchen keine Energie. So

sind durch die Aktivitäten der Community auch exotische Objekte verfügbar. Einige Beispiele: So wird mit „ZiCog“ eine COG zu einer Z80 & 8080 CPU auf welcher auch schon ein CP/M lauffähig ist. Mit der „MoCog“ wird eine COG zu einer Motorola 6809 CPU und mit der SIDCog wird aus einer COG gar ein SID-Chip mit Steuerung analog dem C64 über identische im Hub-RAM eingeblendete Register. Wer sich einige Möglichkeiten in Aktion anschauen möchte, kann das in der Folge 04/2007 von Computer-TV [5] tun, welche den Chip sowie das Demo- und Hydraboard von Parallax vorstellt.



Spin, Assembler und ein Leben ohne Interrupts

Spin verkörpert dabei ein sehr einfaches objektorientiertes Modell und ist nach meinen Erfahrungen sehr gut geeignet, um mit vielen Cores zu jonglieren, ist ausreichend schnell für die meisten Anwendungen und sehr kompakt. Dass wird erreicht, indem der Compiler einen Bytecode erzeugt, welcher dann im Chip von einem Interpreter verarbeitet wird. Das Prinzip ist nicht neu und prominente Beispiele sind Java oder UCSD-Pascal mit p-Code. Der benötigte Spin-Interpreter passt dabei komplett in eine COG und wird beim Systemstart automatisch oder bei Bedarf mit einem Kommando aus dem internen ROM in die gewünschte COG geladen und gestartet. Klingt etwas verwirrend, ist in der Praxis aber sehr einfach: Programm schreiben, per Funktionstaste F10 oder F11 in den Propeller übertragen und schon läuft es. Nach einem Reset lädt der Chip immer einen Interpreter in die erste COG und startet ein vorhandenes Spinprogramm, welches er entweder über die Hostschnittstelle vom Compiler oder aus dem EEPROM empfängt. Läuft auf der ersten COG unser Programm, so können durch cognew-Kommandos beliebige andere Funktionen in weiteren COGs gestartet werden und laufen ab diesem Zeitpunkt parallel auf einem eigenen Prozessor bzw. Interpreter. Um die parallelen Prozesse zu synchronisieren sind acht Semaphoren (Locks) in Hardware verfügbar.

Die Geschwindigkeit: Der Propellerchip läuft normal mit einem internen Takt von 80 MHz. Fast alle RISC-Befehle werden in 4 Takten abgearbeitet, also mit einem Befehlszyklus von 20 MIPS pro Core, woraus sich mit acht Cores die offizielle Angabe von 160 MIPS ergibt. Das ist schnell genug um zum Beispiel in Maschinencode ein VGA-Bild mit 1024x768 in Echtzeit zu erzeugen oder Daten seriell mit 14,4 Megabaud zu versenden. Spin ist um den Faktor 50..100 langsamer, aber in den meisten Anwendungen immer noch ausreichend schnell für den überwiegenden Teil der Anwendung. Dabei ist Spin durch den Bytecode auch noch sehr kompakt, was bei dem kleinen Speicher auch nötig ist. Eine in Spin programmiertes Objekt für eine serielle Schnittstelle belegt gerade mal 49 Longs und erreicht immerhin noch eine Geschwindigkeit von 19.200 Baud.

Im Übrigen werden sehr viele fertige Codeobjekte bei dem Compiler von Parallax schon mitgeliefert oder sind als Beitrag der Community im Object Exchange (OBEX) auf der Webseite des Herstellers zu finden. Alle Objekte dort stehen unter der MIT-Lizenz.

Eine Bemerkung zu den Interrupts: Das völlige Fehlen von Interrupts im Propeller führt immer wieder zu ein wenig Verwirrung. Wie kann man ohne Interrupts leben? Meine provokante These zum Thema: Interrupts sind ein Relikt der Computersteinzeit! ;) Sie sind ein gewolltes, aber nicht wirklich gekonntes Multitasking. Wenn man die Entwicklung von Prozessoren und Mikrocontrollern betrachte und gedanklich die technischen Möglichkeiten und Notwendigkeiten nachvollzieht, so kommt man schnell zu den Wurzeln der Interrupts: Komplexität und Chipfläche waren in der Anfangszeit eine teure und knappe Ressource, an einen solchen Luxus von acht Prozessoren pro Chip, wie wir ihn zur Verfügung haben, war nicht zu denken. Und dennoch mussten diese Computer auf zeitkritische Signale reagieren. Sollten zum Beispiel Daten über eine Schnittstelle empfangen werden, so musste der Prozessor innerhalb einer definierten Zeitspanne diese Daten einlesen und in entsprechenden Puffern speichern. Aber was macht man, wenn nur ein Prozessor zur Verfügung steht, der auch nur ein Programm abarbeiten kann? Richtig: ein Hardwaresignal, welches mit dem entsprechenden Ereignis gekoppelt ist, unterbricht die aktuelle Programmausführung, sichert den relevanten Prozessorstatus und startet eine “Unterbrechungsbehandlungsroutine” – unser Interrupt und seine Interruptroutine. Dieser Mechanismus ist also einfach aus der Not geboren, welche eine Architektur mit sich bringt, die nur einen Prozessor verfügbar hat.

Anders der Propeller: Warum soll ich den Aufwand betreiben und

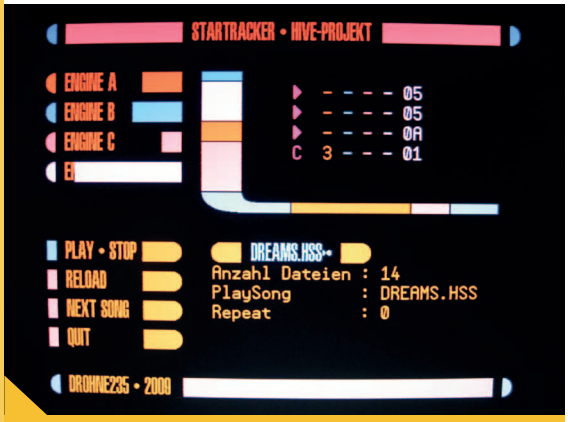


Abb. 4: Screenshot des Modulplayers StarTracker - das erste richtige Programm welches auf dem Hive-Prototypen lief

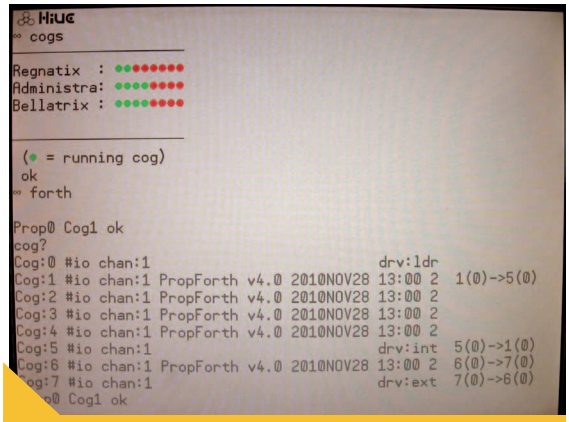
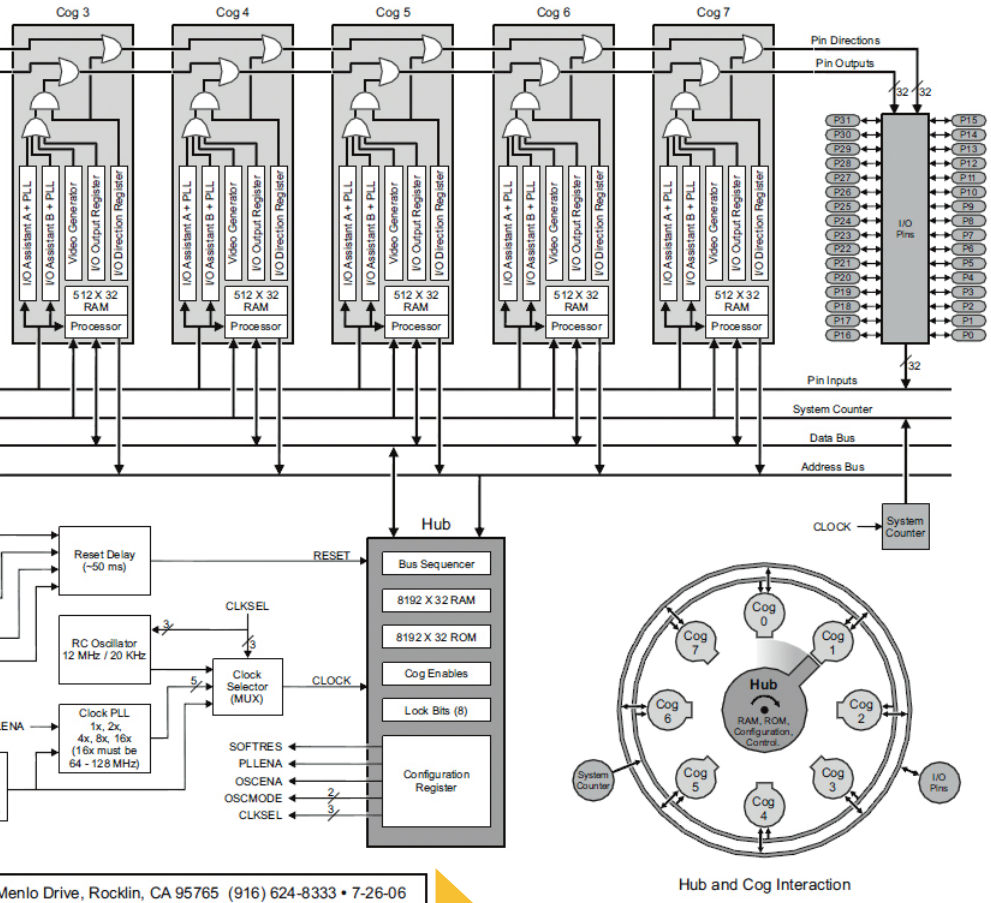


Abb. 5: Screenshot TriOS, Wechsel von der Kommandozeile „Regime“ in das Forth-System mit anzeige der COG-Belegung



Hub and Cog Interaction

Menlo Drive, Rocklin, CA 95765 (916) 624-8333 • 7-26-06

Abb. 6: Blockschaltbild Parallax Propeller (s. Hersteller)

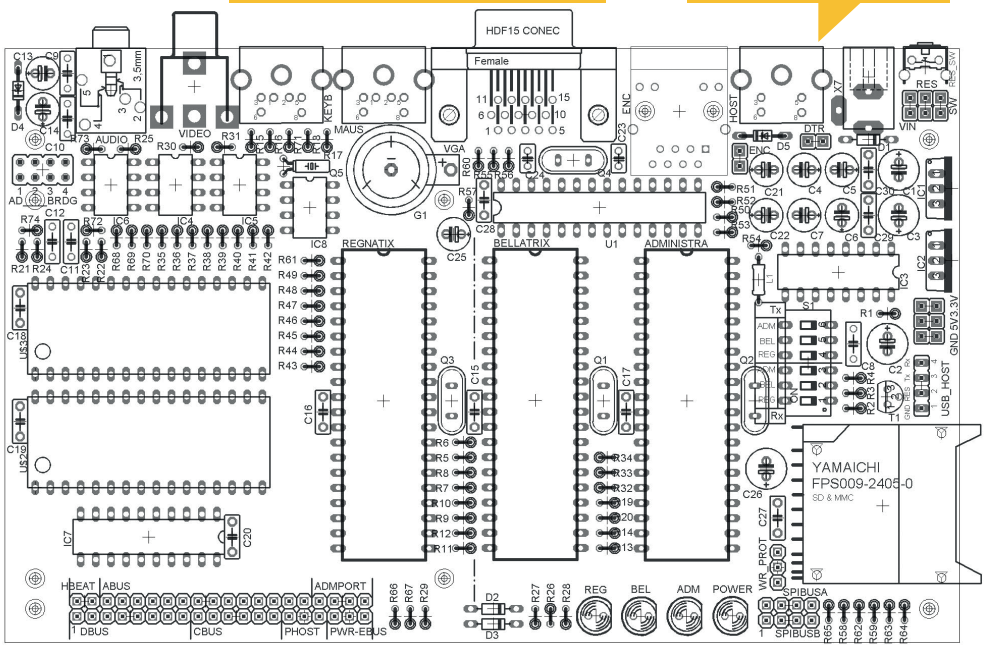


Abb. 7: Bestückungsplan

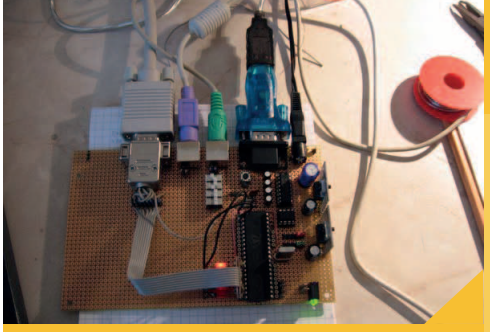


Abb. 8: Der erste Propeller (Bellatrix) auf der Lochrasterplatine mit PS/2- und VGA-Anschluss

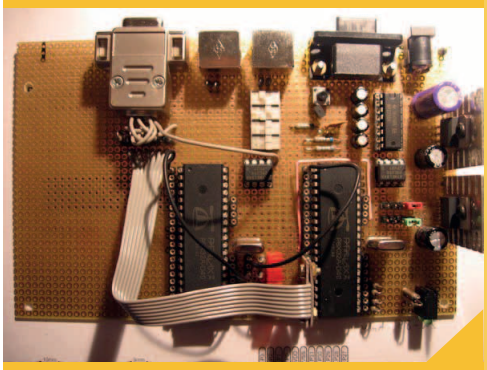


Abb. 9: Der zweite Propeller (Administra) auf der Lochrasterplatine

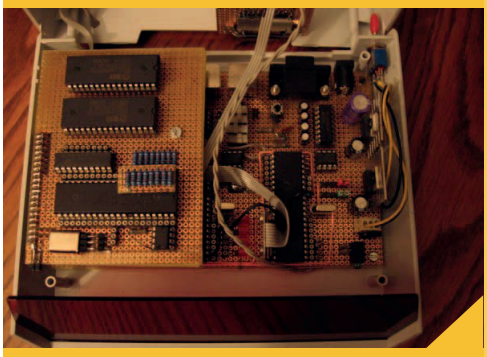


Abb. 10: Der fertige Hive-Prototyp. Regnatix mit externen Speicherbänken ist mit einer aufgesteckt. Oben am Gehäuse das SD-Laufwerk

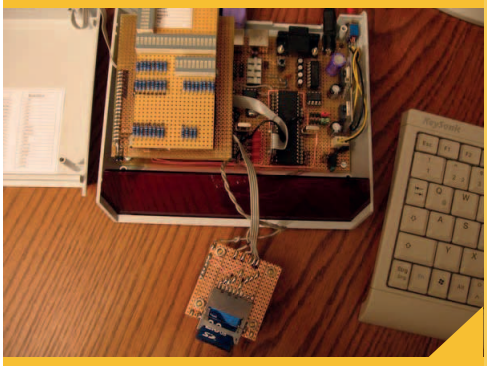


Abb. 11: Aufgesteckter LED-Busmonitor

einen einzelnen Prozessor in seiner Arbeit unterbrechen – mit dem gesamten damit zusammenhängenden Aufwand an Speicherplatz und Rechenzeit – wenn ich auch einen eigenen Prozessor verwenden kann, welcher sich ganz ausschließlich um diese Aufgabe kümmert? Und gerade durch diese Aufteilung der Arbeit auf wirklich real vorhandene Hardware ergibt sich eine sehr deterministische Reaktionszeit des Systems – ein wichtiger Punkt, wenn es um die Bedienung von zeitkritischer Hardware geht. Um auf entsprechende Signale aus der Hardware ohne Polling zu reagieren, ist eine ganze Gruppe von Maschinencodebefehlen vorhanden, die den Programmfluss unterbrechen, bis die definierte Signalkonfigurationen an den Ports oder Timern auftreten und das Programm mit der Behandlung dieser Situation fortfahren kann.

Leider konnte nicht alles Bildmaterial komplett und in Originalgröße abgedruckt werden, da es den Rahmen dieses Journals sprengen würde :)

Deshalb sei an dieser Stelle verwiesen auf die Internetseite des Projekts: http://www.hive-project.de

Die Hive-Hardware

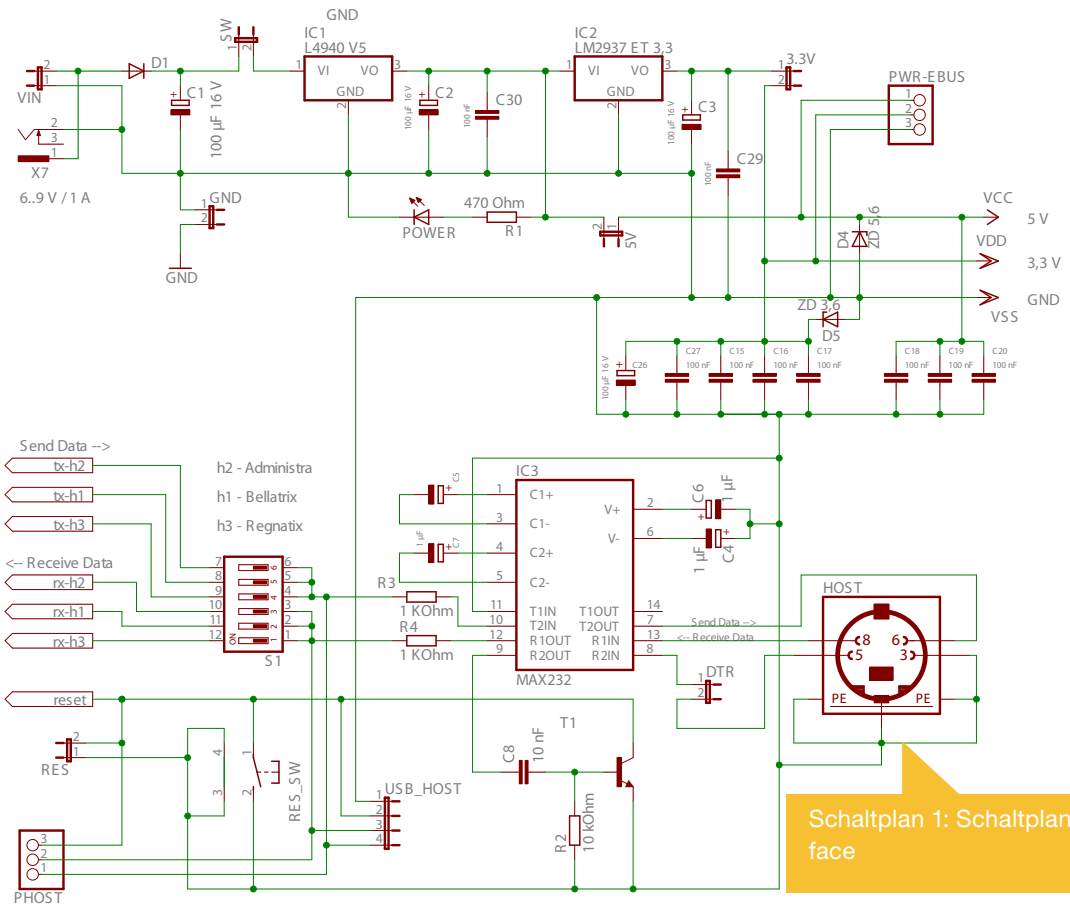
Nun, der Propeller war genau das was ich für meinen minimalistischen Computer brauchte und die Realisierung eines einfachen Computers eine ideale Gelegenheit, mit diesem interessanten Mikrocontroller zu experimentieren. Am Anfang stand die Überlegung zwei Propellerchips nur als intelligente I/O-Schaltkreise zu verwenden und ihnen einen CISC-Prozessor samt Infrastruktur an die Seite zu stellen. Aber diese Variante habe ich verworfen und den CISC einfach durch einen dritten Propellerchip als Master im System ersetzt. Ich persönlich finde, dass die Verwendung von drei gleichartigen Mikrocontrollern einige interessante Vorteile bringt, die ich in der Praxis unbedingt testen wollte.

Ein Vorteil ist die Einfachheit im Verständnis, denn wenn man mit einem Subsystem umgehen kann, versteht man prinzipiell schon viel vom gesamten System. Man muss nicht für jedes Teilsystem einen speziellen „Code“ erlernen, sondern hat eine universelle Basis, die für fast alles im System gültig ist. Dieses Prinzip ist eigentlich dem Propeller selbst geschuldet, denn gegenüber anderen klassisch strukturierten Controllern, die ja auch aus verschiedenen Funktionseinheiten bestehen, gibt es halt im Chip selbst acht völlig gleichartige Subsysteme. Erweitert führte das zur Struktur des Hives mit seinen drei ebenfalls gleichartigen Mikrocontrollern.

Ein anderer Vorteil ist die Einfachheit im Aufbau. Bei einem klassischen Computer, bestehend aus CPU, Speichern, I/O-Schaltkreise (SIO/PIO/CTC) und diverser zusätzlicher Logik (Adressdecoder, DMA, diverse Logik...), müssen erst sehr viele Baugruppen korrekt zusammenspielen, um ein „Erfolgserlebnis“ zu bekommen. Da ist eine relativ hohe Schwelle die man erst überschreiten muss, bis die ersten Zeichen über einen Bildschirm huschen. Der Propeller aber als Mikrocontroller ist ja eine funktionierende Einheit in sich. Schon mit wenigen Bauteilen, also einer kleinen Teilbestückung des Hive-Boards, funktioniert schon etwas und man kann hemmungslos anfangen zu spielen und zu experimentieren. Und natürlich gestaltet sich so auch die Fehlersuche und Inbetriebnahme sehr einfach. Grafik, Sound, Tastatur - alles kann autark und einzeln in Betrieb genommen, programmiert und getestet werden, ohne dass alles erst insgesamt funktional zusammenspielen muss. Da ist Spaß und ein guter Lerneffekt garantiert.

Baut man einen Hive auf und beginnt mit dem Bellatrixchip, so kann schon mit dieser 1/3tel Bestückung experimentiert werden, da Bildausgabe, Tastatur und Maus funktionieren. Wenn man möchte kann man damit sogar schon Boulder Dash spielen - natürlich vorerst ohne Sound, welchen uns erst Administra beschert. Ein anderer und auch entscheidender Grund war die Verfügbarkeit des Propellers in einem bastelfreundlichen DIP40 Gehäuse. Ich hab kein Problem SMD-Bauteile von Hand zu löten, da ich mich lange Jahre beruflich damit beschäftigt habe, aber ein einfaches und ehrliches DIP Gehäuse hat einfach zum minimalistischen Konzept des Gerätes gepasst. Und diese Beschränkung auf einfach lötbare Bauteile hat sich letztlich wirklich bewährt: Mehrere Sammelbestellungen wurden schon organisiert und hunderte Platinen sind im Umlauf. Es haben Interessierte und Neugierige den Hive zusammen gelötet, die bis zu diesem Zeitpunkt noch nicht einmal genau das heiße vom kalten Ende am Lötkolben unterscheiden konnten!

Ich persönlich hatte aber noch einige andere Gründe den Propeller zu verwenden: Ich wollte mich unbedingt mit den Möglichkeiten eines Multicore-Systems beschäftigen. Und sind wir mal ehrlich: Die Vorstellung 24 RISC-Subsysteme unter der Haube zu haben, mit der Einfachheit und dem Charme eines C64, ist schon sehr verlockend! So kann man bei diesem Bastelexperiment wie bei einem klassischen Homecomputer ganz nebenbei untersuchen, wie sich viele parallel laufende unabhängige CPUs anfühlen, wie man sinnvoll Funktionen auf mehrere Prozessoren verteilt und wie man effektiv unabhängige Prozesse synchronisiert.



Schaltplan 1: Schaltplan Stromversorgung und Hostinterface

Erweiterungsbus

Pin	Signal	Signal
1	D0	BEL-HBEAT
2	D1	ADM-HBEAT
3	D2	A7
4	D3	A6
5	D4	A5
6	D5	A4
7	D6	A3
8	D7	A2
9	/WR	A1
10	/PROP1	A0
11	/PROP2	A8
12	BUSCLK	A9
13	/HS	A10
14	REG-HBEAT	AL
15	TX	/RAM1
16	RX	/RAM2
17	/RESET	ADM-P22
18	VCC 5 V	ADM-P21
19	VDD 3,3 V	ADM-P20
20	GND	ADM-P19

SPI Bus

1	ADM-HBEAT	SD-DO
2	VCC	SD-CLK
3	VDD	SD-CMD
4	VSS	SD-D3

SPI Bus

1	VSS
2	RESET
3	TX
4	RX

Adapterkabel Mini-DIN4 nach D-Sub 9pol Seriell

Mini-Din4	D-Sub 9pol
3	5 - GND
5	4 - DTR Data Term Ready
6	2 - RxD Receive Data
8	3 - TxD Transmit Data

Los gehts!

Den Anfang habe ich mit der Verteilung der benötigten Funktionen auf die drei Chips gemacht, denen ich bei dieser Gelegenheit in Erinnerung an diese schöne Tradition der alten Tage auch gleich noch passende Namen verliehen habe: Regnatix, Bellatrix und Administra. Durch diese Verteilung ergab sich eine Übersicht der Portbelegung aller drei Chips. Ich definierte die nötigen Signale, um einen Bus zwischen allen Elementen zu realisieren und entwarf das Busprotokoll für die Kommunikation zwischen ihnen.

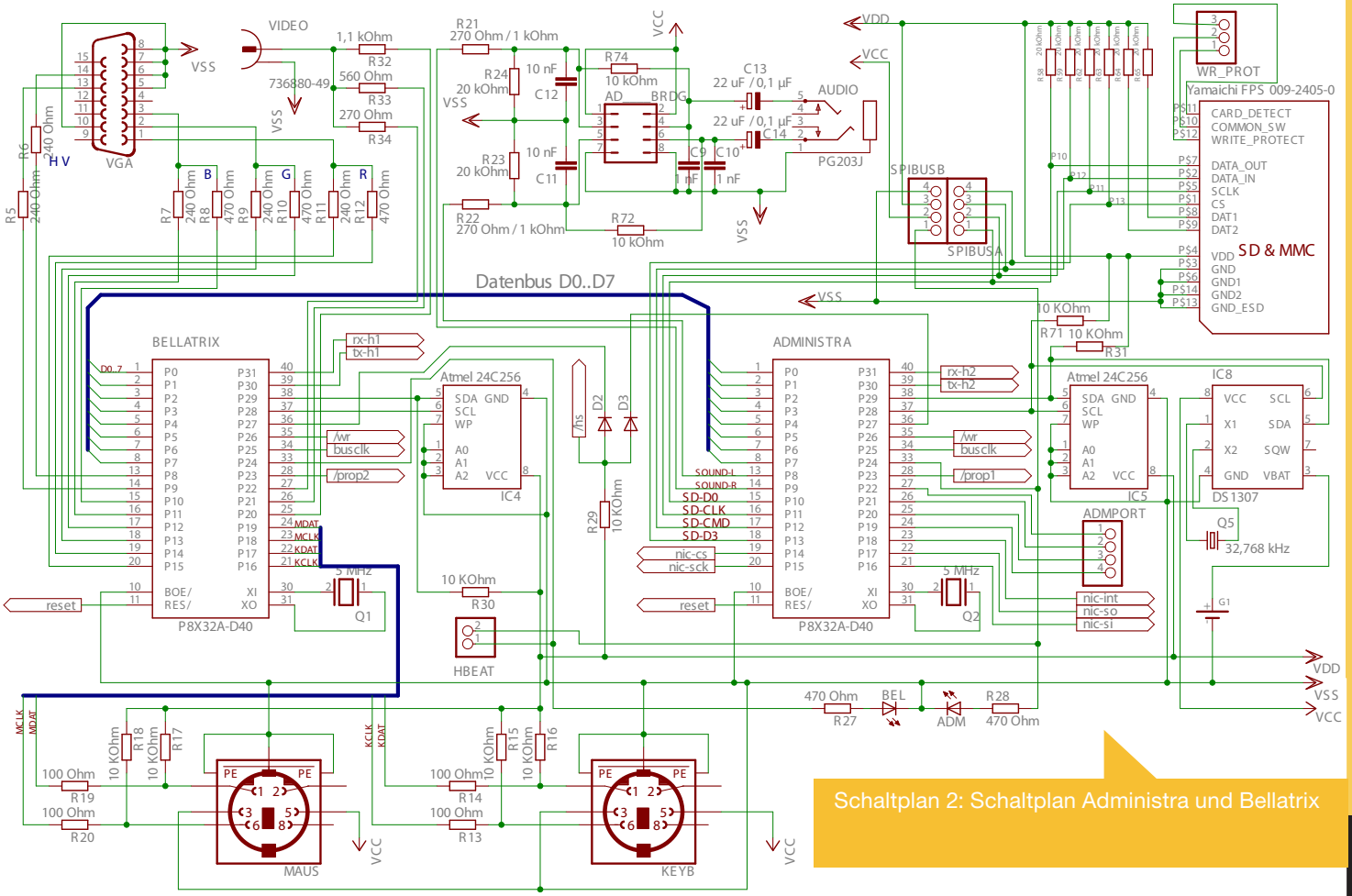
Der erste Chip (Bellatrix) sollte den gesamten Funktionskomplex der Benutzerinteraktion übernehmen, also Grafikkarte, Keyboard und Mausinterface. Im Hinterkopf hatte ich den Gedanken, dass Bellatrix bei geschickter Nutzung in sich eine ganze GUI realisieren könnte. Maussteuerung, Menüs, Fenster und Dialoge blieben dann autark in diesem Baustein. Neben Textmodis können auch diverse Grafikbetriebsarten realisiert werden. Einfache Rastergrafik mit Vektordarstellung oder tilesbasierte Grafik mit Sprites und Scrolling und bis zu 96 Farben gleichzeitig sind im Rahmen der Möglichkeiten realisierbar. Limitierender Faktor ist dabei der Speicher: mit 32 KB RAM für den Grafikpuffer und den Code ist das schon eine kleine Herausforderung.

Ein zweiter Propellerchip (Administra) sollte alle allgemeinen I/O-Aufgaben erledigen: SD-Card mit FAT16/32 Dateisystem, Soundkarte, LAN-Interface sowie ein I/O-Port. Auch hier können recht komplexe Funktionen lokal ausgeführt werden: So kann Administra zum Beispiel eigenständig per Kommando Soundtrackermodule oder SID-Sounds direkt von SD-Card abspielen oder man könnte einen minimalistischen Web- oder FTP-Server direkt und unabhängig in Administra laufen lassen. Auch die Verwaltung des Dateisystems spielt sich im wesentlichen komplett in Administra ab.

Und schlussendlich der dritte Mikrocontroller (Regnatix) steht mit seinen acht COGs vollständig den Anwendungsprogrammen zur Verfügung. Da dieser Schaltkreis im Gegensatz zu den Anderen keine I/O-Aufgaben mehr zu realisieren hat, wurden die dadurch verfügbaren Ports zum Anschluss zweier externen RAM-Bänke von je 512 KB verwendet. Notwendig wurde noch ein 8 Bit Latch für den höherwertigen Adressteil. Durch den Anschluss der RAMs war ein 8Bit-Datenbus nötig, über welchen auch gleich die drei Mikrocontroller untereinander kommunizieren sollten. So der grobe Plan.

Aktuell nutze ich den Hive als reines Master-Slave-System. Administra und Bellatrix stellen als Slave passiv ihre Funktionen im System zur Verfügung und werden durch den „Mastercode“ in Regnatix gesteuert. Im Prinzip ist das aber eine Betriebsart, die rein durch die Software in den drei Mikrocontrollern bestimmt wird. Jeder Chip kann zur Laufzeit und programmgesteuert mit einem vollständig neuen Code von SD-Card geladen werden, wodurch es durchaus möglich ist, aus dem Master-Slave-System auch ein Multimastersystem zu machen, sofern das irgend einen Nutzen bringt.

Die Mikrocontroller werden mit 3,3V versorgt, einige andere Schaltkreise wie die externen RAMs, das Latch, der MAX232 und der Ethernet Controller benötigen allerdings 5V, weshalb beide Spannungen durch zwei Längsregler stabilisiert werden. Wird auf dem Board auch die LAN-Schnittstelle bestückt, so benötigen die Regler einen ausreichenden Kühlkörper. Verpolschutz und Schutzdioden gegen Überspannung sind obligatorisch. Um die drei Mikrocontroller vom PC aus zu programmieren, benötigt der Hive eine serielle Schnittstelle. Schön wäre ein USB zu seriell UART Interface wie der FT232R gewesen, aber es gibt leider keine passenden Schaltkreise in DIL Bauform. So realisierte ich das Interface klassisch mit einem MAX232 für eine



Schaltplan 2: Schaltplan Administra und Bellatrix

RS232 Schnittstelle. Um wählen zu können, welcher der drei Mikrocontroller mit der Schnittstelle verbunden ist, sind die Signale über einen DIP-Schalter geführt.

Als Echtzeituhr ist ein RTC DS1307 an Administra angeschlossen. Dieser Schaltkreis besitzt eine I2C-Schnittstelle und ist parallel zum vorhandenen EEPROM an die schon bestehende I2C-Schnittstelle angeschlossen. Durch die Verwaltung durch Administra verfügt die Software für das Dateisystems gleich über die aktuelle Zeit um die Zeitstempel bei der Dateiarbeit korrekt zu setzen. Ebenfalls durch Administra wird das letzte zusätzliche aktive Bauteil per SPI angesteuert - der Ethernet Controller ENC28J60.

Betriebssystem und Programmierung

Mit der Hardware wucherte auch die Software. Was anfänglich nicht mehr als ein Monitorprogramm für Testzwecke war, bekam immer mehr die Struktur eines sehr einfachen Betriebssystems, welches irgendwann im Forum auf den Namen TriOS getauft wurde. Die Idee hinter TriOS ist eine einfache und universelle Plattform, um Programme von SD-Card zu starten und Dateien und Tools auf dem Laufwerk zu verwalten. Das System sollte klein und einfach sein, schnell starten und soweit wie möglich in SPIN programmiert werden, um Interessierten und natürlich auch mir selbst den Einstieg zu erleichtern. Geschwindigkeit und Rafinesse standen bei TriOS nicht im Vordergrund.

Gestartet wird das Kernsystem komplett aus den drei EEPROMs und ist damit auch ohne Bootvorgang sofort verfügbar. Schaltet man den Hive ein, so meldet sich die integrierte Programmiersprache des Systems: ein angepasstes PropForth. Es fühlt sich wirklich an wie ein alter Homecomputer: Einschalten und programmieren! Im Forth ist typischerweise Interpreter und Compiler integriert. Diese spezielle Version für den Propeller startet bis zu acht unabhängige Forthsysteme in den COGs, wodurch man wirklich interaktiv mit mehreren parallel laufenden Systemen hantieren kann. Das Wörterbuch wird gemeinsam genutzt und der Zugriff für den Compiler durch eine Semaphore gesteuert. Die einzelnen Forthsysteme können über ein einfaches Pipe Konzept miteinander in verschiedener Form verschaltet werden um zu kommunizieren. Eine Forth-COG ist an die serielle Schnittstelle zum Host gekoppelt. Mit einem einfachen Terminalprogramm auf dem Hostcomputer oder einem zweiten Hive als Terminal kann so auf zwei Monitoren gleichzeitig im Forth gearbeitet werden. Das ist schon eine sehr erstaunliche Sache und auch nützlich, wenn man ein Forthcode testen möchte, der die VGA/TV-Ausgabe belegt. Mit PropForth ist auf dem Hive und seinen begrenzten Ressourcen damit auch Multitasking auf Anwendungsebene möglich! Wer Forth nicht mag, kann TriOS auch ohne diese Programmiersprache installieren. Mit dem Forth ist ein Compiler und Interpreter integriert und der Hive kann ohne Unterstützung durch einen Host autark programmiert werden.

Leider würden die vielfälltigen Möglichkeiten dieser Sprache den Rahmen dieses Artikels sprengen. Allerdings war mir die Integration von Forth eine Herzensangelegenheit: So ist der Hive nach vielen Jahren neben dem Jupiter ACE der einzige Homecomputer mit einem integrierten Forth als Programmiersprache. Mit dem Kommando „regime“ kann direkt in eine klassische Kommandozeile gewechselt werden. Dabei wird das Forth beendet und von der SD-Card die Datei „reg.sys“ gestartet, welche die Kommandozeile beinhaltet. Einfache Kommandos zur Navigation im FAT32-Dateisystem mit Unterstützung von Verzeichnissen und zur Verwaltung von Dateien sind vorhanden. Lange Dateinamen werden nicht unterstützt und es gibt momentan auch keinen Pfad-Parser. Leider bietet momentan die verwendete FatEngine nicht die Möglichkeit mehrere Dateien gleichzeitig zu öffnen. Programmdateien werden an ihrem Extender (*.bin, *.bel, *.adm) erkannt, und in dem entsprechenden Chip gestartet, denn wir haben schließlich drei aktive Mikrocontroller im System. Dabei können externe Kommandos und Tools zentral in dem Systemordner auf der Karte gespeichert werden.

Die ausführbaren Dateien können direkt mit dem Propellertool oder alternativ und besser mit „Brad's Spin Tool“ (BST) auf einem Hostcomputer erstellt und gespeichert werden. Der BST-Compiler ist für Windows, Linux und Mac verfügbar und aktuell die beste Wahl, leider aber keine freie Software, da die Quelltexte nicht verfügbar sind. Nicht unerwähnt bleiben soll auch Sphinx-OS, welches ein einfaches OS mit Spin Compiler darstellt und auf einem einzigen Propellerchip läuft. Momentan gibt es aber leider noch keine Portierung für den Hive. TriOS ist einfach strukturiert und eine erste Basis um mit dem Hive zu experimentieren. Tools und Programme können sehr unkompliziert in Spin oder Forth programmiert werden. Für viele Ressourcen gibt es eine breite Palette von Funktionen in der IOS-Bibliothek von TriOS, um mit wenigen Funktionsaufrufen mit Zeichenausgabe, Grafiken, Sound und Dateien zu hantieren.

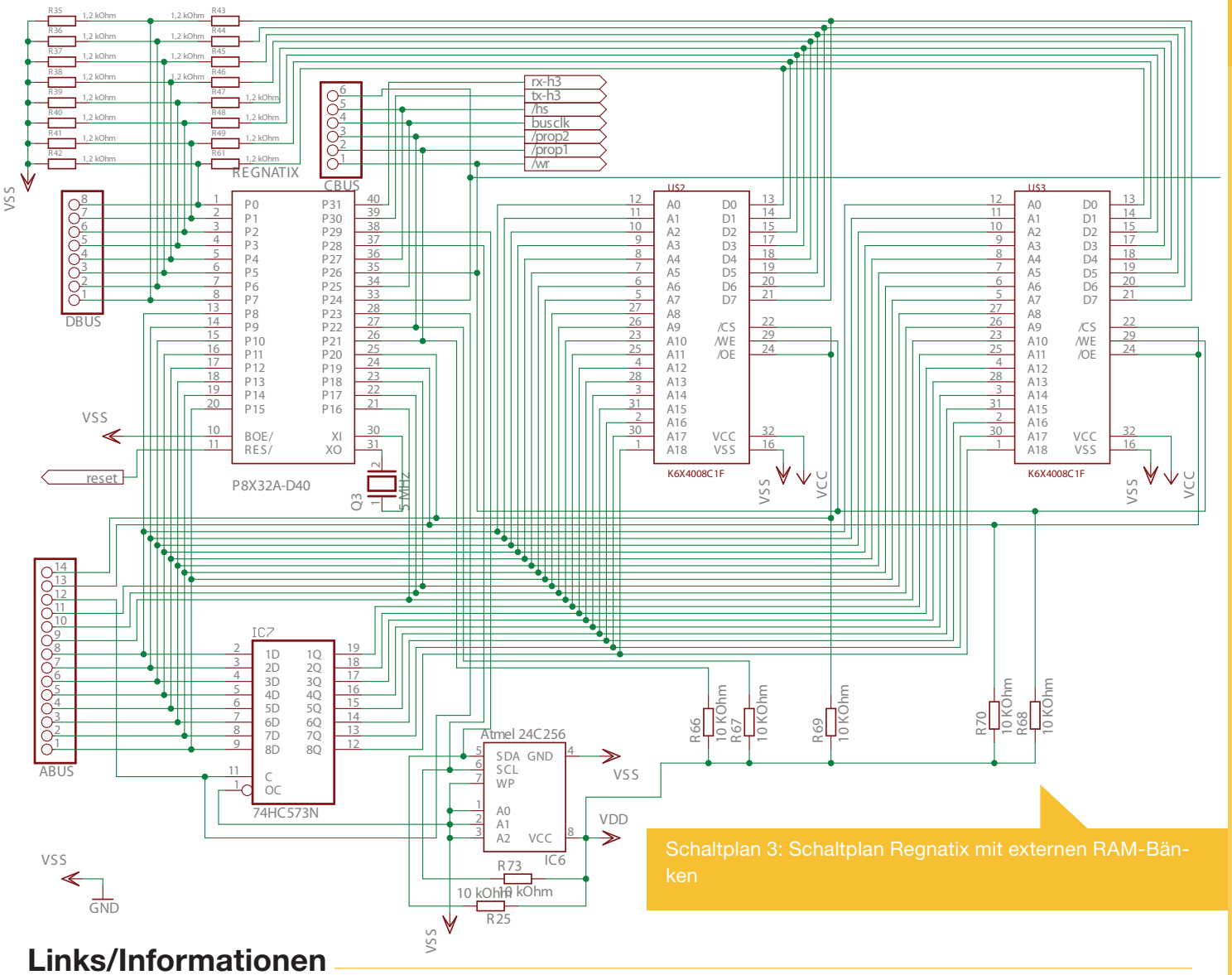
Wo gibts was?

Alle Informationen für den Bau eines Hive findet man auf der Webseite [1] des Projektes.

In regelmäßigen Abständen werden dort Sammelbestellungen durch Interessierte organisiert, meist werden wesentlich mehr Platinen als nötig bestellt – es lohnt sich also im Forum nach einem Board zu fragen. Die Platinen sind in der Regel in guter Industriequalität mit Lötstoplack, Bestückungsaufdruck und durchkontaktiert für unter 15,- € verfügbar, sofern die Bestellung groß genug war. Schalt- und Bestückungspläne, BOM , Aufbauanleitung und Software sind frei im Downloadbereich verfügbar. Die Platinendaten liegen im Eagle-Format vor und können so problemlos von jedem weiterentwickelt werden. Die Entwicklungsumgebungen und Handbücher sind frei beim Hersteller Parallax zu bekommen – entsprechende Links und Hinweise finden sich in dem umfangreichen Aufbaututorial.

Zusätzlich findet man auf der Webseite einige Sammlungen mit Programmen, Demos und Games. Neben der Webseite sind sehr viele Informationen und - ganz wichtig - Hilfestellungen zu allen Problemen rund um den Hive und andere verwandte Projekte im Forum zu finden.

Wer allerdings vor hat einen Hive zu bauen, sollte sich in Ruhe den Eignungstest „Reif für den Hive“ [2] anschauen, um zu prüfen, ob die eigenen Vorstellungen mit den Gegebenheiten und Möglichkeiten im Einklang sind. Der Hive ist und bleibt ein kleiner Experimentalrechner, ein besserer C64, aber er ist zu 100% selbst gebaut und mit eigener Software zum Leben erweckt! Das Motto lautet: „Der Weg ist das Ziel!“, und so lebt das Gerät im Wesentlichen von einem: Dem Willen auch wirklich etwas selbst bauen und programmieren zu wollen.



Schaltplan 3: Schaltplan Regnatix mit externen RAM-Bänken

Links/Informationen

[1] Projekt-Webseite: http://www.hive-project.de

[2] Eignungstest: http://hive-project.de/projekt-info/reif-fur-den-hive

[3] Video mit den ersten Lebenszeichen des Hive-Prototypen: http://youtu.be/efAKdqRwl2U

[4] Ein Video, welches den StarTracker in Aktion zeigt: http://youtu.be/n_-kjd8uOjM

[5] Video Computertrend-TV 04/2007 über den Propellerchip: http://youtu.be/kA8sq_7nYlE

Das Infinity Racing Team der Hochschule Kempten

HS Kempten

Einleitung

Das Infinity Racing Team ist ein studentisches Projekt an der Hochschule Kempten. Ziel ist die Entwicklung, Konstruktion, Fertigung und Vermarktung eines einsitzigen Formel-Rennwagens, um mit diesem erfolgreich am internationalen Konstruktionswettbewerb der Formula Student teilzunehmen. Dieses anspruchsvolle Vorhaben wird jährlich von engagierten und motivierten Studentinnen und Studenten der Hochschule Kempten in Eigeninitiative und Teamarbeit umgesetzt. Somit sammeln die Mitglieder des Rennteams die von vielen Unternehmen hochgeschätzte Projekt- und Praxiserfahrung bereits während ihrer Studienzeit.

Geschichtliches

Ende der 1970er Jahre wurde dieser Konstruktionswettbewerb von der Society of Automotive Engineers als Formula SAE in den USA ins Leben gerufen. Im Jahr 2006 wurde die Formula Student zum ersten Mal vom Verein Deutscher Ingenieure auch in Deutschland ausgerichtet und findet seither jährlich statt. Bereits seit 2008 nimmt das Infinity Racing Team an den internationalen Wettbewerben teil. So wurde das Studententeam der Hochschule Kempten im Jahr 2008 mit ihrem Rennboliden TOMSOI I zum „Best Newcomer“ gekürt und führte somit die erstmalige Teilnahme auf dem Hockenheimring zu vollem Erfolg!

Auch in den Folgejahren wurde eifrig am Fahrzeug entwickelt und so nahm das Team im Jahr 2009 mit einer optimierten und überarbeiteten Version des TOMSOI I an der Formula Student Austria, auf dem Wachauring bei Wien, teil. Auch hier erwies sich die solide Konstruktion des Erstjahresfahrzeugs als konkurrenzfähig und zuverlässig. Parallel wurde von den eifrigen Studenten jedoch schon an einem Nachfolgemodell des TOMSOI I getüftelt. Mit TOMSOI II bestritt das Rennteam souverän die Wettbewerbe in Ungarn und Spanien. Voller Tatendrang und mit vielen neuen Ideen, Eindrücken und Vorstellungen stürzte sich das Infinity Racing Team nach der Rückkehr aus Spanien, im Oktober 2010, in die Konzept- und Entwicklungsphase des TOMSOI III, dem dritten Fahrzeug des Kemptener Teams. Mit diesem Rennwagen nahmen sie an den Wettbewerben in Deutschland, Italien und Spanien teil. Auch in diesem Wettbewerbsjahr konnte Infinity Racing einige erwähnenswerte Erfolge erzielen. In der Saison 2011 konnte das Kemptener Rennteam einen erfolgreichen 35. Platz von insgesamt 78 teilnehmenden Teams in Hockenheim sichern. Während der Wettbewerbsphase wurde der Bolide fortlaufend verbessert. Diese Optimierungsarbeit wurde durch einen 11. Platz in Varano de‘ Melegari belohnt. Den ersten Podiumsplatz in der Geschichte des Infinity Racing Teams konnten die Studenten 2011 am Circuit de Catalunya bei Barcelona feiern. Derzeit ist das Rennteam mit der Konstruktion des vierten Rennwagens, TOMSOI IV, beschäftigt und nimmt auch im Jahr 2012 an drei Wettbewerben teil.

Ein motiveirtes Team

Entscheidend für den Projekterfolg ist jedoch nicht der schnellste Rennwagen, sondern das Gesamtpaket, bestehend aus den Disziplinen Konstruktion, Rennperformance, Finanzplanung und Präsentation des Fahrzeugs führt zum erzielten Erfolg. So müssen die Studenten auf den Wettbewerben auch eine akkurat ausgearbeitete Kostenkalkulation des Gesamtfahrzeugs präsentieren und eine Verkaufs- und Marketingstrategie vorstellen. Letztere sieht eine Kleinserienproduktion des Fahrzeugs von 1.000 Einheiten, im Sektor der Rennfahrzeuge für ambitionierte Hobbyrennfahrer, vor. Während sich die technischen Ressorts um die Entwicklung und Fertigung des Boliden kümmern, schaffen die wirtschaftlichen Ressorts die optimalen Rahmenbedingungen und Voraussetzung für die Umsetzung. So spielt die Organisation und Verwaltung sowie die Finanzierung und das Marketing des Projektes in diesem Bereich eine wesentliche Rolle, welche ebenfalls von Studenten unterschiedlichster Studiengänge in Eigeninitiative höchst erfolgreich gemeistert wird. Im Jahr 2011 hatte das Infinity Racing Team, neben den Wettbewerben, 17 öffentliche Auftritte auf Messen und Ausstellungen. Unter anderem wurden die Fahrzeuge an der Allgäuer Hochschulmesse, der Pyramid Frimenkontaktmesse in Augsburg und am Tag des Sports der Allgäuer Festwoche ausgestellt. Finanziert wird das Projekt hierbei ausschließlich durch zahlreiche Sponsoren und Partner aus der Wirtschaft und Industrie sowie der Hochschule Kempten.

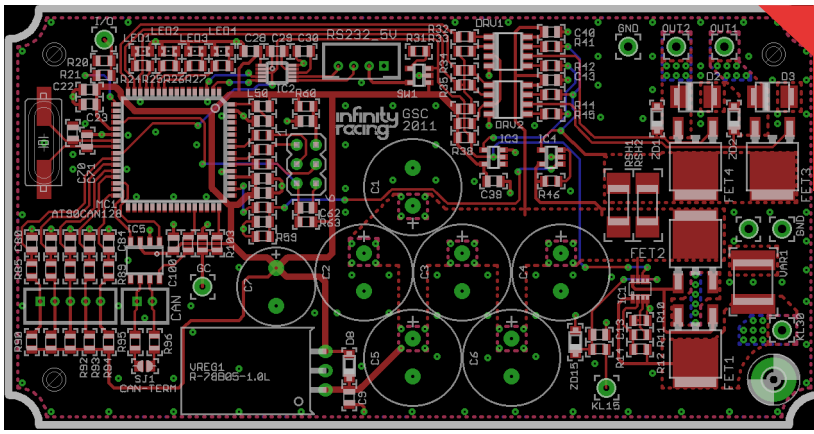


Abb. 1: TOMSOI III in der Boxengasse, Circuit de Catalunya

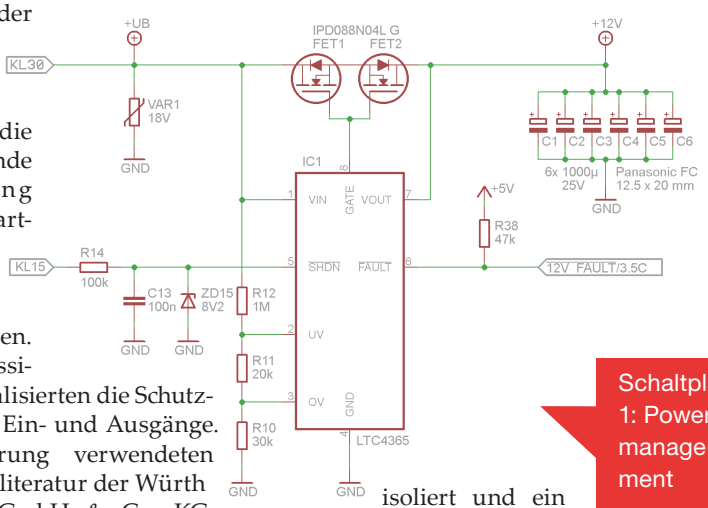
Abb. 2: Teamfoto in Varano de' Melegari

... auch im Bereich Elektronik und Informatik

Für den Rennwagen TOMSOI III entwickelten die Mitglieder des Ressorts Elektronik und Informatik eine per Software gesteuerte Betätigung der Gangschaltung sowie ein Telemetriesystem zur drahtlosen Übertragung verschiedenster Fahrzeugparameter an die Boxengasse. Für den Antrieb des 246 kg wiegenden Boliden sorgt ein 82PS starker Yamaha YZF-R6 Motorradmotor mit 600 ccm. Die maximal erreichbare Leistung begrenzt dabei ein laut Reglement vorgeschriebener air restrictor.

Die rauen Umgebungsbedingungen im Fahrzeug, u.a. hohe Temperaturen und permanente Vibrationen, erfordern den Schutz aller elektronischen Baugruppen. Grundvoraussetzung für die verwendeten Gehäuse und Leitungen war zudem die Beständigkeit und Dichtheit gegen sämtliche Flüssigkeiten (Kraft- und Schmierstoffe, Kühlflüssigkeit etc.). Die Wahl fiel schließlich auf Aluminium-Druckgussgehäuse aus dem Hause Hammond Mfg. welche dank IP65-Zertifizeirung die genannten Anforderungen erfüllten.

Zuverlässiger Betrieb der selbstentwickelten Module, insbesondere bei permanenter Anwesenheit von Störungen magnetischer und elektrischer Wechselfelder im Bordnetz sowie deren Einkopplungseffekte, musste sichergestellt werden. Hauptsächlich verursachen Bursts der elektronischen Zündung, schaltender Relais oder induktiver Drehzahlsensoren (USS ≈ 150V) derartige Störemissionen. Lastabwurf der Lichtmaschine bei Betätigung eines Not-Aus-Schalters oder die stark schwankende Batteriespannung während der Startphase dürfen die Funktion der Module ebenfalls nicht beeinflussen. Varistoren und passive Tiefpassfilter realisierten die Schutzbeschaltungen der Ein- und Ausgänge. Zur Dimensionierung verwendeten die Studenten Fachliteratur der Würth Elektronik eiSos GmbH & Co. KG („Trilogie der Induktiven Bauelemente“).

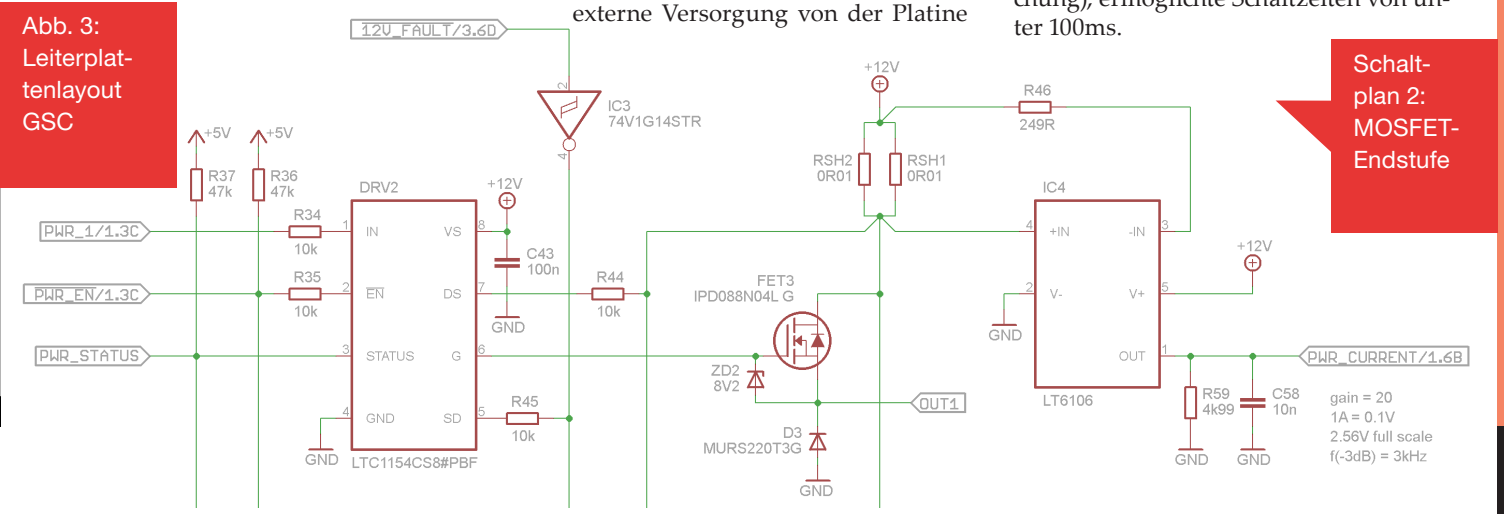


Schaltplan 1: Powermanagement

Bei beiden Modulen verwendete Infinity Racing den Atmel 8-bit Mikrocontroller AT90CAN128. Sein integriertes CAN 2.0 Modul ermöglichte eine einfache und schnelle Anbindung an den 1 Mbit/s-Bus im Rennwagen, welcher das Motorsteuergerät MS4 und das ABS M4 von Bosch Motorsport mit dem Dashboard vernetzt. Das Power Management IC LTC4365 von Linear Technology überwacht die Spannung des Bordnetzes an den Modulen und steuert die Gatespannung zweier MOSFETs in Back-to-Back-Anordnung als ideale Diode. Diese Schaltung realisierte sowohl eine Verpolschutz als auch die Über- und Unterspannungsüberwachung. Sollte die Versorgungsspannung den Grenzwert von 17.5 V überschreiten bzw. 10.5V unterschreiten, wird die externe Versorgung von der Platine isoliert und ein Interrupt am Controller ausgelöst, um relevante Daten im EEPROM ablegen zu können, bevor die Pufferkondensatoren die Dropout-Spannung des 5V-Schaltreglers unterschreiten und der Controller durch den Brownout-Reset zurückgesetzt wird.

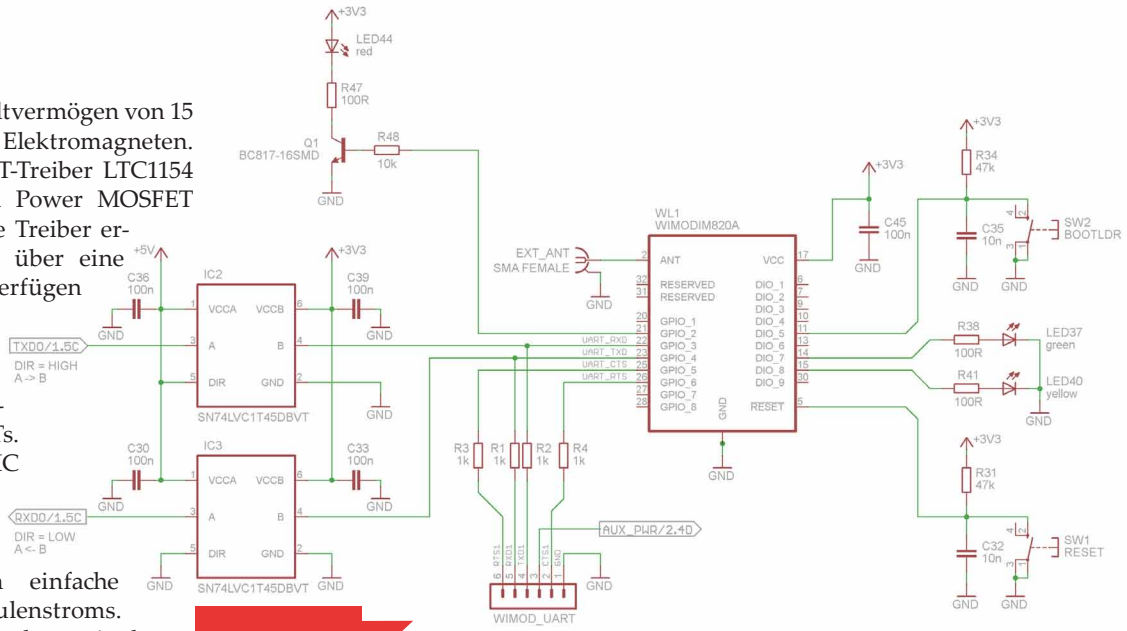
Das GSC-Modul (Gear Shift Controller) erweitert den Mikrocontroller um eine verlustarme Highside-N-Kanal MOSFET-Endstufe. Es überwacht die vom Motorsteuergerät per CAN-Bus übermittelte Motordrehzahl und steuert auf Grund von Schaltbefehlen der Schaltwippen am Lenkrad den Gangwechsel am sequentiellen 6-Gang-Getriebe des Yamaha YZF-R6 Aggregats mit Hilfe eines elektromagnetischen Aktuators. Kupplungsloses Hochschalten über eine Drehmomentreduktion des Motors (Zündunterbrechung), ermöglichte Schaltzeiten von unter 100ms.

Abb. 3: Leiterplattenlayout GSC



Schaltplan 2: MOSFET-Endstufe

Die Elektronik bietet ein Schaltvermögen von 15 Ampere zur Ansteuerung des Elektromagneten. Verwendung fanden MOSFET-Treiber LTC1154 von Linear Technology und Power MOSFET IPD088N04 von Infineon. Die Treiber ermöglichen 100% duty cycle über eine integrierte charge pump, verfügen über Sense-Anschlüsse für eine Überstromerkennung mit Shunt-Widerstand und verhindern so einen Kurzschlussschutz der MOSFETs. Ein LT6106 Current Sense IC zur Strommessung über den ADC des Atmel Controllers, ermöglicht eine Schaltkraftanpassung durch einfache PWM-PI-Regelung des Spulenstroms. Die hohe Schaltleistung erforderte ein besonders sorgfältiges Layout mit möglichst kurzen Hochstrompfaden, außerdem sorgten großflächig dimensionierte Kupferflächen für die Wärmeabfuhr der MOSFETs. Die strikte Trennung der Massepfade für Leistungsbauelemente und digitale Schaltungsteile war obligatorisch. Beim TTX-Modul (Telemtetry Transceiver) handelt es sich prinzipiell um eine CAN-RS232-Bridge mit drahtloser Signalübertragung, welches einen Datenaustausch zwischen dem Rennwagen und einem PC in der Boxengasse ermöglichte. Sämtliche eingehenden CAN-Messages von Motorsteuergerät, ABS und GSC werden gefiltert und zyklisch via SRD-Band (868MHz) an die Box gesendet. Es erlaubt zum Einen die ständige Kontrolle aller Fahrzeugparameter, zum Anderen entlastet es die Fahrer, die sich ausschließlich auf die Stecke und das Fahren konzentrieren können.



Schaltplan 3: WiMOD Funkmodul

Die drahtlose Verbindung wurde über WiMOD iM820A Industriefunkmodule der Firma IMST GmbH hergestellt, welche nach Initialisierung einen transparenten Datenkanal vom UART-Interface des Mikrocontrollers zum auswertenden PC herstellen. Typischerweise liegt die Übertragungsrate bei 38.4 kBaud, bei leicht reduzierter Baudrate ist eine Reichweite von bis zu 1000m bei idealen Bedingungen möglich. □

Wir wünschen dem Infinity Racing Team viel Erfolg und gute Platzierungen für die kommende Saison!

Links

[1] www.infinity-racing.de

[2] http://www.youtube.com/user/InfinityRacingFSAE/videos

[3] www.facebook.com/infinity.racing

Audiotrenner zur Vermeidung von Masseschleifen

Ronny Schmiedel

Einleitung

„Da hat man sich in sein Auto zum „High-End-Autoradio“ noch eine „fette“ Endstufe eingebaut und nun pfeifft zur Musik freudig die Lichtmaschine mit. **Oder:** Nach dem Anschliessen des Computers an die Stereoanlage untermalt ein lästiges Brummen aus der Anlage den lieblichen Klang der Musik.“

Da ist guter Rat teuer. Eine Lösung wäre, 2 Tonköpfe aus alten Kassettenrecodern Kopf an Kopf anzuordnen und darüber das Audiosignal zu leiten a la Kasettenadapter fürs Autoradio (Klang grässlich) oder einen käuflichen Audiotrenner zu nehmen. Im Inneren sind aber meistens 2 Übertrager was zu Klangeinbußen im unterem Frequenzbereich führt.

Ich habe mich erstmal mit der Ursache des Übels beschäftigt. Durch die unterschiedlichen Massen (Minus) im Auto oder unterschiedlichen Steckdosen zwischen Computer und Stereoanlage kommt es zu Potentialunterschieden der Masse der beiden angeschlossenen Geräte. Wenn diese jetzt über die Abschirmung des Audiokabels miteinander verbunden werden, fliesst automatisch ein Ausgleichsstrom. Jetzt könnte man einfach die Abschirmung unterbrechen. Aber Pustekuchen, dann fliesst eben halt der Strom über die Audioleitung und wir haben ein noch grösseres Problem. Dieser Strom kann unter Umständen auch zur Beschädigung eines der angeschlossenen Geräte führen.

Eine ausführlichere Erläuterung ist zu finden auf:

http://www.masseschleife.de

Meine Lösung dieses Problems sieht folgender Maßen aus: Ich benutze ein einfachen Differenz- oder Subtraktionsverstärker aufgebaut mit Operationsverstärkern. Damit werden die Potentialunterschiede aufgehoben und das lästige Brummen oder Pfeiffen gehört der Vergangenheit an. Da der OPV symmetrisch versorgt wird ist noch eine Ladungspumpe mit einem ICL7660 vorhanden für die Erzeugung der -5Volt.

Aufbau

Die Masse wird an die Platine NICHT angeschlossen, ansonsten funktioniert die Schaltung nicht. Die Masse holt sich die Schaltung über die Cinch-Stecker vom Endgerät was am Ausgang der Schaltung angeschlossen wird. Es werden NUR die +12Volt angeschlossen. Die Platine wird in ein TEKO-Metallgehäuse Typ A1 eingebaut. Für den Betrieb der Schaltung am Computer wird noch ein Steckernetzteil mit 12Volt Gleichspannung benötigt.

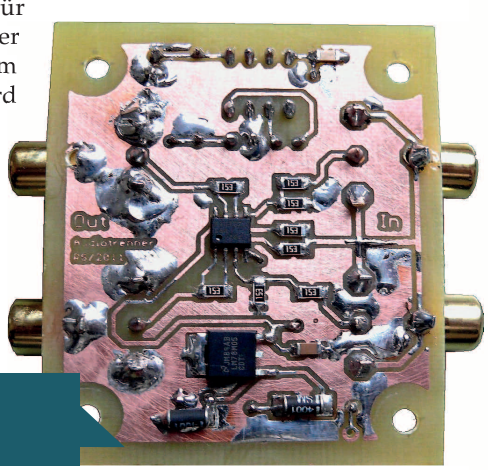


Abb. 1: Platine SMD

Links

Auf meiner Page [1] unter „Audiotrenner“ findet man auch noch eine ältere Version mit nur bedrahteten Bauteilen wer keine SMD-Bauteile löten möchte: □

[1] http://www.ronny-net.de (Gästebuch)

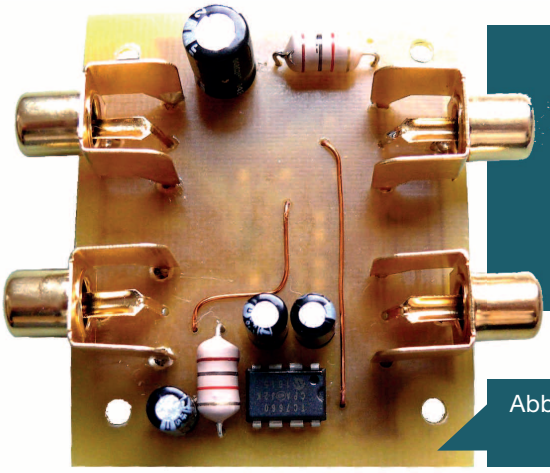


Abb. 2: Platine THT

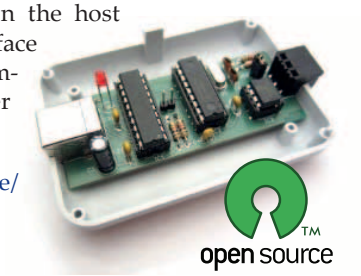
Fazit:
Die Schaltung ist bereits mehrfach im Einsatz und es gibt bis jetzt nur postive Resonanzen.

Marktplatz / Neuigkeiten

Die Ecke für Neuigkeiten und verschiedene Produkte

USBtin - Simple USB to CAN interface

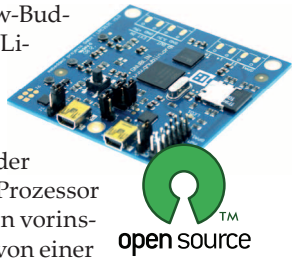
USBtin is a simple to build USB to CAN interface. It can monitor CAN busses and transmit CAN messages. The common CAN baud rates are supported. USBtin implements the USB CDC class and creates a virtual comport on the host computer. The interface is controlled with simple commands over this serial port.



http://www.fischl.de/usbtin/

GNUBLIN Lern- und Ausbildungsplattform

GNUBLIN ist ein „Low-Budget“ embedded GNU/Linux Board. Der Perfekte Einstieg in die Linux Welt auf Mikrocontrollerbasis. GNUBLIN in der Version mit dem ARM9 Prozessor LPC3131 von NXP läuft ein vorinstalliertes Linux, welches von einer SD-Karte gebootet wird. Wesentliche Schnittstellen zum Mikroprozessor wie Digitale Ein- Ausgänge, Analoge Eingänge, I2C, SPI oder eine PWM-Leitung sind nach außen geführt.

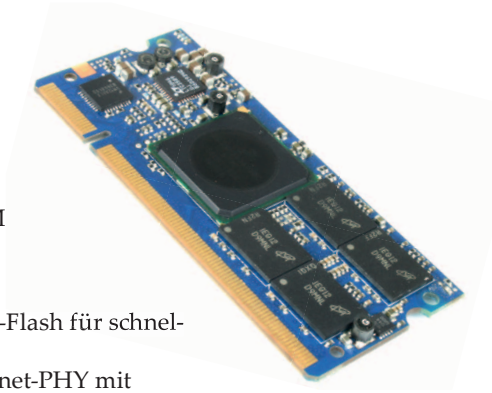


- ARM9 Prozessor mit 180 MHz (LPC3131)
- Integrierter USB-RS232 Wandler
- vorinstalliertes Linux

http://www.gnublin.org

iMX536 Cortex-A8 Prozessor mit bis zu 1GHz Prozessortakt

Die In-Circuit GmbH aus Dresden bietet ihr neuestes Prozessormoul im SODIMM200 Formfaktor an. Dieses Modul verbindet höchste Rechengeschwindigkeit und üppige Speicherausstattung mit einer hohen Schnittstellenauswahl und -Anzahl.



Features:

- iMX536/iMX535 Cortex-A8 Prozessor mit bis zu 1GHz Prozessortakt
- 512 MByte 32-Bit DDR3-RAM
- 4GByte NAND-Flash
- 8MByte 16Bit paralleler NOR-Flash für schnelles Booten
  integrierter (Industrie) Ethernet-PHY mit 10/100MBit
- komplette Spannungsversorgung on-board, Eingangsspannung 5V
- Temperaturbereich -20°C bis 70°C
- nur ca. 2Watt Leistungsaufnahme

Schnittstellen:

- 2xUSB-HOST, 1xUAB-Device, 1xEthernet, 1xSATA, 2xCAN, 4xUART, 2xSPI, I2C, GPIOs

http://ic-board.de/

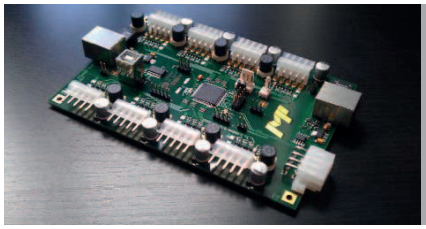
Haben Sie interessante Produkte oder Neuigkeiten? Dann senden Sie uns ein Foto, 3-4 Zeilen Text und einen Link mit Betreff „Marktplatz“ an:

journal@embedded-projects.net

LEDControl 8x3

Der LCDControl8x3 bietet die Möglichkeit 24 LED-Kanäle über PWM anzusteuern. Dabei erfolgt die Datenübermittlung über DMX512, USB, UART, LAN, LVDS, I2C, SPI oder eine Funkschnittstelle. Die SPI-Schnittstelle kan für das Schalten oder Dimmen von 8 230V-Lasten verwendet werden. Außerdem stehen 2 GPIOs, 1 DAC und 3 ADCs zur Verfügung.

http://www.michalak-platzer.com/



Dies ist ein Produkt präsentiert im Experten-Wegweiser „Find your engineer“: http://www.find-your-engineer.de

DAS ORIGINAL SEIT 1994

PCB-POOL®

Beta LAYOUT

Oft kopiert – doch nie erreicht:

	PCB-POOL®	Basista	Euro-circuits	Leiton	WEdirekt	multi-cb
Leiterplatten online kalkulieren	✓	✓	✓	✓	✓	✓
FREE STENCIL	✓	–	–	–	–	–
Bestückung online	✓	–	–	–	–	–
Kostenlose Layoutsoftware	✓	–	–	–	–	–
Bewertungs-Rabattsystem	✓	–	–	–	–	–
Akzeptierte Layoutformate	16	6	1	3	5	3
Kollisionsprüfung zum Anfassen	✓	–	–	–	–	–
Auftragsverfolgung mit Ansprechpartner	✓	–	–	–	–	–
Watch"ur"PCB	✓	–	–	–	–	–
Pünktlich oder kostenlos	✓	–	–	✓	–	–
8h-Eilservice	✓	✓	–	–	–	–
Online Daten-Restore Service	✓	–	–	–	–	–

Das Original seit 1994!

www.pcb-pool.com

Beta LAYOUT

embedded - projects.net
JOURNAL
OPEN SOURCE SOFT-AND HARDWARE PROJECTS

Bitte frei machen

embedded projects GmbH
Holzbachstraße 4
D - 86152 Augsburg

Name / Firma

Straße / Hausnummer

PLZ / Ort

Email / Telefon / Fax

◯ Ich möchte jede zukünftige Ausgabe erhalten

◯ Wir möchten als Hochschule / Ausbildungsbetrieb jede weitere Ausgabe bekommen. Bitte gewünschte Anzahl der Hefte pro Ausgabe ankreuzen. 5 10

◯ Ich möchten im embedded projects Journal werben oder eine Stellenanzeige aufgeben. Bitte schicken Sie mir Infomaterial, Preisliste etc. zu.

Werdet aktiv!

Das Motto: Von der Community für die Community !

Das Magazin ist ein Open Source Projekt.

Falls Du Lust hast, Dich an der Zeitschrift durch einen Beitrag zu beteiligen, würden wir uns darüber sehr freuen. Schreibe deine Idee an:

journal@embedded-projects.net

Regelmäßig

Die Zeitschrift wird über mehrere Kanäle verteilt. Der erste Kanal ist der Download als PDF - Datei. Alle Ausgaben sind auf der Internetseite [1] verfügbar. Diejenigen, die lieber eine Papierversion erhalten möchten, können den zweiten Kanal wählen. Man kann sich dort auf einer Internetseite [2] in eine Liste für die gesponserten Abos oder ein Spendenabo eintragen. Beim Erscheinen einer neuen Ausgabe wird dank Sponsorengeldern an jeden auf der Liste eine Ausgabe des aktuellen Journal versendet. Falls man den Versandtermin verpasst hat, kann man das Heft auch über einen Online - Shop [2] beziehen.

[1] Internetseite (Anmeldeformular gesponserte Abos): http://journal.embedded-projects.net

[2] Online - Shop für Journal:
http://www.embedded-projects.net

Sponsoren gesucht!

Damit wir weiterhin diese Zeitschrift für jeden frei bereitstellen können, suchen wir dringend Sponsoren für Werbe- und Stellenanzeigen. Bei Interesse meldet Euch bitte unter folgender Telefonnummer: 0821 / 2795990 oder sendet eine E-Mail an die oben genannte Adresse.

Impressum

embedded projects GmbH
Holzbachstraße 4
D-86152 Augsburg
Telefon: +49(0)821 / 279599-0
Telefax: +49(0)821 / 279599-20

Veröffentlichung: 4x / Jahr
Ausgabenformat: PDF / Print
Auflagen Print: 2500 Stk.
Einzelverkaufspreis: 1 €

Layout / Satz: EP
Druck: flyeralarm GmbH
Titelfoto: Claudia Sauter

BESSER GLEICH ONLINE KALKULIEREN.

STARRE UND FLEXIBLE LEITERPLATTEN.

ab 12 h Leiterplatten-Express! Pünktlich oder gratis!

LEITON
RECHNEN SIE MIT BESTEM SERVICE

Endlich wird's einfach und Sie bleiben flexibel. Den umständlichen und aufwändigen Kalkulationsprozessen machen wir einen dicken Strich durch die Rechnung. Sie kalkulieren Ihre Leiterplatten online – ganz bequem, ganz schnell. **Einzigartig: Die Online-Kalkulation gilt auch für Serien und flexible Leiterplatten.** Das macht uns weltweit so schnell keiner nach. Einmalig ist zudem der **Leiterplatten-Expressdienst** von LeitOn mit unserer Garantie: Wenn wir bei Expressdiensten den vereinbarten Übergabetermin an den Versender nicht einhalten können, schenken wir Ihnen die bestellten Platinen! Sie möchten mehr darüber wissen? Wir bieten persönliche Beratung am Telefon und einen kompetenten Außendienst. Sie können bei LeitOn immer mit dem besten Service rechnen.

www.leiton.de | **kontakt@leiton.de** | Info-Hotline **+49 (0)30 701 73 49 0**