Hochschule
Augsburg University of
Applied Sciences

Volker Thoms

Mikrocontrollerprogrammierung mit Python

Diplomarbeit

Studienrichtung Informatik

Abgabetermin: 12. Januar 2010

Hochschule Augsburg
University of Applied Sciences
Baumgartnerstraße 16
D 86161 Augsburg

Telefon +49 821 5586-0
Fax +49 821 5586-3222
http://www.hs-augsburg.de
poststelle@hs-augsburg.de

Fakultät für Informatik
Telefon: +49 821 5586-3450
Fax: +49 821 5586-3499

Entwicklung einer Softwareschicht zur Abstraktion der I/O-Funktionalitäten linuxbasierter Mikrocontroller für die Hochsprache Python

Verfasser:
Volker Thoms
Beilingerstr. 35a
86316 Stätzling
unconnected@gmx.de

Erstprüfer und Betreuer: Prof. Dr. Hubert Högl
Zweitprüfer:

Erstellungserklärung

Ich versichere, diese Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet zu haben.

Augsburg, den 10. Januar 2010 Unterschrift

Kurzfassung

Die Firma Atmel präsentiert auf dem *NGW100 Evaluation Board* ihren modernen 32-Bit Mikrocontroller, den AP7000. Die vorinstallierte Linux-Firmware hilft beim Einstieg in die *embedded Linux*-Welt: das Board ist als Router vorkonfiguriert und bietet Zugriff auf einfache I/O-Möglichkeiten, wie die Kontrolle von GPIO-Pins.
Möchte man allerdings andere Funktionalitäten nutzen, wie I2C, SPI oder die PWM-Einheit, stößt man schnell auf Grenzen, da die Möglichkeiten der Kommandozeile zur Programmierung recht beschränkt sind und für einige Peripherieeinheiten gar keine Schnittstelle zur Verfügung steht.
Der Entwickler steht vor der Aufgabe, sich in die *C*-Schnittstellen des Kernels oder gar die Treiberprogrammierung unter Linux einzuarbeiten. Da dies für einen Laien allerdings einen hohen zeitlichen Aufwand bedeutet, wurden in dieser Arbeit **Erweiterungen** für die einsteigerfreundliche Skriptsprache **Python** zusammengestellt (py-smbus, pyserial) und erstellt (py-spi, py-pwm, py-gpio, py-softpwm). Sie bieten einen einfachen Zugriff auf die Hardwareeinheiten. Die Programmierung erfolgte in *C* unter Verwendung der Python/C-API und der Schnittstellen des Kernels. Hierfür war die Analyse und teilweise Anpassung des Kernels erforderlich. Neben der Anwendung von Patches für PWM- und GPIO-Schnittstellen und Änderungen am Startupcode des Boards, wurde ein Kernelmodul geschrieben, um 16 zusätzliche PWM-Kanäle über GPIO-Pins zu emulieren. Weiterhin wurde der CAN-Controller MCP2515 am SPI-Bus aktiviert.
Das Ergebnis der Arbeit ist eine angepasste Version der Linuxdistribution ***OpenWrt*** für den NGW100, die den Pythoninterpreter und die Module zur Ansteuerung der GPIO-, SPI/Microwire-, I2C/SMBus-, UART- und PWM-Einheiten, sowie einige Beispielskripte enthält.

Inhaltsverzeichnis

1 Einleitung

Die Leistungsfähigkeit von Mikrocontrollern hat in den letzten Jahren weiter zugenommen. 32-Bit Architekturen, Taktfrequenzen von mehreren hundert Megahertz und integrierte Speicherverwaltungseinheiten sind keine Seltenheit und ermöglichen den Einsatz von Betriebssystemen, wie zum Beispiel Linux.
Linux ist auf verschiedenen Mikrocontrollerarchitekturen lauffähig. Einmal geschriebene Software ist zwischen ihnen portabel, solange die, vom Kernel bereitgestellten Schnittstellen verwendet werden. Ist für eine Hardwareeinheit allerdings noch kein Treiber vorhanden, steht der Anwendungsentwickler vor der Aufgabe, einen Gerätetreiber zu schreiben.
Um sich in die Benutzung der vorhandenen Kernelschnittstellen oder gar in die Treiberentwicklung unter Linux einzuarbeiten, ist ein hoher Lernaufwand nötig. Die Motivation für diese Arbeit ist es, einem Entwickler ohne Linuxkenntnisse die hardwarenahe Programmierung eines linuxbasierten Mikrocontrollerboards dadurch zu erleichtern, daß eine einfach zu erlernende Programmiersprache (Python) inklusive einfach zu bedienender Schnittstellen zum Zugriff auf die Peripherie zur Verfügung steht.
Die Idee für dieses Projekt stammt von Professor Hubert Högl, der mir auch als Berater und Betreuer zur Seite stand.

1.1 Mikrocontroller

1.1.1 Anwendungen

Mikrocontroller (μC, uC, Microcontrollerunit (MCU)) werden schon lange in vielen verschiedenen Aufgabenbereichen eingesetzt. Hierzu gehören Elektronik im Automotivebereich (Autos, Flugzeuge, Bahnen), industrielle Steuerungs- und Regelungsanlagen, medizinische Anlagen, Multimedia- und Kommunikationsgeräte, Roboterbau und viele weitere.
Die niedrigen Kosten bei hohen Stückzahlen machen sie attraktiv für die Massenproduktion. Durch geringen Stromverbrauch bietet sich ein Einsatz im Dauerbetrieb oder in mobilen Geräten an. Wegen ihrer Robustheit können sie auch in unwirtlichen Umgebungen verwendet werden.
Da Mikrocontroller in ihren Anwendungen häufig mit der realen Welt interagieren, also von Sensoren (Licht, Temperatur, Druck usw.) Daten empfangen und Anweisungen an Aktoren (Motoren, Ventile, Relais usw.) senden, ist Echtzeitfähigkeit, also rechtzeitige Reaktion auf Ereignisse eine häufige Forderung und Optimierungsziel.
Ist ein Mikrocontroller anwendungsgemäß verdrahtet und programmiert, spricht man auch von einem **embedded system** [1].

1.1.2 Aufbau

Mikrocontroller sind kleine Computersysteme, also Recheneinheit, Speicher und Peripherie, die auf einem Chipbaustein vereint sind. Im Unterschied dazu benötigen Mikroprozessorsysteme (μP-Systeme) neben der CPU zusätzliche Controller für Speicher und Grafikeinheit (Northbridge) und weitere Peripherie (Southbridge).
Der Übergang ist dabei fließend und verschwimmt weiter: einerseits werden auf Mikroprozessorsystemen mehr und mehr Komponenten in der CPU integriert, andererseits steigt die Lei-

[1] `http://www.wikipedia.org/wiki/embedded_system`

stungsfähigkeit moderner Mikrocontroller weiter an. Ein Beispiel für einen gemeinsamen Anwendungsbereich sind Netbooks, in denen heutzutage sowohl die klassische x86-Mikroprozessorarchitektur, wie auch die ARM μC-Architektur verbaut wird.
Zur Peripherie, also den Schnittstellen, über die der Controller mit seiner Außenwelt kommunizieren kann, zählen:

- serielle Busse (SPI, I2C, UART, USB, SSC, etc.)
- allgemeine Ein- und Ausgangsleitungen (GPIO)
- Interrupt - Eingänge
- analog/digital und digital/analog - Schnittstellen (ADC's, DAC's, PWM)

Auf die Schnittstellen werde ich in Kapitel 2 der Arbeit im einzelnen zu sprechen kommen.

Hinsichtlich des Speichers wird sowohl flüchtiger Arbeitsspeicher (SRAM, SDRAM), wie auch nicht-flüchtiger Programmspeicher (ROM, EPROM, EEPROM, Flash-memory) verwendet. Der Programmspeicher wird oftmals auch extern, also auf einem eigenen Chip realisiert.

Über den internen Aufbau kann man wenig allgemeine Aussagen machen, da dieser vielfältig ist und stark von der verwendeten Architektur abhängt. So gibt es Architekturen mit 4-, 8-, 16- oder auch 32-Bit Datenbusbreite. Der Prozessortakt reicht von unter 10 bis zu 500MHz.

1.1.3 Atmel AP7000

Das in dieser Arbeit verwendete Board ist der NGW100 von der norwegischen Firma Atmel. Das Board beheimatet eine AP7000 MCU, die eine Referenzimplementierung der AVR32 32-Bit Architektur darstellt. Als obere Grenze für den Takt gibt Atmel 150MHz an. Im Gegensatz zu den, ebenfalls AVR32 basierten Atmel UC3-Mikrocontrollern, verfügt der AP7000 über eine Speicherverwaltungseinheit (MemoryManagementUnit, MMU), um Betriebssysteme bei der Verwaltung von virtuellem Speicher zu unterstützen.
Es existieren die beiden Varianten AP7001 und AP7002, die einen eingeschränkten Funktionsumfang bieten, dadurch aber auch weniger Energie verbrauchen[2].

1.1.4 Programmierung

Die Programmierung von Mikrocontrollern wurde ursprünglich in Assembler durchgeführt. Der Assembler wird meistens kostenlos vom Hersteller angeboten. Einige Hersteller stellen aber auch weitere Werkzeuge, wie Compiler und Entwicklungsumgebungen bereit, um die Programme in einer maschinenunabhängigen Hochsprache wie z.B. C oder einer eigens entwickelten Sprache zu erstellen, zu testen und zu debuggen. Die kompilierten Programme werden in den Programmspeicher des μC geschrieben und beim Start direkt ausgeführt. Es gibt weitere Ansätze, wie Mikrocontroller mit eingebautem Interpreter für die populäre Programmiersprache BASIC [3].

[2] `http://atmel.com/dyn/products/param_table.asp?family_id=682\&OrderBy=part_no\&Direction=ASC`
[3] `http://de.wikipedia.org/wiki/BASIC_Stamp`

Zu den Werkzeugen, die für die AVR-Architekturen[4] von Atmel zur Verfügung stehen, gehört eine angepasste Version des GNU C-Compilers und eine spezielle C-Bibliothek, um die Peripherie der Controller unkompliziert anzusprechen zu können.
Die Nutzung dieser Bibliothek macht die Programmierung sehr komfortabel, da einerseits die Möglichkeiten einer Hochsprache zur Verfügung stehen und andererseits direkter Hardwarezugriff möglich ist.

1.1.5 Betriebssysteme

Mit steigender Leistungsfähigkeit von Mikrocontrollern (hinsichtlich des verfügbaren Speichers, Datenbusbreite, Taktrate etc.) ist es möglich, daß heute eine Reihe von Betriebssystemen (Operating Systems, OSes) für verschiedene Aufgabenbereiche und Architekturen bereit steht.
Microsoft Windows Mobile, NOKIA's SymbianOS oder PalmOS werden in mobilen Geräten, also Handys oder PDA's eingesetzt und sind dafür entwickelt worden, auf Mikrocontrollerarchitekturen zu laufen. Wichtig für entsprechende Einsatzgebiete sind die *echtzeitfähigen* Betriebsysteme wie eCos, FreeRTOS oder vxWorks.

Für die AVR32 Architektur stehen mittlerweile einige Betriebssysteme zur Verfügung:[5]

- Linux (Debian, OpenEmbedded, Buildroot, OpenWrt, ...)
- FreeRTOS
- embOS
- Micrium μC/OS-II
- ThreadX RTOS
- eHalOS

1.2 Linux

Der Betriebssystemkern Linux - im folgenden oft einfach "Kernel" oder "Linux" genannt - wurde 1991 von Linus Torvalds für i386-kompatible Systeme vorgestellt und unter der freien GPL-Lizenz verbreitet. Seitdem wurde Linux auf viele verschiedene Architekturen portiert und läuft heute sowohl auf großen Serveranlagen, wie auch auf kleinen Mikrocontrollern ohne eigene Speicherverwaltung. Heute unterstützt Linux mehr Architekturen und stellt Treiber für mehr Hardware bereit, als jedes andere Betriebssystem.
Linux gab es ursprünglich nur für den x86 kompatiblen PC, es wurde aber auf viele andere Architekturen (z.B. MIPS, ARM, AVR32) portiert. Durch eine Vielzahl von Konfigurationsoptionen läßt sich der Kernel an die Zielhardware anpassen und minimalisieren.
Die Verwendung eines freien Betriebssystems wie Linux eröffnet neue Möglichkeiten, da dadurch eine riesige Menge von Werkzeugen aus der OpenSource - Welt zur Verfügung steht.

Entwicklungszyklus Der Kernel unterliegt einer ständigen Weiterentwicklung. Neue Releases folgen im Abstand von 70 bis 100 Tagen. Die, zur Zeit des Schreibens dieser Arbeit aktuellste

[4] AVR8 und AVR32
[5] Quelle: `http://en.wikipedia.org/wiki/AVR32`

stabile Version ist 2.6.31 während 2.6.32 mit dem fünften “Release Candidate” in den Startlöchern steht. Die zu Beginn des Projektes aktuellste stabile Version war 2.6.30.7 und wird daher im Projekt verwendet.

AVR32 Portierung Der Kernel wurde von Haavard Skinnemoen für die AVR32-Architektur mit den grundlegenden Treibern portiert. Die Anpassungen waren erstmals im 2.6.19 Release enthalten. Daraufhin folgten weitere Treiber für die unterstützte Peripherie. Eine Übersicht der AVR32-Patches ist auf der avr32linux-Website[6] zu finden. In die, im Projekt verwendete Kernelversion 2.6.30.7 sind alle verfügbaren Anpassungen eingeflossen. Eine Untersuchung der Hardwareunterstützung für den AP7000 und seiner Peripherieeinheiten erfolgt in Kapitel 3.

1.2.1 Echtzeitfähigkeit

Man unterscheidet zwischen harten und weichen Echtzeitanforderungen hinsichtlich der Folgen, die das “nicht-einhalten” der geforderten Fristen für die Anwendung hat:

- harte Echtzeitanforderungen liegen vor, wenn eine Verspätung einen Katastrophenfall, also z.B. den Ausfall des Systems oder gar Sach- und Personenschäden zur Folge hat.
- weiche Echtzeitanforderungen liegen vor, wenn eine Verspätung für die Anwendung ohne große Schäden verkraftbar ist. (z.B. Multimediastreaming)

Da der Standard-Linuxkernel nur beschränkt echtzeitfähig ist, sind die Haupteinsatzbereiche momentan eher im Kommunikations- und Multimediabereich zu finden, wo höchstens weiche Echtzeitanforderungen bestehen.
Es gibt mehrere Ansätze, die Linux zu einem Echtzeitbetriebsystem machen, das harten Echtzeitanforderungen genügt. Die entsprechenden Projekte Xenomai [7], RTAI [8] und RTLinux [9] sind allerdings noch nicht für die AVR32-Architektur portiert worden.

Real-Time Linux Projekt

Das Real-Time Linux Projekt von Ingo Molnar und Thomas Gleixner versucht die Echtzeitfähigkeit des Standardkernels zu verbessern. Die bereitgestellten Patches enthalten Änderungen an vielen Stellen des Kernels, allerdings ist bisher nur ein Teil davon in den Hauptzweig übernommen worden. Eine der wichtigsten Änderungen ist die **verbesserte Unterbrechbarkeit** (*Preemptibilität*) des Kernels. Auf der FAQ-Seite des Projekts steht dazu:

> Unter dem Standardkernel ist die Unterbrechung von Kernelaktivität durch einen Prozeß normalerweise nur dann erlaubt, wenn der Kernel auf die Freigabe eines Mutex[10] wartet oder ausdrücklich die Kontrolle an einen Prozeß abgegeben wird. Dieses Verhalten kann zu hohen Latenzzeiten von mehreren hundert Millisekunden führen. [16]

[6] `http://www.avr32linux.org`
[7] `http://www.xenomai.org`
[8] `https://www.rtai.org`
[9] RTLinux wurde 2007 von der Firma Windriver übernommen und wird Dual-Licensed angeboten - siehe `http://www.rtlinuxfree.com/license.html`
[10] `http://de.wikipedia.org/wiki/Mutex`

Der RT-Preempt-Patch führt dahingehend Änderungen durch, daß die Unterbrechung von Kernelaktivität an mehr Stellen möglich wird.
Eine weitere Verbesserung des RT-Projekts, die in den Hauptzweig des Kernels aufgenommen worden ist, sind **hochauflösende Timer** (hrtimer), die die Programmierung mit einer zeitlichen Auflösung im Nanosekundenbereich, unabhängig von Kernel-Ticks, den sogenannten *jiffies* ermöglichen.
Die verbesserte Echtzeitfunktionalität war anfangs nur für die Linux - "Hauptarchitekturen" x86 und amd64 verfügbar, steht - die erwähnten Punkte betreffend - inzwischen aber auch für die AVR32-Architektur zur Verfügung.

1.3 Programmierung linuxbasierter Mikrocontroller

Die Mikrocontrollerprogrammierung unter Linux spaltet sich in zwei Teile: Treiberprogrammierung und Anwendungsprogrammierung. Der Anwendungsprogrammierer muß die Schnittstellen zur Hardware benutzen, die ihm vom Treiber angeboten werden.

Anmerkung: Die Dokumentation des Kernels ist im Verzeichnis `Documentation/` des Kernelquellcodes enthalten. Referenzen auf die Kerneldokumentation sind dadurch erkennbar, daß sie mit diesem Verzeichnisnamen anfangen.

1.3.1 Kernel/Userspace Schnittstellen

Linux ist ein Betriebssystem, das den Speicher in Kernelspace und Userspace trennt. Die Aufgabe des Kernels ist die Verwaltung der Hardware und die Bereitstellung dieser Ressourcen für Anwendungsprogramme, die im Userspace laufen.
Da Anwendungsprogramme keine Möglichkeit haben, direkt auf den Speicherbereich des Kernels zugreifen, stellt der Kernel zur Interaktion Systemaufrufe (system calls) zur Verfügung. Die Systemaufrufe werden in den meisten Fällen auf Knoten im Dateisystem angewandt (open, read, write, release, ioctl, ...), arbeiten aber auch mit anderen Mechanismen, wie z.B. POSIX Signalen zur Inter-Prozeß-Kommunikation (IPC).

UNIX Gerätedateien

UNIX Gerätedateien (*device files*) sind spezielle Dateien, die zur Kommunikation zwischen Anwendungsprogrammen und Hardware dienen. Sie liegen typischerweise im `/dev` - Verzeichnis.
Hinter jeder Gerätedatei steht ein Treiber, der jeweils für Knoten einer bestimmten Klasse[11] zuständig ist. Die Struktur `struct file_operations` wird vom Treiber mit Referenzen auf Funktionen versehen, die bei bestimmten Dateioperationen ausgeführt werden. Listing 1.1 zeigt die Verknüpfung der Systemaufrufe `ioctl`, `open` und `release` mit entsprechenden Treiberfunktionen.

In der ioctl - Methode werden meistens Funktionen implementiert, die zum Beispiel der Konfiguration des Gerätes dienen. Innerhalb dieser Funktion werden für die Übergabe von Daten zwischen

[11] Die Knoten einer Klasse haben eine gemeinsame *major number*. Die *minor number* wird inkrementell zugeteilt.

User- und Kernelspace die Funktionen `get_user` und `put_user` und ihre Varianten verwendet. Auf diese Weise können beliebige Daten zwischen Anwendung und Treiber ausgetauscht werden, wenn die Definition der Struktur der Daten beiden Seiten bekannt ist.

Listing 1.1: Beispiel für die Struktur file_operations (softpwm.c)

```
struct file_operations fops = {
  .owner = THIS_MODULE,
  .ioctl = device_ioctl,
  .open = device_open,
  .release = device_release
};
```

sysfs

Das *process filesystem* (procfs) auf UNIX-Systemen ist dazu gedacht, dem Userspace Informationen über laufende Prozesse im Verzeichnis `/proc` bereitzustellen. Das Programm *ps* bezieht beispielsweise seine Daten von dort. Da dieser Bereich allerdings auch die Möglichkeit mit sich bringt, beliebige Daten zwischen Kernel- und Userspace auszutauschen, wurde er desöfteren dafür "mißbraucht". So fanden sich dort immer neue Einträge, um etwa allgemeine Kernelparameter oder Treiber- und Geräteoptionen zu setzen und abzufragen.
Mit dem Kernel 2.5 sollte zur Reorganisation der bis dato uneinheitlichen Treiber ein neues *Driver Model* [12] eingeführt werden. Um dieses zu debuggen, wurde anfangs wiederum das `/proc` - Verzeichnis benutzt, und dort ein Verzeichnisbaum mit Geräteinformationen eingehängt. Daraufhin wurde auf Drängen von Linus Torvalds das *sys filesystem* (*sysfs*), ursprünglich *device driver filesystem* (ddfs) eingeführt. Wie auch *procfs* liegt *sysfs* im Arbeitsspeicher. Es wird in das Verzeichnis `/sys` eingebunden und repräsentiert den Gerätebaum polyhierarchisch.

```
$ tree -L 1 /sys/
/sys/
|-- block
|-- bus
|-- class
|-- dev
|-- devices
|-- firmware
|-- fs
|-- kernel
|-- module
`-- power
```

Vorraussetzung für die Einbindung in *sysfs* ist, daß die Implementierung des Hardwaretreibers dem gerade angesprochenen *Driver Model* folgt. Für jedes erkannte Gerät wird ein Verzeichnis in `/sys/devices` angelegt. In den übrigen Unterverzeichnissen werden symbolische Links erzeugt, die das Auffinden der Geräte über verschiedene Hierarchien möglich machen. Sowohl für den ganzen Treiber, wie auch für einzelne Geräte können Attribute definiert werden, die durch

[12] Documentation/driver-model/overview.txt

Dateien dargestellt werden. Jede dieser Dateien repräsentiert dabei genau einen Wert der gelesen und/oder geschrieben werden kann. Der Treiberentwickler implementiert die entsprechenden LOAD- und STORE-Routinen im Gerätetreiber. Es ist auch möglich, Attribute zu Gruppen zusammenzufassen, die im Dateisystem als Unterverzeichnisse auftauchen.

1.3.2 uclibc

Die C-Bibliothek ist elementarer Bestandteil eines UNIX Betriebssystems. Die meisten Linuxdistributionen greifen auf die GNU C-Bibliothek *glibc* zurück. Diese hat allerdings einen - für die Verhältnisse von Mikrocontrollern - enormen Platzbedarf. So wird hier häufig die minimalistische *uclibc* [13] verwendet. Sie bietet einen ähnlichen Funktionumfang wie die Standardbibliothek, stellt allerdings weniger Anforderungen hinsichtlich des Speicherverbrauchs, und unterstützt auch Platformen ohne MMU.

1.4 weiterer Aufbau der Arbeit

In Kapitel 2 folgt eine Übersicht über die Peripherieschnittstellen des AP7000. Es wird auf diejenigen Schnittstellen genauer eingegangen, für die Pythonmodule erstellt oder getestet wurden.
In Kapitel 3 wird die Unterstützung des AP7000 und seiner verwendeten Funktionseinheiten durch den Linuxkernel untersucht, wobei besonderes Augenmerk auf die bereitgestellten Schnittstellen zum Userspace gerichtet ist.
Kapitel 4 dokumentiert die Erstellung des SoftwarePWM-Kernelmoduls, das mithilfe eines high resolution timers auf mehreren GPIO's PWM-Signale ausgibt.
Kapitel 5 enthält die API-Dokumentation der erstellten und verwendeten Python-Module.
In Kapitel 6 wird auf die verwendete Linuxdistribution *OpenWrt*, die erstellten Pakete und Kernelpatches eingegangen.
Im Anhang finden sich eine Beschreibung, wie der MCP2515 CAN-Controller an den NGW100 angeschlossen wurde, sowie Quelltexte und Tabellen zur Pinbelegung.

[13] http://www.uclibc.org

2 I/O-Schnittstellen des AP7000

Zur Kommunikation mit anderen elektronischen Geräten hat ein Mikrocontroller in der Regel verschiedene Schnittstellen zur Verfügung. Über diese kann er zum Beispiel mit gleichberechtigten Mikrocontrollern (master-to-master, peer-to-peer) oder mit angeschlossenen Peripheriegeräten (slaves, clients) Daten austauschen, Schaltungen ansteuern oder Sensoren überwachen.

2.1 Pinout des AP7000

Der Atmel AP7000 Mikrocontroller enthält verschiedene Hardwareeinheiten (Peripherieeinheiten, Funktionseinheiten) zur Ein- und Ausgabe von Daten über bestimmte Signale. Das Gehäuse des AP7000 bietet mit 256 Stück allerdings zu wenige Pins, um alle Signalleitungen - inklusive Spannungsversorgung, Oszillatoranschlüssen, Debug- und Testpins, etc. - auf einmal nach außen zu führen. Daher werden jeweils nur die aktivierten Signalleitungen auf die Pins gelegt (*Multiplexing*). Eine Auflistung der Peripherieeinheiten des AP7000 findet sich in Kapitel 9.7 des zugehörigen vorläufigen Datenblatts [2].
Das Multiplexing ist Aufgabe des PIO-Controllers (Parallel Input Output). Er kann jeder Leitung entweder eine von zwei Peripheriefunktionen zuweisen, oder die Leitung als GPIO zur Verfügung stellen.

Anmerkung: Die elektrischen Pegel aller I/O-Leitungen sind 0V (LOW) und 3.3V (HIGH). Der AP7000 ist nicht 5V-tolerant, das heißt, wenn 5V auf einen Eingangspin gelegt werden, kann der AP7000 kaputt gehen.

2.2 Atmel NGW100 Evaluation Board

Das Atmel NGW100 Evaluation Board bietet eine Referenzimplementierung des AP7000 Mikrocontrollers. Es ist ein Entwicklungsboard, das dem Benutzer die Möglichkeit bietet, die Fähigkeiten des AP7000 zu testen und Prototypen für eigene Entwicklungen zu erstellen. Dabei führt das Board eine Auswahl der Pins des AP7000 über drei *Expansion Headers* nach außen. Die *Expansion Headers*, bezeichnet als **J5**, **J6** und **J7**, bestehen aus jeweils 36 doppelreihig angeordneten Pins. Die Verteilung der Jumper auf dem Board und die im Projekt verwendete Belegung der Pins findet sich in Anhang B.1.

2.3 Schnittstellen - Überblick

Es folgt eine Übersicht über serielle, parallele und analoge Schnittstellen und die entsprechenden Funktionseinheiten des AP7000.

2.3.1 Serielle Schnittstellen

Serielle Schnittstellen zeichnen sich dadurch aus, daß ein Bit nach dem anderen übertragen wird. Durch die serielle Übertragung kann die Anzahl benutzter Leitungen gering gehalten werden. Zusätzlich zu den Datenbits werden - je nach Typ der Schnittstelle - Start- und Stopbits, Bits zur Fehlererkennung, etc. übertragen. Eine Übertragungseinheit aus Daten und Kontrollbits wird auch als **frame** bezeichnet. Die seriellen Schnittstellen des AP7000 sind:

- Universal Serial Bus (USB)
- Synchronous Peripheral Interface (SPI)
- Two Wire Interface (TWI)
- Universal Synchronous Asynchronous Receiver Transmitter (USART)
- Synchronous Serial Controller (SSC)
- PS/2 Interface (PSIF)

2.3.2 Parallele Schnittstellen

Parallele Schnittstellen übertragen mehrere Bits parallel, wodurch sie eine hohe Übertragungsgeschwindigkeit erzielen. Jedoch steigt mit der Breite der Parallelität auch die Anzahl der benutzten Leitungen und macht eine entsprechend aufwändigere Hardwareimplementierung erforderlich.

Parallel arbeitende Einheiten des AP7000 sind:

- PIO Controller (PIO) - Der AP7000 bietet 5 PIO (Parallel Input Output) Ports mit jeweils 32 Leitungen
- LCD Controller (LCDC) - Der LCD Controller bietet - je nach Displaytyp - parallele Übertragung verschiedener Breite an (4, 8 oder 24 Bit)
- External Bus Interface (EBI) - Der AP7000 kann durch das EBI Daten mit Flashspeicherbausteinen parallel austauschen
- Image Sensor Interface (ISI) - Das ISI benutzt 12 Datenleitungen parallel

2.3.3 Analogschnittstellen

Analogschnittstellen lassen sich in Digital-zu-Analog-Schnittstellen (DAC), also vom μC aus gesehen als Ausgabe, und Analog-zu-Digital-Schnittstellen (ADC), also vom μC aus gesehen als Eingabe einteilen. Über diese Schnittstellen ist die Kommunikation mit analog arbeitenden Geräten, wie Motoren oder Lautsprechern möglich. In diese Klasse fallen folgende Einheiten des AP7000:

- PWM / Waveform Generator
- AC97 Controller
- Audio Bitstream DAC

2.4 UART

UART (Universal Asynchronous Receiver Transmitter) ist der Name des Bausteins, der eine Verbindung über eine RS-232/V.24 kompatible Schnittstelle[1] ermöglicht. Über diese Schnittstelle wurden für gewöhnlich Modems und Eingabegeräte an Computer angeschlossen. Zwar ist die klassische serielle Schnittstelle die älteste ihrer Art, jedoch wird sie für sogenannte Nullmodem - Verbindungen zwischen zwei PC's oder auch PC's und Mikrocontrollern heute noch häufig verwendet. Da selten alle, vom RS-232-Standard definierten 25 Anschlußpins benötigt werden, hat sich vor allem die platzsparende 9-polige Version (DB9) verbreitet.
In der Praxis werden bei UART oft nur die Sende- und Empfangsleitung (TX/RX) bei 3.3V (also ohne Pegelwandlung) verwendet, um mit Peripherie zu sprechen.

Parameter Um die Kommunikation zwischen zwei Endgeräten zu realisieren, muß sich auf eine Kombination aus folgenden Parametern geeinigt werden:

- **Baudrate**: Die Übertragungsgeschwindigkeit in Bit pro Sekunde, mit der Daten versendet und empfangen werden
- **Data bits**: Anzahl der Datenbits pro Übertragungseinheit
- **Parität**: Übertragung eines Paritätsbits für die gesendeten/empfangenen Datenbits; mögliche Einstellungen sind: gerade Parität, ungerade Parität oder gar keine Paritätsbitübertragung
- **hardware - handshake/flow control**: Für Hardware-Flußkontrolle müssen die Leitungen "Clear To Send" (CTS) und "Request To Send" (RTS) müssen verbunden sein. Soll ein Zeichen übertragen werden, aktiviert der Sender die RTS Leitung. Nach Abschluß der Übertragung aktiviert der Empfänger die CTS Leitung.
- **software - handshake/flow control**: Die Software-Flußkontrolle funktioniert über das Senden reservierter Zeichen von Empfänger zu Sender: Möglich sind die Signalisierung von "buffer full" (der Sender stoppt die Übertragung) oder "continue" (Der Sender nimmt die Übertragung wieder auf)
- **Stop Bits**: Anzahl der Bits, die der Übertragung eines Zeichens folgen (1, 1.5 oder 2)

USART Viele moderne Mikrocontroller haben UART's, die auch *synchrone Kommunikation* unterstützen. Diese Bausteine nennt man "Universal Synchronous Asynchronous Receiver Transmitter", also kurz "**USART**". Die Synchronisation erfolgt über eine separate Taktleitung, die `CLK` oder `SCL` genannt wird.

Der Atmel AP7000 bietet 4 USART Einheiten mit Unterstützung für verschiedene Übertragungsarten: RS232, RS485, MODEM und ISO7816.

[1] `http://de.wikipedia.org/wiki/EIA-232`

2.5 GPIO

Ein **GPIO** (General Purpose Input/Output) ist ein flexibles, softwaregesteuertes digitales Signal. Jeder GPIO repräsentiert ein Bit, das mit einer Signalleitung verbunden ist. Dadurch ermöglichen sie die digitale Ein- und Ausgabe von Signalen ohne Bindung an ein spezielles Protokoll. Fast alle Mikrocontroller bieten zumindest ein paar wenige bis zu mehreren hundert davon an. Die Fähigkeiten von GPIO's variieren von Controller zu Controller.
Der im folgenden verwendete Begriff "GPIO-Pin" impliziert, daß für jeden GPIO Ein-/Ausgang ein Pin auf das Board gelötet ist. Jedoch sind einige GPIO's zum Beispiel mit den LED's auf dem Board verbunden und haben also keinen Pin im eigentlichen Sinne.

Über GPIO Pins können Schaltungen angesprochen werden (Ausgabe, Output) oder logische Signale von diesen empfangen werden (Eingabe, Input).
Als Eingang konfiguriert können GPIO's auch als level- oder flankenabhängig ausgelöste Interruptquelle dienen. Falls diese Funktionalität vom GPIO Controller nicht unterstützt wird, können mithilfe von *Polling* Aktionen bei bestimmten Bedingungen ausgeführt werden. *Polling* bedeutet in diesem Fall das periodische Abfragen des Wertes, der am Eingang anliegt. Einige Anwendungsbeispiele für GPIO's sind:

- Input
  - Sensoren (Lichtschranke, Temperatur)
  - Keypads
- Output
  - Relais
  - LED's
  - Displays

Weiterhin können komplette Protokolle für serielle oder parallele Bustypen mithilfe von GPIO's per Software realisiert werden, wenn keine entsprechende Hardwareeinheit zur Verfügung steht. Bei dieser Technik spricht man von "*Bitbanging*".

Auf der AP7000 Plattform wird der GPIO-Zugriff durch den PIO-Controller mit folgenden Features für jeden GPIO bereitgestellt:

- Lesen des Eingabewerts und Kontrolle des Ausgabewerts
- Bei Signalpegelwechsel (≙edge triggered) ausgelöste Interrupts
- An-/Abschalten des *glitch-filter*: Pulse, die kürzer als ein halber Clockcycle sind werden ignoriert
- An-/Abschalten des *pull-up*-Widerstands[2]

[2]Für die pull-up-Funktionalität wird ein Widerstand zwischen Versorgungsspannung und Pin-Signal geschalten, der die Leitung auf HIGH-Level hält. Wie auch der *glitch-filter* macht dies nur für Pins Sinn, die als Eingang konfiguriert wurden.

2.6 I2C / TWI

Der I2C-Standard (Inter-Integrated Circuit, IIC, I^2C) wurde von Phillips Semiconductors - inzwischen NXP - zur Vernetzung von Funktionseinheiten in TV-Geräten entwickelt. Weitere Hersteller von Elektrogeräten, wie z.B. Siemens oder Motorola haben kompatible Geräte auf den Markt gebracht. Obwohl I^2C kein eingetragenes Warenzeichen ist, verwenden andere Hersteller manchmal andere Bezeichnungen für ihre Implementierung. Atmel verwendet zum Beispiel das Kürzel **TWI** für "*Two Wire Interface*".

Hardware Für I2C werden nur zwei Leitungen benötigt:

- SDA half-duplex Datenleitung
- SCL Clock/Taktleitung

Die Leitungen für SDA und SCL sind open-drain, also über Widerstände mit der Versorgungsspannung V_{CC} verbunden (pull-up Widerstände). Dadurch kann die wired-AND Funktionalität[3] zur Verfügung gestellt werden. Das bedeutet, daß ein niedriger Signalpegel eine höhere Priorität als ein hoher Signalpegel hat. Dieses Verhalten wird für verschiedene Funktionalitäten des I2C-Busses, wie das *Clock-Stretching* und die *Arbitrierung* genutzt. Ein einfacher Aufbau mit einem I2C-Master und zwei Slaves ist in Abbildung 2.1 dargestellt.

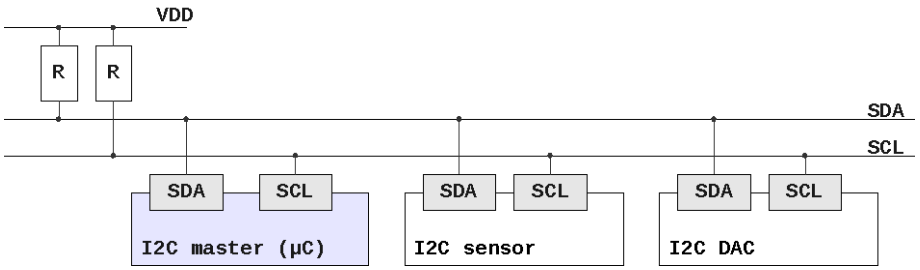

Abbildung 2.1: I2C Bus Layout

Die elektrische Gesamtkapazität des Busses darf 400pF nicht überschreiten.

Geschwindigkeit Der Takt, der durch den Master angegeben wird kann beliebig langsam sein, jedoch sind folgende Geschwindigkeiten in der Spezifikation definiert:

- low-speed mode 10 kbit/s
- standard mode 100 kbit/s
- fast mode 400 kbit/s
- fast mode plus 1 Mbit/s
- high speed mode 3,4 Mbit/s

[3] `http://de.wikipedia.org/wiki/Wired-AND`

Kommunikation Der I2C-Standard [15] legt die Eigenschaften für die Kommunikation über das I2C-Protokoll fest. Die wichtigsten davon sind:

- half-duplex (es gibt nur eine Datenleitung)
- Startbit (Startbedingung) zur Einleitung einer Übertragung
- Stopbit (Stopbedingung) zum Abschluß einer Übertragung
- Byteweise Übertragung, MSB first
- Bestätigung einer Übertragung bzw. Anforderung weiterer Daten über ACK-/NACK-Bit

Ein Mastergerät leitet die Kommunikation mit einem Slave ein, indem es über die Datenleitung ein Startbit **S** sendet und ein Taktsignal auf die Taktleitung legt. Hierrauf werden im Takt die Datenbits übertragen, wobei ein Bit als *stabil* gilt, wenn die Clockleitung auf HIGH steht.
Die Übertragung erfolgt byteweise - das höchstwertige Bit zuerst - vom Master zum Slave oder auch andersherum. Das erste Byte einer Übertragung enthält in der Regel die Adresse eines Slaves (siehe Addressierung). Die Empfängerseite antwortet auf ein empfangenes Byte, indem als neuntes Bit entweder Acknowledge (LOW) oder NotAcknowledge (HIGH) auf die Datenleitung gelegt wird. Eine Übertragung wird durch ein Stopbit **P** abgeschlossen. Startbit, Stopbit und die Übertragung von Datenbits sind in Abbildung 2.2 dargestellt.

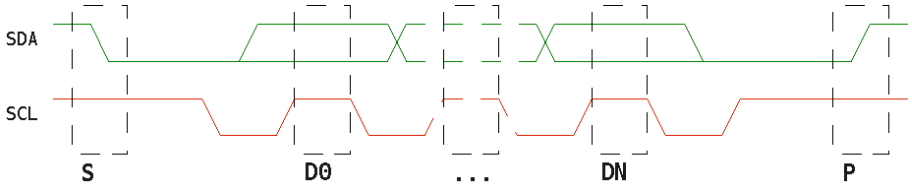

Abbildung 2.2: I2C Bitübertragung der Bits 0 bis N; **S**: Startbit; **P**: Stopbit

Benötigt ein Client für eine Antwort längere Zeit, kann er den Master darüber benachrichtigen, indem er die Clock-Leitung auf LOW zieht. (Clock Stretching bzw. Clock Synchronisation)

Verschiedene Geräte können eine Übertragung einleiten und die Rolle des Masters übernehmen ("Multi Master"). Die Auswahl findet durch Überwachung der Datenleitung statt. Entdeckt ein Master einen LOW Pegel auf der Datenleitung, den er nicht selbst erzeugt hat, geht er von einer Belegung des Busses aus und wartet auf das nächste Stopbit. Starten also zwei Master ihre Aktivität gleichzeitig, so gewinnt der, der als erstes eine `'0'` sendet. (Arbitrierung)

Addressierung Jeder Slave benötigt auf dem Bus eine eindeutige Adresse, die aus 7 Bit besteht. Das begrenzt die Anzahl möglicher Slaves auf 112, da 16 Adressen reserviert sind. Der Master kann also durch Senden eines Bytes ein Gerät auf dem Bus auswählen. Auf die 7 Adressbits des gewünschten Slaves folgt das $R/\overline{W}$ - Bit. Dieses signalisiert dem Slave, ob eine Lese- ($R/\overline{W} = 1$) oder Schreiboperation ($R/\overline{W} = 0$) folgt.
Eine neuere Version des I2C-Standards bietet auch die Möglichkeit 10Bit - Adressen zu verwenden, was allerdings nicht alle Geräte unterstützen. Hierfür wird im ersten Byte das Bitmuster 11110

gesendet, gefolgt von den beiden höchstwertigen Adressbits und dem $R/\overline{W}$ Bit. In einem zweiten Byte folgen die übrigen 8 Adressbits.

2.6.1 SMBus

Der SMBus (System Management Bus) wurde 1995 von Intel zur Kommunikation von z.B. Spannungs- und Temperatursensoren auf einem Motherboard entwickelt.
Der SMBus-Standard [17] ist eine Erweiterung des I2C-Protokolls und setzt strengere Definitionen bei den elektronischen Eigenschaften, den Timings und bei der Adressierung der Slaves an. SMBus definiert eine zusätzliche Busleitung "**SMBALERT#**", über die Slaves den Master über Ereignisse benachrichtigen können. Die Raute am Ende der Bezeichnung bedeutet, daß diese Leitung im Ruhezustand auf HIGH steht. Weiterhin sind verschiedene Lese- und Schreibkommandos standardisiert worden.
Die Application Note 476 [12] des Chipherstellers MAXIM dokumentiert die Unterschiede zwischen I2C und SMBus.

2.7 SPI

SPI (Synchronous Peripheral Interface) ist ein von Motorola entwickeltes Protokoll zur Kommunikation zwischen einem SPI-Master und einem oder mehreren Slaves. Hierbei wurden von Motorola allerdings nur die hardwareseitigen und elektrischen Eigenschaften bestimmt. Über die Form der Daten (z.B. Paketaufbau) wurde nichts festgelegt.

SPI ist:

- ein einfaches Protokoll,
- schnell (>10MHz),
- hardwareseitig einfach zu implementieren,
- vollduplexfähig,
- für kurze Leitungslänge (wenige Zentimeter) ausgelegt.

SPI verwendet 4 Leitungen:

- serial clock (SCLK): Taktübertragung
- master in slave out (MISO, SO: serial out, SDO: serial data out): Eingangsleitung für den Master, Ausgangsleitung für die Slaves
- master out slave in (MOSI, SI: serial in, SDI: serial data in): Ausgangsleitung für den Master, Eingangsleitung für den Slave
- chip select (CS, SS: slave select): Leitung zur Aktivierung eines Slaves
  Das Signal der Chipselect Leitung ist gewöhnlich im Ruhezustand auf HIGH und wird zum Aktivieren eines Slaves auf LOW gezogen. Daher wird es oft mit $\overline{CS}$ oder $\overline{SS}$ bezeichnet.

2.7.1 Verdrahtungsmöglichkeiten

Single Slave Aufbau

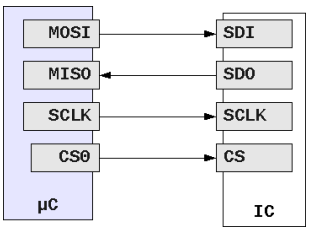

Abbildung 2.3: SPI single slave setup

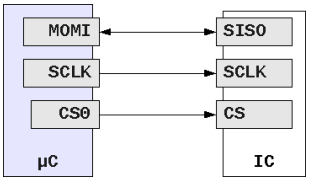

Abbildung 2.4: SPI 3wire setup

Beim Single Slave Betrieb ist nur ein einziges SPI-Gerät an den Master angeschlossen (siehe Abbildung 2.3). Je nach Art des Chips kann eventuell auf eine Leitung verzichtet werden. Gibt ein

Chip keine Daten zurück, ist also nur schreibbar, werden nur **MOSI/SDI**, SCK und CS benötigt. Andere Chips, z.B. einige Sensoren geben nur Daten zurück, benötigen also nur **MISO/SDO**, SCK und CS.
In einigen Fällen wird auch auf die CS-Leitung verzichtet und der entsprechende Eingang des Chips mit Masse oder, im Falle von "CS active high", mit der Versorgungsspannung verbunden.
Es gibt eine weitere Variante "3wire", die auch mit drei Leitungen auskommt. (siehe Abbildung 2.4) Hier können Daten in beiden Richtungen, jedoch nur half-duplex übertragen werden. Ein Pin teilt sich dafür die Rollen von MOSI und MISO und wird daher MOMI bzw. SISO genannt.

Multi Slave Aufbau

Die MOSI-. MISO- und Clocksignale werden direkt auf die jeweiligen Anschlüsse aller SPI-Geräte gelegt. Der Master hat für jedes Gerät eine eigene Chipselect Leitung, über die er den Chip auswählen kann, mit dem er kommunizieren möchte. In Abbildung 2.5 ist ein Aufbau dargestellt, bei dem 3 SPI Slaves an einem Mastergerät angeschlossen sind.

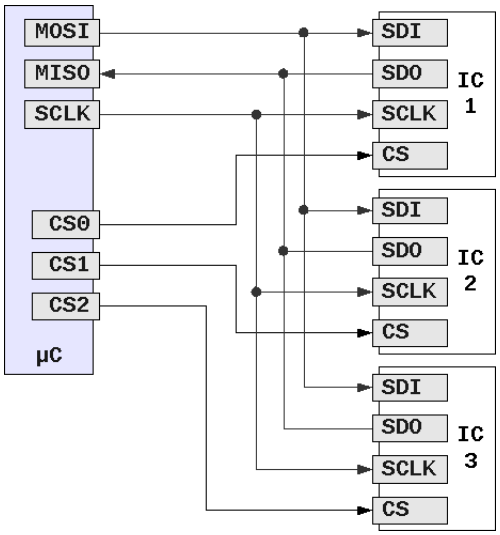

Abbildung 2.5: SPI multi slave setup

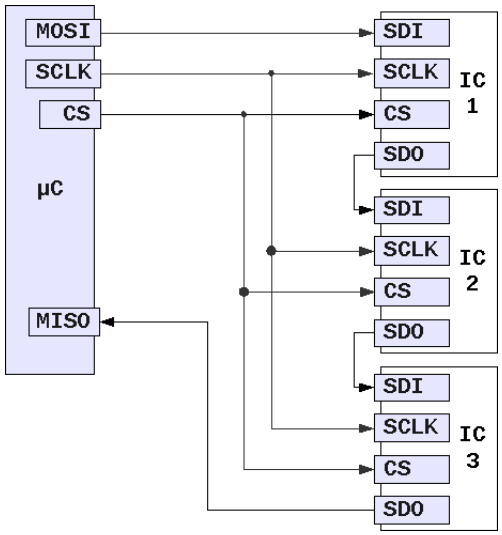

Abbildung 2.6: SPI multi slave in daisy chain setup

Daisy Chain Aufbau

Bestimmte Chips unterstützen das Daisychain Prinzip. Dafür muss ein Chip die Daten, die er über SDI empfängt im nächsten Takt auf SDO ausgeben, solange seine CS-Leitung aktiv ist. Weiterhin muß der Chip die im Eingangsregister liegenden Kommandos ausführen, sobald CS deaktiviert wird. Nur der erste Chip erhält die Befehle direkt von Master. Alle weiteren sind in einer Kette so angeordnet, daß die SDO-Leitung des Vorgängers mit der SDI-Leitung des Nachfolgers verbunden ist.
Der Master aktiviert die CS-Leitung und sendet solange Daten, bis die SDI Register aller Chips gefüllt sind. Bei der Deaktivierung führen diese dann gleichzeitig ihr Kommando aus.

2.7.2 Datenübertragung

Eine Übertragung von Daten wird immer vom Master eingeleitet. Dazu aktiviert er über die CS-Leitung den gewünschten Slave, legt den Takt auf SCLK und schiebt dazu passend die zu sendenden Bits auf MOSI.

SPI - modes

Ein Slave übernimmt ein Bit von seiner Eingangsleitung immer zu einer Taktflanke. Obwohl von Motorola nicht festgelegt wurde, zu welchem Zeitpunkt des Clocksignals ein Bit auf der Leitung als stabil zu gelten hat, haben sich inzwischen 4 Quasi-Standards etabliert: die SPI modes 0 bis 3. Ein Modus ist durch zwei Parameter festgelegt:

- clock polarity (CPOL): Ruhezustand der Clock/Taktleitung (LOW=0, HIGH=1)
- clock phase (CPHA): Bitübernahme (*sample*) zu Beginn (CPHA=0) oder am Ende (CPHA=1) eines Taktes
  Nachschieben des nächsten Bits zur jeweils anderen Taktflanke (*shift*, instabiler Zustand)

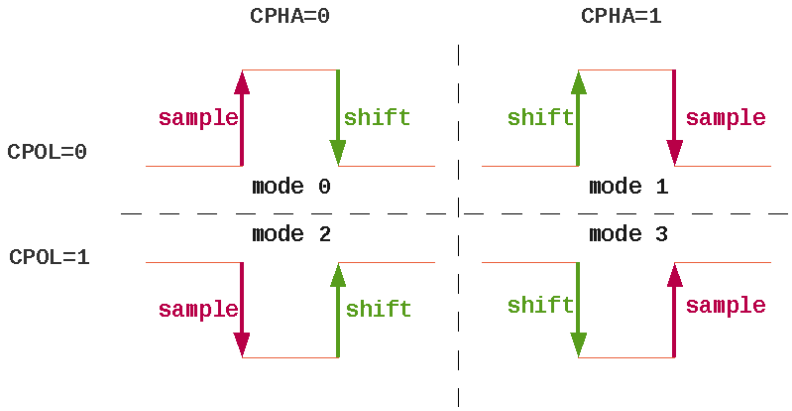

Abbildung 2.7: 4 SPI Modi: verschiedene Kombinationen der Parameter CPOL und CPHA

Protokollkonfiguration

- LSB first: Die meisten SPI-Geräte senden und erwarten normalerweise das höchstwertige Bit (most significant bit, MSB) zuerst. Einige SPI Geräte unterstützen bzw. benötigen es genau andersherum, senden und erwarten also das niederwertigste Bit (least significant bit, LSB) zuerst.
- CS_HIGH: Dieser Parameter bestimmt die Polarität des Slave-/Chip-Select Signals. Bei einigen Geräten muß das CS-Signal im inaktiven Zustand auf HIGH stehen und zur Aktivierung auf LOW gezogen werden (CS active low). Andere SPI-Geräte verlangen ein invertiertes Verhalten, also im Ruhezustand LOW und aktiviert HIGH (CS active high).

- 3wire: Die Ein- und Ausgangsdatenleitungen, also MISO und MOSI können verbunden werden, wodurch sich SPI auch mit nur 3 Leitungen realisieren läßt. Allerdings unterstützt die SPI-Einheit des AP7000, bzw. dessen Linux-Treiber diese Einstellung nicht.

2.7.3 Microwire

Das, von National Semiconductor entwickelte **Microwire** ist zu SPI kompatibel, definiert jedoch (ähnlich SMBus für I^2C) striktere Regeln für die Hard- und Softwareteile des Busses. Damit können Microwiregeräte praktisch an jedem SPI Master betrieben werden.
Der originale Microwire Standard ist kompatibel zum SPI mode 1. Viele ältere SPI Geräte implementieren aber nur den SPI mode 0, der genau andersherum funktioniert. Microwire/Plus behebt dieses Problem, indem ein SKSEL Bit (shift clock select) eingeführt wird, um zwischen den SPI modes 0 und 1 umzuschalten.

2.8 PWM

Pulsbreitenmodulation (PBM), auch PulseWidthModulation (PWM) oder Pulsdauermodulation (PDM) stellt eine Möglichkeit dar, Informationen an analog arbeitende Geräte in Form einer Rechteckspannung zu übertragen. PWM bildet also eine Form des DigitalAnalogConverter (DAC). Das Prinzip ist dabei sehr einfach und bedarf nur einer Leitung, die abwechselnd auf HIGH (pulse, duty) und LOW (pause) gesetzt wird. Die Information steckt nun in der Länge bzw. im Längenverhältnis der einzelnen Phasen zueinander und kann z.B. den Sollwert der Rotationsgeschwindigkeit oder der Auslenkung eines Motors angeben. Dieses Prinzip wird häufig zur Lüftersteuerung in PC's verwendet. Eine weitere Anwendung bilden dimmbare Lichtquellen, wie z.B. LED's oder die Hintergrundbeleuchtung von Displays.

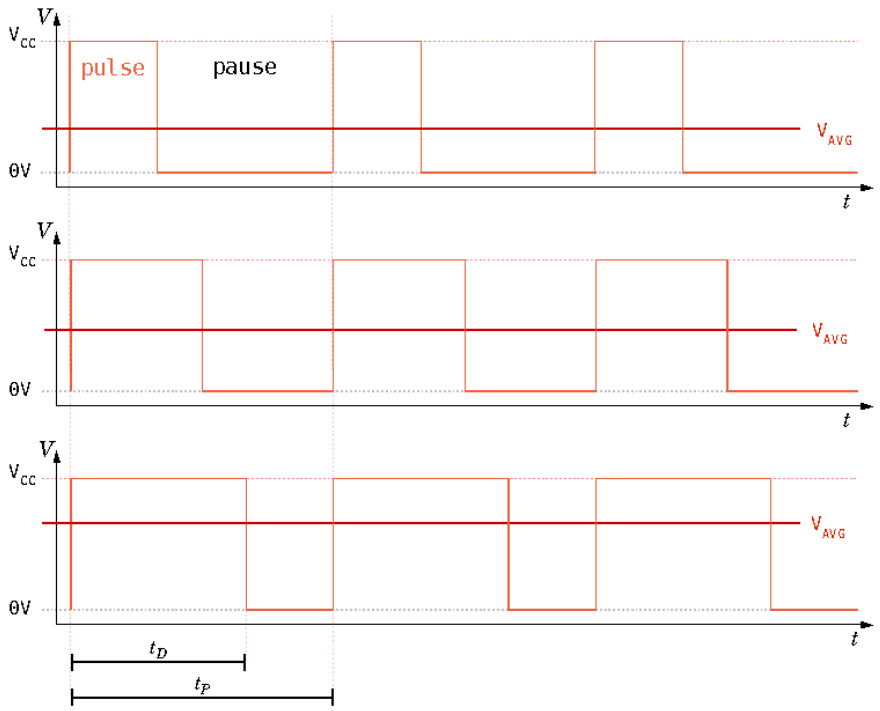

Abbildung 2.8: Pulsbreitenmodulation: V_{avg} stellt die durchschnittliche Spannung dar

Frequenz Die Frequenz des Rechtecksignals ist durch das anzusteuernde Gerät festgelegt und beträgt einige zehn Hertz bis zu mehreren hundert Kilohertz. Einige Geräte können mit verschiedenen Frequenzen arbeiten. Entscheidend für die übertragene Information ist entweder die Pulsdauer oder das Puls / Pause Verhältnis.
Servo- und Schrittmotoren werden häufig mit einer Frequenz von 50Hz angesprochen.

2.8.1 Nachbildung von Spannungskurven

PWM kann auch für die Nachbildung von Spannungkurven (Waveform), wie z.B. Sinus- oder Sägezahnspannung benutzt werden. Für nicht-lineare Verläufe ist im allgemeinen eine Glättung der Ausgabe durchzuführen.

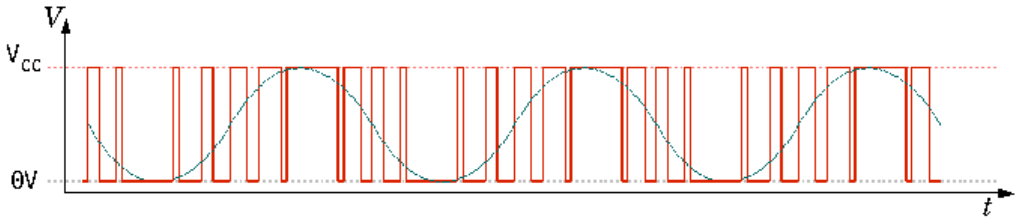

Abbildung 2.9: Pulsbreitenmodulation zur Signalerzeugung (schematische Darstellung)

2.8.2 Implementierung

Viele Mikrocontroller bieten eine PWM Implementierung an, die in einer eigenen Hardwareeinheit realisiert ist und mehrere PWM-Kanäle zur Verfügung stellt. Da die Erzeugung von PWM-Signalen eine hohe zeitliche Genauigkeit erfordert, ist die Nutzung einer Hardwareimplementierung sinnvoll und schont die Rechenkapazitäten des μC.
Die PWM-Einheit des AP700x bietet folgende Features:

- 4 Kanäle
- 20 Bit Counter pro Kanal
- gemeinsamer Clock Generator, der 13 verschiedene Clocks bereitstellt
  - Ein Modulo n Counter, der elf Clocks bereitstellt
  - Zwei unabhängige lineare Divisionseinheiten, die mit der Ausgabe der Modulo n Counter Einheiten arbeiten
- Die Kanäle lassen sich unabhängig voneinander betreiben:
  - An- und Ausschalten einzelner Kanäle
  - separate Clock Auswahl für jeden Kanal
  - unabhängige Perioden- und DutyCycle-Einstellung für jeden Kanal
  - Double Buffering der Perioden- oder DutyCycle-Einstellung für jeden Kanal
  - unabhängige Einstellung der Ausgabepolarität der Waveform für jeden Kanal
  - unabhängige Einstellung der Ausgabeausrichtung der Waveform (mittig oder links) für jeden Kanal

3 Unterstützung des AP7000 im Linux Kernel

Die meisten Teile des, größtenteils in *C* geschriebenen Linuxkernels sind architekturunabhängig und portabel. Da ein Betriebssystem aber eng mit der zugrundeliegenden Plattform zusammenarbeitet, sind unportable, architekturspezifische Teile immer enthalten. Dazu gehören natürlich alle Assembleranweisungen, aber auch *C*-Code zu Speicherverwaltung, Interruptsetup, Debugging etc. Dieser architekturspezifische Code befindet sich im Quellcodebaum jeweils im Unterverzeichnis `arch/<architekturname>`. Linux unterstützt derzeit die Architekturen alpha, arm, avr32, blackfin, cris, frv, h8300, ia64, m32r, m68k, microblaze, mips, mn10300, parisc, powerpc, s390, sh, sparc, x86 und xtensa.

3.1 Überblick

Dieser Abschnitt gibt einen Überblick über die Unterstützung der AVR32-Architektur, des AP7000-Mikrocontrollers und des NGW100-Boards im Linuxkernel.

AVR32: architekturspezifischer Code Der architekturspezifische Code für die AVR32-Architektur liegt in `arch/avr32` und enthält folgende Unterverzeichnisse:

`boards`: boardspezifischer Code für STK1000, NGW100, Hammerhead und weitere AP7000 basierte Boards

`boot`: Startupcode für den UBoot Bootloader

`configs`: Defaultkonfigurationsdateien für verschiedene Boards

`include/asm`: AVR32 spezifische Headerdateien

`kernel`: Anpassungen des Kernelkerns an die AVR32-Architektur

`lib`: Anpassungen von Kernel-Bibliotheksfunktionen an die AVR32-Architektur

`mach-at32ap`: maschinenspezifischer Code für den AT32AP [1]

`mm`: AVR32 Memory Management Unit Support

`oprofile`: Profiling Support für die AVR32-Architektur

AT32AP: maschinenspezifischer Code (AP7000) Der maschinenspezifische Code bietet Unterstützung für die Peripherie und Besonderheiten einer bestimmten Ausführung/Implementierung dieser Architektur. Der maschinenspezifische Code für den AT32AP liegt in `mach-at32ap`. Hier wird die Unterstützung für die Funktionseinheiten des AP7000 bereitgestellt:

`pio.c, pio.h`: PIO Port Multiplexer Support, GPIO API Implementierung

`clock.c, clock.h`: Clock Management Support

`cpufreq.c`: CPU Frequency Scaling Support

[1] bei der AVR32-Architektur ist `mach-at32ap` das einzige Verzeichnis mit maschinenspezifischem Code, wohingegen bei der ARM-Architektur in Linux 2.6.30.7 zum Beispiel 50 verschiedene Maschinentypen unterstützt werden.

`intc.c, intc.h`: Interrupt Controller Support

`extint.c`: External Interrupt Controller Support

`hmatrix.c`: High Speed Busmatrix Support

`hsmc.c, hsmc.h`: Static Memory Controller Support

`pdc.c`: DMA Controller Support

`pm-at32ap700x.S, pm.c, pm.h`: Power Manager Support

`sdramc.h`: SDRAM Controller Support

`at32ap700x.c`: AT32AP700x Support

- Konfiguration und Zugriffsfunktionen für die Clocks
- Konfiguration und Zugriffsfunktionen für die System Peripherie: Power Manager, Realtime Clock, Watchdog Timer, External Interrupt Controller, Interrupt Controller, External Bus Interface, SDRAM, Static Memory Controller, High Speed Bus Matrix, Timer/Counter, PIO, PSIF, USART, Ethernet, SPI, TWI, MMC, LCDC, PWM, SSC, USB Controller, IDE / CompactFlash, NAND-Flash / SmartMedia, AC97 Controller, Audio Bitstream DAC, GCLK

`include/mach`: AT32AP spezifische Headers

NGW100: boardspezifischer Code Der boardspezifische Code dient der Initialisierung der Peripherie, die auf einem bestimmten Mikrocontrollerboard vorhanden ist und berücksichtigt meist schon den gewünschten Anwendungszweck. Der boardspezifischen Code befindet sich im Unterverzeichnis `arch/<architekturname>/boards/<boardname>` des Quellcodebaums - beim NGW100 also in `arch/avr32/boards/atngw100`. Hier wird stark auf die, von architektur- und maschinenspezifischem Code bereitgestellten Strukturen und Funktionen zurückgegriffen. Es finden sich folgende Dateien:

`flash.c`: Flashspeicher Initialisierung

`setup.c`: Board Setupcode

`evklcd10x.c`: Setupcode für das Atmel EVKLCD10X Addonboard

Die Anweisungen in `setup.c` werden beim Bootvorgang abgearbeitet. Hier erfolgt die Aktivierung der Funktionseinheiten, die für den gewünschten Anwendungsfall benötigt werden.

3.2 USART

3.2.1 Konfiguration

In der Datei `arch/avr32/mach-at32ap/at32ap700x.c` werden die Funktionen zur Aktivierung der seriellen Ports `at32_map_usart` und `at32_add_device_usart` bereitgestellt. Hier erfolgt auch die Konfiguration der vier USART Schnittstellen des AP7000 mit folgenden Parametern:

- Benutzung von DMA
- I/O-Basis
- Interruptnummer
- Pinauswahl/Multiplexing

Listing 3.1: USART0 Konfiguration (at32ap700x.c)

```
static struct atmel_uart_data atmel_usart0_data = {
        .use_dma_tx     = 1,
        .use_dma_rx     = 1,
};
static struct resource atmel_usart0_resource[] = {
        PBMEM(0xffe00c00),
        IRQ(6),
};
DEFINE_DEV_DATA(atmel_usart, 0);
DEV_CLK(usart, atmel_usart0, pba, 3);
...
static inline void configure_usart0_pins(int flags)
{
        u32 pin_mask = (1 << 8) | (1 << 9); /* RXD & TXD */
        if (flags & ATMEL_USART_RTS)    pin_mask |= (1 << 6);
        if (flags & ATMEL_USART_CTS)    pin_mask |= (1 << 7);
        if (flags & ATMEL_USART_CLK)    pin_mask |= (1 << 10);

        select_peripheral(PIOA, pin_mask, PERIPH_B, AT32_GPIOF_PULLUP);
}
```

3.2.2 Aktivierung

Die Schnittstellen werden im Setupcode des Boards aktiviert. Je nach Bedarf muß `arch/avr32/boards/setup.c` angepaßt werden.
Um die Konsolenausgabe über den zweiten seriellen Port USART1 (`/dev/ttyS0`) möglichst früh zu aktivieren, erfolgt die Aktivierung in der Funktion `setup_board`. Diese wird vor der Funktion `atngw100_init` aufgerufen, in der die weiteren seriellen Ports und die übrige Hardware registriert werden.

Listing 3.2: USART1 Aktivierung (setup.c:setup_board)

```
at32_map_usart(1, 0, 0);  /* USART 1: /dev/ttyS0, DB9 */
at32_setup_serial_console(0);
```

In `atngw100_init` wird USART3 mit Flags für aktiverte HW-Handshake- und Clockleitungen (`ATMEL_USART_RTS`, `ATMEL_USART_CTS` und `ATMEL_USART_CLK`) auf `/dev/ttyS2` abgebildet. Per `at32_add_device_usart` werden die Ports als *platform devices* [2] registriert.

Listing 3.3: USART Aktivierung Registrierung (setup.c:atngw100_init)

```
/* USART 3: /dev/ttyS2 */
at32_map_usart(3, 2, ATMEL_USART_RTS | ATMEL_USART_CTS |
    ATMEL_USART_CLK);

at32_add_device_usart(0);
at32_add_device_usart(1);
at32_add_device_usart(2);
```

3.2.3 Userspace Interface

Die aktivierten Anschlüsse werden dem Anwender als "TeleType" - Gerätedateien (`/dev/ttyS*`) bereitgestellt. Zum Ansprechen der Geräte wird unter POSIX-kompatiblen Betriebssystemen im allgemeinen die *termios*-Bibliothek genutzt, die Unterstützung für alle Einstellungen und Übertragungsarten mitbringt.

[2] Documentation/driver-model/platform.txt

3.3 I2C

Die I2C-Unterstützung im Kernel ist vorwiegend zur Kommunikation mit SMBus-Geräten auf Motherboards entstanden. In der Application Note AVR32412 - AVR32 AP7 TWI Driver [5] dokumentiert Atmel, wie die I2C/TWI-Einheit des AP7000 unter Linux benutzt wird. Im Dokument werden die I2C-Bustreiber `i2c-gpio` und `i2c-atmeltwi` beschrieben. Zum `i2c-atmeltwi` Treiber ist der avr32linux Webseite die Anmerkung "*Has some issues*" zu entnehmen, was auch der Grund ist, daß der Treiber nicht im Hauptzweig des Kernels aufgenommen wurde. Im Projekt wird daher der, im Kernel 2.6.30.7 enthaltene `i2c-gpio`-Treiber benutzt, der die I2C-Funktion über zwei GPIO-Pins per *Bitbanging* (siehe 2.5) realisiert.

3.3.1 Konfiguration

Die Konfiguration der I2C-Schnittstellen erfolgt in `arch/avr32/boards/setup.c`. Dabei wird die Struktur `i2c_gpio_platform_data` mit Informationen zu den benutzten Pins und dem gewünschten clock-delay gefüllt.

Listing 3.4: i2c-0 und i2c-1 Konfiguration (setup.c)

```
static struct i2c_gpio_platform_data i2c_gpio_data0 = {
  .sda_pin    = GPIO_PIN_PA(6),
  .scl_pin    = GPIO_PIN_PA(7),
  .sda_is_open_drain  = 1,
  .scl_is_open_drain  = 1,
  .udelay     = 2,  /* close to 100 kHz */
};

static struct i2c_gpio_platform_data i2c_gpio_data1 = {
  .sda_pin    = GPIO_PIN_PB(7),
  .scl_pin    = GPIO_PIN_PB(8),
  .sda_is_open_drain  = 1,
  .scl_is_open_drain  = 1,
  .udelay     = 2,  /* close to 100 kHz */
};

static struct platform_device i2c_gpio_device0 = {
  .name   = "i2c-gpio",
  .id   = 0,
  .dev    = {
    .platform_data  = &i2c_gpio_data0,
  },
};

static struct platform_device i2c_gpio_device1 = {
  .name   = "i2c-gpio",
  .id   = 1,
  .dev    = {
    .platform_data  = &i2c_gpio_data1,
  },
};
```

3.3.2 Aktivierung

Die Aktivierung und Registrierung der I2C Schnittstellen des AP7000 erfolgt ebenfalls in `arch/avr32/boards/setup.c`. Für Bus 0 wird zusätzlich zu SDA und SCL die `SMBALERT#` Leitung auf PB28 aktiviert. Alle verwendeten Pins sollten mit einem externen pullup-Widerstand versehen sein. Die SDA- und SCL-Pins von i2c-0 haben diese Widerstände auf dem NGW100 (siehe [3]) schon. Für die von i2c-1 verwendeten Pins müssen diese von der Schaltung bereitgestellt werden.

Listing 3.5: i2c-0 und i2c-1 Aktivierung (setup.c:atngw100_init)

```
at32_select_periph(GPIO_PIOB_BASE, 1 << 28, 0, AT32_GPIOF_PULLUP);
at32_select_gpio(i2c_gpio_data0.sda_pin,
  AT32_GPIOF_MULTIDRV | AT32_GPIOF_OUTPUT | AT32_GPIOF_HIGH);
at32_select_gpio(i2c_gpio_data0.scl_pin,
  AT32_GPIOF_MULTIDRV | AT32_GPIOF_OUTPUT | AT32_GPIOF_HIGH);
platform_device_register(&i2c_gpio_device0);

at32_select_gpio(i2c_gpio_data1.sda_pin,
  AT32_GPIOF_MULTIDRV | AT32_GPIOF_OUTPUT | AT32_GPIOF_HIGH);
at32_select_gpio(i2c_gpio_data1.scl_pin,
  AT32_GPIOF_MULTIDRV | AT32_GPIOF_OUTPUT | AT32_GPIOF_HIGH);
platform_device_register(&i2c_gpio_device1);
i2c_register_board_info(0, i2c_info, ARRAY_SIZE(i2c_info));
```

3.3.3 Userspace Interface

Als Schnittstelle zum Userspace stellt der Kerneltreiber **`i2c-dev`** für jede I2C-Businstanz eine Gerätedatei `/dev/i2c-X` (`X`=Busnummer) bereit. Für einfache Ein-/Ausgabe stehen die Aufrufe read() und write() zur Verfügung. Allerdings ist die Auswahl des Slaves nur durch die ioctl()-Kommandos `I2C_SLAVE` oder `I2C_SLAVE_FORCE` möglich.
C-Programme können den Kernelheader `i2c-dev.h` einbinden, um über die Gerätedatei Busseinstellungen vorzunehmen oder komplexere Übertragungen per ioctl() durchzuführen. Von der Datei `i2c-dev.h` gibt es zwei Versionen:

> ... Please note that there are two files named "i2c-dev.h" out there, one is distributed with the Linux kernel and is meant to be included from kernel driver code, the other one is distributed with i2c-tools and is meant to be included from user-space programs. You obviously want the second one here.[3]

Die in den i2c-tools enthaltene `i2c-dev.h`, die die `I2C_SMBUS`-Kommandos weiter abstrahiert, wird auch von der Python-Erweiterung `py-smbus` genutzt. Mit dem `I2C_SMBUS`-Kommando können verschiedene, durch den SMBus-Standard festgelegte Übertragungen oder I2C-Blocktransfers veranlasst werden.

[3] aus Documentation/i2c/dev-interface

3.4 SPI

Die Treiber für die AP7000 SPI Einheiten sind im Kernel enthalten.

3.4.1 Konfiguration

Die Konfiguration der SPI Schnittstellen des AP7000 erfolgt im boardspezifischen Setupcode in `arch/avr32/boards/setup.c`. Für jede SPI-Einheit wird eine `spi_board_info` Struktur mit Angaben zu den angeschlossenen Geräten gefüllt. Diese beinhalten mindestens den Treibernamen (`modalias`) und die Chipselect-Nummer, sowie optional die Taktgeschwindigkeit und weitere treiberspezifische Angaben.

Listing 3.6: spi-0 Konfiguration (setup.c)

```
static struct spi_board_info spi0_board_info[] __initdata = {
  {
    .modalias = "mtd_dataflash",
    .max_speed_hz = 8000000, // set to 8.00MHz
    .chip_select  = 0,
  },
  {
    .modalias = "spidev",
    .chip_select  = 1,
  },
  {
    .modalias      = "mcp251x",
    .platform_data = &mcp251x_info,
    .irq           = AT32_EXTINT(0),
    .max_speed_hz  = 2*1000*1000,
    .chip_select   = 2,
  },
};
```

3.4.2 Aktivierung

Der Aufruf des Befehls at32_add_device_spi() führt die Initialisierung durch.

Listing 3.7: spi-0 und spi-1 Aktivierung (setup.c:atngw100_init)

```
at32_add_device_spi(0, spi0_board_info, ARRAY_SIZE(spi0_board_info));
at32_add_device_spi(1, spi1_board_info, ARRAY_SIZE(spi1_board_info));
```

3.4.3 Userspace Interface

Der `spi-dev`-Treiber stellt für jedes, mit dem Treibernamen "`spidev`" konfigurierte SPI-Gerät eine Gerätedatei `/dev/spidevX.Y` bereit. Über diese Datei können die Einstellungen zur Protokollkonfiguration vorgenommen und Daten per read() und write() übertragen werden.
Um Daten von einem SPI-Gerät zu lesen bedarf es normalerweise eines write()-Befehls, der ein

Kommando oder eine Adresse enthält und eines darauf folgenden read()-Befehls, mit dem die angefragten Daten gelesen werden. Zwischen diesen beiden Befehlen wird allerdings das CS-Signal deaktiviert, was viele Geräte zum Abbruch einer Aktion veranlaßt.
Per ioctl()-Aufruf lassen sich Vollduplex-Übertragungen durchzuführen. Es können mehrere Bytes gesendet und empfangen werden, während CS weiter aktiv bleibt.
Um eine Vollduplex-Übertragung durchzuführen, muß die Struktur spi_ioc_transfer gefüllt werden; dem ioctl() Aufruf wird eine Referenz darauf mitgegeben.

Listing 3.8: Definition der Struktur spi_ioc_transfer (spidev.h)

```
struct spi_ioc_transfer {
  __u64   tx_buf;
  __u64   rx_buf;

  __u32   len;
  __u32   speed_hz;

  __u16   delay_usecs;
  __u8    bits_per_word;
  __u8    cs_change;
  __u32   pad;
};
```

- Die Adressen in rx_buf und tx_buf zeigen auf Speicherbereiche für Empfangs- und Sendepuffer; sie dürfen identisch sein.
- speed_hz: Angabe der maximalen Übertragungsgeschwindigkeit
- delay_usec: Wartezeit nach der Übertragung
- bits_per_word: Angabe der Bits pro SPI-Wort
- cs_change: Angabe, ob nach der Übertragung des Wort die CS Leitung deaktiviert wird
- len: Länge der Puffergrößen von rx_buf und tx_buf in Bytes

Die SPI Implementierung für den AP7000 unterstützt die per-transfer Angaben für speed_hz und bits_per_word nicht. Die Felder können freigelassen oder mit dem Wert '0' belegt werden.

3.5 PWM

Linux 2.6.30.7 enthält zwar einen einfachen Treiber für die PWM-Einheit des AP7000, doch stellt dieser nur eine interne Kernelschnittstelle zur Verfügung. Damit sind die PWM-Einheiten nur von Kerneltreibern aus nutzbar.

3.5.1 Bill Gatliff's Generic PWM API

Um PWM-Hardwareeinheiten verschiedener Mikrocontroller über eine gemeinsame User- und Kernelspaceschnittstelle ansprechen zu können, hat Bill Gatliff die "**Generic PWM API**" entwickelt, und als Patch zur Verfügung gestellt. Momentan (Dezember 2009) läuft eine Diskussion über den Reifegrad und die Aufnahme in den Hauptzweig des Kernels.[4]
Die Beschreibung des Patches enthält folgende Punkte:

- Bereitstellung einer API um PWM Einheiten hinter einer gemeinsamen User- and Kernelschnittstelle zu konsolidieren
- Emulierung von PWM Hardware auf einem GPIO Pin mithilfe eines high-resolution timers
- Ersatz des alten Atmel PWMC Treibers, durch einen API konformen
- LED "`dimmer`" Trigger, der die PWM API nutzt, um die Helligkeit je nach Systemauslastung zu variieren
- Einbinden des PWM API Codes in das Kbuild[5] System

3.5.2 Aktivierung

Eine Aktivierung der PWM-Kanäle in `arch/avr32/boards/setup.c` ist für die *Generic PWM API*, ebenso wie für den originalen Atmel Treiber notwendig:

Listing 3.9: Aktivierung der 4 PWM Kanäle (setup.c:atngw100_init)

```
at32_add_device_pwm((1 << 0)|(1 << 1)|(1 << 2)|(1 << 3));
```

3.5.3 Userspace Interface

Die API macht die Aktivierung und Konfiguration der PWM-Kanäle über eine *sysfs*-Schnittstelle möglich. Für jeden PWM-Kanal wird ein Verzeichnis (`atmel_pwmc.0:X`, `X`: Kanalnummer) unter `/sys/devices/virtual/pwm` eingehängt. Für jeden Kanal werden dort folgende Attribute bereitgestellt:

- `duty_ns` (read / write): Einstellung des DutyCycle in Nanosekunden
- `period_ns` (read / write): Einstellung der Periodendauer in Nanosekunden

[4] http://lwn.net/Articles/357837/ bzw. http://thread.gmane.org/gmane.linux.kernel.embedded/2330
[5] Kbuild: Kernel Konfigurations- und Makesystem

- `polarity` (read / write): Hat das **polarity** Attribut den Wert 1, wird die Polarität der Ausgabe invertiert (also Pulse=LOW und Pause=HIGH statt Pulse=HIGH und Pause=LOW).
- `request` (read / write): Initialisierung des PWM-Kanals durch Lesen des Attributs; Freigabe durch Schreiben eines beliebigen Wertes.
- `run` (write only): Nach Initialisierung (`request`) kann der PWM-Kanal durch das Schreiben des `run` Attributs gestartet (`1`) und gestoppt (`0`) werden.

3.6 GPIO

In der Datei `arch/avr32/mach-at32ap/pio.c` wird die Linux GPIO API[6] inklusive Interruptunterstützung für die AP700x Plattform implementiert.

3.6.1 Userspace Interface

Um dem Userspace GPIO Pins zur Verfügung zu stellen, gibt es verschiedene Möglichkeiten.

gpio-dev driver

Im vorinstallierten Firmwareimage des NGW100 ist der, von Atmel bereitgestellte **gpio-dev** Treiber aktiv. Seine Verwendung ist im Dokument "AVR32 AP7 Linux GPIO driver" [4] dokumentiert. Allerdings ist die Schnittstelle AVR32-spezifisch und wird deshalb nie auf anderen Plattformen zur Verfügung stehen:

> The GPIO dev interface is not in the mainline kernel and will never be there either. This interface is AVR32 specific and it is not available on other platforms. This chapter is here because many use this interface already and maybe do not want to switch to another solution.

Im Dokument wird als Alternative empfohlen, zur Ausgabe von Signalen das *LED Framework* und zur Eingabe den `gpio-keys` Treiber zu verwenden.

Der **`gpio-keys`**-Treiber ermöglicht die Eingabe von interruptfähigen GPIO Pins und stellt zur Kommunikation mit dem Userspace ein *input device*[7] zur Verfügung. Um Pins mit dem gpio-keys Treiber zu verwenden, wird der Board - Setup Code angepaßt (siehe Anhang C).
Der **`gpio-buttons`**-Treiber stellt, ebenso wie der `gpio-keys` Treiber ein *input device* zur Verfügung. `gpio-buttons` ermöglicht allerdings die Eingabe von GPIO Pins, die *keine* Interrupts erzeugen können.

sysfs interface

Die Ein- und Ausgabe von GPIO Signalen ist auch über eine *sysfs*-Schnittstelle möglich. Diese soll auch der zukünftige Standardweg sein, um GPIO's anzusprechen. Die Dokumentation ist aus `Documentation/gpio.txt` zu entnehmen:

Im Verzeichnis `/sys/class/gpio` finden sich drei Arten von Einträgen:

1. `export`/`unexport` helfen, die Kontrolle über GPIO's zu erlangen oder abzugeben.
2. GPIO-Instanzen
3. GPIO-Controller-Instanzen (`gpio_chip`)

[6] Documentation/gpio.txt
[7] Documentation/input/input.txt

Diese Einträge sind zusätzlich zu den normalen *sysfs*-Einträgen für die Controller vorhanden.

Die *control files* `export` und `unexport` in `/sys/class/gpio/` sind nur schreibbar:

- `export` Durch das Schreiben einer Nummer in diese Datei kann vom Userspace aus die Kontrolle über den entsprechenden GPIO vom Kernel angefordert werden.
  Beispiel: `echo 19 > export` erzeugt den "gpio19" Knoten für GPIO #19, wenn der Pin zur Verfügung steht.
- `unexport` Macht den Export rückgängig.
  Beispiel: `echo 19 > unexport` entfernt den "gpio19" Knoten, der über "export" exportiert wurde.

Die Knoten für die einzelnen GPIO-Pins haben Pfade wie `/sys/class/gpio/gpio42/` (für GPIO #42) und besitzen die folgenden Attribute, die schreib- und lesbar sind:

`/sys/class/gpio/gpioN/`

- `direction`: Ein Lesen gibt "in" oder "out" zurück. Das Attribut kann normalerweise geschrieben werden. Das Setzen auf "out" initialisiert das Signal mit LOW. Werden die Werte "low" oder "high" geschrieben, wird der GPIO als Ausgang mit dem entsprechenden Wert konfiguriert.
- `value`: Ein Lesen gibt entweder 0 (LOW) oder 1 (HIGH) zurück. Ist der GPIO als Ausgang konfiguriert, kann das `value` Attribut geschrieben werden. Jeder Wert ungleich Null wird als HIGH interpretiert.
- `edge`: Ein Lesen gibt entweder "none", "rising", "falling", oder "both" zurück. Dieselben Werte können geschrieben werden, um die Pegelwechsel auszuwählen, bei denen ein poll()-Systemaufruf auf dem `value`-Knoten zurückkehrt.
  Dieser Knoten existiert nur, wenn der GPIO als interrupterzeugender Eingang konfiguriert werden kann.

Der im Projekt benutzte Kernel (2.6.30.7) benötigte einen Patch um die Systemaufrufe select() und poll() auf `/sys/class/gpio/gpioXX/value` zu unterstützen. Dieser Patch ist mittlerweile in den Hauptzweig des Kernels eingeflossen (2.6.32), und stellt den neuen *sysfs*-Eintrag `/sys/class/gpio/gpioXX/egde` bereit.

GPIO Controller haben Pfade wie `/sys/class/gpio/chipchip42/` (für den Controller, der die GPIO's ab #42 verwaltet) und besitzen die folgenden read-only Attribute:

- `base`: Nummer des ersten GPIO's, der durch diesen Chip verwaltet wird
- `label`: Zu Diagnosezwecken bereitgestellt (nicht unbedingt eindeutig)
- `ngpio`: Anzahl der GPIO's, die durch diesen Chip verwaltet werden (N .. N + ngpio - 1)

4 SoftwarePWM Kernelmodul

Um auf dem NGW100 mehr als die vier, durch die Hardwareeinheit bereitgestellten PWM Kanäle zur Verfügung zu haben (z.B. zur Ansteuerung von Motoren), habe ich einen Kerneltreiber entwickelt, der auf 16 weiteren GPIO-Pins PWM-Signale bereitstellt. Der Treiber wurde mithilfe eines *high resolution timer*'s implementiert und stellt eine Gerätedatei (`/dev/softpwm`) bereit, über die die Pulsbreite der Kanäle eingestellt werden kann.

Schrittweite Servomotoren sprechen meist auf Änderungen in einem kleinen Bereich der Gesamtperiodendauer an, der zwischen 1ms und 2ms liegt.[1] Das heißt, daß der Motor bei einer Pulsbreite von 1.5ms zentriert ist, bei einer Pulsbreite von 1ms voll auf die eine Seite ausschlägt, bei einer Breite von 2ms auf die andere. Auf diesen Bereich entfällt nur ein Bruchteil der verfügbaren Schritte; im Falle eines aktiven Bereiches von einer Millisekunde und einer Gesamtperiodendauer von 20 Millisekunden ergeben sich 5%. ($1ms/20ms = 5\%$)
Um eine Granularität von 100 Schritten zur Motorsteuerung zu erreichen, ist also eine Gesamtgranularität von 2000 Schritten erforderlich. Ist der aktive Bereich (Abbildung 4.1) wie beim FreeBird Project mit 0.4ms noch kleiner, ergeben sich 40 Schritte.

> The FreeBird depends on two Parallax continuous rotation servo's to move around.
> The Parallax continuous rotation servo has to be first 'centred' by applying a signal with an ON time of 1.5ms and period of 20ms. The servo potentiometer has to be adjusted in such a way that the servo remains motionless. Once this calibration is over, a signal with an ON time of 1.7ms should rotate the servo in a counter-clockwise direction and a signal with an ON time of 1.3ms will make the servo rotate in a clockwise direction.[2]

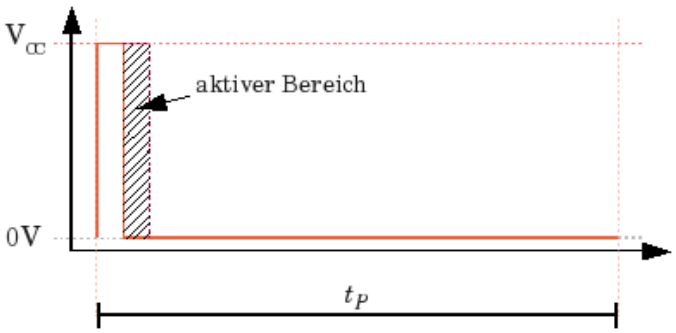

Abbildung 4.1: Pulsbreitenmodulation zur Motoransteuerung - aktiver Bereich

4.1 Implementierung

Den ersten Ansatz der Implementierung im Userspace unter Benutzung der in *uclibc* enthaltenen **librt** *C*-Bibliothek habe ich aufgegeben. Zwar werden dem Userspace durch librt hoch auflösende Timer durch Funktionen wie `nanosleep` zur Verfügung gestellt, jedoch nimmt der ständige

[1] http://www.societyofrobots.com/member_tutorials/node/231
[2] http://avr32linux.org/twiki/bin/view/Main/PramodeCE?skin=georgiablue.nat%2cnat#Controlling_a_Parallax_continuou

Kontextwechsel zwischen User- und Kernelspace beim Setzen der Pins zu viel Rechenzeit in Anspruch. Die Auslastung des Systems steigt - bei niedriger Qualität des Signals - stark an.

Um die Emulation möglichst performant und ressourcenschonend zu realisieren habe ich folgende Einschränkungen gemacht:

- Die Kanäle müssen gemeinsam auf einem PIO-Port liegen (Port E)
- Die Anzahl der Kanäle (16) ist zur Compilierzeit festgelegt (Port E 1-16)
- Die Frequenz der Kanäle ist auf 50 Hz, also auf eine Periodendauer von 20ms festgelegt
- Die Granularität für die Angabe der Pulsbreite beträgt 2000 Schritte; das ergibt eine Schrittweite von $10\mu s$

Diese Parameter können zur Kompilierzeit per `#define`-Direktive geändert werden.

Pinauswahl Die Signale werden auf den Pins von Port E ausgegeben, da die auf diesen Port gemultiplexten Funktionseinheiten (LCDC, EBI) im Projekt nicht verwendet werden. Dort steht eine ununterbrochene Folge von GPIO's zur Verfügung, auf die sich ein 16 Bit-Muster einfach anwenden läßt. Da Pin 0 von Port E als write-protect - Pin für das Memory Card Interface verwendet wird, werden Pins 1 bis 16 benutzt.

Tabelle 4.1: Software PWM Kanäle - Pinbelegung auf J7

SPWM Kanal	Pin	Pin	SPWM Kanal
2	1	2	3
4	3	4	5
6	5	6	
	7	8	
7	9	10	8
9	11	12	10
11	13	14	
	15	16	
12	17	18	13
14	19	20	15
	21	22	
	23	24	
0	25	26	1
	27	28	
	29	30	
	31	32	
	33	34	
	35	36	

Um Strom- und Spannungsspitzen zu Zyklusbeginn zu vermeiden, sind die Periodenstarts der PWM - Kanäle um jeweils 20 Schritte, also $200\mu s$ verschoben. Die Verschiebung in Schritten im Makro `PARAM_PINDELAY` definiert (siehe Listing 4.1).

Listing 4.1: set_dutycycle (softpwm.c)

```
static void set_dutycycle(struct pin * mypin, unsigned int pulse_width)
{
  mypin->stop = (pulse_width + mypin->offset * PARAM_PINDELAY) %
     PARAM_GRANU;
  if ( mypin->start > mypin->stop ) {
    mypin->rev = 1;
  } else {
    mypin->rev = 0;
  }
}
```

4.1.1 High Resolution Timer

Die Erzeugung eines PWM-Signals erfordert, je nach Periodendauer und Granularität eine hohe zeitliche Auflösung. Diese kann durch Benutzung der hoch auflösenden Timer (*hrtimer*) erreicht werden. Auf der AP7000 Plattform wird die Timer/Counter-Einheit **TC0** als clocksource (clock device) für die Implementierung der *hrtimer* genutzt.

Die theoretisch höchste Frequenz, die sich auf diese Weise erreichen läßt, ist der Kehrwert des minimalen Zeitoffsets, das angegeben werden kann. Bei der Beantwortung der Frage, wie groß dieses minimale Zeitoffset ist, hilft die Masterarbeit "Better Real-Time Capabilities For The AVR32 Linux Kernel" von Morton Engen [13]:

> min_delta_ns: Minimum event delta (offset into the future) which can be scheduled

Das minimale Offset, das unterstützt wird läßt sich über den *procfs* - Eintrag `timer_list` ermitteln. Auf dem NGW100 ergibt

```
cat /proc/timer_list | grep min_delta_ns
```

ein minimales Offset von `min_delta_ns = 30518`.
Zum Vergleich: Das minimale Offset auf dem Entwicklungsrechner ist `min_delta_ns = 1199`.

Die vom hrtimer aufgerufene Funktion wird im Kontext des Timer-Softinterrupt abgearbeitet und erbt dessen Priorität [3].

Die Interruptroutine wird zu jeder Änderung aufgerufen. Um den Timer neu zu starten muß die Routine den Wert `HRTIMER_RESTART` zurückgeben. Die Auslaufzeit des Timers wird um den Abstand zu nächsten Änderung erhöht. Die Variable `j` ist der Änderungszähler, anhand dessen das aktuelle Muster und das Offset zum nächsten Event aus dem Array gewählt wird.
Das Muster wird in das SODR ("Set Output Data Register") von Port E geschrieben, um eventuelle Änderungen von LOW auf HIGH durchzuführen. Für Änderungen von HIGH auf LOW muß das invertierte Muster in das CODR ("Clear Output Data Register") geschrieben werden.
Diese Operationen können in zukünftigen Versionen des Treibers möglicherweise durch einen

[3] Quelle: http://www.linux-magazin.de/Heft-Abo/Ausgaben/2007/01/Kern-Technik

Befehl ersetzt werden (AP7000 Datenblatt [2] - 19.5.6: Synchronous Data Output). An dieser Stelle ist das Kernelmodul architekturspezifisch und unportabel. Es kann aber recht einfach an andere Plattformen angepaßt werden, die die Möglichkeit zum Setzen mehrerer GPIO's auf einmal bieten.

Listing 4.2: hrtimer Interruptroutine (softpwm.c)

```
static int update_registers( struct hrtimer *hrt )
{
  static u8 j = 0;

  __raw_writel( (changes[j].mask) << 1, PORTE + PIO_SODR);
  __raw_writel( ( ( (u32) ( (u16) ~changes[j].mask ) ) ) << 1, PORTE +
      PIO_CODR);

  hrtimer_add_expires_ns(hrt, changes[j].diff);

  j++;
  j %= num_changes;

  return HRTIMER_RESTART;
}
```

Auf dem Oszilloskop sind Schwankungen des Signals zu beobachten. Diese werden nachfolgend genauer untersucht.

4.1.2 Userspace Interface

Die Schnittstelle zum Userspace stellt die Gerätedatei `/dev/softpwm` zur Verfügung. Die ioctl()-Kommandos `SOFTPWM_IOC_SETDUTY` und `SOFTPWM_IOC_ENABLE` werden im Header `gpio-softpwm.h` definiert:

`SOFTPWM_IOC_ENABLE`: Über diesen Aufruf können die PWM-Kanäle ein- und ausgeschalten werden, wobei der Timer gestoppt oder gestartet wird. Als Argument dienen die, in `gpio-softpwm.h` definierten Makros `SOFTPWM_ON` und `SOFTPWM_OFF`.

`SOFTPWM_IOC_SETDUTY`: Um die Pulsbreite eines Pins zu ändern, wird dem Kommando eine Referenz auf eine Struktur des Typs `spwm_ioctl` mitgegeben. Diese enthält zwei Integer, und zwar das Offset des Pins und die gewünschte Pulsbreite in Schritten. `spwm_ioctl` wird ebenfalls in `gpio-softpwm.h` definiert:

Listing 4.3: Definition von spwm_ioctl (gpio-softpwm.h)

```
struct spwm_ioctl {
  unsigned short pin_offset;
  unsigned short pulse_width;
};
```

Die Änderung der Pulsbreite eines Pins resultiert in einer Neuberechnung der Informationen, die die Interrupt-Routine benutzt (Funktion `calc_changes`). Dies ist ein aufwändiger Vorgang, bei dem für jeden Granularitätsschritt und für jeden Pin überprüft wird, ob der Pin gesetzt sein muß.

4.2 Bewertung des Signalqualität / Jitteruntersuchung

Bei ungewünschten Schwankungen in einem Signal spricht man von **jitter**. Da bei der SoftwarePWM-Lösung Betriebssystem und CPU bei der Signalerzeugung miteinbezogen werden, spielt das zeitliche Auflösungsvermögen des OS eine tragende Rolle bei der zeitlichen Genauigkeit.
Zur Untersuchung der Schwankungen stand ein Logic Analyzer (LA) zur Verfügung, mit dem man über einen gewissen Zeitraum die Pegeländerung nanosekundengenau aufzeichen kann.
Die Messung erstreckt sich über 58 PWM Perioden, also 116 Übergänge. Das Testsignal hat eine Pulsbreite von 10ms und eine Periodendauer von 20ms. Das heißt, daß zwischen jedem Pegelwechsel im besten Fall eine Ruhezeit von exakt 10ms, also $1.00e^7$ns liegen sollte.
Allerdings pendeln die Ruhezeiten zwischen $9.975e^6$ und $1.0015e^7$; siehe hierzu Abbildung 4.2.
Die Differenz zwischen den Zeiten, die zu lang, und denen, die zu kurz sind beträgt circa 30000 ns, also ungefähr der Wert, der dem `min_delta_ns` des NGW100 entspricht.
Da der hrtimer mit absoluten Zeitangaben arbeitet, gleichen sich die verkürzten und die verlängerten Ruhezeiten nach einer gewissen Zeit (2-8 Übergänge) wieder aus; für die 116 Übergänge ergibt sich ein Durchschnitt der Ruhezeiten von 9999863 ns bei einer Standardabweichung von ~13882 ns.
Auf dem Oszilloskop ist eine Schwankung der Frequenz zwischen 49.98 und 50.02 Hz zu beobachten.

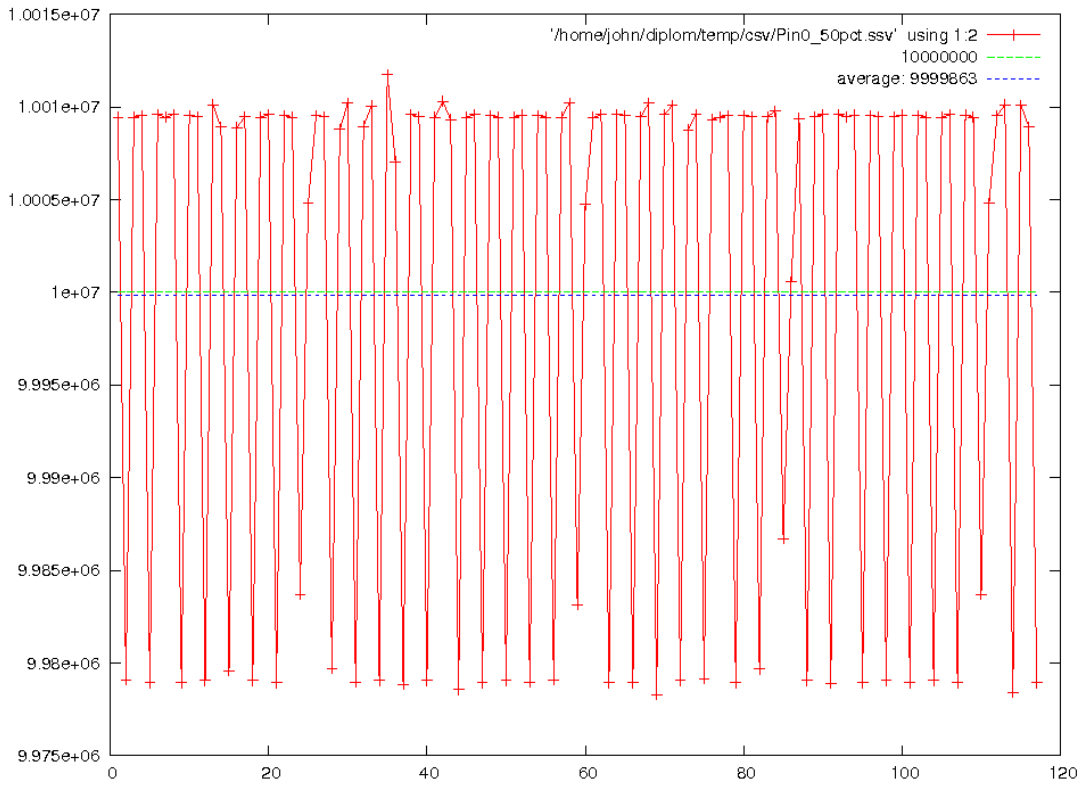

Abbildung 4.2: Schwankungen der Ruhezeiten eines Kanals bei einer Peridendauer von 20ms und einer Pulsbreite von 10ms

5 Python API

5.1 Python

Die Programmiersprache **Python**[1] wurde 1990 von Guido van Rossum am Centrum voor Wiskunde en Informatica (CWI), einem Forschungsinstitut in Amsterdam entwickelt, um die Lücke zwischen Shellskripts und compilierten C Programmen zu überbrücken. Dieselbe Aufgabe soll Python auf dem AP7000 erfüllen. Python ist eine objektorientierte Skriptsprache und heute auf einer Vielzahl von Plattformen (Unix/Linux, Windows, MacOS, etc.) verfügbar.

> Ein Python-Skript wird vom Interpreter in Bytecode übersetzt und ausgeführt. Es existieren Python-Implementierungen in C (CPython und Stackless), Java (Jython), C#/Mono (IronPython) und Python selber (PyPy). Dadurch lässt sich Python sehr gut als eingebettete Skriptsprache in den unterschiedlichsten Umgebungen einsetzen und mit den entsprechenden Programmiersprachen (C/C++, Java, C#, etc.) erweitern.[2]

In dieser Arbeit wird die am weitesten verbreitete Implementierung in C, CPython benutzt.

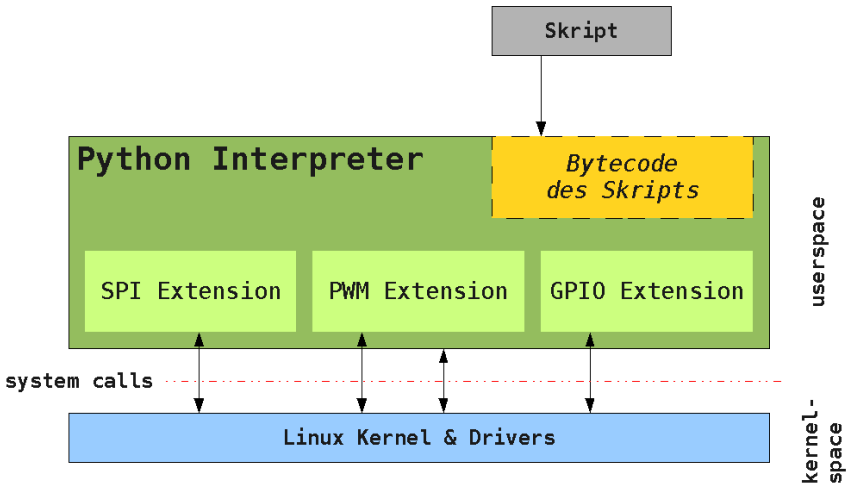

Abbildung 5.1: Der Pythoninterpreter mit Erweiterungsmodulen

5.2 Erweiterung von Python

Die Erweiterung von Python kann einerseits, wie bei jeder objektorientierten Programmiersprache durch Definition und Implementierung neuer Klassen mit Python selbst bewerkstelligt werden. Diesen Weg gehen beispielsweise die Autoren des **pyserial** Moduls, welches komplett in Python geschrieben wurde. Bei der skriptseitigen Erweiterung ist man allerdings immer durch die Möglichkeiten beschränkt, die der Interpreter zur Verfügung stellt.
Mithilfe der Python C-API [10] ist es einfach möglich, den Interpreter selbst mit Modulen (Extension Modules) zu erweitern. Diese können als statisch oder dynamisch ladbare Bibliotheken bereitstehen oder direkt in den Interpreter compiliert und gelinkt werden.

[1] `http://www.python.org`
[2] von `http://www.python.de`

5.2.1 distutils

Mit den **distutils** steht ein Pythonpaket zur Verfügung, das Routinen zur Übersetzung und Installation von Pythonerweiterungen bietet.

> The distutils package provides support for building and installing additional modules into a Python installation. The new modules may be either 100%-pure Python, or may be extension modules written in C, or may be collections of Python packages which include modules coded in both Python and C. [3]

Der Autor eines Moduls erstellt ein kleines Pythonskript `setup.py`, in dem die Funktion `setup` aufgerufen wird, die die Einstellungen für die auszuführende Aktion vornimmt. Die auszuführende Aktion wird dem Skript als Kommandozeilenparameter mitgegeben. Eine ausführliche Dokumentation findet sich im Dokument *Extending and Embedding Python* [9] der Python Dokumentation.

Listing 5.1: py-spi: setup.py

```
from distutils.core import setup, Extension

setup(  name="spi",
        version="1.1",
        description="Python bindings for Linux SPI access through spi-
            dev",
        author="Volker Thoms",
        author_email="vthoms@hs-augsburg.de",
        license="GPLv2",
        include_dirs=["/usr/src/linux/include"],
        scripts=['scripts/spitest.py','scripts/spitest2.py'],
        ext_modules=[Extension("spi", ["spimodule.c"])])
```

Beispiele für Aufrufe von setup.py:

- `python setup.py compile`
  - Übersetzen des Moduls
- `python setup.py install`
  - Installation des Moduls

5.2.2 Python/C-API

Die Python/C-API bietet für Spracherweiterungsmodule die Schnittstelle zum CPython-Interpreter. Sie bietet Funktionen und Makros, die beim Datentransfer von und zu Python, beim Arbeiten mit Python-Datentypen, oder auch bei der Threadprogrammierung helfen. Zur Verwendung muß die Headerdatei `Python.h` eingebunden werden.

5.2.3 Module

Die erstellten Module sind im Python-Pfad auf dem Rootfilesystem enthalten und können von Pythonskripten aus per '`import modulname`' eingebunden werden. Der Quellcode ist auf der CDROM in `trees/pyap7k/<modulname>/src` zu finden.

[3] `http://docs.python.org/library/distutils.html`

5.3 Python Extension: socket

Das **socket** Modul kapselt den Zugriff auf die BSD socket-Schnittstelle. Es läuft auf allen modernen Unix Systemen, Windows, Mac OS X, BeOS, OS/2 und möglicherweise weiteren Plattformen und in der Standardinstallation von Python enthalten.
Sockets ermöglichen eine Client/Server basierte Programmierung, wobei unter Linux die Adressfamilien AF_INET, AF_INET6, AF_UNIX, AF_PACKET, AF_NETLINK und AF_TIPC zur Verfügung stehen. Das heißt, Sockets der Familie 29 (AF_CAN) werden in der aktuellen Version leider **nicht** unterstützt [4].

5.3.1 API Dokumentation

Die Verwendung des **socket**-Moduls ist in der Python Library Reference, 18.2 [5] ausführlich und mit Beispielen dokumentiert.

5.3.2 Test

Zum Test des Moduls habe ich das erste Beispiel aus der Dokumentation benutzt. Die Kommunikation über die Ethernetverbindung funktionierte, wobei der Serverteil auf dem Entwicklungsrechner und der Clientteil auf dem NGW100 lief.

5.4 Python Extension: pyserial

Das **serial**-Modul stellt die Klasse `serial.Serial` bereit. Ein Objekt diesen Typs kapselt den Zugriff auf einen seriellen Port. Auf POSIX kompatiblen System wird die termios-Bibliothek als Backend genutzt, deren Möglichkeiten an den Pythonprogrammierer weitergereicht werden.

5.4.1 API Dokumentation

pyserial ist auf der Homepage [6] mit Beispielen ausführlich dokumentiert.

[4] `https://lists.berlios.de/pipermail/socketcan-users/2009-June/000970.html`
[5] `http://docs.python.org/library/socket.html`
[6] `http://pyserial.sourceforge.net`

5.5 Python Extension: py-smbus

Die SMBus-Pythonerweiterung von Mark M. Hoffman ist in den i2c-tools des **lm-sensors** Projekts enthalten.
Das **smbus**-Modul stellt die Klasse `smbus.SMBus` bereit. Ein Objekt vom Typ `SMBus` kapselt den Zugriff auf eine I2C- bzw. SMBus-Businstanz; als Backend dienen die vom i2c-dev Kerneltreiber bereitgestellten Gerätedateien `/dev/i2c-X` (`X`=Busnummer). Es werden die SMBus- und I2C-Blocktransfer-Funktionen an den Pythonprogrammierer weitergereicht, die im Header `i2c-dev.h` der i2c-tools deklariert sind.

Einschränkungen: Das SMBus-Pythonmodul ist - trotz der Unterstützung für zwei I2C-Befehle - kein I2C-Modul. Alle Geräte, die mir zur Verfügung standen konnten aber mit SMBus-Kommandos erfolgreich angesprochen werden.

5.5.1 API Dokumentation

Attribute

`pec`
Packet Error Checking wurde in Revision 1.1 der SMBus Spezifikation aufgenommen. PEC fügt zu den Transfers, die es unterstützen eine 8bit CRC-Prüfsumme direkt vor dem terminierenden Stopbit hinzu.

Methoden

`open``(int bus): None`

- *`bus`*: Busnummer - die Busnummer kann bei der Instanzierung auch der `__init__()`-Funktion mitgegeben werden

Öffnet die entsprechende Gerätedatei

`close``(): None`
Schließt die aktuelle Gerätedatei

SMBus Übertragungskommandos [7]

`write_quick``(int addr): None`

- *`addr`*: Client Adresse

SMBus Quick Command:
Das **write_quick** Kommando sendet ein einzelnes Bit zu einem Gerät. Das Bit ist durch das Rd/Wr-Bit spezifiziert und im *`addr`* Parameter mitgegeben.

```
A Addr Rd/Wr [A] P
```

[7] Die Dokumentation der SMBus Kommandos wurde großteils aus der Kerneldokumentation (*Documentation/i2c/smbus-protocol*) übernommen und ins Deutsche übersetzt.

read_byte(int *addr*): int

- *addr*: Client Adresse

SMBus Receive Byte:
Das **read_byte** Kommando liest ein einzelnes Byte von einem Gerät, ohne dabei ein Register auszuwählen. Einige einfache Geräte sind simpel genug aufgebaut, daß dieses Kommando zur kompletten Ansteuerung genügt. Einige andere Geräte nutzen dieses Kommando um dasselbe Register, wie im vorherigen SMBus Kommando noch einmal zu lesen.

```
S Addr Rd [A] [Data] NA P
```

write_byte(int *addr*, int *val*): None

- *addr*: Client Adresse
- *val*: zu sendendes Byte

SMBus Send Byte:
Das **write_byte** Kommando ist das Pendant zu **read_byte**. Es wird ein einzelnes Byte an ein Gerät gesendet ohne dabei ein Register auszuwählen. Einige einfache Geräte sind simpel genug aufgebaut, daß dieses Kommando zur kompletten Ansteuerung genügt. Einige andere Geräte nutzen dieses Kommando um in dasselbe Register, wie im vorherigen SMBus Kommando noch einmal zu schreiben.

```
S Addr Wr [A] Data [A] P
```

read_byte_data(int *addr*, int *cmd*): int

- *addr*: Client Adresse
- *cmd*: Befehl zur Registerauswahl

SMBus Read Byte:
Das **read_byte_data** Kommando liest ein einzelnes Byte vom ausgewählten Register *cmd* eines Gerätes *addr*.

```
S Addr Wr [A] Comm [A] S Addr Rd [A] [Data] NA P
```

write_byte_data(int *addr*, int *cmd*, int *val*): None

- *addr*: Client Adresse
- *cmd*: Befehl zur Registerauswahl
- *val*: zu sendendes Byte

SMBus Write Byte:
Das **write_byte_data** Kommando ist das Pendant zu **read_byte_data**. Es wird ein einzelnes Byte *val* in das durch *cmd* ausgewählte Register eines Gerätes *addr* geschrieben.

```
S Addr Wr [A] Comm [A] Data [A] P
```

read_word_data(int *addr*, int *cmd*): int

- *addr*: Client Adresse
- *cmd*: Befehl zur Registerauswahl

SMBus Read Word:
Das **read_word_data** Kommando ist ähnlich dem **read_byte_data**. Es werden Daten von einem Register *cmd* vom Gerät *addr* gelesen. Die Daten sind diesmal allerdings ein komplettes word (16 Bit).

```
S Addr Wr [A] Comm [A] S Addr Rd [A] [DataLow] A [DataHigh] NA P
```

write_word_data(int *addr*, int *cmd*, int *val*): None

- *addr*: Client Adresse
- *cmd*: Befehl zur Registerauswahl
- *val*: zu sendendes 16 Bit Wort

SMBus Write Word:
Das **write_word_data** Kommando ist das Pendant zu **read_word_data**. Es werden 16 Bit Daten *val* in ein Register *cmd* eines Gerätes *addr* geschrieben.

```
S Addr Wr [A] Comm [A] DataLow [A] DataHigh [A] P
```

process_call(int *addr*, int *cmd*, int *val*): int

- *addr*: Client Adresse
- *cmd*: Befehl zur Registerauswahl
- *val*: zu sendendes Word

Das **process_call** Kommando wurde in Version 2.0 in die SMBus Spezifikation aufgenommen. Es kombiniert die **write_word_data** und **read_word_data** Kommandos. An das ausgewählte Register *cmd* werden 16 Bit Daten (*val*) gesendet und 16 Bit gelesen. [8]

```
S Addr Wr [A] Comm [A] DataLow [A] DataHigh [A] S Addr Rd [A] [DataLow] A
    [DataHigh] NA P
```

read_block_data(int *addr*, int *cmd*): list

- *addr*: Client Adresse
- *cmd*: Befehl zur Registerauswahl

SMBus Block Read:
Das **read_block_data** Kommando liest einen Block von bis zu 32 Byte Länge aus dem ausgewähltem Register *cmd* vom Gerät *addr*. Die Anzahl der empfangenen (weiteren) Bytes wird vom Gerät im Count - Byte zurückgegeben und ist im ersten Element der von der Funktion zurückgegebenen Liste enthalten.

```
S Addr Wr [A] Comm [A] S Addr Rd [A] [Count] A [Data] A [Data] A ... A [
    Data] NA P
```

[8] Anmerkung: Im original py-smbus Modul wird `None` zurückgegeben. In der Version auf der CD-ROM wurde das geändert, aber nicht getestet.

write_block_data(int *addr*, int *cmd*, list *vals*): None

- *addr*: Client Adresse
- *cmd*: Befehl zur Registerauswahl
- *vals*: Liste von Integern (mindestens einer, maximal 32)

SMBus Block Write:
Das **write_block_data** Kommando ist das Pendant zu **read_block_data**. Es werden bis zu 32 Byte *vals* in das Register *cmd* von Gerät *addr* geschrieben. Die Länge der Liste wird automatisch festgestellt und in das Count Byte geschrieben.

```
S Addr Wr [A] Comm [A] Count [A] Data [A] Data [A] ... [A] Data [A] P
```

block_process_call(int *addr*, int *cmd*, list *vals*): list

- *addr*: Client Adresse
- *cmd*: Befehl zur Registerauswahl
- *vals*: Liste von mindestens einem und maximal 32 Integern

Das **block_process_call** Kommando wurde in Version 2.0 in die SMBus Spezifikation aufgenommen. Es kombiniert die **write_block_data** und **read_block_data** Kommandos. An das ausgewählte Register *cmd* werden **M** Byte Daten (*val*) gesendet und **N** Byte gelesen. Dabei gelten folgende Bedingungen für **M** und **N**: $M \geq 1, N \geq 1, M + N \leq 32$

```
S Addr Wr [A] Comm [A] Count [A] Data [A] ... S Addr Rd [A] [Count] A [
    Data] ... A P
```

I2C Block Transfers Die folgenden I2C Block Transfers werden vom SMBus Layer unterstützt, sind jedoch nicht Bestandteil der SMBus Spezifikation.
Prinzipiell ist die Anzahl der übertragbaren Bytes nicht beschränkt, doch die SMBus-Schicht führt eine Grenze von 32 Bytes ein.

read_i2c_block_data(int *addr*, int *cmd* [, int *len* = 32]): list

- *addr*: Client Adresse
- *cmd*: Befehl zur Registerauswahl
- *len*: Anzahl der zu lesenden Bytes

Das **read_i2c_block_data** Kommando liest einen Block von Bytes aus einem, durch *cmd* ausgewählten Register von einem Gerät *addr*. Nachdem die gewünschte Anzahl von Bytes empfangen wurde, sendet der Master ein NA Bit. Die Anzahl der empfangenen Bytes ist im ersten Element der von der Funktion zurückgegebenen Liste enthalten.

```
S Addr Wr [A] Comm [A] S Addr Rd [A] [Data] A [Data] A ... A [Data] NA P
```

write_i2c_block_data(int *addr*, int *cmd*, list *data*): None

- *addr*: Client Adresse
- *cmd*: Befehl zur Registerauswahl
- *data*: Liste der zu schreibenden Bytes

Das **write_i2c_block_data** ist das Pendant zu **read_i2c_block_data**. Es werden Bytes in das Register *cmd* von Gerät *addr* geschrieben. Es werden Kommandolängen von 0, 2 oder mehr Byte unterstützt, da diese nicht von den Datenbytes unterscheidbar sind.

```
S Addr Wr [A] Comm [A] Data [A] Data [A] ... [A] Data [A] P
```

Tabelle 5.1: Symbolerklärung

Symbol	Länge	
S	1 bit	Start Bit
P	1 bit	Stop Bit
Rd/Wr	1 bit	Read/Write Bit. Rd entspricht 1, Wr entspricht 0.
A, NA	1 bit	Acknowledge / Not Acknowledge Bit
Addr	7 bits	I2C 7 Bit Adresse; erweiterbar zu 10 Bit Adresse
Comm	8 bits	Kommando-/Befehlsbyte; zur Auswahl eines Registers eines Gerätes
Data	8 bits	Ein Datenbyte (DataLow, DataHigh für 16 bit Daten)
Count	8 bits	Ein Byte, das die Länge einer Block Operation enthält
[..]		vom I2C Gerät gesendete Daten

5.6 Python Extension: py-spi

Das **spi**-Modul stellt die Klasse `spi.SPI` bereit. Ein Objekt vom Typ SPI kapselt den Zugriff auf ein SPI Gerät, das durch Bus- und Gerätenummer festgelegt ist. Als Backend dienen die vom spidev-Kerneltreiber bereitgestellten Gerätedateien `/dev/spidevX.Y` (`X`=bus.`Y`=client). Es können die Geräteeinstellungen getätigt und Daten half- oder full-duplex übertragen werden.

5.6.1 Attribute

Mithilfe der Attribute lassen sich Einstellungen für die SPI - Übertragung festlegen. Beim Aufruf der **`open`** Methode werden die Attribute mit den aktuellen Werten der vom Gerät bezogenen Einstellungen initialisiert. Das Schreiben der Attribute resultiert in einem ioctl()-Aufruf, der die jeweilige Einstellung an den Treiber weiterreicht.

`mode`
: Das **mode** Attribut enthält einen Python Integer für den aktuellen SPI Modus. Die gültigen Werte sind - entsprechend 2.7.2 - 0 bis 3 und bestimmen Clockphase und Clockpolarity.

`lsb`
: Das **lsb** Attribut enthält einen Python Bool und gibt an, ob das niedrigstwertige Bit (Least Significant Bit First, **lsb** = True), oder das höchstwertige Bit (Most Significant Bit First, **lsb** = False) zuerst gesendet werden soll.

`bpw`
: Das **bpw** Attribut gibt die Anzahl der Bits pro Wort (Bits Per Word) an. (SPI default: 8)

`msh`
: Das **msh** Attribut enthält die aktuelle Einstellung für die maximale Übertragungsgeschwindigkeit in Hz (Maximum Speed in Hz).

5.6.2 Methoden

`__init__``(int bus, int client) : SPI`
: liefert ein neues SPI - Objekt zurück; sind die Parameter *`bus`* und *`client`* angegeben, wird die **`open`** Methode aufgerufen und die Parameter werden weitergereicht.

`open``(int bus, int client): None`

- *`bus`*: Busnummer
- *`client`*: Clientnummer (Chip Select)

Öffnet die entsprechende Gerätedatei `/dev/spidevX.Y`

`close``(): None`
: Schließt die aktuelle Gerätedatei

einfache Übertragungen

readbytes(int *len*): list

- *len*: Anzahl der zu lesenden Bytes

Das **readbytes** Kommando liest *len* SPI Bytes vom aktuellen SPI Gerät mit den aktuellen Einstellungen. $1 \leq len \leq 32$

writebytes(list *values*): None

- *values*: Liste der zu schreibenden Bytes

Das **writebytes** Kommando ist das Pendant zu **readbytes**. Es werden die Bytes *values* zum Gerät gesendet.

Vollduplex Übertragungen Die folgenden Übertragungen laufen immer voll duplex ab. Das heißt, daß für jedes über MOSI gesendete Wort eines über MISO empfangen wird. Die msg-Funktion startet für jedes Element der übergebenen Liste einen eigenen Transfer der Länge 1, während die msg2-Funktion alle Elemente der Liste als SPI-Wörter in einer Übertragung bündelt. Die zurückgegebene Liste ist genauso lang wie die mitgegebene und enthält die, über MISO empfangenen Wörter. Die Wortlänge für Sende- und Empfangsliste ist momentan auf Bytes (8 Bit) beschränkt.
Das erste Element in der Empfangsliste enthält also das Byte, das über MISO empfangen wurde, während das erste Byte aus der Sendeliste über MOSI gesendet wurde.

msg(list *tx*, int *delay*) : list

- *tx*: Liste von Python Integern der zu schreibenden SPI Wörter;
- *delay*: Python Integer, der das delay (die Wartezeit) nach einem Block in μs angibt. In der aktuellen Implementierung ist wird eine delay-Angabe für alle Übertragungseinheiten benutzt.

msg2(list *tx*) : list

- *tx*: Liste von Python Integern der zu schreibenden SPI Wörter

Das msg2 Kommando kombiniert die zu übertragenden Wörter in einer Übertragung, was die *per-transfer*-Angaben delay und cs_change in ihrer Wirksamkeit beschränkt.

5.7 Python Extension: py-pwm

Das **pwm** - Modul stellt die Klasse `pwm.PWM` bereit. Ein Objekt vom Typ PWM kapselt den Zugriff auf einen PWM Hardwarekanal, der durch die Kanalnummer festgelegt ist. Als Backend dient die, durch die *Generic PWM API* bereitgestellte, *sysfs*-Schnittstelle.

5.7.1 Attribute

`duty` Das **duty** Attribut enthält einen Python Integer mit dem Wert für die Pulsdauer (duty count. duty cycle) in Nanosekunden. Der Wert kann gelesen und geschrieben werden.

`period` Das **period** Attribut enthält einen Python Integer mit dem Wert für die Periodenlänge in Nanosekunden. Der Wert kann gelesen und geschrieben werden.

`polarity` Das **polarity** Attribut enthält einen Python Integer, dessen Wert die Polarität des PWM Signals angibt. (1: active high; 0: active low) Der Wert kann gelesen und geschrieben werden.

5.7.2 Methoden

`__init__( int channel)` Das **__init__** Kommando erzeugt ein neues Objekt vom Typ PWM. Es öffnet den PWM Kanal *`channel`*.

`start``()` Das **start** Kommando startet die PWM-Ausgabe mit den, durch die Attribute **duty**, **period** und **polarity** festgelegten Einstellungen.

`stop``()` Das **stop** Kommando stoppt die PWM-Ausgabe.

Listing 5.2: Python PWM Beispiel

```
import pwm
import time

a = pwm.PWM(0)

a.period = 1000000
a.polarity = 0

while(True):
  a.start()
  i = 0
    while (i < a.period):
    a.duty = i
    time.sleep(0.001)
    i += 1000
  a.stop()
  time.sleep(1)
```

5.8 Python Extension: py-gpio

Das **gpio** Modul stellt die Klasse `gpio.GPIO` bereit, die genau einen GPIO-Pin abstrahiert. Als Backend wird die GPIO *sysfs*-Schnittstelle genutzt, deren Möglichkeiten an den Pythonprogrammierer weitergegeben werden.

5.8.1 Methoden

__init__`( int` *gpio*`[, string` *direction*`, string` *trigger*`])`

- *gpio* [erforderlich]
  mögliche Werte: Integer
  Beschreibung: gpio - Nummer nach dem GPIO-Framework Schema
- *direction* [optional]
  mögliche Werte: “in”,“out”, “high”, “low”
  Beschreibung: Richtung des gpio-Pins
- *trigger* [optional] “both”, “none”

Das **__init__** Kommando erzeugt ein neues Objekt vom Typ GPIO. Ist der GPIO nicht verfügbar, versucht das Modul ihn zu exportieren.
Werden die optionalen Argumente `direction` und `trigger` nicht angegeben, werden sie mit den Werten belegt, die für den Pin zur Zeit der Objekterzeugung gelten.

5.8.2 Attribute

value `[1|0]` Das Lesen des **value** Attributs gibt ein Integerobjekt mit Wertemenge [0,1] zurück. Das Setzen ist nur möglich, wenn der GPIO durch Übergabe des **direction** Attributes im Konstruktor oder durch späteres Setzen als **Ausgang** konfiguriert wurde.

direction `[''in''|''out''|''low''|''high'']`
Über das **direction** Attribut läßt sich die Richtung des GPIO-Pins einstellen. Soll der GPIO als Eingang dienen, muß das Attribut auf “in” gesetzt werden. In dieser Einstellung kann auch die Callback - Funktionalität genutzt werden.
Alle anderen Werte konfigurieren den GPIO als Ausgang; “out” und “low” setzen das Signal initial auf LOW; “high” auf HIGH.

trigger `[''both''|''none'']`
Über das **trigger** Attribut läßt sich einstellen, bei welchen Signalpegeländerungen die Callbackfunktion aufgerufen werden soll. Mögliche Werte sind `none` (keine Aktivierung) oder `both` (Aktivierung bei HIGH-LOW und LOW-HIGH Übergängen).

callback Wird das **callback** Attribut auf ein *callable python object* gesetzt, wird dieses aufgerufen, wenn die Triggerbedingung erfüllt ist. Ein Setzen auf `None` schaltet den Callbackmechanismus für diesen Pin aus.

5.8.3 Callback Mechanismus

Ist der Pin als Eingang konfiguriert, kann der Callback-Mechanismus verwendet werden. Damit kann auf eine Änderung des Eingangswertes (z.B. der **SMBALERT#**-Leitung) mit dem Aufruf eines ausführbaren Pythonobjekts (Callbackfunktion) reagiert werden.

Der Callback-Mechanismus wurde mittels eines Python-Threads implementiert. Dieser läuft solange, wie mindestens eine Callbackfunktion registriert ist. In dem Thread werden in einer Endlosschleife die `value`-Attribute der GPIO's mithilfe des Systemaufrufs `poll` auf Änderungen überprüft.

Da *`both`* momentan der einzige funktionierende Trigger ist, wird der Callbackfunktion als erstes Argument der aktuelle Wert des GPIO mitgegeben, woraus sich auf die Art des Übergangs (HIGH→LOW oder LOW→HIGH) schließen läßt.

Listing 5.3: Python GPIO callback Beispiel

```
# GPIO's #27 (output) and #95 (input) are connected over a resistor
#
# everytime #27's value changes (fourty times) #95's
# callback function will be triggered
#

import gpio
from time import sleep

def mycb(opt):
  if (opt == 0):
    print "cb: down"
  else:
    print "cb: up"

b = gpio.GPIO(27, "out")
a = gpio.GPIO(95, "in")

a.callback = mycb
a.trigger = "both"

i = 0
while(i < 20):
  b.value = 0
  sleep(0.05)
  b.value = 1
  sleep(0.05)
  i += 1
```

5.9 Python Extension: py-softpwm

Das **softpwm** Modul stellt die Klasse `softpwm.softpwm` bereit, die es ermöglicht, das Software-PWM Kernelmodul über das Characterdevice `/dev/softpwm` zu steuern und dessen Möglichkeiten an den Pythonprogrammierer weiterzugeben.
Das Modul ermöglicht es die Pulsbreite der Kanäle zu ändern. Für die Auswahl des Pins wird der Offset **off** zum Startpin angegeben. Die Angabe der Pulsbreite **width** erfolgt in 2000 Schritten von 0 bis 1999. Da die Periodendauer von 50Hz im Kernelmodul festgelegt ist, läßt sie sich nicht ändern. Der Kerneltreiber wird beim Start des Systems automatisch geladen.

5.9.1 Methoden

set`(int` *`off`*`, int` *`width`*`)` Das Kommando **set** setzt die Pulsbreite (duty cycle) *`width`* für den Pin *`off`*. Gültige Werte für die Pulsbreite sind Integerwerte von 0 bis 1999; gültige Werte für das Pinoffset sind Integerwerte von 0 bis 15.

enable`(bool` *`val`*`)` Je nach dem Wert von *`val`* startet (*`val`*`=1`) und stoppt (*`val`*`=0`) das Kommando **enable** die PWM-Ausgabe. Wird die PWM Ausgabe gestoppt, werden die Ausgangspegel auf LOW gesetzt werden.

Listing 5.4: Python SoftwarePWM Beispiel

```
import time
import softpwm

a = softpwm.softpwm()
i,j = 0

while(True):
  while (i <= 100):
    for pin in range(0,15):
      a.set(pin, i + 1000 )
    time.sleep(0.1)
    i += 1
  time.sleep(2.0)
  while (i >= 0):
    for pin in range(0,15):
      a.set(pin, i + 1000 )
    time.sleep(0.1)
    i -= 1
  time.sleep(2.0)
```

5.10 Portabilität

Eines der Hauptargumente für die Verwendung eines Betriebssystems ist die Portabilität, also die Möglichkeit zur Wiederverwendung geschriebenen Codes auf anderen Plattformen und Architekturen. Prinzipiell sind alle aufgeführten Python-Erweiterungen auch auf anderen Systemen als dem AP7000 verwendbar, sofern die entsprechende Hardware angeboten wird, diese von Linux unterstützt wird und der Pythoninterpreter ausführbar ist. Neben den portablen socket- und pyserial-Modulen werden von folgenden Erweiterungen (zukünfige[9]) Standard-Kernelinterfaces genutzt:

- `py-smbus`
- `py-pwm`
- `py-gpio`
- `py-spi`

Software PWM Eine Ausnahme stellt die SoftwarePWM-Erweiterung `py-softpwm` dar, da das, als Backend benutzte `gpio-softpwm` Kernelmodul keine Standard-Kernelschnittstelle ist und auch nicht werden wird.
Ein *hrtimer*-basiertes SoftwarePWM-Kernelmodul für *einen* PWM-Kanal wird in der aktuellen Version des *Generic PWM API*-Patches mitgeliefert. Mit einer Einbindung des 16-Kanal-Kernelmoduls in die API wäre die Python-Erweiterung `py-softpwm` hinfällig, da `py-pwm` verwendet werden kann.

[9] Die `py-pwm` Erweiterung nutzt die *Generic PWM API*, die momentan noch kein fester Bestandteil des Kernels ist (siehe Abschnitt 3.5)

6 OpenWrt

Die Linuxdistribution OpenWrt diente als Grundlage für die Arbeit auf dem NGW100. Das folgende Kapitel soll beim Nachvollziehen der Entwicklungsarbeit helfen und gibt eine Übersicht über erstellte und geänderte Pakete sowie Kernel-Patches.

6.1 Die Distribution

OpenWrt (http://www.openwrt.org) ist eine Linuxdistribution, die speziell an DSL-Router angepaßt ist. Im Projekt wird der git-Snapshot r17602 der aktuellen Version 8.09 'kamikaze' verwendet.

Mit dem wireless Router WRT54GL verletzte die Firma Linksys (Cisco Systems) die Lizenzbedingungen der GPL. Das Resultat war die Freigabe des Quellcodes der - auf Opensource Software basierten - Firmware und damit der Beginn von OpenWrt. [1]

> Zunächst unterstützte OpenWrt nur die Linksys WRT54G-Serie, inzwischen läuft OpenWrt auf einer Vielzahl von Routern der Unternehmen Linksys, ALLNET, ASUS, Belkin, Buffalo, Microsoft, FON, Netgear und Siemens. An der Portierung auf weitere Plattformen wie die AVM Fritz!Box wird gearbeitet.[2]

Für die unterstützten Geräte kann ein fertiges Firmware-Image heruntergeladen und als Ersatz für die Originalfirmware auf dem eigenen Router installiert werden. In der aktuellen Version ist eine Weboberfläche enthalten, mit der grundlegende und erweiterte Routereinstellungen getätigt werden können.
Über einen SSH-Login kann man das System auch kommandozeilenbasiert konfigurieren und ist dabei nicht durch die Masken der Weboberfläche eingegrenzt. Softwarepakete können mittels der *ipkg* Paketverwaltung hinzugefügt und entfernt werden.

6.2 OpenWrt Entwicklung

Das Buildsystem von OpenWrt ist - vergleichbar mit Buildroot [3] und OpenEmbedded [4] - eine CrossCompile Umgebung, mit der Rootdateisysteme (*root filesystems*) und *firmware images* für verschiedene Architekturen erstellt werden können.

cross compilation Da die Ressourcen auf einem Mikrocontrollersystem selten ausreichen, um die Firmware dort nativ in angemessener Zeit (Rechenleistung) bzw. überhaupt (verfügbarer Speicher) zu kompilieren, wird diese Aufgabe im allgemeinem auf einem leistungsstarken, separatem Computer, dem Entwicklungsrechner (development host) durchgeführt.
Allerdings ist das Kompilieren einer Mikrocontroller-Firmware auf einem Desktoprechner nicht ohne weiteres möglich. Wird ein Programm für eine Architektur A auf einer Maschine der Architektur B übersetzt und gelinkt, spricht man von 'cross compilation'. Es wird eine 'cross compiler

[1] Quelle: http://www.linux-magazin.de/NEWS/Linksys-Router-WRT160-NL
[2] Zitat von http://de.wikipedia.org/wiki/OpenWrt
[3] http://buildroot.net
[4] http://www.openembedded.net

toolchain' benötigt, die aus Kernel-Headers, C-Bibliothek, Compiler, Assembler und Linker besteht.[5]

Patches Oftmals gibt es Probleme und Inkompatibilitäten, wenn Programme für eine neue Architektur übersetzt oder dort ausgeführt werden sollen. Weiterhin können Optimierungen und architekturunabhängige Bugfixes sinvoll bzw. nötig sein.
Bei der Abarbeitung von Makefiles muß außerdem berücksichtigt werden, daß eventuell gewisse Tests nicht durchführbar sind.
Es werden dann Patches benötigt, um Quellcode und Makefiles an die Besonderheiten des Zielsystems anzupassen.

Das Buildsystem von OpenWrt ist eine Sammlung von Makefiles, Skripts und Patches mit folgenden Features:

- dialogbasierte Konfiguration
- Verwaltung von Paketrepositories
- Toolchainerstellung
- ipkg Paketerstellung
- Erstellen von firmware images

6.2.1 unterstützte Plattformen

Die Auswahl der Zielplattform findet im Konfigurationsmenü statt. Auch die AVR32 Architektur wird mit einem target für den NGW100 unterstützt. Folgende, weitere Profile - sortiert nach Architekturen - werden angeboten:

arm: at91, gemini, goldfish, iop32x, kirkwood, orion, pxa, s3c24xx

armeb: ixp4xx

cris: etrax

i386: olpc, rdc, x86

mips: adm5120/router_be, amazon, ar71xx, atheros, brcm63xx, ifxmips, octeon, sibyte

mipsel: adm5120/router_le, ar7, au1000, brcom-2.4, brcm47xx, cobalt, ramips, rb532

m68k: coldfire

powerpc: ppc40x, ppc44x, mpc52xx

ubicom32: ubicom32

[5] Ein alternativer Ansatz ist es, einen Emulator für die Zielplattform zu benutzen. Einen solchen gibt es für AVR32 leider noch nicht, doch wird daran im Rahmen des Debian-AVR32 Projekts (`http://avr32.debian.net`) gearbeitet.

6.2.2 Einrichten der Umgebung

Für die folgenden Aktionen wird von einem Arbeitsverzeichnis <trees> ausgegangen.

Das OpenWrt Buildsystem kann entweder als gepacktes Archiv (Releases) oder über die Versionsverwaltung (aktuelle Entwicklerversion) bezogen werden. Die gängige Versionsverwaltung für OpenWrt ist Subversion, von einem Entwickler wird aber auch ein git-Repository verwaltet.
Für folgende Kommandos muß in das Verzeichnis <trees> gewechselt werden.

```
$ git clone git://nbd.name/openwrt.git
```

Damit existiert ein Verzeichnis <trees>/openwrt, das das OpenWrt-Buildsystem beinhaltet.

6.2.3 Paketverwaltung & Repositories

Im OpenWrt Baum sind nur die nötigsten Pakete enthalten. Zur Aktivierung weiterer Paketrepositories, muß die Datei feeds.conf angepaßt werden. Im folgenden Beispiel werden die git-Version des OpenWrt-Hauptpaketrepositories packages und das pyap7k-Repository registriert. Das packages-Repository stellt eine Vielzahl von Paketen zur Verfügung, unter anderem den Pythoninterpreter mit einigen Spracherweiterungen (siehe Kapitel 6.2.4).

```
src-git packages git://nbd.name/packages.git
src-link pyap7k ../../pyap7k
```

Die erste Spalte gibt das Protokoll, die zweite den Namen, und die dritte Spalte die Bezugsquelle für das Repository an.
Der src-link Eintrag erwartet im Verzeichnis <trees> das Verzeichnis pyap7k mit entsprechender Unterverzeichnisstruktur <category>/<package>. In den <package>-Verzeichnissen befinden sich die Makefiles mit Anweisungen zu Erstellung und Installation des Pakets, sowie Quellcodedateien für Pakete, die ihre Quellen nicht aus dem Internet beziehen.

Zur Verwaltung von Paketinformationen und Repositories bietet OpenWrt das Skript scripts/feeds an, das im Abschnitt 2.1.1 der OpenWrt Dokumentation [14] dokumentiert ist.

```
$ cd <trees>/openwrt
```

`$ scripts/feeds update -a`
: aktualisiert alle, in feeds.conf eingetragenen Paketquellen, und führt dafür z.B. einen pull-request für git-Repositories aus. Eine Kopie des Repositories liegt dann im Repository-Cache feeds/tmp. Für einen src-link-Eintrag wird dort nur ein symbolischer Link erstellt.

`$ scripts/feeds install -a`
: installiert alle Paketinformationen aus allen Repositories und registriert sie mit dem Buildsystem. Es werden vom Verzeichnis package/feeds aus symbolische Links auf den Repository-Cache erstellt. Nur für installierte Paketinformationen können in der menübasierten Konfiguration (make menuconfig) die Pakete zur Erstellung und Installation ausgewählt werden.

6.2.4 OpenWrt Paket-Makefiles

Das Buildsystem für OpenWrt-Pakete basiert auf Buildroot 2[6]. In den Makefiles stehen Informationen zum Download, Paketbeschreibung und die Anweisungen zum Kompilieren und Installieren des jeweiligen Pakets. Die OpenWrt Dokumentation [14] erläutert die Erstellung von Makefiles für Autoconf- und Kernelmodul-Pakete (Abschnitt 2.1.2 und 2.1.4).

Python und OpenWrt

Im verwendeten git-Snapshot des `package`-Repositories ist Python in der Version 2.6.1 und einige Erweiterungen enthalten. Das aktuelle Release 1 von kamikaze enthält Python in Version 2.5.4.

```
pybluez/        pymysql/        python-dbus/
pycairo/        pypcap/         python-gnome-desktop2/
pyevent/        pyserial/       pyyaml/
pygobject/      pysqlite/
pygtk/          python/
```

Da OpenWrt einen speziellen Pakettyp für Pythonpakete bereitstellt, der *distutils*-basiertes Erstellen der Erweiterungen (siehe 5.2.1) unterstützt, ist das Kompilieren von Python und zugehörigen Modulen ohne Umstände möglich.
Die Pythoninstallation von OpenWrt ist - um Platz zu sparen - auf die nötigsten Erweiterungen reduziert (python-mini). Die fehlenden Bibliotheken werden in eigenen Unterpaketen zur Installation angeboten.

6.3 geänderte Pakete

Die geänderten Makefiles der Pakete `python` und `pyserial` liegen im temporären Repository-Cache von OpenWrt [7] für das `package`-Repository.

6.3.1 pyserial und termios.so

Die Aktivierung des pyserial Moduls erforderte eine Anpassung der Makefiles von `lang/python` und `lang/pyserial` im `packages`-Repository.
Die pyserial-Erweiterung arbeitet auf POSIX-kompatiblen Systemen, wie Linux mit der termios-Erweiterung, um auf die seriellen Ports zuzugreifen. Die in Python enthaltene Datei `termios.so`, wird zwar übersetzt und gelinkt, jedoch nicht in das Paket gepackt und dementsprechend auch nicht installiert.

Die Anweisungen zur Erstellung eines Pakets `python-termios` werden zum Python-Makefile hinzugefügt:

[6] http://buildroot.uclibc.org
[7] Repository-Cache: `<trees>/openwrt/feeds`

Listing 6.1: lang/python/Makefile Patch

```
87a88,92
> define Package/python-termios
> $(call Package/python/Default)
>  TITLE:=Python support for terminal io
>  DEPENDS+=+python-mini
> endef
372a378,381
> define PyPackage/python-termios/filespec
> +|/usr/lib/python$(PYTHON_VERSION)/lib-dynload/termios.so
> endef
>
378a388
> $(eval $(call PyPackage,python-termios))
385a396
> $(eval $(call BuildPackage,python-termios))
```

Im Makefile von `pyserial` wird die Abhängigkeit von `python-termios` definiert.

Listing 6.2: lang/pyserial/Makefile Patch

```
27c27
<   DEPENDS:=+python-mini
---
>   DEPENDS:=+python-mini +python-termios
```

6.4 pyap7k Repository

Das `pyap7k`-Repository enthält die, für das Projekt erstellten OpenWrt-Pakete. Sie sind in die Kategorien `lang` und `utils` unterteilt. In der `lang`-Kategorie sind die Python-Erweiterungen `py-smbus`, sowie die übrigen selbst erstellten Module inklusive Quellcode zu finden. Die `utils`-Kategorie enthält alles weitere, wie Testprogramme und das `gpio-softpwm` Kernelmodul inklusive Quellcode. Die pyap7k-Repository befindet sich auf der CDROM in `trees/pyap7k`.

6.4.1 lang - Kategorie

`py-gpio`: Python GPIO Erweiterung (sysfs interface)

`py-pwm`: Python PWM Erweiterung (sysfs interface)

`py-smbus`: Python I2C/SMBus Erweiterung (device file interface)

`py-softpwm`: Python Software PWM Erweiterung (device file interface)

`py-spi`: Python SPI Erweiterung (device file interface)

6.4.2 utils - Kategorie

`can-modules`: Makefile für die SocketCAN - Kernelmodule

`can-utils`: Makefile für die SocketCAN Utilities [8]

`can-test`: Makefile für die SocketCAN Testprogramme [8]

`gpio-softpwm`: Makefile und Quellcode für das Software PWM Kernelmodul

`interfacetest`: Makefile und Quellcode für zwei C-Programme zum Test der SPI- und I2C-Schnittstellen

`iproute2_can`: Eine Kopie des iproute2 [9] - Makefiles von OpenWrt, das Patches für SocketCAN Unterstützung enthält. Es stellt folgende Pakete in der Kategorie `Network` bereit:

- ip_can: Routing control utility
- tc_can: Traffic control utility
- genl_can: General netlink utility frontend

`spidev-test_avr32`: Makefile und Quellcode des spidev-test Programms aus der Kernel-Dokumentation [10], angepaßt für atmel_spi

6.5 Kernel Patches

Zusätzlich, zu den beiden vorhandenen avr32-spezifischen Patches [11] wurden folgende Kernelpatches in `<trees>/openwrt/target/linux/avr32/patches` hinzugefügt:

`003_atmelspi_ngw100.patch`: Aktivierung von SPI-0 und PWM-Kanälen; Deaktivierung von USB0 und ETH1[12] in `arch/avr32/boards/atngw100/setup.c`

`102_gpiolib-allow-poll-2-on-gpio-value.patch`: Unterstützung des poll(2)-Befehls auf dem `value`-Attribut der GPIO *sysfs*-Schnittstelle

`120_atmel_pwm_c.patch`: Anpassung der Datei `drivers/misc/atmel_pwm.c` an die *GENERIC PWM API*

`121_pwm_c.patch`: Die Datei `drivers/pwm/pwm.c` der *GENERIC PWM API*

`122_pwm_h.patch`: Die Datei `drivers/pwm/pwm.h` der *GENERIC PWM API*

`123_atmel_pwm_Makefile.patch`: Anpassungen an die *GENERIC PWM API* der Datei `drivers/misc/Makefile`

[8] hier liegt möglicherweise noch ein crosscompile - Problem vor
[9] `http://linux-net.osdl.org/index.php/Iproute2`
[10] Documentation/spi/spidev_test.c
[11] Behebung eines Konflikts der USART-Einstellungsflags und Einstellung der Flashmap für das OpenWrt-Rootdateisystem
[12] Im Projekt nicht benötigt

`124_at32ap700x_c.patch`: Anpassungen an die *GENERIC PWM API* der Datei `arch/avr32/mach-at32ap/at32ap700x.c`

`125_socketcan.patch`: SocketCAN-Patch

`126_mcp251x_setup.patch`: Aktivierung des MCP2515 CAN-Controllers an der SPI-Schnittstelle in `arch/avr32/boards/atngw100/setup.c`

7 Zusammenfassung, Ausblick

7.1 Zusammenfassung

Problemstellung Die Verwendung eines Betriebssystems auf einem Mikrocontroller bietet eine Fülle neuer Möglichkeiten. Die hardwarenahe Programmierung verkompliziert sich jedoch durch Trennung des Speichers in Kernelspace und Userspace, da der Programmierer sich an die Vorgaben des Betriebssystems zur Hardwareansteuerung halten muß. Dies erschwert den Einstieg für Entwickler, die keine Erfahrung mit den I/O-Mechanismen des Linuxkernels haben.

Ziele Die Grundlage dieser Arbeit bildet eine Idee von Prof. Dr. Hubert Högl: Es sollen Erweiterungen für die einsteigerfreundliche Skriptsprache *Python* bereitgestellt werden, um dem Pythonprogrammierer Schnittstellen zu verschiedenen I/O-Funktionalitäten eines linuxbasierten Mikrocontrollers zu bieten. Vorrangig gefordert waren Erweiterungsmodule zur Kontrolle über die GPIO-Pins, die Peripheriebusse I2C und SPI, sowie die UART-, PWM- und Ethernet-Einheiten. Die Programmierung mit einer Skriptsprache ist außerdem sehr komfortabel, da das Programm nicht nach jeder Änderung neu übersetzt werden braucht.
Die Arbeit sollte auf dem Atmel NGW100 Evaluation Board durchgeführt werden, auf dem ein 32Bit AP7000 Mikrocontroller verbaut ist (auch von Atmel), der mit 140MHz arbeitet.
Ein Manko des AP7000 ist die geringe Anzahl von nur vier bereitgestellten PWM-Kanälen. PWM wird häufig zur Ansteuerung von Motoren genutzt. So bestand ein Ziel der Arbeit auch darin, weitere PWM-Kanäle durch Softwareemulation über GPIO-Pins bereitzustellen.

Vorgehen An erster Stelle stand die Einarbeitung in die Übertragungsmechanismen von UART, SPI, I2C und PWM, sowie die Untersuchung der entsprechenden Peripherieeinheiten des AP7000 (Kap. 2).
Es folgte die Analyse der Unterstützung des AP7000 und seiner Peripherie durch den Linuxkernel (Kap. 3). Hierbei stellte sich heraus, daß Anpassungen der PWM-Infrastruktur und eine kleine Erweiterung der GPIO-Bibliothek des Kernels nötig waren. Für beides wurden Patches im Internet gefunden.
Der Entwurf des SoftwarePWM-Kernelmoduls (`gpio-softpwm`) verlangte eine Auseinandersetzung mit der Echtzeitfähigkeit des Standard-Linuxkerns. Bei der Implementierung wurde auf die Benutzung eines hochauflösenden Timers (hrtimer) zurückgegriffen (Kap. 4).
Die Programmierung der Pythonerweiterungen erfolgte in *C* unter Verwendung der Python-C/API einerseits und der Kernelschnittstellen andererseits.
Zur Erstellung eigener Pakete für die verwendete Linuxdistribution *OpenWrt* war eine Einarbeitung in das OpenWrt-Makefilesystem notwendig (Kap. 6).

Das Pythonmodul zur Ansteuerung von SMBus-kompatiblen I2C-Bausteinen (`py-smbus`) wurde während der Recherche beim lm_sensors-Projekt gefunden. Dieses Modul konnte teilweise als Vorlage für die selbst erstellten Module `py-spi`, `py-pwm`, `py-softpwm` und `py-gpio` verwendet werden.

Ergebnisse Im Rahmen dieser Arbeit wurde das Buildsystem von OpenWrt konfiguriert und damit ein *Rootdateisystem* für das AP7000 basierte *NGW100 Evaluation Board* erstellt. Dieses beinhaltet den Pythoninterpreter und die Erweiterungsmodule zur Ansteuerung der UART-, SPI-,

I2C[1]-, GPIO- und PWM-Hardwareeinheiten. Ebenso ist das `gpio-softpwm`-Kernelmodul inklusive Pythonschnittstelle enthalten. Zusätzlich wurde ein CAN-Controller an den SPI-Bus des NGW100 angeschlossen und mittels SocketCAN aktiviert.

Erstellte Module Mit den gesammelten und selbst erstellten Python-Modulen war ein Zugriff auf die Mikrocontroller-typischen Schnittstellen, wie GPIO, I2C und SPI möglich. Es konnten verschiedene Peripheriegeräte angesprochen und die PWM-Ausgabe über die Hardware-Einheit und das SoftwarePWM-Modul gesteuert werden. Die Erweiterungen haben Prototypen-Status, wurden also **nicht** ausgiebig getestet und enthalten vermutlich noch einige Fehler. Auch ist eine Untersuchung der Latenzzeiten (z.B. beim Setzen von GPIO-Werten) noch nicht erfolgt. Die I2C-, SPI- und GPIO-Module wurden mit einigen einfachen Schaltungen getestet.
Die Signalqualität des `gpio-softpwm`-Kerneltreibers wurde mit einem Logic Analyzer untersucht, mit dem die nanosekundengenaue Aufzeichnung logischer Signale möglich war.

7.2 Ausblick

Stabilitätstests Die erstellten Module stellen *Prototypen* dar, die zeigen sollen, daß von Python aus die Ansteuerung von Peripherie getätigt werden kann. Ein **ausführlicher Test** der Erweiterungen, vor allem hinsichtlich des zeitlichen Verhaltens steht aus.

Vorschläge zur Weiterentwicklung der Schnittstellen

- das gpio-softpwm Kernelmodul:
  - Einbindung in die *Generic PWM API*
  - Optimierung der `update_registers` Interrupt Routine
  - Optimierung der `calc_changes` Funktion, die das Muster neu berechnet, das in `update_registers` verwendet wird
- die py-smbus Pythonerweiterung:
  - Implementierung weiterer I2C-Kommandos
- die py-pwm Pythonerweiterung:
  - py-pwm übergibt die zu setzenden Werte ohne weitere Überprüfung an die *Generic PWM API sysfs*-Schnittstelle. Es könnten eine Überprüfung der Werte und korrekte Fehlerbehandlung implementiert werden.
- die py-gpio Pythonerweiterung:
  - Es werden nicht alle Möglichkeiten genutzt, die der PIO-Controller anbietet, wie z.B. das An- und Abschalten des glitch-filters oder des pull-up Widerstands. Um diese anzubieten, könnte eventuell der unportable `gpio-dev`-Kerneltreiber von Atmel genutzt werden. Alternativ könnte die *sysfs*-Funktionalität für GPIO's erweitert werden.

[1] Es wird nicht die echte I2C-Hardwareeinheit, sondern der `i2c-gpio` Treiber genutzt.

- die py-spi Pythonerweiterung:
  - `msg2`: Die Größe der Übertragungseinheiten ist momentan auf 8 Bit festgelegt, kann aber in Zukunft noch erweitert werden. Möglicherweise können diese Funktionen dann als speicherschonende `msg8bit2`-Funktionen weiterbestehen.
  - `msg`: Um eine feinere Kontrolle über die Übertragungen anzubieten, könnte eine weiterentwickelte `msg`-Funktion eine Liste von `spi_msg`-Pythonobjekten übergeben bekommen. Der neu bereitzustellende Python-Typ `spi_msg` könnte folgende Elemente beinhalten:
    * Liste der Werte (Python Integer)
    * `speed_hz` - Angabe
    * `bpw` - Angabe
    * `delay` - Angabe
    * `cs_change` - Angabe

Die Entwicklung weiterer Module, z.B. für die *1-wire*-Schnittstelle, die USB-Schnittstelle oder das *Image Sensor Interface* (kombiniert mit Pythonbibliotheken zur Bildverarbeitung wie z.B. numpy), könnte folgen.

Distribution Momentan wird nur ein Rootdateisystem zur Verfügung gestellt, das per NFS eingebunden werden muss (siehe README im Anhang D.1). Um einen wirklich einfachen Einstieg zu ermöglichen, sollte eine Image und eine Anleitung zum Boot über eine SD-Karte erstellt werden.

Unterstützung bei der Pythonentwicklung Die Module bieten zwar eine Grundlage für die Pythonentwicklung auf dem NGW100, zur Verwendung sind aber noch immer rudimentäre Linuxkenntnisse - etwa zum Kopieren von Dateien oder zum Ausführen von Pythonskripts - nötig.

Zur Unterstützung bei der Entwicklung von Pythonanwendungen auf dem NGW100 könnte die USB-Gadget-Funktionalität[2] folgendermaßen eingerichtet werden:

Der NGW100 verhält sich beim Start verschieden - je nachdem, ob er mit dem Entwicklungsrechner verbunden ist oder nicht. Hierfür müßte die OpenWrt-Bootsequenz entsprechend angepaßt werden.

Wird der NGW100 im "normalen" Modus gestartet, führt er ein - als Defaultskript ausgewähltes - Pythonskript aus.
Erkennt er beim Start eine Verbindung mit dem Entwicklungsrechner[3], so führt er nicht das Defaultskript aus (es könnte Fehler enthalten und das Board unansprechbar machen), sondern bietet folgende Optionen an:

- Dateitransfer über *usb-storage*
- Auswahl des Defaultskripts

[2] siehe `http://www.linux-usb.org/gadget/` - der AP7000 wird unterstützt
[3] bzw. mit einer speziellen Software, die auf dem Entwicklungsrechner läuft

- Starten und Stoppen von Pythonskripts - bei Übertragung der Ausgabe zum Entwicklungsrechner (→ Debugging)
- Konfiguration des NGW100
- Python Prompt (à la *ipython*[4])

Während der erste Punkt mit jedem Rechner funktioniert, der USB-Sticks unterstützt, wird für die restlichen Punkte entweder die entsprechende Software auf dem Entwicklungsrechner benötigt oder der NGW100 muß diese Möglichkeiten über die serielle Konsole oder eine Weboberfläche anbieten. Dabei könnte die serielle Konsole per `gadget_serial`[5] auf der Clientseite und `usb_serial`[6] auf der Hostseite auch über die USB-Verbindung laufen.

[4] `http://ipython.scipy.org`
[5] Documentation/usb/gadget_serial.txt
[6] Documentation/usb/usb-serial.txt

A Anhang: SocketCAN

Der CAN-Bus wurde 1983 von Bosch für die Vernetzung von Steuergeräten in Automobilen entwickelt und 1987 zusammen mit Intel vorgestellt. [1] Zu den Eigenschaften des CAN Bus zählen Robustheit/Fehlertoleranz (der CAN Bus kann selbst bei Wegfallen einer Leitung weiterarbeiten) und einfacher Hardwareaufbau.

A.1 Socket-CAN

Die Unterstützung für CAN - Hardware unter Linux war bisher wenig strukturiert. So gab es viele unterschiedlich aufgebaute Treiber, die über verschiedene Kernel-/Userspace Schnittstellen angesprochen wurden. Um einen einheitlichen Zugriff auf verschiedene CAN-Controller zu ermöglichen, hat Oliver Hartkopp das SocketCAN-Projekt ins Leben gerufen. Die Schnittstelle zum Userspace stellt die BSD-Socket API dar.

> 1. What is Socket CAN
>
> The socketcan package is an implementation of CAN protocols (Controller Area Network) for Linux. CAN is a networking technology which has wide-spread use in automation, embedded devices, and automotive fields. While there have been other CAN implementations for Linux based on character devices, Socket CAN uses the Berkeley socket API, the Linux network stack and implements the CAN device drivers as network interfaces. The CAN socket API has been designed as similar as possible to the TCP/IP protocols to allow programmers, familiar with network programming, to easily learn how to use CAN sockets.[2]

A.2 Kernelunterstützung

Socket-CAN findet schrittweise Einzug in den Linux-Kernel. In der verwendeten Kernelversion 2.6.30.7 ist die Protokoll- und Adressfamilienunterstützung für CAN Sockets (AF_CAN=PF_CAN=29) eingepflegt. Weiterhin steht der **vcan** Treiber für virtuelle CAN Geräte zur Verfügung.

A.2.1 MCP2515

Der MCP2515 ist ein CAN Controller von Microchip Technology Inc., der über die SPI-Schnittstelle angesprochen wird. Eine weitere Leitung wird als Interruptleitung benutzt. Zur Aktivierung des Chips muß zuerst der Treiber compiliert werden. Da die Konfiguration von SPI-Geräten im boardspezifischen Setupcode stattfindet, wurde der Treiber nicht nachträglich und separat compiliert, sondern per Patch in den Kernelcode integriert:

Listing A.1: mcp251x Konfiguration (setup.c)

```
#include <socketcan/can/platform/mcp251x.h>
...
static int mcp251x_setup(struct spi_device *spi)
{
```

[1] Quelle: http://de.wikipedia.org/wiki/Controller_Area_Network [6]
[2] Aus der socketcan README

```
  return 0;
}

static struct mcp251x_platform_data mcp251x_info = {
        .oscillator_frequency = 8000000,
        .board_specific_setup = &mcp251x_setup,
        .model = CAN_MCP251X_MCP2515,
        .power_enable = NULL,
        .transceiver_enable = NULL,
};

static struct spi_board_info spi0_board_info[] __initdata = {
  ...
        {
                .modalias      = "mcp251x",
                .platform_data = &mcp251x_info,
                .irq           = AT32_EXTINT(0),
                .max_speed_hz  = 2*1000*1000,
                .chip_select   = 2,
        },
};
```

In den Bootnachrichten des Kernels kann man beobachten, wie nach der Unterstützung für Internetschnittstellen, der CAN-Kern geladen wird.

```
NET: Registered protocol family 17
NET: Registered protocol family 15
can: controller area network core (rev 20090105 abi 8)
```

Ist der MCP2515 richtig angeschlossen (SPI Port 0, Chip Select 2) und die Interruptleitung mit GPIO #59 (EXTINT2) verbunden, so wird er richtig erkannt.

```
CAN device driver interface
mcp251x spi0.2: mcp251x_hw_probe: 0x80 - 0x07
mcp251x spi0.2: probed
```

A.3 Userspace

Die Aktivierung von CAN-Geräten kann über eine *sysfs*-Schnittstelle erfolgen (deprecated). Die empfohlene Alternative ist die Verwendung des, um CAN-Unterstützung erweiterten und im **iproute2**-Paket enthaltenden Werkzeugs **ip**. (siehe 6.4.2)

```
ip link set can0 up type can bitrate 200000
```

Ein Blick in die Info-Meldungen des Kernels zeigt die erfolgreiche Aktivierung von der can0-Schnittstelle an:

```
mcp251x spi0.2: mcp251x_do_set_bittiming: BRP = 1, PropSeg = 8,
 PS1 = 8, PS2 = 3, SJW = 1
```

A.3.1 Link Status

Ein CAN-Knoten kann drei verschiedene Zustände einnehmen: ERROR-ACTIVE, ERROR-PASSIVE und BUS-OFF. ERROR-ACTIVE ist der initiale Zustand, in den der Knoten nach einem Reset wechselt. Tritt eine Häufung von Sende- oder Empfangsfehlern auf (≥ 127), wird in den passiven Modus gewechselt. Treten weitere Sendefehler auf (≥ 255), geht der Controller in den BUS-OFF Modus, aus dem nur ein Reset fährt. [3]

```
root@pyngw:/$ ip -details link show can0
3: can0: <NOARP,UP,LOWER_UP,40000> mtu 16 qdisc pfifo_fast state
   UNKNOWN qlen 10
    link/[280]
    can state ERROR-ACTIVE restart-ms 0
    bitrate 200000 sample-point 0.850
    tq 250 prop-seg 8 phase-seg1 8 phase-seg2 3 sjw 1
    : tseg1 3..16 tseg2 2..8 sjw 1..4 brp 1..64 brp-inc 1
    clock 4000000
```

[3] http://www.softing.com/home/en/industrial-automation/products/can-bus/more-can-bus/error-handling/error-states.php?navanchor=3010510

B Anhang: Pin Belegung

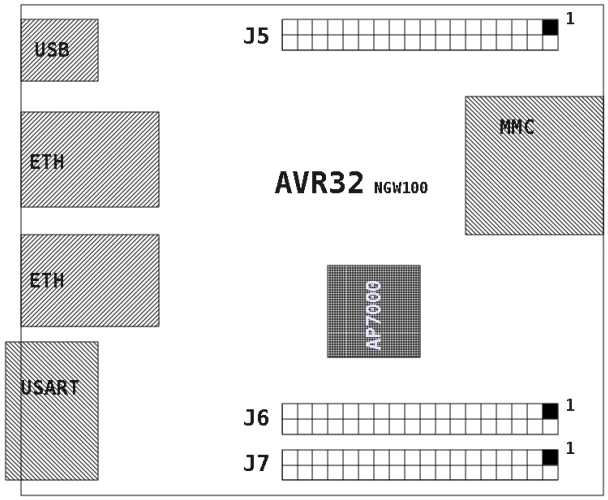

Abbildung B.1: NGW100 Schema

Tabelle B.1: J5, J6 Pinbelegung

J5				J6			
function	pin	pin	function	function	pin	pin	function
3.3V	1	2	GND	3.3V	1	2	GND
SPI0: MISO	3	4	SPI0: MOSI	3.3V	3	4	GND
SPI0: SCK	5	6	SPI0: NPCS0	N.C.	5	6	N.C.
SPI0: NPCS1	7	8	SPI0: NPCS2	I2C1: SDA	7	8	I2C1: SCL
I2C0: SDA	9	10	I2C0: SCL	GPIO: 41	9	10	GPIO: 42
USART0: RXD	11	12	USART0: TXD	GPIO: 43	11	12	GPIO: 44
PWM: C2	13	14	PWM: C3	GPIO: 45	13	14	GPIO: 46
GPIO: 23	15	16	GPIO: 24	USART3: CTS	15	16	USART3: RTS
GPIO: 25	17	18	GPIO: 26	USART3: TXD	17	18	USART3: RXD
SPI1: NPCS3	19	20	PWM: C0	USART3: CLK	19	20	GPIO: 52
PWM: C1	21	22	GPIO: 30	GPIO: 53	21	22	GPIO: 54
GPIO: 31	23	24	SPI1: MISO	GPIO: 55	23	24	GPIO: 56
SPI1: MOSI	25	26	SPI1: NPCS0	EXTINT 0, GPIO: 57	25	26	EXTINT 1, GPIO: 58
SPI1: NPCS1	27	28	SPI1: NPCS2	EXTINT 2, GPIO: 59	27	28	WAKE_N
SPI1: SCK	29	30	GPIO: 38	N.C.	29	30	N.C.
SPI0: NPCS3	31	32	N.C.	N.C.	31	32	N.C.
3.3V	33	34	GND	3.3V	33	34	GND
3.3V	35	36	GND	3.3V	35	36	GND

Tabelle B.2: J7 Pinbelegung

J7			
function	pin	pin	function
SPWM: 2	1	2	SPWM: 3
SPWM: 4	3	4	SPWM: 5
SPWM: 6	5	6	GPIO: 95
GPIO: 96	7	8	GPIO: 97
SPWM: 7	9	10	SPWM: 8
SPWM: 9	11	12	SPWM: 10
SPWM: 11	13	14	GPIO: 103
GPIO: 104	15	16	GPIO: 105
SPWM: 12	17	18	SPWM: 13
SPWM: 14	19	20	SPWM: 15
GPIO: 145	21	22	GPIO: 146
GPIO: 112	23	24	GPIO: 113
SPWM: 0	25	26	SPWM: 1
GPIO: 84	27	28	GPIO: 85
GPIO: 86	29	30	N.C.
N.C.	31	32	N.C.
3.3V	33	34	GND
3.3V	35	36	GND

Die Datenblätter aller Chips sind auf der beiliegenden CDROM zu finden. Die Beispielskripts befinden sich im erstellten *rootfs* in `/pyap7k/py-spi` und `/pyap7k/py-smbus`. Folgende SPI- und I2C-Peripheriegeräte konnten erfolgreich angesprochen werden:

B.1 SPI

- MCP3304: Analog-Digital Konverter
- DS1305: Real Time Clock

DS1305 Die DS1305 RTC benötigt für den Takt einen 32.768kHz Oszillator, der an X1 und X2 angeschlossen wird. Es kann zwischen den Standard-SPI- (SERMODE=Vcc) und 3wire-Übertragungsmodi (SERMODE=GND) gewählt werden.

Tabelle B.3: DS1305 - Pinbelegung

3.3V	Vcc2	1		16	Vcc1	GND
GND	Vbat	2		15	PF#	
32k768osz1	X1	3	DS1305	14	VccIF	3.3V
32k768osz2	X2	4		13	SD0	SPI1 - MISO
	N.C.	5		12	SDI	SPI1 - MOSI
8k7 zu 3.3V	INT0#	6		11	SCLK	SPI1 - SCK
	INT1#	7		10	CE	SPI1 - NPCS1
GND	GND	8		9	SERMODE	GPIO59:PB27

MCP3304 Am MCP3304 8-Kanal Analog Digital Konverter war ein einfacher Spannungsteiler angeschlossen.

Tabelle B.4: MCP3304 - Pinbelegung

SPANNUNGS	CH0	1		16	Vdd	3.3V
-	CH1	2		15	Vref	3.3V
TEILER	CH2	3	MCP3304	14	Agnd	GND
	CH3	4		13	CLK	SPI1 - SCL
	CH4	5		12	DO	SPI1 - MISO
	CH5	6		11	DI	SPI1 - MOSI
	CH6	7		10	CS#	SPI1 - NPCS1
	CH7	8		9	Dgnd	GND

B.2 I2C

- PCA9555: GPIO Expander
- DS1803: digitales Potentiometer
- Tiny24: onboard Controller

DS1803 Der DS1803 Baustein bietet zwei digitale Potentiometer an (H0,L0,W0 und H1,L1,W1). Das erste Potentiometer (0) ist in eine einfache Reihenschaltung (LED+Vorwiderstand) eingebunden, so daß die Änderung des Widerstands an der Helligkeit der LED beobachtet werden kann.

Tabelle B.5: DS1803 - Pinbelegung

	N.C.	1		16	Vcc	3.3V
	H1	2		15	N.C.	
	L1	3	DS1803	14	H0	
	W1	4		13	L0	LED
GND	A2	5		12	W0	R
GND	A1	6		11	N.C.	
GND	A0	7		10	SDA	I2C0 - SDA
GND	GND	8		9	SCL	I2C0 - SCL

PCA9555 Der PCA9555 bietet mit 2x8 GPIO's, die als Ein- und Ausgabepins dienen können. Weiterhin besteht die Möglichkeit, den Host über eine SMBALERT#-Leitung (Pin 1 des Chips, verbunden mit GPIO #59 des NGW100) über Änderungen der, als Eingang konfigurierten Pins zu benachrichtigen.

Tabelle B.6: PCA9555 - Pinbelegung

GPIO59:PB27	INT#	1		24	Vdd	3.3V
GND	A1	2		23	SDA	I2C0 - SDA
GND	A2	3		22	SCL	I2C0 - SCL
	IO0_0	4	PCA9555	21	A0	GND
	IO0_1	5		20	IO1_7	
	IO0_2	6		19	IO1_6	
	IO0_3	7		18	IO1_5	
	IO0_4	8		17	IO1_4	
	IO0_5	9		16	IO1_3	
	IO0_6	10		15	IO1_2	
	IO0_7	11		14	IO1_1	
3.3V	Vss	12		13	IO1_0	

C Anhang: Listings

Listing C.1: gpio-buttons Setupcode

```
#if defined(CONFIG_KEYBOARD_GPIO) || defined(
    CONFIG_KEYBOARD_GPIO_MODULE)
#include <linux/input.h>
#include <linux/gpio_keys.h>

/*
 * GPIO Keys
 */

static struct gpio_keys_button at32_buttons[] = {
  {
    .code           = BTN_1,
    .gpio           = GPIO_PIN_PA(23),
    .active_low     = 1,
    .desc           = "Button 1",
    .type   = EV_KEY,
    .wakeup         = 1,
    .debounce_interval = 20,
  },
  {
    .code           = BTN_2,
    .gpio           = GPIO_PIN_PA(24),
    .active_low     = 1,
    .desc           = "Button 2",
    .type   = EV_KEY,
    .wakeup         = 1,
    .debounce_interval = 20,
  },
  {
    .code           = BTN_3,
    .gpio           = GPIO_PIN_PA(25),
    .active_low     = 1,
    .desc           = "Button 3",
    .type   = EV_KEY,
    .wakeup         = 1,
    .debounce_interval = 20,
  },
  {
    .code           = BTN_4,
    .gpio           = GPIO_PIN_PA(26),
    .active_low     = 1,
    .desc           = "Button 4",
    .type   = EV_KEY,
    .wakeup         = 1,
    .debounce_interval = 20,
  }
};

static struct gpio_keys_platform_data at32_button_data = {
```

```
        .buttons        = at32_buttons,
        .nbuttons       = ARRAY_SIZE(at32_buttons),
};

static struct platform_device at32_button_device = {
        .name           = "gpio-keys",
        .id             = -1,
        .num_resources  = 0,
        .dev            = {
                .platform_data  = &at32_button_data,
        }
};

static void at32_add_device_buttons(struct gpio_keys_platform_data *
   data)
{
  int i;

  for (i=0; i < data->nbuttons; i++) {
    at32_select_gpio(data->buttons[i].gpio, AT32_GPIOF_DEGLITCH);
  }

        platform_device_register(&at32_button_device);
}
#endif
```

D Anhang: README

Die folgende README Datei liegt im Rootverzeichnis der CDROM.

Listing D.1: pyngw README

```
pyngw: PYTHON ON THE AVR32


-------------------------------------------------------------------------------
0) Introduction

This README provides information on how to get the Atmel NGW100 running the
python-enabled OpenWrt root filesystem (pyngw) over NFS.

The NGW100 hosts an Atmel AP7000 uC which offers several I/O-Ports for
peripheral communication. The rootfs contains python extensions, that allow to
use some of the uC's (low-level) I/O-functionality from Python.


-------------------------------------------------------------------------------
I) CDROM contents

The directory tree contains all the software and documentation that is used
in the pyap7k project.

thesis.pdf:      thesis paper

short.pdf:       thesis paper - abstract only

arbeit/thesis:   thesis paper's LaTex sourcecode

arbeit/drawings: drawings in OpenDocument Graphic format

arbeit/mypins.ods:
                 pin layout in Opendocument Spreadsheet format

documentation:   documentation that is referred to in the thesis paper and
                 additional documents

download/code:   example C-code from internet ressources

download/packages:
                 i2c-tools, lm_sensors, socketCAN, tinyCAN

trees/openwrt.tar.bz2:
                 git-Snapshot of OpenWRT 8.09r1 'kamikaze' development tree to
                 build kernel and root filesystem

                 * attention *
                 do not update the 'packages' package repository; it contains
                 adapted versions of python and pyserial Makefiles which would
                 get lost

trees/pyap7k:    pyap7k package repository (including C-sourcecodes)

trees/pyap7k.tar.bz2:
                 pyap7k package repository

trees/vusb_ngw100:
```

```
                sourcecode for Atmel Mega8 firmware that features SPI and I2C
                interfacetest with VUSB (www.obdev.at) library usage
                documentation can be found in
                                documentation/hoegl/Wiki_G216_Merkle.tar.gz
```

```
II) Building

It is assumed that you are running an up-to-date Linux Distribution like
Ubuntu 9.xx or Debian >=5.0. I'm currently using Ubuntu 9.10, but other,
non-Debian distributions should work as well (like Fedora Core, OpenSUSE,
 Gentoo, etc). Well, package names may differ.

II.A) hardware requirements

The time needed for the build may vary widely depending on your hardware setup,
but any modern Giga[Hertz|Byte]-workstation should do the job.

! note: there are about 2.5 Gigabytes of free diskspace needed for the build

II.B) software requirements

If the makeprocess complains about missing tools or header files, install them.

You can make make to test for the requirements by typing '$ make prereq'

The following packages are needed to be installed on a Ubuntu 9.10:

 * flex
 * g++
 * gcc
 * gawk
 * git-core
 * libz-dev
 * ncurses-dev
 * patch
 * subversion
 * unzip
 * wget

II.C) make

Create a working directory on your harddrive. Let's call it <trees>.

Unpack the files openwrt.tar.bz2 and pyap7k.tar.bz2 from the trees/
directory on the CDROM to the <trees> directory on your harddrive.

Change to the directory '<trees>/openwrt' and type 'make' to build
or 'make menuconfig' to view and change the build settings.

! note: you must be logged in as none-root for building

After the successful build the root filesystem is in
  <trees>/openwrt/build_dir/target-avr32_uClibc-0.9.30.1/rootfs
a kernel should be found in
  <trees>/openwrt/bin/openwrt-avr32-uImage
```

```
III) Preparing the NFS

! note: you need to must be logged in as root (or use sudo) for creating
        symbolic links in /home, editing  /etc/exports and restarting the
        nfs-kernel-server service

Assumption: CWD = <trees>/openwrt

1) create a link from
        build_dir/target-avr32_uClibc-0.9.30.1/rootfs
  to
        /home/ngw100

  $ sudo ln -s build_dir/target-avr32_uClibc-0.9.30.1/rootfs /home/ngw100

2) copy kernel image into rootfs
  $ cp bin/openwrt-avr32-uImage /home/ngw100/uImage

3) export /home/ngw100 via NFS

! note: you may need to install the nfs-kernel-server package if you haven't
        done so already ($ sudo apt-get install nfs-kernel-server)

  * add the following line to /etc/exports

  /home/ngw100  192.168.78.0/24(rw,sync,no_subtree_check,no_root_squash)

! note: replace '192.168.78.0/24' with a mask appropriate for your network
    setup
        see 'man 5 exports' for details

  * restart NFS server

  $ sudo su -
  $ echo "/home/ngw100/192.168.78.0/24(rw,sync,no_subtree_check,no_root_squash)
      " >> /etc/exports
  $ service nfs-kernel-server restart
  $ exit
```

```
IV) Starting the NGW100

IV.A) Uboot version

Uboot has to support boot (especialy fetching the kernel) over NFS for the
following steps. Older versions of Uboot do not support this. Upgrading Uboot
is easily done with Uboot itself; follow the Uboot documentation for this.
The image file that I used can be found in the download/ directory called
  "download/flash-upgrade-atngw100-v2008.10.uimg"

IV.B) Uboot enviroment settings

I hope you are familiar with the common uboot commands 'setenv', 'printenv',
```

```
'askenv' and so on...

change IP addresses to match your setup

nfsboot= \
  set ipaddr 192.168.78.101; \
  set bootargs root=/dev/nfs nfsroot=192.168.78.23:/home/ngw100 \
    ip=192.168.78.101::192.168.78.23:::eth0:off \
    console=ttyS0 init=/etc/preinit; \
  nfs 0x10400000 192.168.78.23:/home/ngw100/uImage; \
  bootm

bootcmd=run nfsboot
```

```
V) Running Python

Solder some circuits, write the Python scripts and put them (somewhere)
into the rootfs. Please note that py-gpio, py-spi, py-pwm, py-softpwm are in
beta status and have to be seen as _unstable_. A serial terminal can be used
for shell access. Change to the directory that contains your python scripts and
run them, for example:
  pyngw $ python gpiotest.py

For API documentation of the modules, have a look at the thesis paper.
example scripts are in /pyap7k/py-[gpio|pwm|smbus|softpwm|spi]
```

Literaturverzeichnis

[1] ATMEL CORPORATION: *AVR32100: Using the AVR32 USART*.
http://www.atmel.com/dyn/resources/prod_documents/doc32006.pdf,
CDROM: documentation/Atmel/32Bit/doc32006.pdf, 2005.

[2] ATMEL CORPORATION: *AT32AP7000 Preliminary - Revision K 09/07*.
http://www.atmel.com/dyn/resources/prod_documents/doc32003.pdf,
CDROM:documentation/Atmel/32Bit/doc32003.pdf, 2006.

[3] ATMEL CORPORATION: *AT32NGW100 schematics*.
http://www.atmel.com/dyn/resources/prod_documents/AT32NGW100_schematics.pdf,
CDROM:documentation/Atmel/32Bit/AT32NGW100_schematics.pdf, 2008.

[4] ATMEL CORPORATION: *AVR32408: AVR32 AP7 Linux GPIO driver*.
http://www.atmel.com/dyn/resources/prod_documents/doc32073.pdf,
CDROM:documentation/Atmel/32Bit/doc32073.pdf, 2008.

[5] ATMEL CORPORATION: *AVR32412: AVR32 AP7 TWI Driver*.
http://www.atmel.com/dyn/resources/prod_documents/doc32083.pdf,
CDROM:download/documentation/Atmel/32Bit/doc32083.pdf, 2008.

[6] DE.WIKIPEDIA.ORG: *Controller Area Network Wikipedia Artikel*.
http://de.wikipedia.org/wiki/Controller_Area_Network.

[7] DE.WIKIPEDIA.ORG: *Universal Asynchronous Receiver Transmitter Wikipedia Artikel*.
http://de.wikipedia.org/wiki/UART.

[8] EN.WIKIPEDIA.ORG: *I2C Wikipedia Artikel*.
http://en.wikipedia.org/wiki/I2C.

[9] GUIDO VAN ROSSUM, FRED L. DRAKE, JR., EDITOR: *Extending and Embedding Python Release 2.6.4*.
http://docs.python.org/ftp/python/doc/2.6.4/python-2.6.4-docs-pdf-a4.tar.bz2
CDROM:documentation/python/docs-pdf/extending.pdf, 2008.

[10] GUIDO VAN ROSSUM, FRED L. DRAKE, JR., EDITOR: *The Python/C API Release 2.6.4*. `http://docs.python.org/ftp/python/doc/2.6.4/python-2.6.4-docs-pdf-a4.tar.bz2`, CDROM:`documentation/python/docs-pdf/c-api.pdf`, 2008.

[11] LINUX KERNEL DEVELOPERS: *Kernel 2.6.30 Documentation*. `http://lxr.linux.no/linux+v2.6.30.7/Documentation`.

[12] MAXIM INTEGRATED PRODUCTS: *APPLICATION NOTE 476 Comparing the I2C Bus to the SMBus*. `http://www.maxim-ic.com/appnotes.cfm?appnote_number=476`, 2000.

[13] MORTON ENGEN, NTNU DET SKAPENDE UNIVERSITET: *Masters Thesis: Better Real-Time Capabilities For The AVR32 Linux Kernel*. `http://ntnu.diva-portal.org/smash/get/diva2:124182/FULLTEXT01`, 2007.

[14] OPENWRT GROUP, THE: *OpenWRT documentation*. `http://downloads.openwrt.org/docs/openwrt.pdf`, CDROM:`documentation/other/openwrt.pdf`, 2009.

[15] PHILIPS SEMICONDUCTORS: *THE I2C-BUS SPECIFICATION VERSION 2.1*. `www.nxp.com/acrobat_download2/literature/9398/39340011.pdf`, 2000.

[16] RTLINUX.NET: *RT Linux Wiki FAQ*. `http://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions`, 2009.

[17] SBS IMPLEMENTERS FORUM: *System Management Bus (SMBus) Specification Version 2.0*. , 2000.

Alle Internet-Ressourcen wurden am *05.01.2010* auf ihre Verfügbarkeit geprüft.
Die Linux Cross-Reference (LXR) [11] bietet nicht nur die Dokumentation verschiedener Kernelversionen und -releases, sondern stellt auch eine verlinkte Referenz des Kernelquellcodes bereit.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Index