Betriebssysteme
Übungen

Barry Linnert

Wintersemester 2017/18

Demonstration am Beispiel von LED

- Aufgabe/Ziel: Eine der Leuchtdioden soll leuchten.

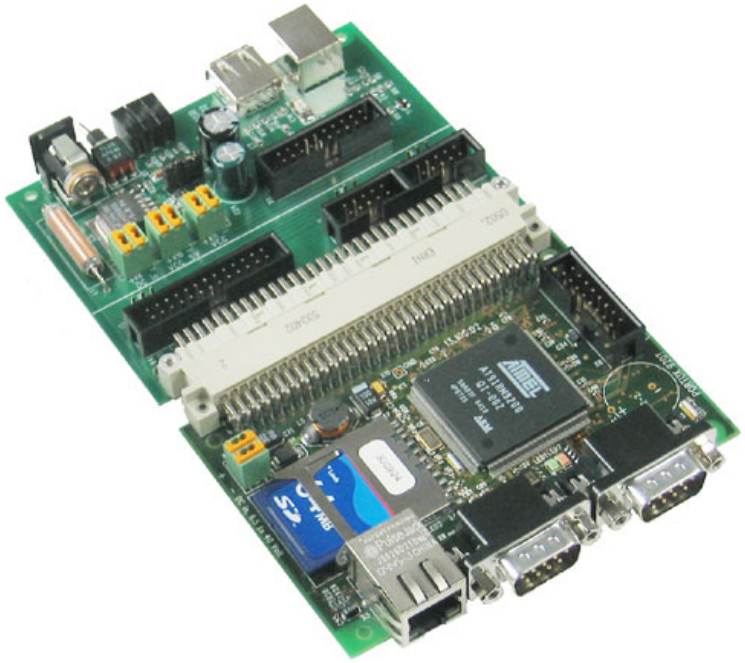
  Wie gehen wir diese Aufgabe an?

Herangehensweise

- Ansteuerung LEDs
  - Wie sind die LEDs angebunden?
  - Wie programmiere ich die Schnittstelle?
  - Welchem Protokoll muss ich folgen?
  - erst anschließend Code/Skripte schreiben
- Integration
  - Wie kommt der Code auf die Zielplattform?
  - Wohin muss der Code auf der Zielplattform?
  - Wie wird der Code gestartet?
  - erst anschließend Code/Skripte schreiben

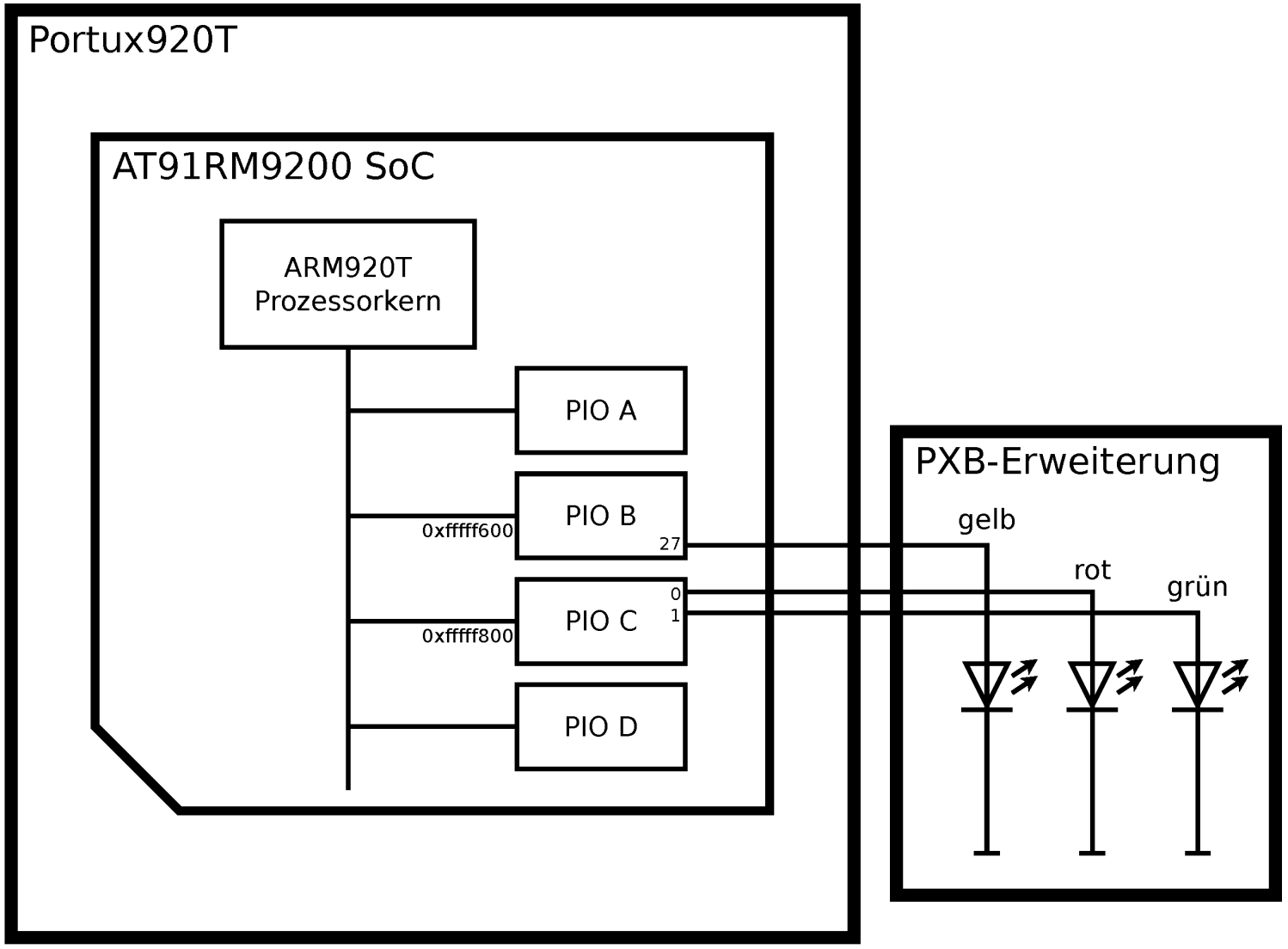
Verfügbare Dokumentation

- Handbücher zu
  - Portux920T
  - PXB-Erweiterungskarte
  - AT91RM9200
  - ARM920T
  - ARMv4



Quelle: taskit

Ergebnis der Recherche



Ein-/Ausgabe bei ARM

- Prozessorkern kommuniziert dem Rest der Welt über
  - Daten- und Adressbus
  - Interrupt-Leitungen
  - (sowie Debug- und Coprozessor-Schnittstellen)

- Peripherie wird in den Speicher eingeblendet (memory mapped I/O)
  - jedes Gerät hat Register für Konfiguration und Datenaustausch
  - Register werden in Adressraum des Prozessorkerns eingeblendet
  - Registerzugriff führt zu entsprechenden Reaktionen des Geräts
  - viele Geräte können bei Statusänderung Interrupt auslösen

Programmierung PIO

- Ein Parallel-I/O-Controller verwaltet 32 I/O-Pins
  - jeder I/O-Pin ist u. U. mehrfach belegt (Gerät A, Gerät B, oder PIO)
  - Pin ist Eingabe- oder Ausgabe-Pin
  - Ausgabe-Pin ist entweder an oder aus
  - Pin wird in verschiedenen Registern durch entsprechende Bit-Position repräsentiert
  - zu einer Einstellung gibt es meist drei Register:

    zum Setzen, zum Löschen, zum Status erfragen
- Somit muss:
  - dem PIO die Kontrolle über LED-Pin gegeben werden (via PIO_PER)
  - der LED-Pin als Ausgabe-Pin konfiguriert werden (via PIO_OER)
  - die LED schließlich angeschaltet werden (via PIO_SODR)

Leuchtdiode anschalten I

- Direkte Variante: led.c

```
#define PIOB 0xfffff600
#define YELLOW_LED (1 << 27)
#define PIO_PER 0x00
#define PIO_OER 0x10
#define PIO_SODR 0x30

static inline
void write_u32 (unsigned int addr, unsigned int val) {
    *(volatile unsigned int *)addr = val;
}

void yellow_on (void) {
    /* Initialisieren */
    write_u32 (PIOB + PIO_PER, YELLOW_LED);
    write_u32 (PIOB + PIO_OER, YELLOW_LED);
    /* Anschalten */
    write_u32 (PIOB + PIO_SODR, YELLOW_LED);
}
```

Leuchtdiode anschalten II

- Variante mit Array: led.c

```
#define PIOB 0xfffff600
#define YELLOW_LED (1 << 27)
#define PIO_PER (0x00 / 4)
#define PIO_OER (0x10 / 4)
#define PIO_SODR (0x30 / 4)

static volatile
unsigned int * const piob = (unsigned int *)PIOB;

void yellow_on(void)
{
    /* Initialisieren */
    piob[PIO_PER] = YELLOW_LED;
    piob[PIO_OER] = YELLOW_LED;
    /* Anschalten */
    piob[PIO_SODR] = YELLOW_LED;
}
```

Leuchtdiode anschalten III

- Variante mit Struktur: led.c

```
#define YELLOW_LED (1 << 27)
struct pio {
    unsigned int per;
    unsigned int unused0[3];
    unsigned int oer;
    unsigned int unused1[7];
    unsigned int sodr;
};
static volatile
struct pio * const piob = (struct pio *)PIOB;

void yellow_on(void) {
    /* Initialisieren */
    piob->per = YELLOW_LED;
    piob->oer = YELLOW_LED;
    /* Anschalten */
    piob->sodr = YELLOW_LED;
}
```

Leuchtdiode anschalten

- `YELLOW_LED` sagt mehr als `1<<27` sagt mehr als `0x08000000` sagt mehr als `134217728`.

Warum `volatile`?

- Compiler geht von gewissen Annahmen aus:
  - Speicher ändert sich nicht von selbst
  - Speicher wird von keinem anderen gelesen
- Erlaubt diverse Optimierungen, zum Beispiel:
  - bei mehrfachem Schreiben derselben Speicherstelle nur letzten Wert schreiben
  - bei mehrfachem Lesen derselben Speicherstelle nur einmal lesen
  - Zwischenspeichern von Speicherstellen in Registern
  - Umsortieren von Speicherzugriffen
- Annahmen werden durch `volatile` außer Kraft gesetzt:
  - jede Verwendung einer volatile Speicherstelle führt zu einem tatsächlichen Zugriff
  - Zugriffsreihenfolge auf volatile Speicherstellen bleibt erhalten

- (Achtung: zur Synchronisierung von Threads oder Prozessoren ist `volatile` das falsche Mittel!)

Übersetzen

- GCC + GNU Binutils als Cross-Compiler
  - Tools für Linux-Pool liegen im KVV
  - Präfix zum Unterscheiden von der normalen Variante:
    `arm-none-eabi-`

- Kompilieren mit:

```
arm-none-eabi-gcc -Wall -Wextra
-ffreestanding -mcpu=arm920t -O2 -c led.c
```

`-Wall`	mehr Warnungen
`-Wextra`	noch mehr Warnungen
`-ffreestanding`	keine „Standard-Umgebung“ annehmen
`-mcpu=arm920t`	erzeuge Code für unseren Prozessorkern
`-O2`	erzeuge optimierten Code
`-c`	stoppe nach dem Kompilieren und Assemblieren

Generierten Maschinencode prüfen

- Disassemblieren von Objektdateien mit:

  `arm-none-eabi-objdump -d led.o`

- Nutzen:
  - Binärdarstellung einer Anweisung ermitteln
  - Inline-Assembler-Blöcke verifizieren
  - Optimierungspotential erkennen

- Ausgabe objdump

```
led.o: file format elf32-littlearm
Disassembly of section .text:
00000000 <yellow_on>:
0: e3e03000 mvn r3, #0
4: e3a02302 mov r2, #134217728 ; 0x8000000
8: e50329ff str r2, [r3, #-2559] ; 0x9ff
c: e50329ef str r2, [r3, #-2543] ; 0x9ef
10: e50329cf str r2, [r3, #-2511] ; 0x9cf
14: e12fff1e bx lr
```

Integration

- Zielplattform startet automatisch U-Boot, was unser Betriebssystem laden kann.
- Dazu wird bei unserer Variante folgendes benötigt:
  - Speicherabbild
  - Ladeadresse
  - Einsprungspunkt
- Diese Dinge hängen zusammen:
  - Maschinencode arbeitet (i. d. R.) mit absoluten Adressen, kann also nicht an beliebige Adresse geladen werden.
  - Speicherlayout wird beim Linken festgelegt.
  - Speicherabbild geht vom ersten bis zum letzten durch das Layout definierte und belegte Byte.
  - Ladeadresse ist also die erste belegte Adresse im Layout.
  - Einsprungspunkt ergibt sich durch Layout und Platzierung des Codes innerhalb des Layouts.

Speicherlayout

- Beim Festlegen des Layouts gilt es zu beachten:
  - Belegter Speicher muss sich im RAM befinden.
  - RAM ist (zu Beginn) nicht leer: U-Boot lädt unser Betriebssystem!
  - Aufgrund des Umwegs über ein Speicherabbild ist ein kompaktes Layout von Vorteil.
  - (Kompliziertere Layouts müssen über einen eigenen Lader realisiert werden!)

- Internes RAM (16 KiB): `0x0020 0000 – 0x0020 3FFF`
- Externes RAM (64 MiB): `0x2000 0000 – 0x23FF FFFF`

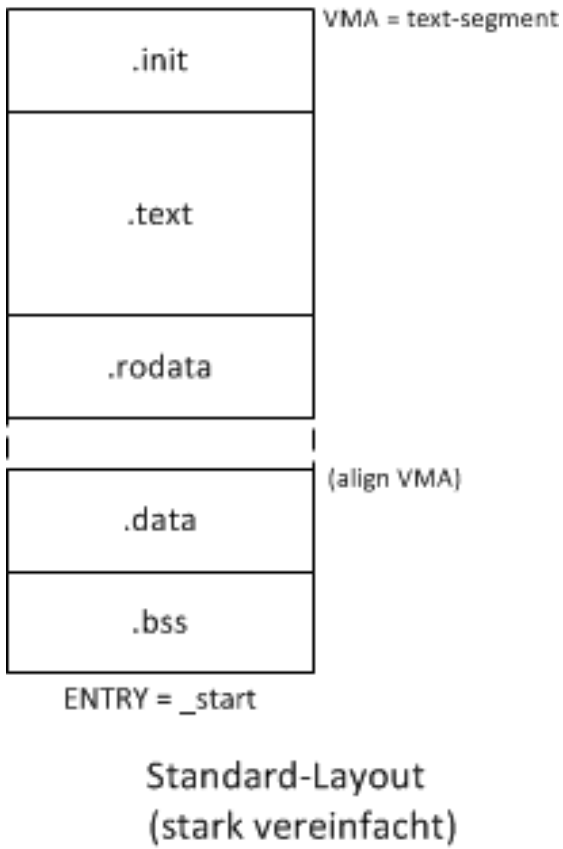
Speicherlayout

- Interrupt Vektor Tabelle (32 Byte) belegt später:
  `0x0020 0000 – 0x0020 001F`
- U-Boot (16 MiB mit Lücke) belegt noch:
  `0x2100 0000 – 0x21FF FFFF`
- (aus Netz geladenes Image ab `0x2100 0000`; Code und Daten sind hinten)

- Es empfiehlt sich, das Layout irgendwo zu dokumentieren, insb. da später noch einiges dazu kommt.

Speicherlayout

- Layout wird beschrieben durch ein Linker-Script.
- Durch Standard-Linker-Script beschriebenes Layout
  - ist kompliziert, kann aber genutzt werden (siehe Ausgabe `arm-none-eabi-ld -verbose`)
  - Beginn kann durch Linker-Parameter `-Ttext-segment=<adresse>` gesetzt werden
  - Einsprungspunkt ist `_start`, kann durch Linker-Parameter `-e<symbol>` gesetzt werden



Standard-Layout
(stark vereinfacht)

Speicherlayout

- Selbst definiertes Linker-Script
  - erlaubt feinere Kontrolle (für komplexere Dinge)
  - hier: gleicher Einsprungspunkt, hart kodierte Adresse für Beginn des Code-Segments, `.init` vor `.text`
  - nicht aufgeführte Sektionen werden hinten angefügt

- Nahezu äquivalentes Script: kernel.lds

```
ENTRY(_start)
SECTIONS
{
. = 0x20000000;
.init : { *(.init) }
.text : { *(.text) }
}
```

Einsprungspunkt

- Irgendeine beliebige C-Funktion ist **nicht** geeignet
  - C-Compiler fügt i. d. R. Instruktionen vor dem eigenen Code ein (z. B. solche, die den noch nicht initialisierten Stack benutzen)
  - Ende springt an unbekannte Adresse „zurück“!
- Es wird ein Mini-Lader benötigt
  - nimmt notwendige Initialisierungen vor
  - verzweigt in normalen C-Code
  - tut bei Rückkehr aus C-Code etwas „sinnvolles“

Assembler-Variante: start.S

```
.section .init
.global _start
_start:
    bl yellow_on
.Lend:
    b .Lend
```

GCC C-Variante: start.c

```
#include "led.h"
__attribute__((naked, section(".init")))
void _start(void) {
    yellow_on();
    for(;;);
}
```

Einsprungspunkt

- Variante 1: Statischer Einsprungspunkt, Code entsprechend platzieren
- Variante 2: Code platzieren lassen, Einsprungspunkt auslesen und U-Boot übergeben (keine init-Sektion nötig, Auslesen z. B. via `arm-none-eabi-objdump -f kernel`)

Linken

- Linken entweder mit:

```
arm-none-eabi-ld -Tkernel.lds \
-o kernel <objektdateien>
```

- Oder mit:

```
arm-none-eabi-ld \
-Ttext-segment=<adresse> \
-o kernel <objektdateien>
```

- Ergebnis studieren, bspw. mit:

```
arm-none-eabi-objdump -fhd kernel
```

Linken

- Ausgabe objdump

```
kernel: file format elf32-littlearm
architecture: armv4t, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x20000000
Sections:
Idx Name Size VMA LMA File off Algn
     0 .init     00000008 20000000 20000000 00008000 2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
     1 .text     00000018 20000008 20000008 00008008 2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
     2 .comment  00000011 00000000 00000000 00008020 2**0
                 CONTENTS, READONLY
     3 .ARM.attributes 0000002f 00000000 00000000 00008031 2**0
                 CONTENTS, READONLY
Disassembly of section .init:
20000000 <_start>:
20000000: eb000000 bl 20000008 <yellow_on>
20000004: eafffffe b 20000004 <_start+0x4>
Disassembly of section .text:
20000008 <yellow_on>:
20000008: e3e03000 mvn r3, #0
2000000c: e3a02302 mov r2, #134217728 ; 0x8000000
20000010: e50329ff str r2, [r3, #-2559] ; 0x9ff
20000014: e50329ef str r2, [r3, #-2543] ; 0x9ef
20000018: e50329cf str r2, [r3, #-2511] ; 0x9cf
2000001c: e12fff1e bx lr
```

Fehler beim Linken

- Unter Umständen generiert der Compiler automatisch Referenzen auf euch unbekannte Funktionen (z.B. `__aeabi_idiv` oder `memcpy`).
- Für `memcpy`, `memmove`, `memset` und `memcmp`:
  - Compiler nimmt an, dass diese Funktionen existieren
  - sind normalerweise in der C-Library
  - muss selbst implementiert werden, falls notwendig
- Für alles andere:
  - Compiler Support Routines (z. B. Integer Division, Softfloat, Thumb-Interworking)
  - sind in der libgcc.a; kann dem Linker als weitere Datei mitgegeben werden
  - genauer Pfad: siehe Ausgabe von

    ```
    arm-none-eabi-gcc -print-libgcc-file-name
    ```

Speicherabbild

- Speicherabbild erstellen mit:

```
arm-none-eabi-objcopy -Obinary \
--set-section-flags \
.bss=contents,alloc,load,data  kernel kernel.bin
```

- Parameter `–Obinary` sorgt für Speicherabbild als Ausgabe
- Sonderfall BSS-Sektion
  - hat nur Größe, keinen Inhalt
  - enthält nicht initialisierte sowie mit Null initialisierte Daten
  - Konvention: zugeordneter Speicherbereich wird beim Laden mit Nullen gefüllt
  - wird von objcopy normalerweise ignoriert, da kein Inhalt
  - `--set-section-flags .bss=contents,alloc,load,data` sorgt dafür, dass die BSS-Sektion in jedem Fall Teil des Speicherabbilds ist

Image für U-Boot und serielle Konsole

Freie Universität Berlin

- U-Boot-Image erstellen mit:

```
mkimage -A arm -T standalone -C none \
-a <ladeadresse> -e <einsprungspunkt> \
-d kernel.bin kernel.img
```

-A arm	spezifiziert die Zielplattform; wird durch U-Boot beim Laden verifiziert
-T standalone	spezifiziert die Art, wie das Image geladen/gestartet wird
-C none	kernel.bin ist nicht komprimiert
-a <ladeadresse>	Ladeadresse, bspw. 0x2000 0000
-e <einsprungspunkt>	optional, wenn weggelassen identisch zur Ladeadresse

- Image auf Server kopieren mit:

```
arm-install-image kernel.img
```

- Serielle Konsole bspw. mit: `veryminicom`

QEMU

- QEMU kann ELF-Dateien direkt laden.
- Letztlich muss QEMU nur mit den richtigen Parametern gestartet werden.
- Zur Vereinfachung enthält unsere Toolchain bereits einen Wrapper für QEMU.

- QEMU starten mit:

```
qemu-bsprak -kernel kernel
```

  - Hinweis: QEMU beenden mit CTRL+A und dann X (ohne CTRL).

QEMU

- Es gibt experimentellen Support zur Visualisierung der LEDs (und des Displays)
  - aktivieren über zusätzlichen Parameter `-piotelnet`
  - anschließend in einer weiteren Konsole:
    ```
    telnet localhost 44444
    ```
  - Hinweis: Nur eine Instanz pro Hostsystem wg. fester Portnummer.

Hinweise zum Programmierstil

- Coding Style
  - ordentlich eingerückt; aussagekräftige aber nicht ausschweifende Bezeichner; konsistent angewendet
  - Code idealerweise selbsterklärend; Kommentare für das Was und Warum, nicht das Wie
  - defensiv; keine Warnungen beim Kompilieren
  - für Anregungen: Linux Kernel Coding Style

- Komplett ohne Assembler funktioniert es nicht!
  - C für alles, was in C machbar ist (ohne Hacks)
  - Assembler für maschinennahe Operationen (bspw. beim Behandeln von Ausnahmen)
  - Herausforderung besteht in der richtigen Aufteilung