

# U1 1. Übung

- Organisatorisches
- Arbeitsumgebung
- Toolchain: vom Quellcode zum ladbaren Binärabbild
- Arbeiten mit dem Debugger
- Aufgabe 0

## U1-1 Organisatorisches

# U1-1 Organisatorisches

- keine Vorlesung
  - ◆ Grundlegende C-Kenntnisse aus der Vorlesung werden vorausgesetzt...
  - ◆ ...oder müssen sich selbst angeeignet werden
- Übungsablauf wie im Sommersemester
  - ◆ Besuch einer Tafelübung (Anwesenheitspflicht)
    - Anmeldung im Waffel zur Abgabe von Aufgaben zwingend erforderlich
  - ◆ Besuch von Rechnerübungen (2 Termine) nach Bedarf
  - ◆ **eigenständige Arbeit** außerhalb der Übungen erforderlich!
- Erwerb von Bonuspunkten
  - ◆ Bonuspunkte aus dem Sommersemester können nicht übernommen werden
  - ◆ Neuerwerb von Bonuspunkten durch Bearbeitung der Übungsaufgaben
- Alle Aufgaben sind **einzeln** abzugeben

# U1-1 Umsetzung von Evaluationsanmerkungen

- viele neue Boards...
- mehr Wert auf eigenständige (real-world nahe ;-)) Arbeit
  - ◆ selbstständiges Nachschlagen im Datenblatt
  - ◆ möglichst keine 1:1 übernehmbaren Codebeispiele in den Folien
  - ◆ Bereitstellung einer Schnittstellen-Dokumentation (in späteren Aufgaben)
- größere Programme aus mehreren Modulen
- Makefiles
- bei Bedarf zwei Übungsleiter in den Rechnerübungen
  - ◆ wenn bis 30 Minuten nach Übungsbeginn (hh:30 Uhr) keine Teilnehmer anwesend sind, können die Übungsleiter die Rechnerübung vorzeitig beenden

## U1-2 Arbeitsumgebung

- 20 neue SPiC-Boards, entwickelt von Susanne Brunner
  - ◆ umfangreichere Peripherie als auf den alten Boards
  - ◆ ATmega32 Mikrocontroller (MCU)
- zugehörige Hardware-Debugger
  - ◆ Entwicklung direkt auf den Boards
  - ◆ Nutzung des Simulators entfällt
- Entwicklung ausschließlich unter Linux

## U1-2 UNIX/Linux Benutzerumgebung

- Projektverzeichnisse zur Bearbeitung der Übungsaufgaben
  - `/proj/i4spic/login-Name`
  - werden nach Anmeldung im Waffel innerhalb eines Tages erzeugt
  - Wechseln des Arbeitsverzeichnisses dorthin (`cd = change directory`):  
`cd /proj/i4spic/login-Name`
  - Einrichten eines Verzeichnisses für Aufgabe 1:  
`mkdir aufgabe1`
  - Anzeigen der Dateien in einem Verzeichnis:  
aktuelles Verzeichnis anzeigen: `ls` oder `ls -l`  
Verzeichnis `aufgabe1` anzeigen: `ls -l aufgabe1`
- Aufgaben sollten ausschließlich im Projektverzeichnis bearbeitet werden
  - Nur vom eigenen Benutzer lesbar
  - Suchverzeichnis des Abgabeprogramms

## U1-2 Kommandointerpreter (Shell)

- Kommandointerpreter (Shell)
  - Programm, das Kommandos entgegennimmt und ausführt
  - verschiedene Varianten, am häufigsten unter Linux: `bash` oder `tcsh`
- Sonderzeichen
  - ◆ einige Zeichen haben unter UNIX besondere Bedeutung
  - ◆ Funktionen:
    - Korrektur von Tippfehlern
    - Einwirkung auf den Ablauf von Programmen
- Übersicht: ( `<CTRL>` = `<STRG>` )
 

|                                |                                                                      |
|--------------------------------|----------------------------------------------------------------------|
| <code>&lt;BACKSPACE&gt;</code> | letztes Zeichen löschen (manchmal auch <code>&lt;DELETE&gt;</code> ) |
| <code>&lt;CTRL&gt;</code> - C  | Interrupt - Programm wird abgebrochen                                |
| <code>&lt;CTRL&gt;</code> - Z  | Stop - Programm wird gestoppt (Fortsetzen mit <code>f9</code> )      |
| <code>&lt;CTRL&gt;</code> - D  | End-of-File                                                          |

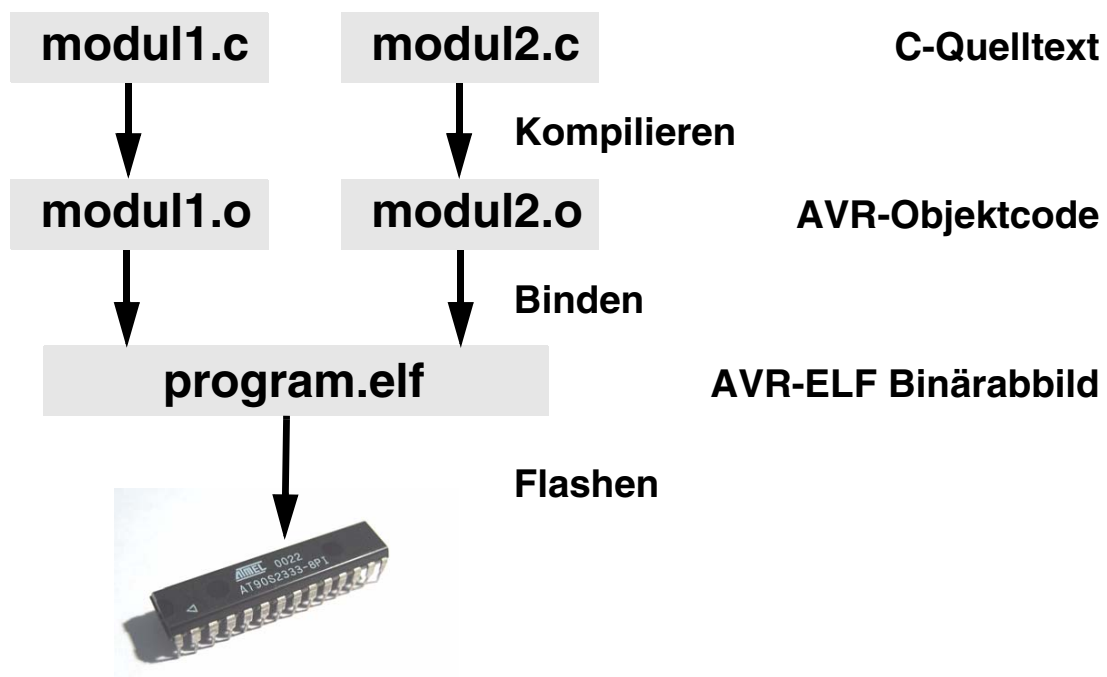
## U1-3 Toolchain: Vom Quellcode zum geflashten Binärabbild

- (1) Erstellen des Programmquellcodes mit einem Texteditor (z.B. vim oder **kate**)
  - ◆ das Programm besteht ggf. aus mehreren Modulen ( = **.c-Dateien** )
  - ◆ und ggf. einer Schnittstellenbeschreibung/Header pro Modul ( = **.h-Dateien** )
  - ◆ modulare Programmierung ist Gegenstand einer späteren Übung
- (2) Übersetzen der C-Module zu Objektdateien mit einem C-Compiler (GCC)
  - ◆ jedes C-Modul wird zu einer Objektdatei ( = **.o-Datei** ) kompiliert
  - ◆ da wir Binärcode für den AVR erzeugen wollen, verwenden wir einen AVR-Crosscompiler (avr-gcc)
 

```
avr-gcc -c -o modul1.o modul1.c
```
- (3) Linken/Binden der Objektdateien zu einem ladbaren ELF-Binärabbild (**.elf-Datei**)
  - ◆ mit GCC oder LD
 

```
avr-gcc -o program.elf modul1.o modul2.o
```
- (4) Flashen des Binärabbaus auf den Mikrokontroller
  - ◆ z.B. mit avarice oder avrdude

## U1-3 Toolchain Überblick



## U1-3 Texteditoren

- verschiedene Editoren unter UNIX verfügbar
  - ◆ vim
  - ◆ emacs
- für Newbs zu empfehlen: **kate**
  - ◆ Starten
    - durch Eingabe von `kate` in einer Shell
    - oder über Auswahllisten von KDE
- Abspeichern der Quelltexte in Dateien mit der Endung **.c** im Projektverzeichnis
  - ◆ die zu entwickelnden Module und Dateinamen sind in der Aufgabenstellung vorgegeben

## U1-3 Kompilieren der C-Module

- C-Quellcode wird mit einem C-Compiler (z.B. GCC) zu Binärcode für die Zielarchitektur (hier: 8-bit AVR) übersetzt: **avr-gcc**
- Jede **.c**-Datei wird in eine Objektdatei übersetzt: Compileroption **-c**
- Referenzen auf externe Symbole werden hierbei noch nicht aufgelöst
- Weitere Compiler Flags
  - ◆ `-mmcu=atmega32`: teilt dem Compiler den Typ der Ziel-CPU mit
  - ◆ `-ansi`: wählt den C-Standard ISO C90
  - ◆ `-Wall`: aktiviert viele Warnungen, die auf evtl. Programmierfehler hinweisen
  - ◆ `-pedantic`: aktiviert weitere Warnungen in Bezug auf ISO-C-Konformität
  - ◆ `-O0` bzw. `-Os`: Optimierungen deaktivieren bzw. nach Größe optimieren
    - ☞ Debuggen mit `-O0 -g`, Testen (**volatile** richtig verwendet?) mit `-Os`
- Übersetzung eines C-Moduls **modul.c** dann zu **modul.o** mit Aufruf:
 

```
avr-gcc -Os -c -mmcu=atmega32 -ansi -pedantic -Wall modul.c
```

## U1-3 Binden der Objekdateien

- Im Bindeschritt werden offene Symbolreferenzen aufgelöst
- Binden z.B. mit **avr-gcc**
- Beispiel: Programm aus den Modulen **modul1.o** und **modul2.o**
  - ◆ mit avr-gcc (**-o** bestimmt den Namen der Zielfeldatei, hier **program.elf**):
 

```
avr-gcc -mmcu=atmega32 -o program.elf modul1.o modul2.o
```
- GCC kann auch in einem Schritt kompilieren und binden:
 

```
avr-gcc -mmcu=atmega32 ... -o program.elf modul1.c modul2.c
```

  - ◆ Vorteil: Übersetzer sieht komplettes Programm ➔ globale Optimierungen
    - in aktuellen GCC Versionen: Compiler-Flags **-combine -fwhole-program**
  - ◆ Nachteil: Alle Module müssen komplett übersetzt werden, auch wenn man nur ein Modul verändert hat
  - ◆ Für kleine Programme ist diese Variante aber oft die bevorzugte Wahl
  - ◆ In SPiC: your choice...

## U1-3 Flashen des ELF-Images

- Ablegen des Binärabbilds im Flash-ROM des Mikrokontrollers
  - ◆ z.B. mit unserem Debugger und dem Programm **avarice**
  - ◆ wir haben das entsprechende Kommando in einem **Makefile** abgelegt
  - ◆ Beispiel: Flashen des Binärabbilds **program.elf**:
 

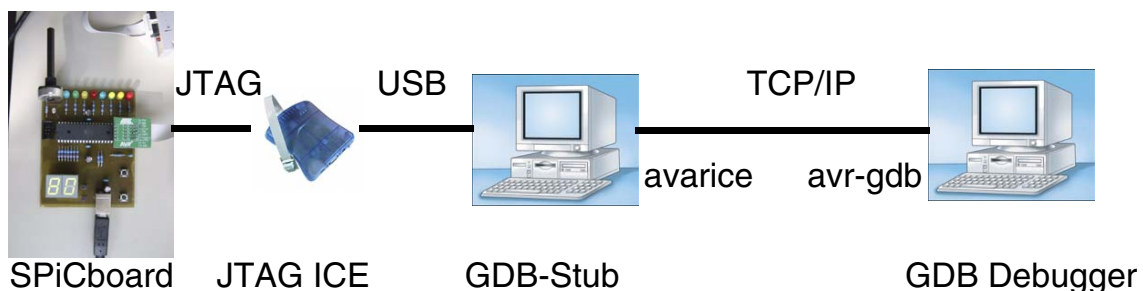
```
make -f /proj/i4spic/pub/i4/debug.mk program.elf.flash
```
- Nach jedem Reset lädt der **Bootloader** des Mikrokontrollers die relevanten Sektionen in den RAM und startet die Ausführung

## U1-4 Debugger

- Der Debugger vereinfacht die Fehlersuche im Programm
  - ◆ ähnlich wie der Debugger des Simulators im Sommersemester
  - ◆ schrittweises Abarbeiten des Programms
  - ◆ Beobachten der Werte von Variablen
  - ◆ Haltepunkte (Breakpoints), auch abhängig von Bedingungen
- Die JTAG-Debugger erlauben das Debugging der Ausführung direkt auf dem Mikrokontroller
- Die Debugger/Boards können gegen Hinterlegung eines staatlichen Ausweisdokuments **für die Arbeit im CIP-Pool** ausgeliehen werden
  - ◆ Abholung/Rückgabe bei M. Stilkerich (Büro 0.044, RRZE Erdgeschoss)
  - ◆ die Debugger sind am gleichen Tag zurückzugeben

## U1-4 Anschluss des Debuggers

- Verbinden des Debuggers mit dem JTAG-Anschluss auf dem SPiCboard
- Board und Debugger an zwei USB-Ports des Rechners anschließen
  - ◆ Achtung: der Debugger funktioniert nicht zuverlässig an den SunRays, daher "richtige" Rechner verwenden
- Das Programm **avarice** öffnet einen GDB-Stub und fungiert so als Mittler zwischen JTAG-Debugger und Software-Debugger (**avr-gdb**)
- GDB-Stub-Rechner und Debug-Rechner sind normalerweise identisch



## U1-4 Verwendung des Debuggers

- Flashen des Binärabbilds **program.elf** in den Mikrokontroller
  - ◆ das Binärabbild sollte mit Debug Symbolen erzeugt werden:
    - ☞ zusätzliches Compiler-Flag **-g** bei der Übersetzung verwenden
  - ◆ Compiler-Optimierungen sollten deaktiviert werden: **-O0**
- Starten des GDB-Stubs **avarice**

```
make -f /proj/i4spic/pub/i4/debug.mk dbgstub
```
- Starten des Debuggers **avr-gdb** auf dem gleichen Rechner

```
avr-gdb program.elf
```

  - ◆ das hier verwendete Binärimage muss mit dem in den Mikrokontroller geflashten Abbild übereinstimmen!
- Verbinden des Debuggers mit dem GDB-Stub

```
target remote localhost:4242
```
- Das Programm ist gestoppt an der ersten Instruktion

## U1-4 Wichtige GDB-Kommandos

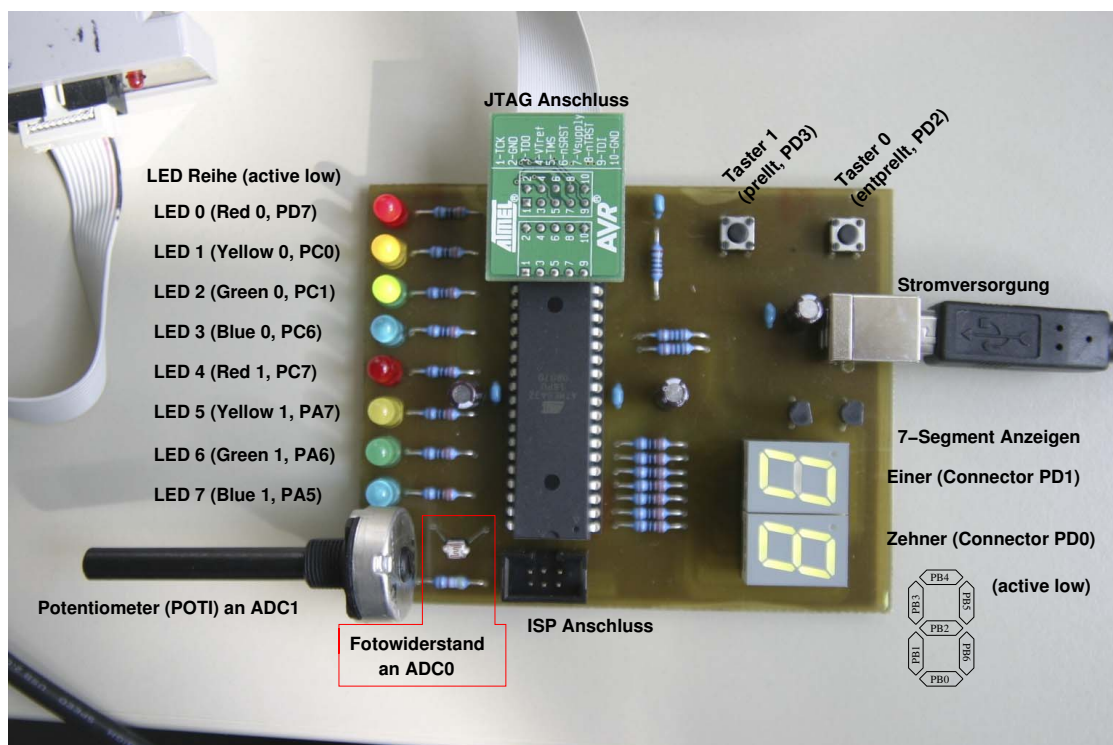
- Schrittweises Abarbeiten des Programms
  - ◆ **n**: führt nächste Zeile C-Code aus, übergeht Funktionen
  - ◆ **s**: wie **n**, steigt aber in Funktionen ab
- Setzen von Breakpoints (Haltepunkten)
  - ◆ Anzahl durch die Hardware auf 3 beschränkt
  - ◆ **b [Dateiname:]Funktionsname [condition]**
  - ◆ **b Dateiname:ZeilenNr. [condition]**
  - ◆ Die Ausführung stoppt bei Erreichen der angegebenen Stelle
  - ◆ wenn **condition** angeben (C-Ausdruck) nur dann, wenn Bedingung erfüllt ist
  - ◆ Breakpoints anzeigen: **info breakpoints**
  - ◆ Breakpoint löschen (Nr. des Breakpoints aus Anzeige): **d BreakpointNr**
- Fortsetzen der Programmausführung bis zu Haltepunkt: **c**



## U1-4 Wichtige GDB-Kommandos

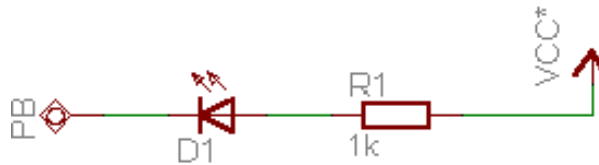
- Watchpoints: Stop der Ausführung bei Zugriff auf eine bestimmte Variable
  - ◆ **watch *expr***: Stoppt, wenn sich Wert des C-Ausdrucks ***expr*** ändert
  - ◆ **rwatch *expr***: Stoppt, wenn ***expr*** gelesen wird
  - ◆ **awatch *expr***: Stoppt bei jedem Zugriff (kombiniert **rwatch** und **watch**)
  - ◆ ***expr*** ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
  - ◆ Achtung: für jedes Byte des Ausdrucks wird ein Hardware-Breakpoints verbraucht, **watch** auf einen **int** belegt also zwei Hardware-Breakpoints!
- Weitere im Reference-Sheet (👉 Doku-Bereich der Webseite)

## U1-5 SPiCboard



## U1-5 Hardware

- acht Leuchtdioden
  - ◆ Achtung: die LEDs werden durch low-Pegel eingeschaltet

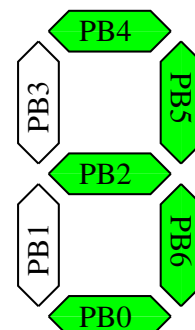


- zwei Geräte am A/D-Wandler
  - ◆ Potentiometer: Regelbarer Spannungsteiler
  - ◆ Fotowiderstand: von Helligkeit abhängiger Widerstand
- zwei Taster
  - ◆ ein Taster ist entprellt, der andere muss durch Software entprellt werden
  - ◆ der nicht-entprellte Taster darf nicht an einer Interruptleitung betrieben werden

## U1-5 7-Segment-Anzeigen

- zwei 7-Segment-Anzeigen
- bestehen aus 7 Balken, die ebenfalls durch low-Pegel aktiviert werden
  - ◆ für jedes darzustellende Zeichen muss man sich also überlegen, welche Balken leuchten sollen
  - ◆ an diese Pins legt man einen low-Pegel an

- Beispiel (Erinnerung: active low!): Ziffer 3
  - 0b 1000 1010 = 0x8a
  - ☞ Der Wert 0x8a in Port B zeigt die Ziffer 3 an.



- teilen sich einen Port
  - ◆ Umschaltung zwischen den Anzeigen mit Transistoren
  - ◆ low-Pegel an Transistorpin verbindet entsprechende Anzeige mit Port B
  - ◆ schnelle Umschaltung zwischen den Anzeigen für Auge nicht wahrnehmbar
  - ◆ quasi-paralleler Betrieb also z.B. durch einen regelmässigen Timerinterrupt

# U1-6 Aufgabe 0

- Einfacher Countdown (von 9 bis 0)
  - ◆ es wird nur eine 7-Segementanzeige verwendet (Umschaltung nicht nötig!)
- Ziele der Aufgabe: Kennenlernen der Arbeitsumgebung
  - ◆ wie bekomme ich mein Programm auf den Mikrokontroller
  - ◆ wie verwende ich einen Debugger, um den Programmablauf zu beobachten
  - ◆ wie gebe ich meine Aufgabe ab
- Notwendiges bzw. anzueignendes Wissen
  - ◆ Hardware Ports konfigurieren
  - ◆ Umgang mit Hexadezimalzahlen
  - ◆ siehe ATmega32 Datenblatt, S. 49-54
    - ☞ zu finden im Doku-Bereich auf der SPiC-Webseite
  - ◆ alternativ: SPiC Übung U4 aus dem Sommersemester