U1 1. Übung

- Organisatorisches
- Arbeitsumgebung
- Aufgabe 1

U1-1 Login in die Windows-Umgebung

- Zur Nutzung der Windows-PCs zunächst mit dem Kommando
  `/local/ciptools/bin/setsambapw`
  ein Windows-Passwort setzen.
  - ➤ **Achtung**: Passwörter werden erst nach etwa 10 Minuten aktiv!
- Setzen Sie **jetzt** soweit noch nicht geschehen im Raum 01.155 Ihr Windows-Passwort
- Melden Sie sich dann an der Windows-Domäne **ICIP** an

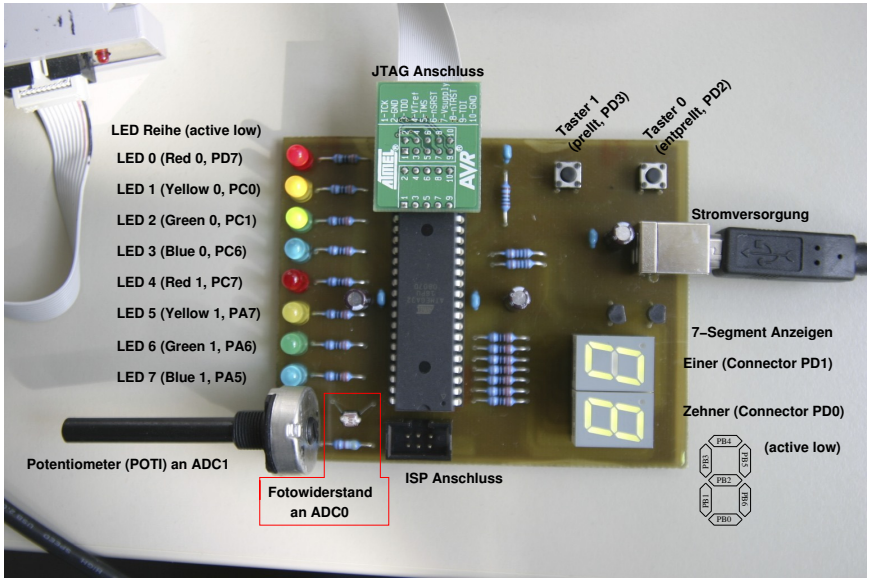
U1-2 Organisatorisches

- keine Vorlesung
  - ◆ Grundlegende C-Kenntnisse aus der Vorlesung werden vorausgesetzt...
  - ◆ ...oder müssen sich selbst angeeignet werden
- Übungsablauf wie im Sommersemester
  - ◆ Besuch einer Tafelübung (Unentschuldigtes Fehlen -> 0P)
    - Anmeldung im Waffel zur Abgabe von Aufgaben zwingend erforderlich
  - ◆ Besuch von Rechnerübungen 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

U1-2 Rechnerübungen

- Termine:
  - ◆ Di 16:00 - 18:00 Raum 01.153
- wenn bis 30 Minuten nach Übungsbeginn (*hh*:30 Uhr) keine Teilnehmer anwesend sind, können die Übungsleiter die Rechnerübung vorzeitig beenden
- Die Debugger/Boards können gegen Hinterlegung eines Ausweises **für die Arbeit im CIP-Pool** ausgeliehen werden
  - ◆ Abholung/Rückgabe bei Moritz Strübe (Büro 0.041, RRZE Erdgeschoss)
  - ◆ die Debugger sind am gleichen Tag zurückzugeben

U1-2 Entwicklungsboards



U1-3 SPiCboard-Bibliothek

- Funktionsbibliothek zur einfachen Verwendung der Hardware
  - Konfiguration des Mikrokontrollers durch den Anwender entfällt
  - Soll während des Semesters "aufgeräumt" werden (Details folgen)
- Dokumentation zu den einzelnen Funktionen online verfügbar

`http://www4.cs.fau.de/Lehre/WS10/V_SPIC/Uebung/doc`

1 Projektverzeichnisse

- Spezielle Projektverzeichnisse zur Bearbeitung der Übungsaufgaben
  - ➤ Unter Linux: `/proj/i4(g)spic/`*login-Name*
  - ➤ Unter Windows: Laufwerk P:
- Aufgaben sollten ausschließlich im Projektverzeichnis bearbeitet werden
  - ➤ Nur vom eigenen Benutzer lesbar
  - ➤ Suchverzeichnis des Abgabeprogramms
- Werden nach Waffel-Anmeldung binnen eines Tages erzeugt

2 UNIX-Heimverzeichnisse

- Unter Windows: Laufwerk H:
- Zugriff auf Ihre Dateien aus der Linux-Umgebung

3 Vorgabeverzeichnis

- Unter Linux: `/proj/i4spic/pub`
- Unter Windows: Laufwerk Q:
- Hilfsmaterial zu einzelnen Übungsaufgaben
  - ➤ z.B. `/proj/i4spic/pub/aufgabe0`
- Testprogramm für die Entwicklungsboards
  - ➤ `/proj/i4spic/pub/boardtest`
- Hilfsbibliothek (libSPiCboard) und Dokumentation
  - ➤ `/proj/i4spic/pub/i4`
- Werkzeuge zur Entwicklung unter Linux
  - ➤ `/proj/i4spic/pub/tools`

U1-4 Windows-Umgebung

- Verwendung der Entwicklungsumgebung AVR-Studio
  - vereint Editor, Compiler und Debugger in einer Umgebung
  - komfortable Nutzung der JTAGICE-Debugger

1 Compiler

- Um auf einem PC Programme für den AVR-Mikrokontroller zu erstellen, wird ein ***Cross-Compiler*** benötigt
  - Ein Cross-Compiler ist ein Compiler, der Code für eine Architektur generiert, die von der Architektur des Rechners, auf dem der Compiler ausgeführt wird, verschieden ist.
  - Hier: Compiler läuft auf Intel x86 und generiert Code für AVR.
  - AVR-GCC (GNU Compiler Collection)
    - Funktioniert unter Windows und Linux

2 AVR Studio

- Entwicklungsumgebung von Atmel
- Simulator und Debugger für alle AVR-Mikrokontroller
- Start über das Startmenü
  Start - Alle Programme - Atmel AVR Tools - AVR Studio 4
- Anlegen eines neuen Projekts
  - Projekttyp: AVR GCC
  - Projektname: *aufgabe0*
  - *Create Initial File* und *Create Folder* aktivieren
  - Initial File: *ledon.c*
  - Location: *P:*
- Auswahl der Plattform *JTAGICE mkII* mit dem Gerät *ATMega32*
  - bei Verwendung der (kleinen) Programmer *AVRISP mkII*

3 AVR Studio-Projekteinstellungen

- Setzen der Projekteinstellungen (*Project - Configuration Options*)
  - Frequency: *1000000* Hz (1 MHz)
  - Optimization: *-O0*
  - **Wichtig:** Die vier Haken rechts daneben deaktivieren
- Bereich *Include Directories*: `Q:\i4\` aufnehmen
- Bereich Libraries (bei Verwendung der SPiCboard-Bibliothek)
  - `Q:\i4\` in *Library Search Path* aufnehmen
  - `libspicboard.a` bei *Available Link Objects* anwählen, *Add Library* klicken
- Compileroptionen im Bereich *Custom Options*
  - *-std=gnu99* entfernen
  - *-ansi* hinzufügen
  - *-pedantic* hinzufügen
  - *-ffreestanding* hinzufügen

4 Erstellen einer main()-Funktion

- Auf dem Mikrokontroller ist die *main()*-Funktion vom Typ
  `void main(void);`
  - ➤ Sollte niemals zurückkehren (wohin?)
  - ➤ Rückgabetyp daher nicht sinnvoll
  - ➤ Freistehende Umgebung (*-ffreestanding*)
- Beispiel: Grüne LED einschalten

```
#include <led.h>

void main(void) {
    int i=0;
    sb_led_on(GREEN0);
    while(1) { /* Endlosschleife*/
        i++;
    }
}
```

- Kompilieren des Programmes mit F7 oder *Build - Build*

5 Flashen des Programms auf die MCU

- Mit dem Debugger/Programmer verbinden
  - *Tools - Program AVR - Auto Connect*
  - ggf. aktuell angeschlossenen Debugger/Programmer auswählen
- Reiter *Main*
  - *Device*: ATmega32
  - wenn Debugger angeschlossen: *Programming Mode*: JTAG mode
  - wenn Programmer angeschlossen: *ISP Frequency* max. 250 kHz
- Reiter *Program*, Bereich *Flash*
  - *Input HEX File*: aktuell übersetztes Programm auswählen
  - normalerweise im Unterverzeichnis *default* des Projektverzeichnisses
  - z.B. *blink.hex*
- *Program* klicken, Programm wird auf die MCU geflasht

6 Starten des Simulators/Debuggers

- Start des Debugging-Modus über *Debug - Start Debugging*
  - Das Programm wird automatisch neu geflasht
- Das Programm wird zunächst beim Betreten von `main()` angehalten
  - Normal laufen lassen mit *F5* oder *Debug - Run*
  - Schrittweise abarbeiten mit
    - ➤ *F10* (step over): Funktionsaufrufe werden in einem Schritt bearbeitet
    - ➤ *F11* (step into): Bei Funktionsaufrufen wird die Funktion betreten
- Die I/O-Ansicht (rechts) gibt Einblick in die Zustände der I/O-Register
- Breakpoints erlauben, das Programm an einer bestimmen Stelle zu stoppen
  - Codezeile anklicken, dann *F9* oder *Debug - Toggle Breakpoint*
  - Programm laufen lassen (*F5*), stoppt wenn ein Breakpoint erreicht wird

7 Programm abgeben

- Abgabeskript funktioniert nur in Linux-Umgebung
- Remote-Login auf den Linux-Rechnern über SSH (Terminalfenster)
  - im Startmenü *Secure Shell Client* starten
  - verbinden mit einem beliebigen Linux-Rechner, z.B. *faui01*
  - Login mit den UNIX-Zugangsdaten
- Im Terminal-Fenster folgendes Kommando ausführen:

```
/proj/i4(g)spic/pub/abgabe aufgabe0
```

  - hierbei die aktuelle Aufgabe einsetzen
- Der Quelltext wird zur Erkennung von Plagiaten an die den Server der Uni Karlsruhe übertragen -> Keine vertraulichen Daten in den Datein speichern!

U1-5 UNIX/Linux Benutzerumgebung

- 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>)

<BACKSPACE>	letztes Zeichen löschen (manchmal auch **<DELETE>**)
<CTRL> - C	Interrupt - Programm wird abgebrochen
<CTRL> - Z	Stop - Programm wird gestoppt (Fortsetzen mit `fg`)
<CTRL> - D	End-of-File

1 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)

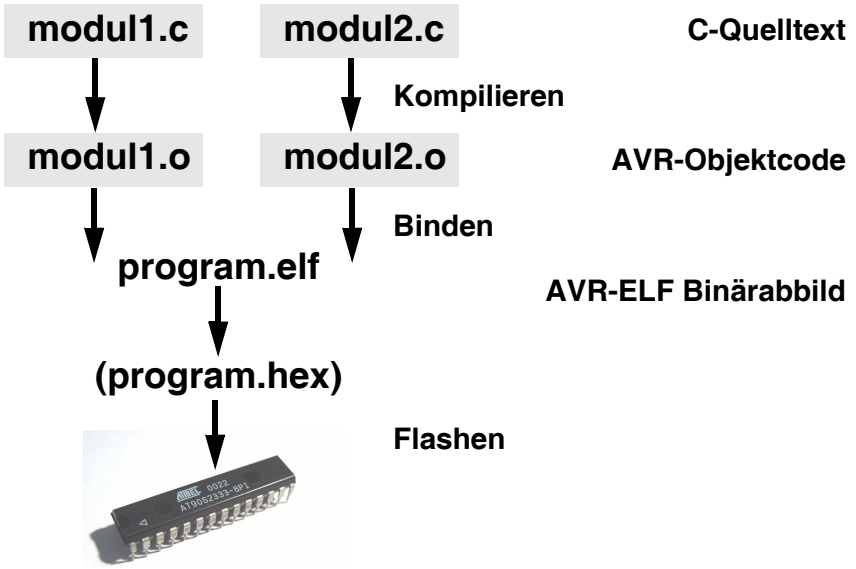
```
avr-gcc -c -o modul2.o modul2.c
```

(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 Binärabbilds auf den Mikrokontroller
- z.B. mit avarice oder avrdude

2 Toolchain-Überblick



3 Texteditoren

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

4 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 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
```

5 Binden der Objektdateien

- 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 Zieldatei, 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 bessere Wahl

6 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.hex**:

```
make -f /proj/i4spic/pub/i4/debug.mk program.hex.flash
```

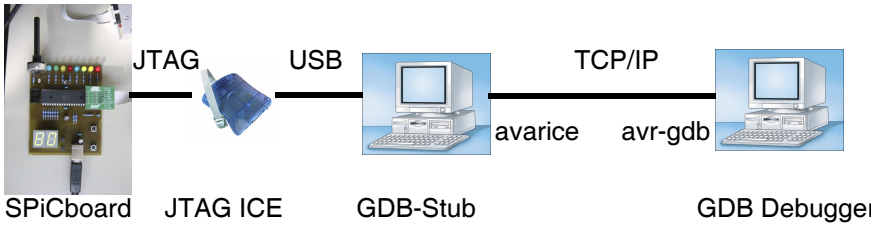
- Nach jedem Reset lädt der **Bootloader** des Mikrokontrollers die relevanten Sektionen in den RAM und startet die Ausführung
- Für einfache Programme (nur eine C-Datei **program.c**) übernimmt obiger Aufruf zum Flashen auch die Übersetzung

7 Debugging unter Linux (*Hintergrundinfo*)

- Der Debugger vereinfacht die Fehlersuche im Programm
  - 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
- Unter Linux ist das Debugging leider mit Schmerzen verbunden
  - Stepping durch den Code sehr langsam
  - GDB-Stub stürzt gelegentlich ab

7 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



7 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

7 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**

7 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 Ausdruks wird ein Hardware-Breakpoints verbraucht, **watch** auf einen **int** belegt also zwei Hardware-Breakpoints!
- Weitere im Reference-Sheet (☞ Doku-Bereich der Webseite)