

# U1 Organisatorisches

---

- Tafelübungen: Windows-Umgebung
  - ◆ Räume 01.155-N und 01.153
  - ◆ Integrierte Entwicklungsumgebung *AVR Studio*
  
- Rechnerübungen (SPiC)
  - ◆ Raum 01.155
  - ◆ Verwendung einer Windows-VM unter Linux
  
- Ziele der heutigen Tafelübung
  - (1) Wie schreibe ich ein Programm unter Windows?
  - (2) Wie lade ich dieses Programm auf das Entwicklungsboard?
  - (3) Wie arbeite ich das Programm schrittweise ab (Fehlersuche/Debugging)?
  - (4) Wie gebe ich eine Übungsaufgabe ab (unter Linux)?

# 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 Tafelübung

---

- Keine Anwesenheitspflicht
- Es wird jedoch eine zufällig ausgewählte Lösung besprochen; bei Abwesenheit gibt es gegebenenfalls 0 Punkte
  - ◆ Gegebenenfalls vorher beim Übungsleiter abmelden
- Fragen bitte im Forum der FSI EEI stellen:
  - ◆ `http://eei.fsi.uni-erlangen.de/forum/16`
  - ◆ Spezielles an `i4spic@informatik.uni-erlangen.de`

# U1-3 Rechnerübungen (SPiC)

---

## ■ Termine:

- ◆ Mo 12:00-14:00
- ◆ Di 08:00-10:00
- ◆ Mi 14:00-16:00
- ◆ Do 10:00-12:00
- ◆ Fr 10:00-12:00

## ■ Sonstiges:

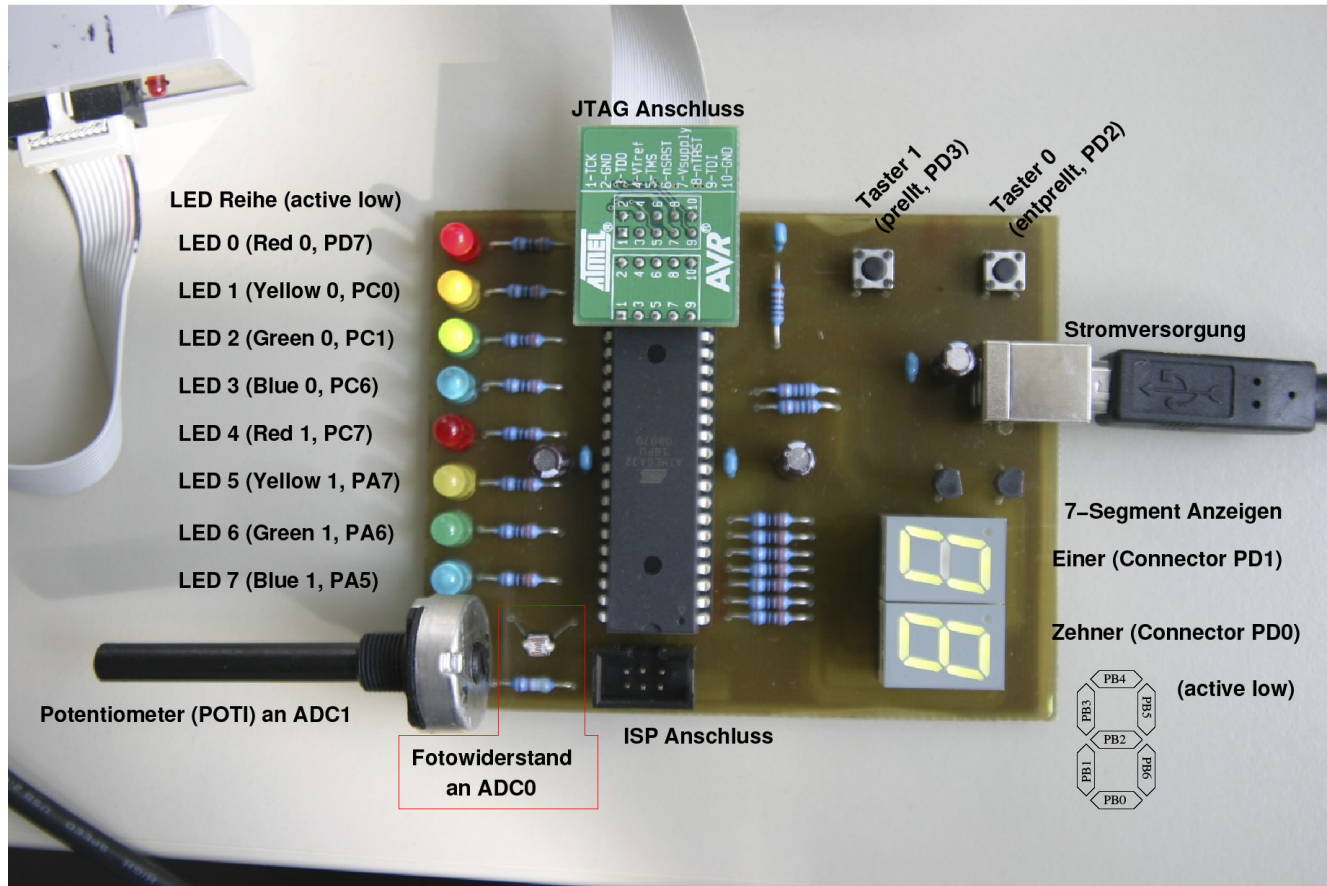
- ◆ Wenn bis 30 Minuten nach Übungsbeginn (*hh:30 Uhr*) keine Teilnehmer anwesend sind, können die Übungsleiter die Rechnerübung vorzeitig beenden
- ◆ In 01.155 sind Linux-Rechner. Es gibt daher die Möglichkeit in einer *Virtuellen Maschine* unter Windows zu arbeiten. Alternativ ist auch das Arbeiten unter Linux möglich. Siehe Anhang.
- ◆ Genaue Anleitung zur KVM auf der Webseite

# U1-4 Entwicklungsboards

---

- 30 SPiC-Boards
  - ◆ ATmega32-Mikrocontroller (MCU)
  
- 22 Hardware-Debugger (groß mit 10-poligem Stecker)
  - ◆ Entwicklung direkt auf den Boards
  - ◆ Überwachung des Programms während der Ausführung (Debugging)
  - ◆ Nutzung des Simulators entfällt
  - ◆ Anschluss an den JTAG-Port des Boards
  
- 10 ISP-Programmer (klein mit 6-poligem Stecker)
  - ◆ kein Debugging möglich
  - ◆ Anschluss an den ISP-Port des Boards
  - ◆ USB-Kabel

# U1-4 Entwicklungsboards



# U1-5 SPiCboard-Bibliothek

---

- Funktionsbibliothek zur einfachen Verwendung der Hardware
  - ◆ Konfiguration des Mikrocontrollers durch den Anwender entfällt
- Verwendung v. a. in der Anfangsphase
- Dokumentation zu den einzelnen Funktionen online verfügbar

[http://www4.cs.fau.de/Lehre/SS11/V\\_SPIC/Uebung/doc](http://www4.cs.fau.de/Lehre/SS11/V_SPIC/Uebung/doc)

[http://www4.cs.fau.de/Lehre/SS11/V\\_GSPIC/Uebung/doc](http://www4.cs.fau.de/Lehre/SS11/V_GSPIC/Uebung/doc)

# 1 Projektverzeichnisse

---

- Spezielle Projektverzeichnisse zur Bearbeitung der Übungsaufgaben
  - Unter Linux: `/proj/i4gspic/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/i4gspic/pub`
- Unter Windows: Laufwerk Q:
- Hilfsmaterial zu einzelnen Übungsaufgaben
  - z. B. `/proj/i4gspic/pub/aufgabe0`
- Testprogramm für die Entwicklungsboards
  - `/proj/i4gspic/pub/boardtest`
- Hilfsbibliothek (Board-Support-Package) und Dokumentation
  - `/proj/i4gspic/pub/i4`
- Werkzeuge zur Entwicklung unter Linux
  - `/proj/i4gspic/pub/tools`

# U1-6 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-Mikrocontroller 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 Studio + WinAVR (Windows)
  - ◆ GNU Compiler Collection (GCC) unter Linux

## 2 AVR Studio

---

- Entwicklungsumgebung von Atmel
- Simulator und Debugger für alle AVR-Mikrocontroller
- Start über das Startmenü  
*Start - Alle Programme - Atmel AVR Tools - AVR Studio 5.0*

### 3 AVR Studio einrichten (einmalig)

- Importieren des I4-Projekts (einmalig)
  - ◆ File -> Import Project Template
  - ◆ Q:\tools\AVRStmpl.zip
  - ◆ Add to Folder: <Root>
  - ◆ OK
  
- Importieren weiterer Einstellungen
  - ◆ Tools -> Import and Export Settings...
  - ◆ Import selected environment settings
  - ◆ No, just import new settings, overwriting my current settings
  - ◆ Browse...
  - ◆ Q:\tools\SPIC.vsssettings
  - ◆ All Settings
  - ◆ Finish

### 3 Anlegen eines neuen Projekts

---

#### ■ Projekt erstellen

- ◆ File -> New -> Project...
- ◆ Projekt: SPiC
- ◆ Name: aufgabe0
- ◆ Location: P:\
- ◆ Kein Häkchen bei "Create directory for solution"
- ◆ OK

#### ■ Datei anlegen

- ◆ Project -> Add New Item... (Ctrl + Shift + A)
- ◆ C File
- ◆ Name: Entsprechend der Aufgabenstellung

## 4 Erstellen einer main()-Funktion

- Auf dem Mikrocontroller ist die *main()*-Funktion vom Typ *void main(void)* ;
  - Sollte niemals zurückkehren (wohin?)
  - Rückgabetyt 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 (AVRISP)

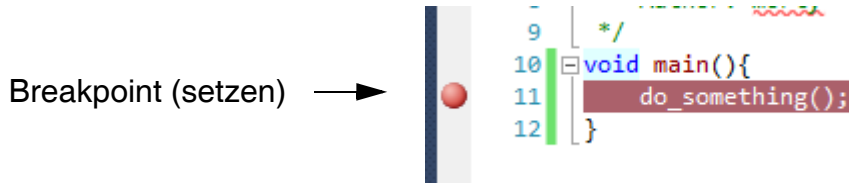
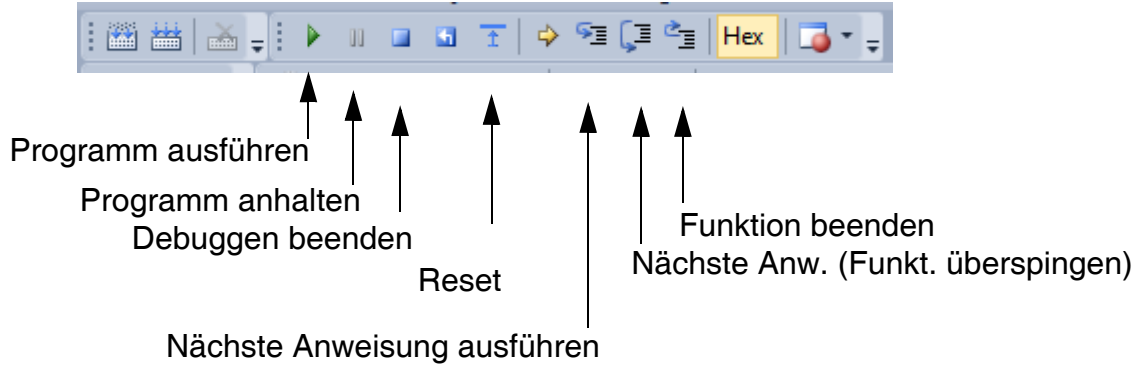
- Mit dem Button: Flash\_ISP
- Alternativ:
  - ◆ Tools -> AVR Programming
  - ◆ Tool: AVRISP
  - ◆ Device: ATmega32
  - ◆ Apply
  - ◆ Verbindung mit Device ID: Read überprüfen
  - ◆ -> Memories
  - ◆ Flash: P:\AufgabeX\Build\Programm.hex
  - ◆ -> Program

## 6 Starten des Simulators/Debuggers (JTAG)

- Debugger wählen: Project -> Properties (ALT-F7)
  - ◆ Debugging -> Select Debugger -> JTAGICE
- Debug -> Start Debugging (F5) - Programm wird übersetzt und geladen
  - ◆ 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
  - ◆ Die Register können auch direkt geändert werden.
- Breakpoints unterbrechen das Programm einer bestimmten Stelle
  - ◆ Codezeile anklicken, dann *F9* oder *Debug - Toggle Breakpoint*
  - ◆ Programm laufen lassen (*F5*), stoppt wenn ein Breakpoint erreicht wird



# 7 Starten des Simulators/Debuggers 2



# U1-7 Programm abgeben

---

- Abgabeskript funktioniert nur in Linux-Umgebung
- Remote-Login auf den Linux-Rechnern
  - ◆ im Startmenü *PuTTY* starten
  - ◆ verbinden mit einem beliebigen Linux-Rechner, z. B. `faii0sr0`
    - Von daheim: `faii0sr0.cs.fau.de`
  - ◆ Gegebenenfalls speichern
  - ◆ Login mit den UNIX-Zugangsdaten
- Im Terminal-Fenster folgendes Kommando ausführen:
  - `/proj/i4gspic/pub/abgabe aufgabe0`
  - ◆ hierbei die aktuelle Aufgabe einsetzen

# U1-8 UNIX/Linux

---

- In der Übung arbeiten wir mit der *AVR-Studio*-Umgebung unter Windows
  - ◆ Grundsätzlich gibt es auch die Möglichkeit, unter Linux zu arbeiten
  - ◆ In der Übung nicht behandelt
  - ◆ Bei persönlichem Interesse mögen die folgenden Informationen hilfreich sein
  - ◆ Ein Schnelleinstieg bedindet sich auf der Webseite

# U1-9 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

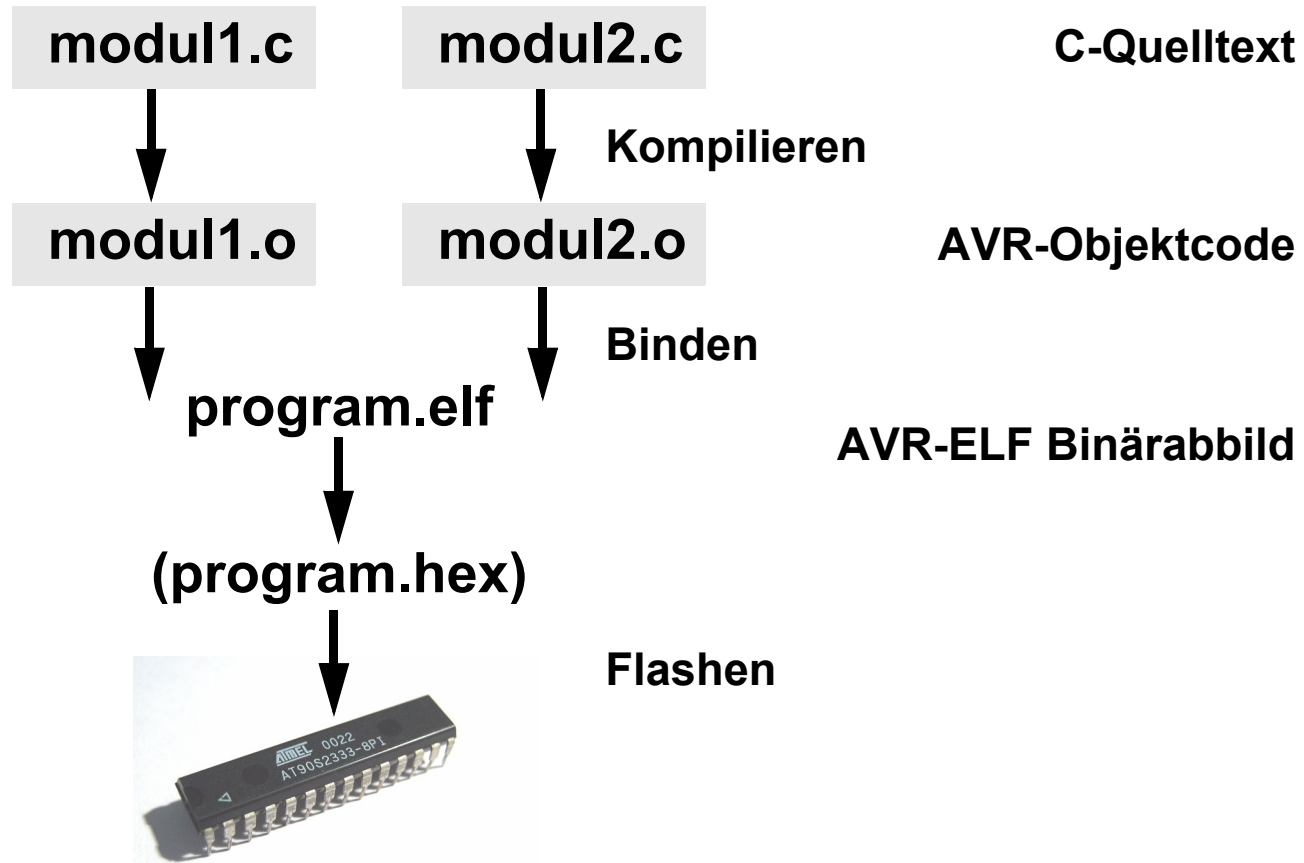
# 1 Toolchain: Vom Quellcode zum geflashten Binärabbild

- (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 Mikrocontroller
  - ◆ 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 Bineschritt 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 Zielfile, 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 Mikrocontrollers
  - ◆ 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/i4gspic/pub/i4/debug.mk program.hex.flash
```
- Nach jedem Reset lädt der **Bootloader** des Mikrocontrollers 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

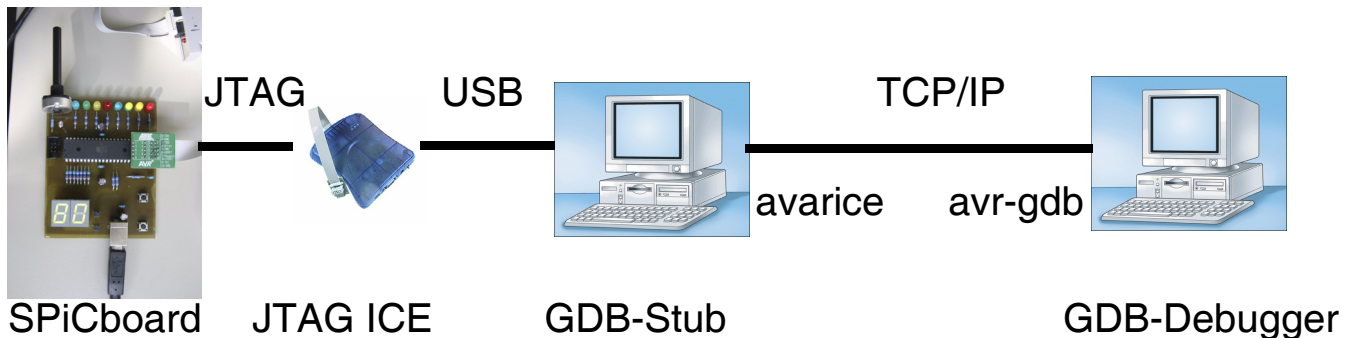
## 7 Debugging unter Linux (*Hintergrundinfo*)

---

- 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 Mikrocontroller
- 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 Mikrocontroller
  - ◆ 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/i4gspic/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 Mikrocontroller 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:Zeilenr. [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 Ausdrucks wird ein Hardware-Breakpoints verbraucht, **watch** auf einen **int** belegt also zwei Hardware-Breakpoints!
  
- Weitere im Reference-Sheet ( Doku-Bereich der Webseite)