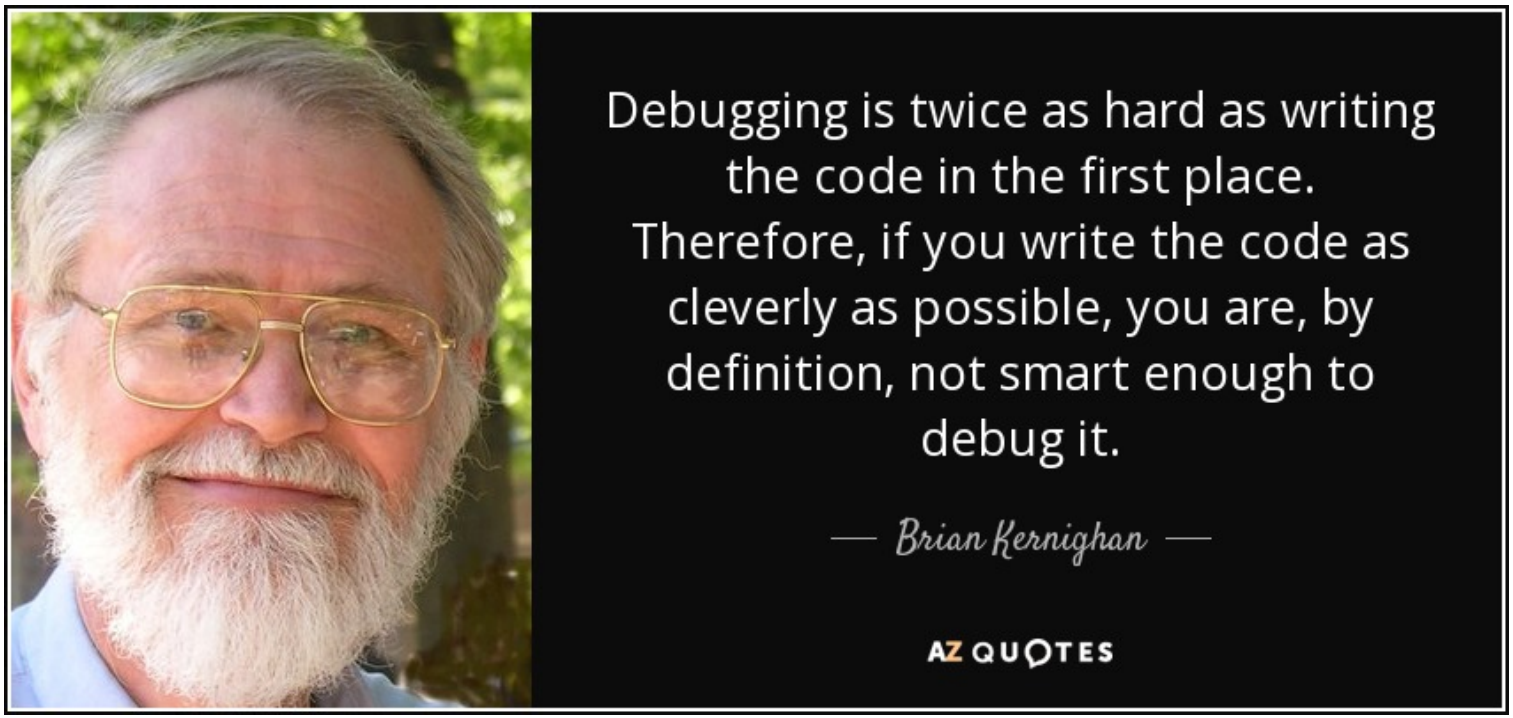
Hardwarenahe Programmierung I

U. Kaiser, R. Kaiser, M. Stöttinger, S. Reith

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022

4. Hochsprachen



https://www.azquotes.com/quote/669106

Motivation

- Fazit Maschinenprogrammierung
  -  Sehr effizient (in der Programm**ausführung**)
  -  Maximale Freiheitsgrade
  -  Maschinenabhängig → nicht *portabel*
  -  Zu geringes Abstraktionsniveau → komplexe Algorithmen sind kaum mehr beherrschbar

⇒ Bedarf nach **Hoch**sprachen, Ziele dabei ...

- Maschinenunabhängig → *portabel*
- Höheres Abstraktionsniveau

- ... im Tausch gegen (*tradeoff*):
  - Einschränkung der Freiheitsgrade (z.B. durch Typkonzept)
  - Effizienz
- Mögliche Ansätze:
  - Automatische Übersetzung Hochsprache → Maschinensprache
  - Direkte Ausführung der Hochsprache durch *Interpreter*
  - Kombination: Übersetzung in eine Zwischensprache und direkte Ausführung der Zwischensprache

Compilersprachen

- Ein **Compiler** übersetzt ein Hochspracheprogramm in Maschinencode
  - → Liest *Quelltext*
  - → Schreibt Maschinencode (z.B. Assembler-Text)
    - Prüft dabei die (syntaktische) Korrektheit des Programms
    - Grundsätzliche Arbeitsweise: Endlicher Automat[1]
- Maschinenunabhängigkeit bei guter Effizienz
- Programme müssen zur Ausführung erst „compiliert“ werden
- Übertragung auf eine neue Architektur erfordert Anpassung eines Teils (des *Backends*) des Compilers
- Sonderfall **Cross**-Compiler: Das Compiler-Programm läuft auf einem Entwicklungsrechner (typischerweise: Intel-PC), generiert Code für eine andere Architektur (z.B. `avr-gcc`, vgl. Praktikum)
- Programm-Quelltexte bleiben (idealerweise) unverändert
- Beispiele: C, C++, Rust, Pascal, Modula-2, Ada, FORTRAN, ...

[1] Siehe Vorlesung „Compilerbau“

Interpreter- / Skriptsprachen

- Ein **Interpreter** führt die Anweisungen eines Hochspracheprogramm **direkt** aus
- Solche Hochsprachen werden auch als **Skriptspachen** bezeichnet
- Maschinenunabhängig, aber weniger effizient als Compiler (Wg. Übersetzungs- bzw. interpretier-Aufwand während der Programmausführung)
- Arbeitsweise: ebenfalls ein endlicher Automat[2]
- Übertragung auf eine neue Architektur erfordert daher neu-compilation des Interpreters (sofern ein Compiler für die Architektur existiert ...)
- Vorteil: unmittelbares, interaktives Arbeiten möglich
- Nachteil: geringere Geschwindigkeit, höherer Ressourcenbedarf
- Beispiele: Python, PHP, Shell, BASIC, ...

[2] Dieser ist meist in einer Compiler-Hochsprache implementiert.

Virtuelle Maschinen

- (S.o.) ein Computer ist ein endlicher Automat

⇒ kann durch ein Programm nachgebildet („*modelliert*“) werden

- Ein solches Modell *emuliert* das Verhalten des Computers exakt: es ist eine **virtuelle Maschine**
- Beispiele für Emulatoren:

Programm	**Emulierte Architektur**	**siehe**
Spike	RISC-V	`https://github.com/riscv-software-src`
Spim	MIPS	`http://spimsimulator.sourceforge.net/`
simulavr	Atmel AVR	`https://www.nongnu.org/simulavr/`
bochs	IBM PC	`https://bochs.sourceforge.io/`
qemu	verschiedene	`https://www.qemu.org/`
...	...	

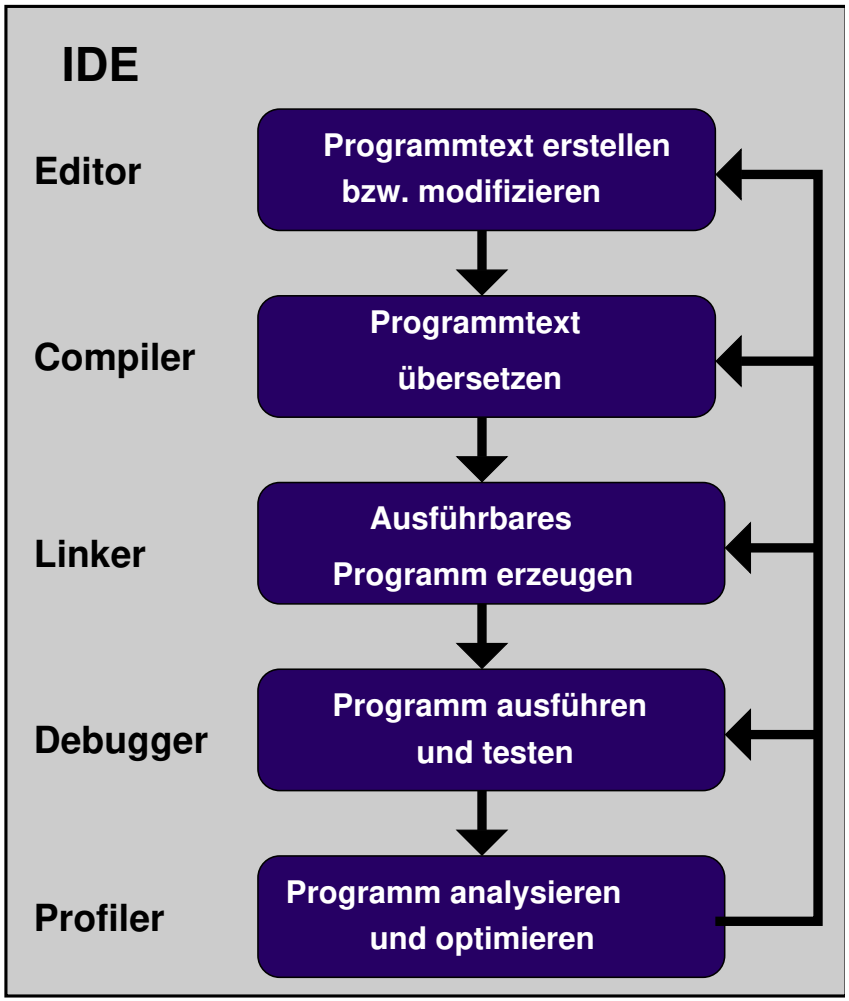
- In gleicher Weise ist auch ein Interpreter ein Modell einer Maschine (einer, die in Hochsprache programmiert wird)
- Auch beliebige Zwischensprachen und nicht technisch existierende Maschinen können modelliert werden

⇒ Kombination aus Compiler und Interpreter:

- Beispiele:
  - Java: Compiler erzeugt *Bytecode*, dieser wird von einer Java Virtual Machine (JVM) interpretiert
  - UCSD-p-Pascal: Compiler erzeugt *p-Code*, wird von p-Maschine interpretiert

Entwicklungszyklus

- Zum Erstellen von Hochsprache-Programmen wird eine Reihe von Software-Werkzeugen benutzt
- Oft sind die Werkzeuge in eine Entwicklungsumgebung (IDE = *Integrated Development Environment*) integriert.
- Beispiele: Eclipse, VSCode, Geany, Kate, ...

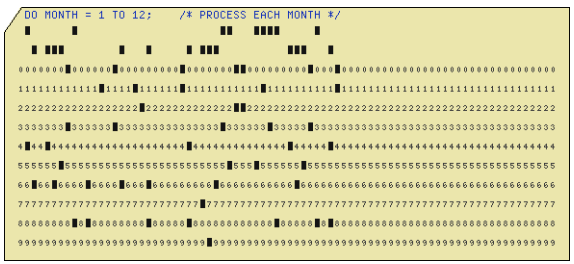


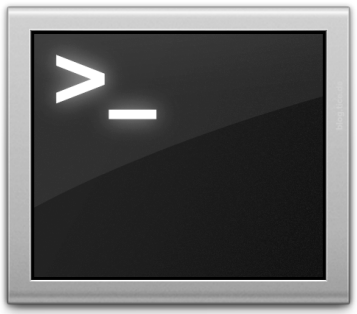
Editor

Hochschule RheinMain

- Editor [3]: Werkzeug zum Erstellen und Bearbeiten von (Programm-)Texten
  - Textdateien: ausschließlich ASCII-Zeichen

  → Können auch mit `cat`, `less` oder `more` ausgegeben werden
- Klassifikation:
  - Zeileneditoren: `edlin`, `ed`
    - ★ Historisch: für Fernschreiber und Lochkarten
    - ★ Verwendung heute: In der Kommandozeile (z.B. Shell)





[3] („Herausgeber“, „Erzeuger“)

Editor

- Editor [3]: Werkzeug zum Erstellen und Bearbeiten von (Programm-)Texten
  - Textdateien: ausschließlich ASCII-Zeichen

  → Können auch mit `cat`, `less` oder `more` ausgegeben werden
- Klassifikation:
  - Zeileneditoren: `edlin`, `ed`
    - ★ Historisch: für Fernschreiber und Lochkarten
    - ★ Verwendung heute: In der Kommandozeile (z.B. Shell)
  - Bildschirmorientierte (*full-screen*) Editoren: `vi`, `emacs`
    - ★ Cursorsteuerung über „Escape-Sequenzen“

    → Anpassung an Terminal über „Termcap“-Environment
    - ★ Heute meist kein Problem, da de facto alle Terminals ANSI-kompatibel sind

[3] („Herausgeber“, „Erzeuger“)

Editor

- Editor [3]: Werkzeug zum Erstellen und Bearbeiten von (Programm-)Texten
  - Textdateien: ausschließlich ASCII-Zeichen

  → Können auch mit `cat`, `less` oder `more` ausgegeben werden
- Klassifikation:
  - Zeileneditoren: `edlin`, `ed`
    - ★ Historisch: für Fernschreiber und Lochkarten
    - ★ Verwendung heute: In der Kommandozeile (z.B. Shell)
  - Bildschirmorientierte (*full-screen*) Editoren: `vi`, `emacs`
    - ★ Cursorsteuerung über „Escape-Sequenzen“

    → Anpassung an Terminal über „Termcap“-Environment
    - ★ Heute meist kein Problem, da de facto alle Terminals ANSI-kompatibel sind
  - Editoren für GUIs: Geany, Kate, XEmacs, Notepad
    - ★ Nur in grafischer Benutzerumgebung lauffähig (X, Windows, etc.)
    - ★ Fließender Übergang zu IDEs

[3] („Herausgeber“, „Erzeuger“)

Eigenschaften von Editoren

- Wünschenswerte Funktionen:
  - Suchen/Ersetzen, evtl. mit regulären Ausdrücken
  - Syntax-Highlighting
  - Codevervollständigung
  - Shell-Fenster
  - Programmierbare Tasten/Makros
  - Vorgängerversion der Datei wird als Backup-Datei (`*.bak`, `* `) beibehalten
- Sehr große Vielfalt an verfügbaren Editoren
- `vi`: auf fast allen UNIX-Systemen verfügbar und ohne GUI funktionsfähig

→ Zumindest rudimentäre Kenntnisse in der Bedienung von `vi` sollte man haben

Verwendung von Editoren

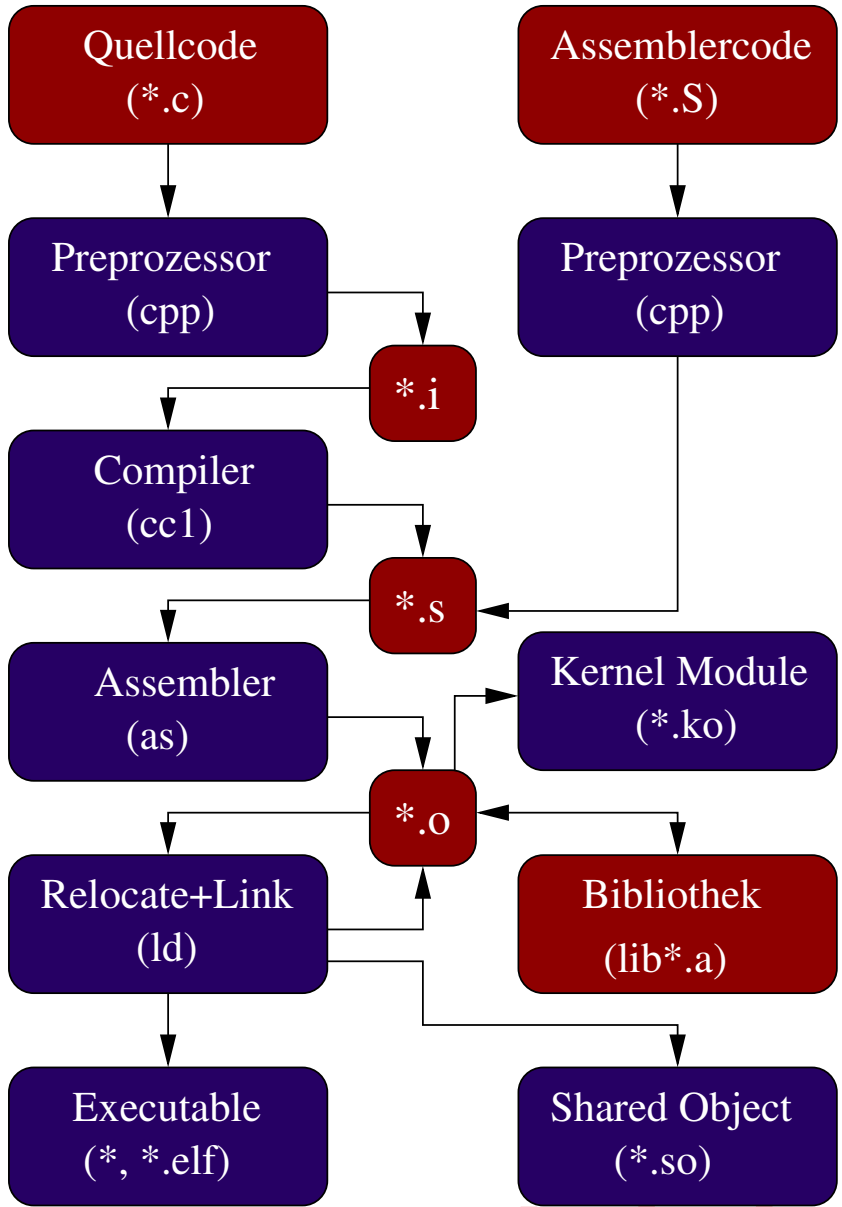
- Erstellen und Bearbeiten von Header-Dateien und Quellcode-Dateien
  - **Header-Dateien** (engl. Headerfiles) sind Dateien, die Informationen zu Datentypen und Datenstrukturen, Schnittstellen von Funktionen etc. enthalten.
    → allgemeine Vereinbarungen, die an verschiedenen Stellen (d. h. in verschiedenen Quellcode- und Headerfiles) einheitlich und konsistent benötigt werden.
  - **Quellcode-Dateien** (engl. Sourcefiles) enthalten den eigentlichen Programmtext.
    → zunächst im Vordergrund
- Erkennbar anhand der Dateinamensendung:
  - „`.h`“: Headerdatei
  - „`.c`“: C-Quellcode-Datei
  - „`.cpp`“, „`.cc`“: C++-Quellcode-Datei

Compiler, Konkret: C-Compiler

Arbeitsschritte:

1. Preprozessor (cpp):
   Text-Vorverarbeitung
   (#-Direktiven)
2. Compiler (cc1):
   C-Text → Assembler-Text
3. Assembler (as):
   Asm-Text → Objektdatei
4. Linker (ld):
   Objektdatei(en)
   +Bibliothek(en)
   → Executable

gcc (bzw. cc) ist ein „*Chain Processor*"



Dateizuordnungen

- Der Chain-Prozessor entscheidet anhand der Dateiendung, was zu tun ist:

Name	Bedeutung	Aktion
`*.c`	C-Quellcode	Preprocess & Compile
`*.C`, `*.cc`, `*.cxx`	C++-Quellcode	Preprocess & (C++-)Compile
`*.i`	Preprozessierter C-Quellcode	Compile
`*.ii`	Preprozessierter C++-Code	(C++-)Compile
`*.s`	Preprozessierter Asm-Code	Assemble
`*.S`	Assembler-Code	Preprocess & Assemble
`*.o`	Objektdatei	Link
`lib*.a`	Bibliothek von Objektdateien	Link
`*.so`	Shared Bibliothek	Mitlinken

Linker (1)

Hochschule RheinMain

- Was ist eigentlich eine Objektdatei?

Vergleich: `hello.o` vs. `hello.elf`

`objdump -d hello.o`

```
00000000 <main>:
 0: 55                push %ebp
 1: 89 e5             mov  %esp,%ebp
 3: 83 e4 f0          and  $0xfffffff0,%esp
 6: 83 ec 10          sub  $0x10,%esp
 9: c7 04 24
    00 00 00 00       movl $0x0,(%esp)
10: e8 fc ff ff ff    call 11 <main+0x11>
15: c9                leave
16: c3                ret
```

`objdump -d hello.elf`

```
080483c4 <main>:
80483c4: 55                push %ebp
80483c5: 89 e5             mov  %esp,%ebp
80483c7: 83 e4 f0          and  $0xfffffff0,%esp
80483ca: 83 ec 10          sub  $0x10,%esp
80483cd: c7 04 24
         a0 84 04 08       movl $0x80484a0,(%esp)
80483d4: e8 1f ff ff ff    call 80482f8 <puts@plt>
80483d9: c9                leave
80483da: c3                ret
..............
```

→ „Fast fertiger" Binärcode

- Bisher nur Platzhalter für Adressbezüge
- Keine absolute Position im Speicher
- Bibliotheksfunktionen (hier: `puts`) nicht enthalten

→ Allgemein: Referenzen zu externen Symbolen

Linker (2)

- Ein **Linker** (dt.: „Binder“) verbindet die einzelnen Objektdateien zu einem fertigen (ausführbaren) Programm.
- Dabei werden ggf. auch weitere Funktions- oder Klassen**bibliotheken** hinzugebunden:
  - Bibliotheken sind Sammlungen von Objektdateien mit kompilierten Funktionen, zu denen oft kein Quellcode verfügbar ist, und die z. B. vom Betriebssystem oder dem C-Laufzeitsystem zur Verfügung gestellt werden.
- Der Linker nimmt dabei noch ausstehende, übergreifende Prüfungen vor
- Auch dabei kann es noch zu Fehlern kommen, z.B. wenn der Linker in der Zusammenschau aller Objectfiles feststellt, dass versucht wird, eine Funktion zu verwenden, die es nirgendwo gibt („undefined reference“).

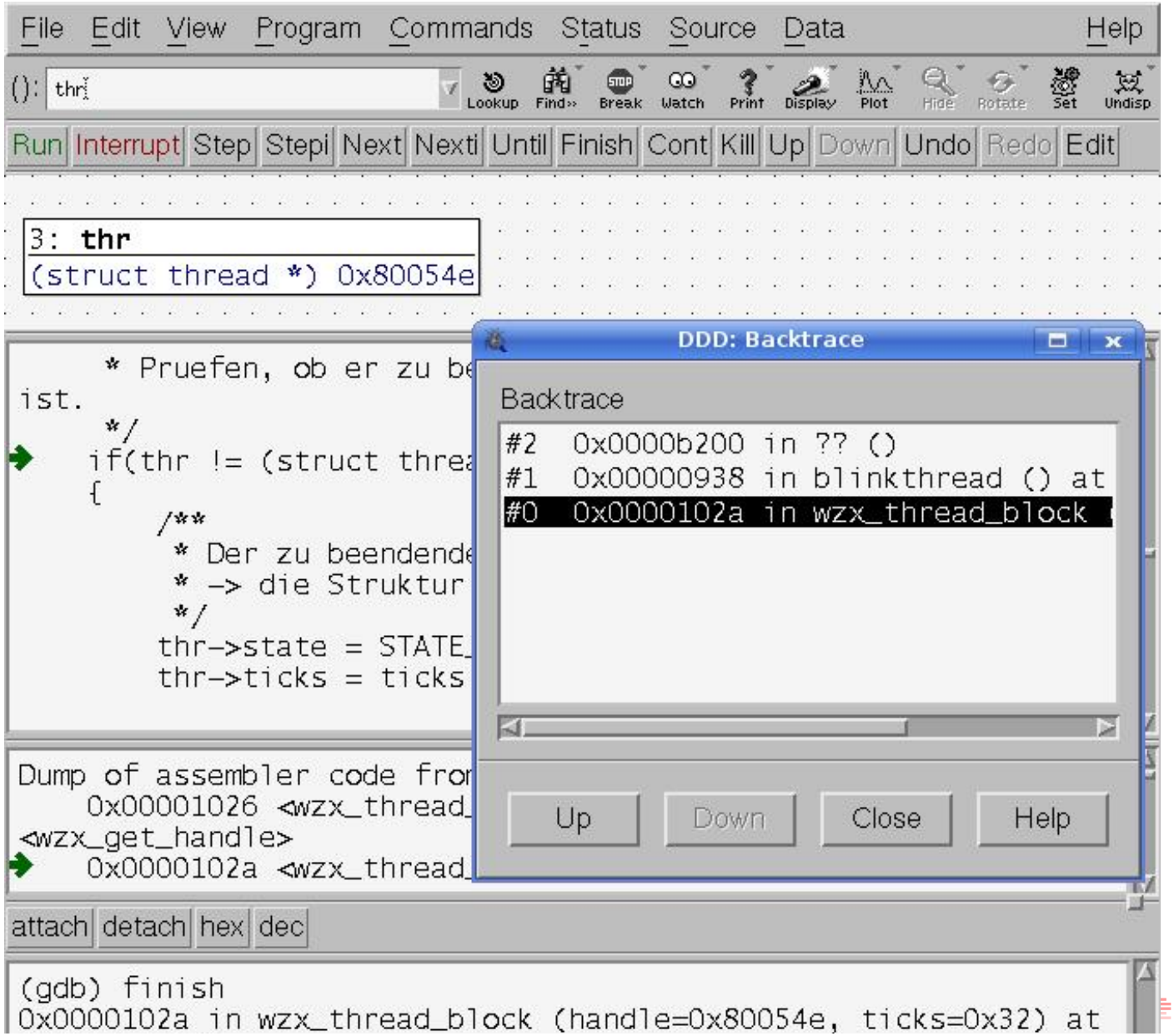
Debugger

- Eigentlich: engl. *Bug* = Käfer, Insekt, Wanze, Laus
- In der IT: *Bug* = Programmierfehler

→ *Debugging*: Finden und Entfernen von Fehlern

- Der **Debugger** dient zum Testen von Programmen:
  - *beobachten* der erstellten Programme bei ihrer Ausführung
  - *eingreifen* in das laufende Programm, z.B. durch Ändern von Variablenwerten
- Nicht nur zur Lokalisierung von Programmierfehlern, sondern auch zur Analyse eines Programms durch Nachvollzug des Programmablaufs
- Auch hilfreich zum interaktiven Erlernen einer Programmiersprache (ähnlich Skriptsprache, s.o)
- GNU-Debugger `gdb`: Bedienung über Kommandozeile
- Verschiedene grafische Frontends: DDD, Insight, gdbgui, ...

Beispiel: GDB-Frontend *DDD*



Self-Hosted oder Cross?

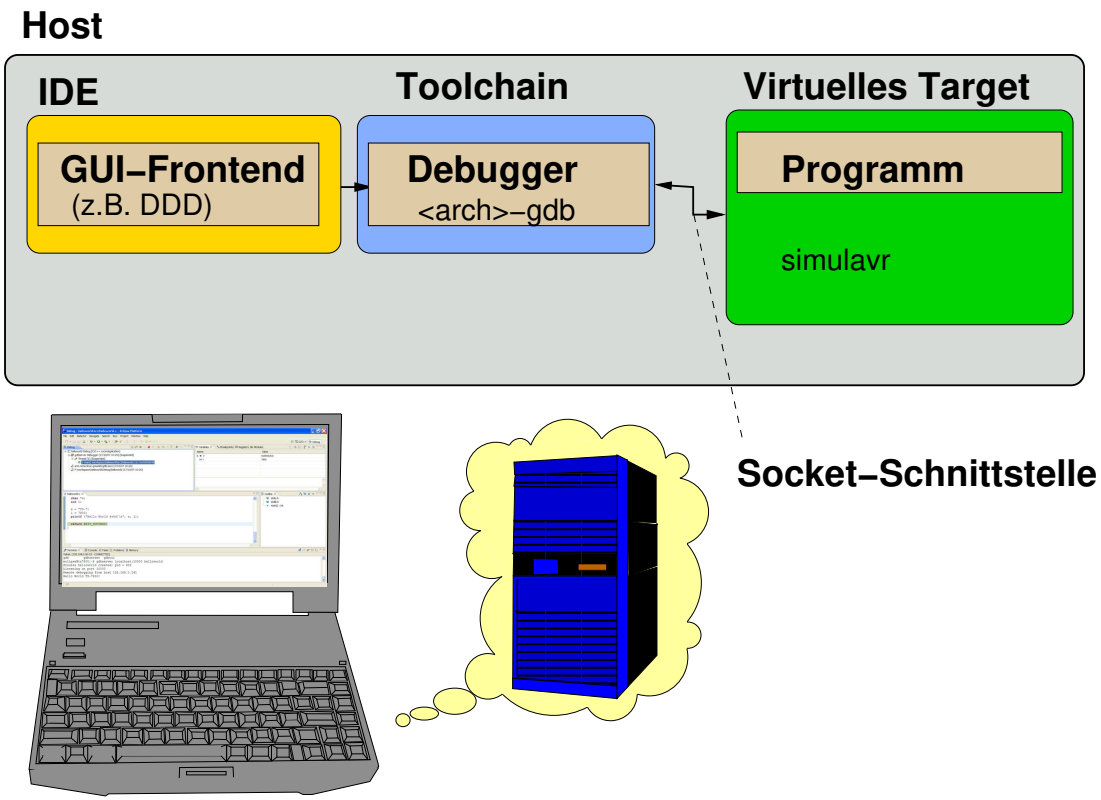
- Mikrocontroller verfügen häufig nicht über Ressourcen, einen Debugger zu betreiben

⇒ Lösung: **Cross**-Debugger

- „Fernsteuern“ der Programmausführung
- Dazu nötige Basisfunktionen:
  - Prozessor anhalten / weiterlaufen lassen
  - Register des (angehaltenen) Prozessors lesen / schreiben
  - Daten- **und** Programmspeicher Lesen **und** Schreiben (Programmspeicher-Schreibzugriff ist für Breakpoints erforderlich)
  - Ausnahmebedingungen (z.B. Speicherzugriffsfehler, Nulldivision, Ungültiger Maschinenbefehl) abfangen und Prozessor anhalten
- *self-hosted* Debugger erreichen dies über spezielle Betriebssystemfunktionen (z.B. Linux: `ptrace(2)`)
- Ein *cross*-Debugger benötigt dazu einen (i.d.R. externen) „Debug-Server“

Cross-Debugging (1)

- Möglichkeit: Zielrechner (*Target*) existiert „nur“ als virtuelle Maschine
- z.B. **simulavr**: Emuliert AVR Mikrocontroller
- Kommunikation mit Debugger TCP/IP- oder UNIX Domain Socket

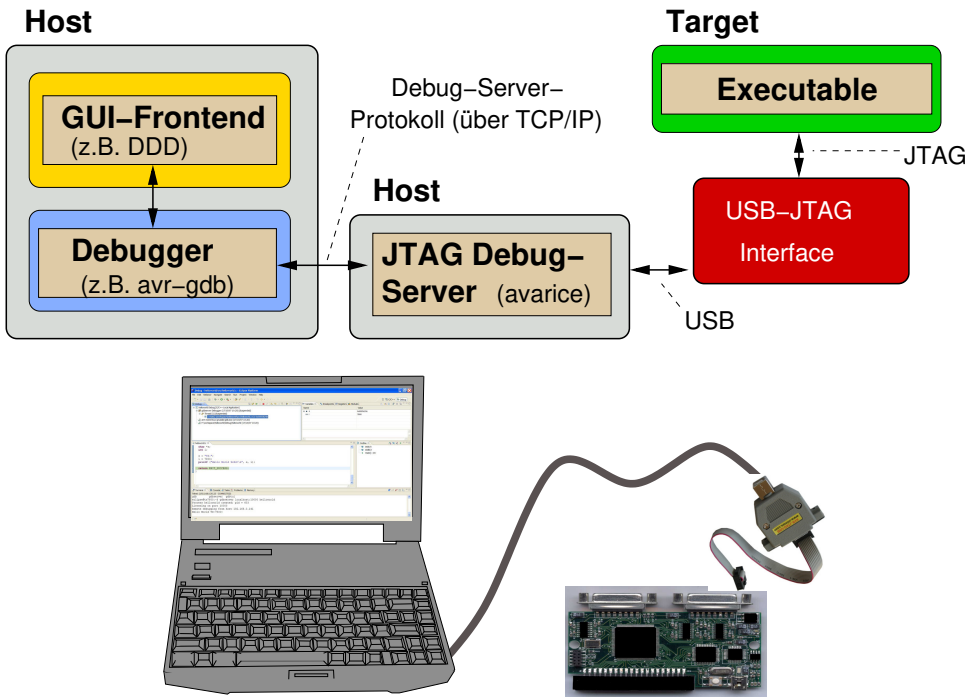


Cross-Debugging (2)

Hochschule RheinMain

- Möglichkeit: Zielrechner verfügt über geeignete Debug-Schnittstelle
- z.B. **avarice**: Debug-Server für AVR ($\geq$ ATmega16) mit JTAG-Schnittstelle
- Kommuniziert mit Debugger über Netzwerk (*TCP/IP-Socket*)

→ Kann auf demselben oder einem anderen Hostrechner arbeiten



Profiler

- Ein **Profiler** überwacht Programme zur Laufzeit
- Erstellt *Laufzeitprofile*: Informationen über
  - die verbrauchte Rechenzeit
  - den in Anspruch genommenen Speicher
- Häufig nicht beides gleichzeitig optimierbar
- In Linux: `gprof`

Weitere Werkzeuge

Weitere Dienstprogramme („GNU binutils“)

- `objdump` – In Objektdateien enthaltene Informationen anzeigen (Sektionen, deren Inhalte, Attribute, etc. Auch: Dissassembler)
- `addr2line` – Konvertieren von Speicheradressen → Quellcode-Zeilen
- `gcov` – Überdeckungsanalyse
- `size` – Sektions- und Gesamtgrößen anzeigen
- `nm` – Symbolinformationen anzeigen
- `strings` – ASCII-Strings innerhalb von Binärdateien anzeigen

Zusammenfassung

- Hochsprachen bieten bessere Portabilität und ermöglichen ein höheres Abstraktionsniveau als Maschinensprache.
- Neben den in dieser LV vorrangig betrachteten Compilersprachen gibt es auch Skriptsprachen und auf virtuellen Maschinen basierende Sprachen.
- Bei der Programmentwicklung kommen Werkzeuge zum Einsatz:
  - Editor, Compiler, Linker, Debugger, Profiler, diverse Analysetools