

# Betriebssystembau (BSB)

## VL 2 – Einstieg in die Betriebssystementwicklung

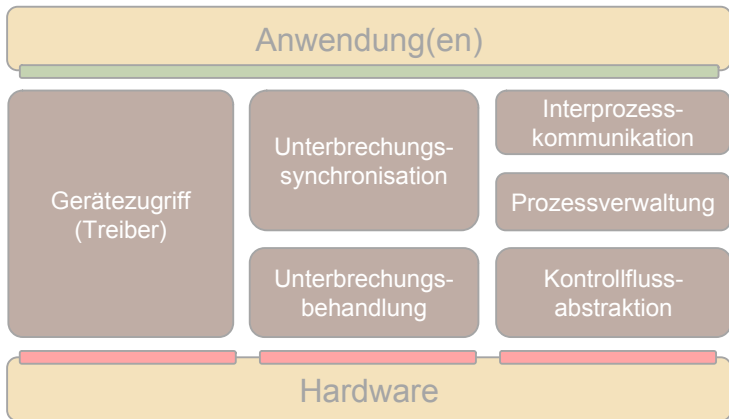
**Alexander Krause**

Lehrstuhl für Informatik 12 – Arbeitsgruppe Systemsoftware / IRB  
Technische Universität Dortmund

<https://sys.cs.tu-dortmund.de/de/lehre/ws25/bsb>

WS 25 – 14. Oktober 2025

# Überblick: Einordnung dieser VL



Betriebssystementwicklung



# Agenda

---

Einordnung

Übersetzen und Linken

Booten

Debugging

Zusammenfassung



# Agenda

---

Einordnung

Übersetzen und Linken

Booten

Debugging

Zusammenfassung



# BS-Entwicklung (oft ein harter Kampf)

- Erste Schritte  
wie bringt man sein System auf die Zielhardware?
  - Übersetzung
  - Bootvorgang
- Testen und *Debugging*  
was tun, wenn das System nicht reagiert?
  - „printf“ *debugging*
  - Emulatoren
  - *Debugger*
  - *Remote-Debugger*
  - Hardwareunterstützung



# Agenda

---

Einordnung

Übersetzen und Linken

Booten

Debugging

Zusammenfassung



# Übersetzung – *Hello, World?*

```
#include <iostream>
```

```
int main () {  
    std::cout << "Hello, World" << std::endl;  
}
```

```
> g++ -o hello hello.cc
```

- Annahme:
  - das Entwicklungssystem läuft unter Linux/x86
  - das Zielsystem ist ebenfalls ein PC
- Läuft dieses Programm auch auf der „nackten“ Hardware?
- Kann man Betriebssysteme überhaupt in einer Hochsprache entwickeln?



# Übersetzung – Probleme u. Lösungen

- kein dynamischer Binder vorhanden
  - alle nötigen **Bibliotheken statisch einbinden**.
- libstdc++ und libc benutzen Linux Systemaufrufe (insbesondere write)
  - die normalen C/C++ **Laufzeitbibliotheken können nicht benutzt werden**. Andere haben wir (meistens) nicht.
- generierte Adressen beziehen sich auf virtuellen Speicher! ("nm hello | grep main" liefert "0804846c T main")
  - die Standardeinstellungen des Binders können nicht benutzt werden. **Man benötigt eine eigene Binderkonfiguration**.
- der Hochsprachencode stellt Anforderungen (Registerbelegung, Adressabbildung, Laufzeitumgebung, Stapel, ...)
  - ein eigener **Startup-Code** (in Assembler erstellt) muss die Ausführung des Hochsprachencodes vorbereiten



# Agenda

---

Einordnung

Übersetzen und Linken

**Booten**

Debugging

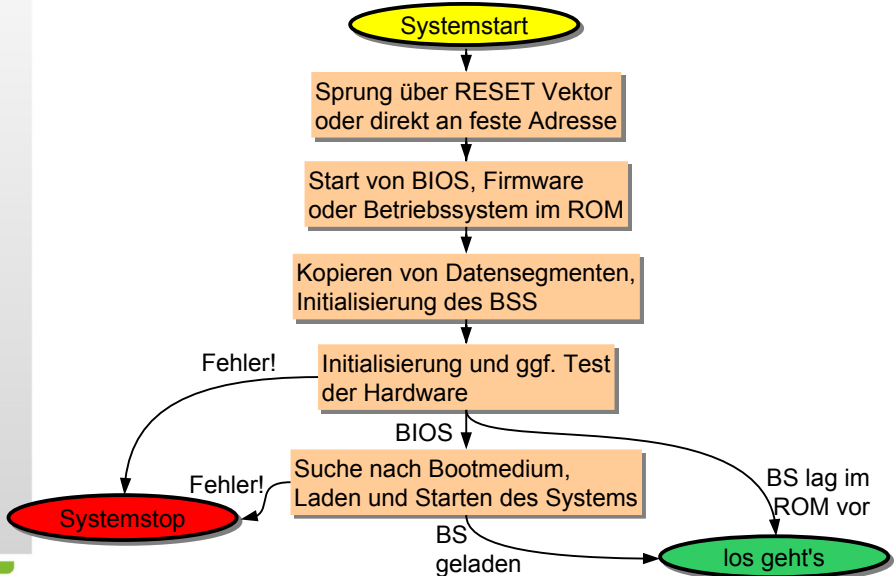
Zusammenfassung



„**Bootstrapping** (englisches Wort für Stiefelschlaufe) bezeichnet einen Vorgang bei dem ein einfaches System ein komplexeres System startet. Der Name des Verfahrens kommt von der **Münchhausen-Methode**.“

„Die **Münchhausen-Methode** bezeichnet allgemein, dass ein System sich selbst in Gang setzt. Die Bezeichnung spielt auf die deutsche Legende von **Baron Münchhausen** an, der sich an seinen eigenen Haaren aus einem Sumpf gezogen haben soll. In der amerikanischen Fassung benutzte er seine Stiefelschlaufen, was die englische Bezeichnung **Bootstrapping** für diese Methode begründete.“

# Bootvorgang



# Bootvorgang beim PC – Bootsektor

- das PC BIOS lädt den 1. Block (512 Bytes) des Bootlaufwerks an die Adresse 0x7c00 und springt dorthin (blind!)
- Aufbau des „Bootsektors“:

## FAT Diskette (DOS/Windows)

| Offset | Inhalt                                 |
|--------|----------------------------------------|
| 0x0000 | jmp boot; nop; (ebxx90)                |
| 0x0003 | Systemname und Version                 |
| 0x000b | Bytes pro Sektor                       |
| 0x000d | Sektoren pro Cluster                   |
| 0x000e | reservierte Sektoren (für Boot Record) |
| 0x0010 | Anzahl der FATs                        |
| 0x0011 | Anzahl der Stammverzeichniseinträge    |
| 0x0013 | Anzahl der logischen Sektoren          |
| 0x0015 | Medium-Deskriptor-Byte                 |
| 0x0016 | Sektoren pro FAT                       |
| 0x001a | Anzahl der Köpfe                       |
| 0x001c | Anzahl der verborgenen Sektoren        |
| 0x001e | boot :                                 |
|        | ...                                    |
| 0x01fe | 0xaa55                                 |



# Bootvorgang beim PC – Bootsektor

- das PC BIOS lädt den 1. Block (512 Bytes) des Bootlaufwerks an die Adresse 0x7c00 und springt dorthin
- Aufbau des „Bootsektors“:

**Alternative**  
(OOSTuBS):

Wichtig ist eigentlich nur der Start und die „**Signatur**“ (0xaa55) am Ende. Alles weitere benutzt der **Boot-Loader**, um das eigentliche System zu laden.

| Offset | Inhalt                             |
|--------|------------------------------------|
| 0x0000 | <code>jmp boot;</code>             |
| 0x0004 | Anzahl der Spuren                  |
| 0x0006 | Anzahl der Köpfe                   |
| 0x0008 | Anzahl der Sektoren                |
| 0x000a | reservierte Sektoren (Setup-Code)  |
| 0x000c | reservierte Sektoren (System)      |
| 0x000e | BIOS Gerätecode                    |
| 0x000f | Startspur der Diskette/Partition   |
| 0x0010 | Startkopf der Diskette/Partition   |
| 0x0011 | Startsektor der Diskette/Partition |
| 0x0010 | <code>boot:</code><br>...          |
| 0x01fe | <b>0xaa55</b>                      |



- einfache, systemspezifische *Boot Loader*
  - Herstellung eines definierten Startzustands der Hard- und Software
  - ggf. Laden weiterer Blöcke mit *Boot Loader Code*
  - Lokalisierung des eigentlichen Systems auf dem Boot-Medium
  - Laden des Systems (mittels Funktionen des BIOS)
  - Sprung in das geladene System
  
- "*Boot Loader*" auf nicht boot-fähigen Disketten
  - Ausgabe einer Fehlermeldung und Neustart
  
- *Boot Loader* mit Auswahlmöglichkeit  
(z.B. im *Master Boot Record* einer Festplatte)
  - Darstellung eines Auswahlmenüs
  - Nachbildung des BIOS beim Booten des ausgewählten Systems
    - Laden des jeweiligen Bootblocks nach 0x7c00 und Start



# Agenda

---

Einordnung

Übersetzen und Linken

Booten

**Debugging**

- Wie entwanzt man ein BS?

- „printf“-Debugging

- Software-Emulatoren

- Debugger

- Source-Level-Debugging

- Remote-Debugging

- Debugging Deluxe

Zusammenfassung



# Debugging



# Der erste dokumentierte „Bug“

9/9

0800 Antam started  
 1000 " stopped - antam ✓  
 1300 (032) MP-MC 1.58247000 9.037846995 correct  
 (033) PRO 2 2.130476415 (2) 4.615925059(-2)  
 correct 2.130676415  
 Relays 6-2 in 033 failed special speed test  
 in relay " " test -  
 Relays changed  
 1100 Started Cosine Tape (Sine check)  
 1525 Started Multi-Adder Test.  
 1545  Relay #70 Panel F  
 (moth) in relay.  
 First actual case of bug being found.  
 1630 Antam started.  
 1700 closed down.

Relay 3145  
 Relay 3370

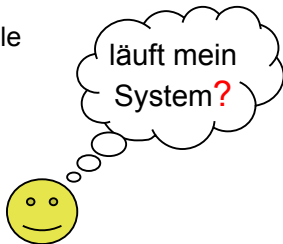
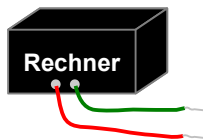


Admiral Grace Hopper

Quelle: Wikipedia

# „printf – Debugging“

- gar nicht so einfach, da es `printf()` per se nicht gibt!
  - oftmals gibt es nicht mal einen Bildschirm
- `printf()` ändert oft auch das Verhalten des *debuggee*
  - mit `printf()` tritt der Fehler nicht plötzlich nicht mehr / anders auf
  - das gilt gerade auch bei der Betriebssystementwicklung
- Strohhalm
  - eine blinkende LED
  - eine serielle Schnittstelle

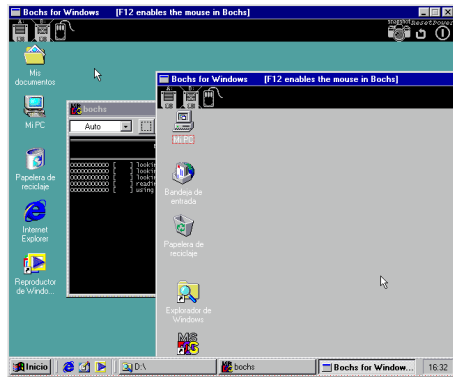


- ahmen reale Hardware in Software nach
  - einfacheres Debugging, da die Emulationssoftware in der Regel kommunikativer als die reale Hardware ist
  - kürzere Entwicklungszyklen
- Vorsicht: am Ende muss das System auf realer Hardware laufen!
  - in Details können sich Emulator und reale Hardware unterscheiden!
  - im fertigen System sind Fehler schwerer zu finden als in einem inkrementell entwickelten System
- übrigens: "virtuelle Maschinen" und "Emulatoren" sind **nicht** gleichbedeutend
  - in VMware wird z.B. kein x86 Prozessor emuliert, sondern ein vorhandener Prozessor führt Maschinencode in der VM direkt aus



# Emulatoren – Beispiel "Bochs"

- emuliert i386, ..., Pentium, AMD64 (Interpreter)
  - optional MMX, SSE, SSE2 und 3DNow! Instruktionen
  - Multiprozessoremulaton
- emuliert kompletten PC
  - Speicher, Geräte (selbst Sound- und Netzwerkkarte)
  - selbst Windows und Linux Systeme laufen in Bochs
- implementiert in C++
- Entwicklungsunterstützung
  - Protokollinformationen, insbesondere beim Absturz
  - eingebauter Debugger (GDB-Stub)



Bochs in Bochs



# Debugging

- ein *Debugger* dient dem Auffinden von Softwarefehlern durch Ablaufverfolgung
  - in Einzelschritten (*single step mode*)
  - zwischen definierten Haltepunkten (*breakpoints*), z.B. bei
    - Erreichen einer bestimmten Instruktion
    - Zugriff auf ein bestimmtes Datenelement
- **Vorsicht:** manchmal dauert die Fehlersuche mit einem Debugger länger als nötig
  - wer gründlich nachdenkt kommt oft schneller zum Ziel
    - Einzelschritte kosten viel Zeit
    - kein Zurück bei versehentlichem Verpassen der interessanten Stelle
  - beim printf-Debugging können Ausgaben besser aufbereitet werden
  - Fehler im Bereich der Synchronisation nebenläufiger Aktivitäten sind interaktiv mit dem Debugger praktisch nicht zu finden
- **praktisch: Analyse von "core dumps"**
  - beim Betriebssystembau allerdings weniger relevant



# Debugging – Beispielsitzung

Setzen eines  
Abbruchpunktes

Start des  
Programms

Ablaufverfolgung  
im Einzelschritt-  
modus

Fortsetzung des  
Programms

```
spinczyk@fau48:~> gdb hello
GNU gdb 6.3
...
(gdb) break main
Breakpoint 1 at 0x8048738: file hello.cc, line 5.
(gdb) run
Starting program: hello

Breakpoint 1, main () at hello.cc:5
5          cout << "hello" << endl;
(gdb) next
hello
6          cout << "world" << endl;
(gdb) next
world
7      }
(gdb) continue
Continuing.

Program exited normally.
(gdb) quit
```



# Debugging – Funktionsweise (1)

- praktisch alle CPUs unterstützen das *Debugging*
  - Beispiel: Intels x86 CPUs
    - die **INT3** Instruktion löst "*breakpoint interrupt*" aus (ein *TRAP*)
      - wird gezielt durch den *Debugger* im Code platziert
      - der *TRAP-Handler* leitet den Kontrollfluss in den *Debugger*
    - durch Setzen des **Trap Flags (TF)** im Statusregister (EFLAGS) wird nach **jeder** Instruktion ein "*debug interrupt*" ausgelöst
      - kann für die Implementierung des Einzelschrittmodus genutzt werden
      - der *TRAP-Handler* wird nicht im Einzelschrittmodus ausgeführt
    - mit Hilfe der **Debug Register DR0-DR7** (ab i386) können bis zu vier Haltepunkte überwacht werden, ohne den Code manipulieren zu müssen
      - erheblicher Vorteil bei Code im ROM/FLASH oder nicht-schreibbaren Speichersegmenten
- nächste Folie



## al

## 2 Entwicklung – Debugging

2-24

|     |    |     |    |     |    |     |    |   |   |       |   |   |   |    |    |    |    |    |    |   |   |   |   |
|-----|----|-----|----|-----|----|-----|----|---|---|-------|---|---|---|----|----|----|----|----|----|---|---|---|---|
| LEN | RW | LEN | RW | LEN | RW | LEN | RW | - | - | G     | - | - | - | GL | GL | GL | GL | GL | GL |   |   |   |   |
| 3   | 3  | 2   | 2  | 1   | 1  | 0   | 0  |   |   | D     |   |   |   | E  | E  | 3  | 3  | 2  | 2  | 1 | 1 | 0 | 0 |
|     |    |     |    |     |    |     |    |   |   | DR7   |   |   |   |    |    |    |    |    |    |   |   |   |   |
| 31  |    |     |    |     |    |     |    |   |   | 16 15 |   |   |   |    |    |    |    |    |    | 0 |   |   |   |

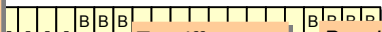


## die Debug Register des 80386

### Breakpoint Register

|                               |     |
|-------------------------------|-----|
| breakpoint 0: lineare Adresse | DR0 |
| breakpoint 1: lineare Adresse | DR1 |
| breakpoint 2: lineare Adresse | DR2 |
| breakpoint 3: lineare Adresse | DR3 |
| reserviert                    | DR4 |
| reserviert                    | DR5 |

Abbruch-Ereignis  
00: Befehlsausführung  
01: Schreiben  
10: I/O (ab Pentium)  
11: Schreiben/Lesen

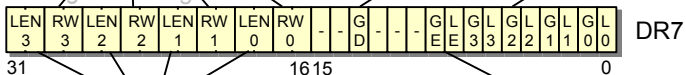


16 15

Zugriffssperre  
für DR0-7

Breakpoint-Freigabe  
(lokal, global)

### Debug Steuerregister



31

16 15

0

DR7

Länge des überwachten  
Speicherbereichs

exakter Daten-Breakpoint  
(lokal, global)

# Debugging – Funktionsweise (3)

- besonders effektiv wird Debugging, wenn das Programm im Quelltext visualisiert wird (*source-level debugging*)
  - erfordert Zugriff auf den Quellcode und Debug-Informationen
  - muss durch den Übersetzer unterstützt werden

```
lohmann@fau148:~> g++ -o hello -g hello.cc
lohmann@fau148:~> objdump --section-headers hello
hello:      file format elf32-i386
Sections:
Idx Name          Size      VMA           LMA           File off  Algn
...
26 .debug_aranges 00000098  00000000  00000000  00000ca0  2**3
    CONTENTS, READONLY, DEBUGGING
27 .debug_pubnames 00000100  00000000  00000000  00000d38  2**0
    CONTENTS, READONLY, DEBUGGING
28 .debug_info     000032b8  00000000  00000000  00000e38  2**0
    CONTENTS, READONLY, DEBUGGING
29 .debug_abbrev   00000474  00000000  00000000  000040f0  2**0
    CONTENTS, READONLY, DEBUGGING
30 .debug_line     000003ac  00000000  00000000  00004564  2**0
    CONTENTS, READONLY, DEBUGGING
31 .debug_frame    0000008c  00000000  00000000  00004910  2**2
    CONTENTS, READONLY, DEBUGGING
32 .debug_str      000001c7  00000000  00000000  0000499c  2**0
    CONTENTS, READONLY, DEBUGGING
```

# Remote Debugging

- bietet die Möglichkeit Programme auf Plattformen zu *debuggen*, die (noch) kein interaktives Arbeiten erlauben
  - setzt eine Kommunikationsverbindung voraus (seriell, Ethernet, ...)
  - erfordert einen Gerätetreiber
  - der Zielrechner kann auch ein Emulator sein (z.B. Bochs)
- die *Debugging*-Komponente auf dem Zielsystem (*stub*) sollte möglichst einfach sein



- das Kommunikationsprotokoll  
("GDB Remote Serial Protocol" - RSP)
  - spiegelt die Anforderungen an den gdb *stub* wieder
  - basiert auf der Übertragung von ASCII Zeichenketten
  - Nachrichtenformat: **\$**<Kommando oder Antwort>**#**<Prüfsumme>
  - Nachrichten werden unmittelbar mit **+** (OK) oder **-** (Fehler) beantwortet
- Beispiele:
  - **\$g#67** ► Lesen aller Registerinhalte
    - Antwort: **+** **\$**123456789abcdef0...**#**... ► Reg. 1 ist 0x12345678, 2 ist 0x9...
  - **\$G123456789abcdef0...#...** ► Setze Registerinhalte
    - Antwort: **+** **\$**OK**#**9a ► hat funktioniert
  - **\$m4015bc,2#5a** ► Lese 2 Bytes ab Adresse 0x4015bc
    - Antwort: **+** **\$**2f86**#**06 ► Wert ist 0x2f86



## Remote Debugging – Beispiel gdb (2)

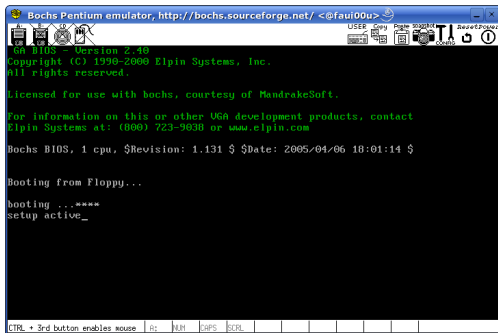
- das Kommunikationsprotokoll – kompletter Umfang
  - Register- und Speicherbefehle
    - lese/schreibe alle Register
    - lese/schreibe einzelnes Register
    - lese/schreibe Speicherbereich
  - Steuerung der Programmausführung
    - letzte Unterbrechungsursache abfragen
    - Einzelschritt
    - mit Ausführung fortfahren
  - Sonstiges
    - Ausgabe auf der *Debug* Konsole
    - Fehlernachrichten
- allein "schreibe einzelnes Register", "lese/schreibe Speicherbereich" und "mit Ausführung fortfahren" müssen notwendigerweise vom *stub* implementiert werden



# Remote Debugging – mit Bochs

- durch geeignete Konfigurierung vor der Übersetzung kann der Emulator Bochs auch einen gdb *stub* implementieren

```
> bochs-gdb build/bootdisk.img  
...  
Waiting for gdb connection on  
localhost:10452
```



The screenshot shows a window titled "Bochs Pentium emulator, http://bochs.sourceforge.net/ <@fau100u>". The window contains a text-based interface for the BIOS. The text displayed is as follows:

```
GA BIOS - Version 2.40  
Copyright (C) 1990-2000 Elpin Systems, Inc.  
All rights reserved.  
  
Licensed for use with bochs, courtesy of MandrakeSoft.  
  
For information on this or other UGA development products, contact  
Elpin Systems at: (800) 723-9038 or www.elpin.com  
  
Bochs BIOS, 1 cpu, $Revision: 1.131 $ $Date: 2005/04/06 18:01:14 $  
  
Booting from Floppy...  
booting ...*****  
setup active_
```

At the bottom of the window, there is a status bar with the text "CTRL + 3rd button enables mouse" and a row of buttons: "H:", "NUM", "CHPS", "SCRL", and several empty buttons.

# Remote Debugging – mit Bochs

```
> gdb build/system
GNU gdb 6.3-debian
...
(gdb) break main
Breakpoint 1 at 0x11fd8: file main.cc, line 38.
(gdb) target remote localhost:10452
Remote debugging using localhost:10452
0x0000fff0 in ?? ()
(gdb) continue
Continuing.

Breakpoint 1, main () at main.cc:38
38      Application application(appl_stack+sizeof(appl_stack));
(gdb) next
43      for (y=0; y<25; y++)
(gdb) next
44      for (x=0; x<80; x++)
(gdb) next
45      kout.show (x, y, ' ', CGA_Screen::STD_ATTR);
(gdb) continue
Continuing.
```

- viele Prozessorhersteller integrieren heute Hardwareunterstützung für *Debugging* auf ihren Chips (*OCDS – On Chip Debug System*)
  - BDM, OnCE, MPD, JTAG
- i.d.R. einfaches serielles Protokoll zwischen *Debugging*-Einheit und externem *Debugger* (Pins sparen!)
- Vorteile:
  - der *Debug Monitor* (z.B. *gdb stub*) belegt keinen Speicher
  - Implementierung eines *Debug Monitors* entfällt
  - Haltepunkte im ROM/FLASH durch Hardware-Breakpoints
  - Nebenläufiger Zugriff auf Speicher und CPU Register
  - mittels Zusatzhardware ist zum Teil auch das Aufzeichnen des Kontrollflusses zwecks nachträglicher Analyse möglich

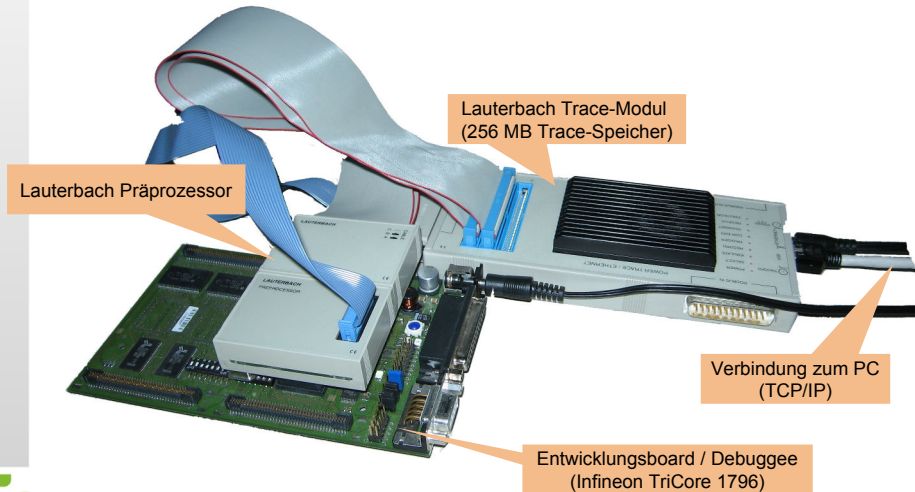


- *"Background Debug Mode"* - eine *on-chip debug* Lösung von Motorola
- serielle Kommunikation über drei Leitungen (DSI, DSO, DSCLK)
- BDM Kommandos der 68k und ColdFire Prozessoren
  - RAREG/RDREG – Read Register
    - lese bestimmtes Daten- oder Adressregister
  - WAREG/WDREG – Write Register
    - schreibe bestimmtes Daten- oder Adressregister
  - READ/WRITE – Read Memory/Write Memory
    - lese/schreibe eine bestimmte Speicherstelle
  - DUMP/FILL – Dump Memory/Fill Memory
    - lese/fülle einen ganzen Speicherblock
  - BGND/GO – Enter BDM/Resume
    - Ausführung stoppen/wieder aufnehmen



# Debugger Deluxe: Hardware-Lösung

## ■ Lauterbach Hardware-Debugger



# Debugger Deluxe: Lauterbach-Frontend

TRACE32

File Edit View Var Break Run CPU Misc Trace Perf Cov TriCore Window Help

Registers / SPOTLIGHT

| Register | Value    | Address  | Comment |
|----------|----------|----------|---------|
| V        | D0       | FFFC     | A1      |
| SV       | D2       | FFFC00FA | A2      |
| AV       | D3       | FFFC00F2 | A3      |
| SAV      | D4       | FFFC0000 | A4      |
| PRS      | D5       | FFFC00A0 | A5      |
| IO       | D6       | 8        | A6      |
| IS       | D7       | FFFC0002 | A7      |
| GW       | D8       | 0        | A8      |
| CDE      | D9       | 0        | A9      |
| CDC      | D10      | 0        | A10     |
| D11      | 0        | 0        | A11     |
| D12      | 0        | 0        | A12     |
| D13      | 0        | 0        | A13     |
| D14      | 0        | 0        | A14     |
| D15      | A010622  | A15      | A15     |
| PSW      | 08000BFF | PC       | PC      |
| PKCI     | 0040007F | TSP      | TSP     |
| FCR      | 0000007E | ICR      | ICR     |
| LCK      | 00000002 | BTIV     | BTIV    |

os::krm::theTasks = {  
 pri\_ = 3,  
 state\_ = SUSPENDED,  
 func\_ = 0x04000290,  
 stack\_ = 0x00002150,  
 interrupted\_ = 0}

Step Over Next Return Up Go Break Mode Find

| addr        | line     | code | label           | mnemonic              | comment                |
|-------------|----------|------|-----------------|-----------------------|------------------------|
| P:D:00020C  | 4-08     |      | ld16.bu         | d4,[a15]0x8           |                        |
| P:D:00020E  | 004E0010 |      |                 | 0x0400037A            | ; os::krm::SchadImp1:: |
| P:D:00020E2 | 3000     |      | ret16           |                       |                        |
| P:D:00020E4 | 3000     |      | callback::ret16 |                       |                        |
| P:D:00020E6 | FF000091 |      | main:           | movh.a                | a15,0xF000             |
| P:D:00020EA | 0482     |      |                 | mov16                 | d4,0x0                 |
| P:D:00020EC | 40000091 |      |                 | movh.a                | a4,0x0000              |
| P:D:00020F0 | 8030FF09 |      | lea             | a15,[a15]0x2FC        |                        |
| P:D:00020F4 | 52304409 |      | lea             | a4,[a4]0x2170         |                        |
| P:D:00020F8 | F04C     |      | ld16.v          | d15,[a15]             |                        |
| P:D:00020FA | F601FF87 |      | insert          | d15,d15,0x0F,0x0C,0x1 |                        |
| P:D:00020FE | 0F68     |      | st16.v          | [a15],d15             |                        |
| P:D:000300  | 00940060 |      | call            | 0x04000428            | ; os::krm::OSControl:: |
| P:D:0003004 | 1292     |      | mov16           | d2,0x0                | ; return,0             |
| P:D:0003006 | 3000     |      | ret16           |                       |                        |
| P:D:0003008 | 02200060 |      | __main:         | call                  | 0x04000762             |
| P:D:000300C | 02010060 |      |                 | call                  | 0x0400070E             |
| P:D:0003010 | 02440060 |      |                 | call                  | 0x04000798             |
| P:D:0003014 | 01EC0060 |      |                 | call                  | 0x040006EC             |
| P:D:0003018 | 001E0010 |      |                 | j                     | 0x04000354             |
| P:D:000301C | 1282     |      | hw::irq::mov16  |                       | d2,0x1                 |
| P:D:000301E | 0000     |      | nop16           |                       |                        |
| P:D:0003020 | 080C     |      | ji16            |                       | a11                    |
| P:D:0003022 | 1282     |      | hw::irq::mov16  |                       | d2,0x1                 |
| P:D:0003024 | 0000     |      | nop16           |                       |                        |
| P:D:0003026 | 080C     |      | ji16            |                       | a11                    |
| P:D:0003028 | 1282     |      | hw::irq::mov16  |                       | d2,0x1                 |
| P:D:000302A | 0000     |      | nop16           |                       |                        |
| P:D:000302C | 080C     |      | ji16            |                       | a11                    |
| P:D:000302E | 1282     |      | hw::irq::mov16  |                       | d2,0x1                 |
| P:D:0003030 | 0000     |      | nop16           |                       |                        |
| P:D:0003032 | 080C     |      | ji16            |                       | a11                    |
| P:D:0003034 | 1282     |      | hw::irq::mov16  |                       | d2,0x1                 |
| P:D:0003036 | 0000     |      | nop16           |                       |                        |
| P:D:0003038 | 080C     |      | ji16            |                       | a11                    |
| P:D:000303A | 40000091 |      | hw::irq::movh.a | a4,0x0000             | ; _tc_reg_A11          |
| P:D:000303E | 52104409 |      | lea             | a4,[a4]0x2150         |                        |

B::

emulate trigger devices trace Data Var PERF SYSTEM Step Go Break Register sYmbol other previous

D:D0002028 \Measure\main:os::krm::theTasks stopped MIX UP

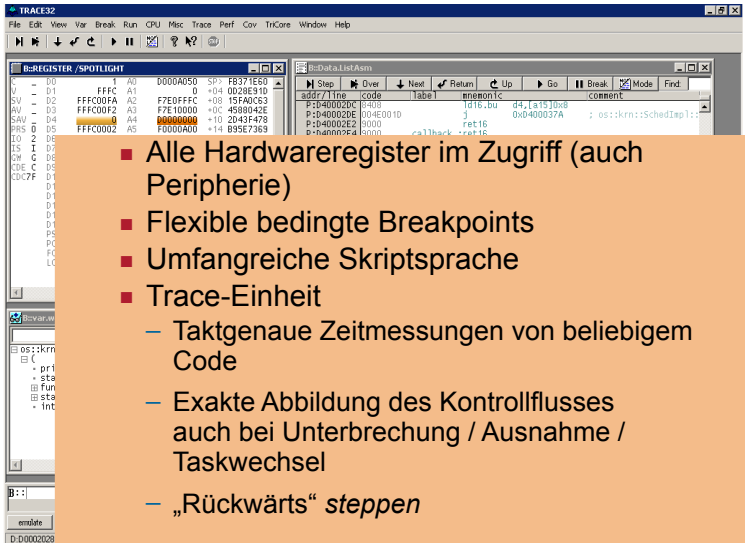
# Debugger Deluxe: Lauterbach-Frontend

The screenshot displays the TRACE32 debugger interface. The main window shows a list of instructions with their addresses, assembly code, and execution times. The list is titled "B::trace.list address list.asm ti.zero ti.back". The instructions are as follows:

| record    | address         | ti.zero | ti.back  |
|-----------|-----------------|---------|----------|
| *****     |                 |         |          |
| -00000125 | P: A1000040     | 0.000   |          |
|           | dsync           |         |          |
| -00000124 | P: A1000044     | 0.240us | 0.240us  |
|           | nop16           |         |          |
| -00000123 | P: A1000046     | 0.240us | <0.020us |
|           | nop16           |         |          |
| -00000117 | P: A1000048     | 0.480us | 0.240us  |
|           | bsr16 0x2       |         |          |
| -00000112 | P: A100004A     | 0.960us | 0.480us  |
|           | call 0xA1000E4E |         |          |
| -00000099 | P: A1000E4E     | 3.140us | 2.180us  |
|           | ret16           |         |          |
| -00000085 | P: A100004E     | 6.020us | 2.880us  |
|           | rsilcx          |         |          |
| -00000081 | P: A1000052     | 6.260us | 0.240us  |
|           | nop16           |         |          |
| -00000080 | P: A1000054     | 6.500us | 0.240us  |
|           | rfe16           |         |          |
| *****     |                 |         |          |

An orange callout box points to the instruction list with the text: „TRACE32“ erlaubt das Aufzeichnen von Programmabläufen. Zeiten werden mit hoher Auflösung protokolliert.

# Debugger Deluxe: Lauterbach-Frontend



The screenshot displays the TRACE32 debugger interface. The top menu bar includes File, Edit, View, Var, Break, Run, CPU, Misc, Trace, Perf, Cov, TriCore, Window, and Help. Below the menu is a toolbar with various icons for navigation and execution. The main window is divided into several panes:

- B:REGISTER / SPOTLIGHT**: Shows a list of registers (C, V, SV, AV, SAV, PRS, IO, TS, CDE, CDC7F) and their values. The SP register is highlighted with a value of 00000000.
- B:Data.ListAsm**: Shows a list of memory addresses and their corresponding assembly instructions. The first instruction is `addr7 line code label mnemonic comment` with values `00000000 00000000 00000000 00000000 00000000 00000000`.
- Var**: Shows a list of variables and their values. The first variable is `os::krm` with a value of `(`.

Overlaid on the screenshot is a list of features:

- Alle Hardwareregister im Zugriff (auch Peripherie)
- Flexible bedingte Breakpoints
- Umfangreiche Skriptsprache
- Trace-Einheit
  - Taktgenaue Zeitmessungen von beliebigem Code
  - Exakte Abbildung des Kontrollflusses auch bei Unterbrechung / Ausnahme / Taskwechsel
  - „Rückwärts“ *steppen*
  - ...



# Agenda

---

Einordnung

Übersetzen und Linken

Booten

Debugging

**Zusammenfassung**



- Betriebssystementwicklung unterscheidet sich deutlich von gewöhnlicher Applikationsentwicklung:
  - Bibliotheken fehlen
  - die „nackte“ Hardware bildet die Grundlage
- die ersten Schritte sind oft die schwersten
  - Übersetzung
  - Bootvorgang
  - Systeminitialisierung
- komfortable Fehlersuche erfordert eine Infrastruktur
  - Gerätetreiber für *printf-Debugging*
  - STUB und Verbindung/Treiber für *Remote Debugging*
  - Hardware Debugging-Unterstützung wie mit BDM
  - Optimal: Hardware-Debugger wie Lauterbach

