

Betriebssysteme (BS)

VL 2.3 – Betriebssystementwicklung – Debugging

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 21 – 25. Oktober 2021



https://www4.cs.fau.de/Lehre/WS21/V_BS

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



9/9

0800 Antan started

1000 " stopped - antan ✓

1300 (032) MP - MC ~~1.982647000~~
2.130476415 (2) 4.615925059(-2)

(033) PRO 2 2.130476415
conch 2.130676415

Relays 6-2 in 033 failed special speed test
in relay " 10.000 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 Antan started.

1700 closed down.

Relay 3375

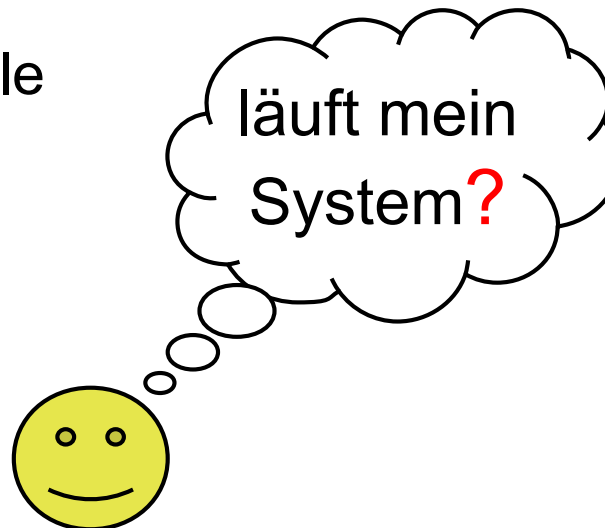
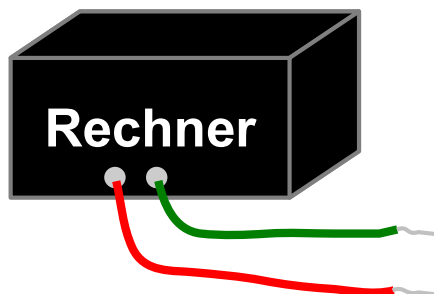


vs/dl



„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



(Software-)Emulatoren

- 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



Debugging – Funktionsweise (2)

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

Debug Statusregister

-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	B T	B S	B D	-	-	-	-	-	-	-	-	-	B 3	B 2	B 1	B 0	DR6
31																16	15														0	

Debug Steuerregister

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



Debugging – Funktionsweise (2)

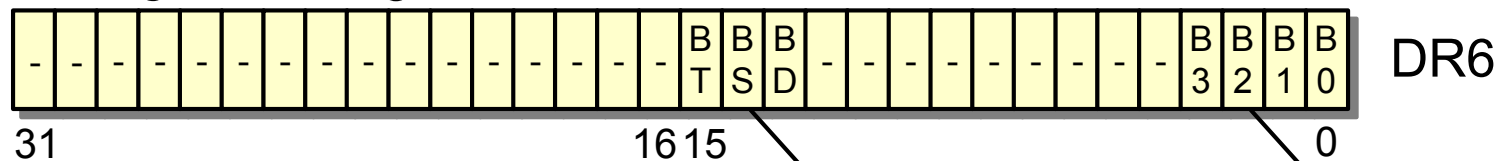
die Debug Register des 80386

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

gibt dem *Trap Handler*
Auskunft über die
Ursache des *Traps*

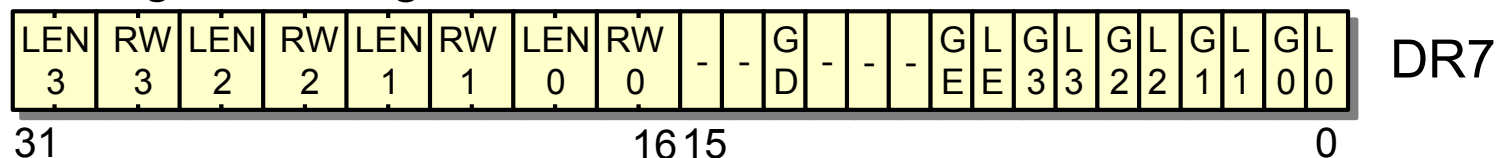
Debug Statusregister



Einzelsschritt

Breakpoint 0-3

Debug Steuerregister



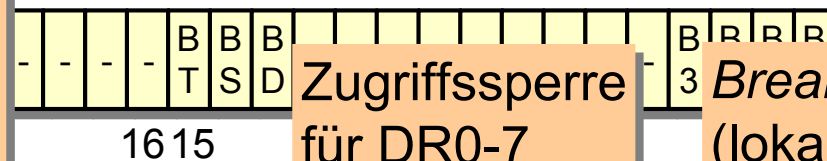
Debugging – Funktionsweise (2)

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

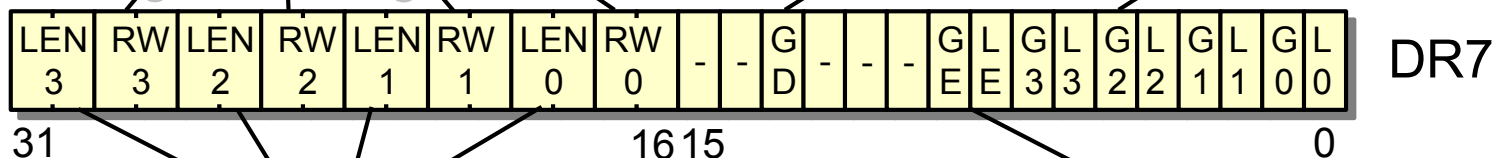
Abbruch-Ereignis
00: Befehlsausführung
01: Schreiben
10: I/O (ab Pentium)
11: Schreiben/Lesen



Zugriffssperre
für DR0-7

Breakpoint-Freigabe
(lokal, global)

Debug Steuerregister



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

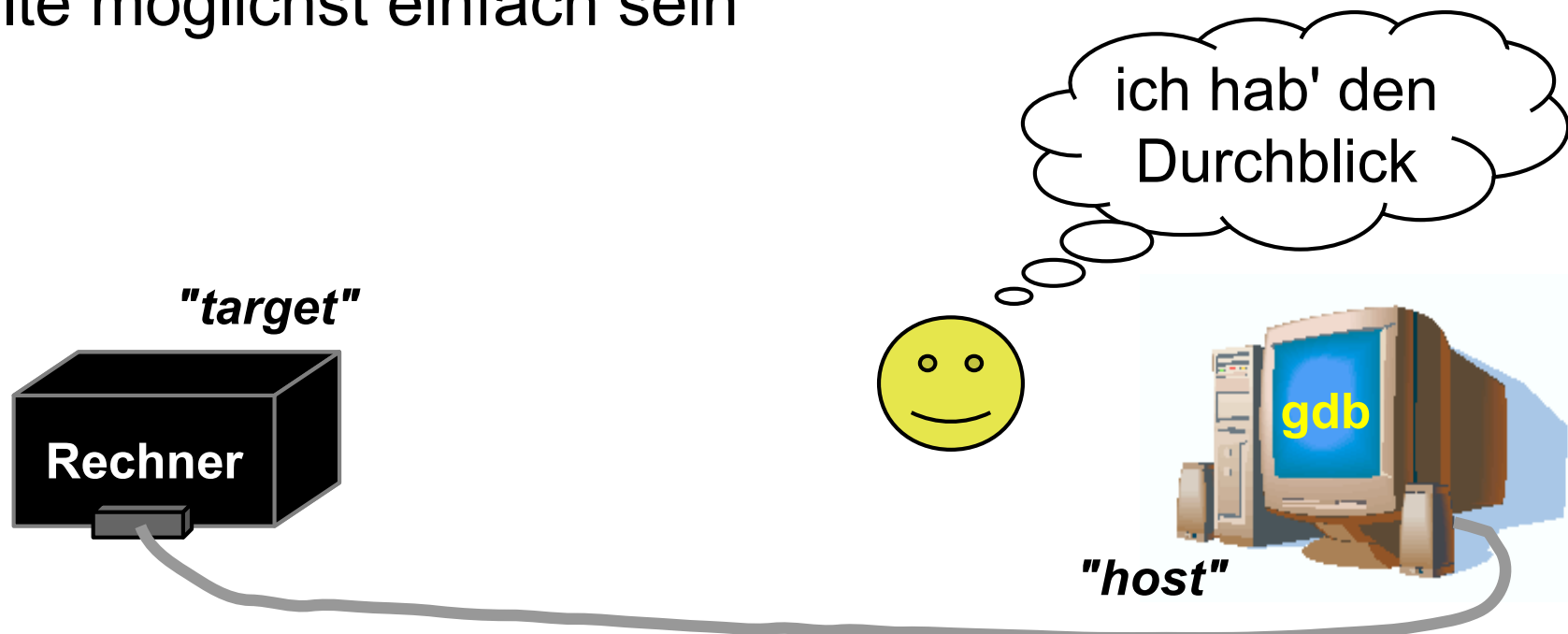
```
lohmann@fai48:~> g++ -o hello -g hello.cc
lohmann@fai48:~> objdump --section-headers hello
hello:      file format elf32-i386
Sections:
Idx Name          Size      VMA           LMA           File off  Algn
...
 26 .debug_aranges 00000098  00000000  00000000  00000ca0  2**3
      CONTENTS, READONLY, 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
```

```
lohmann@fai48:~>
```



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



Remote Debugging – Beispiel gdb (1)

- 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)

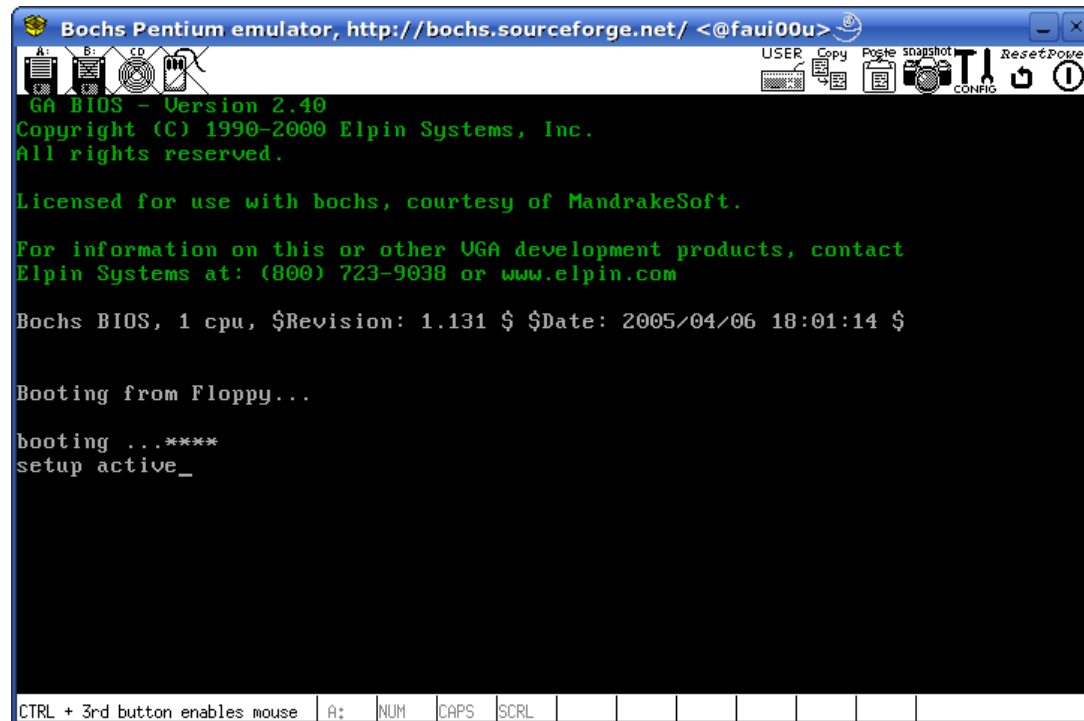
- das Kommunikationsprotokoll – kompletter Umfang
 - Register- und Speicherbefehle
 - lese/schreibe alle Register
 - lese/schreibe einzelnes Register
 - lese/schreibe Speicherbereich
 - Steuerung der Programmausführung
 - letzte Unterbrechungsursache abfragen
 - Einzelschritt
 - mit Ausführung fortfahren
 - 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
```



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



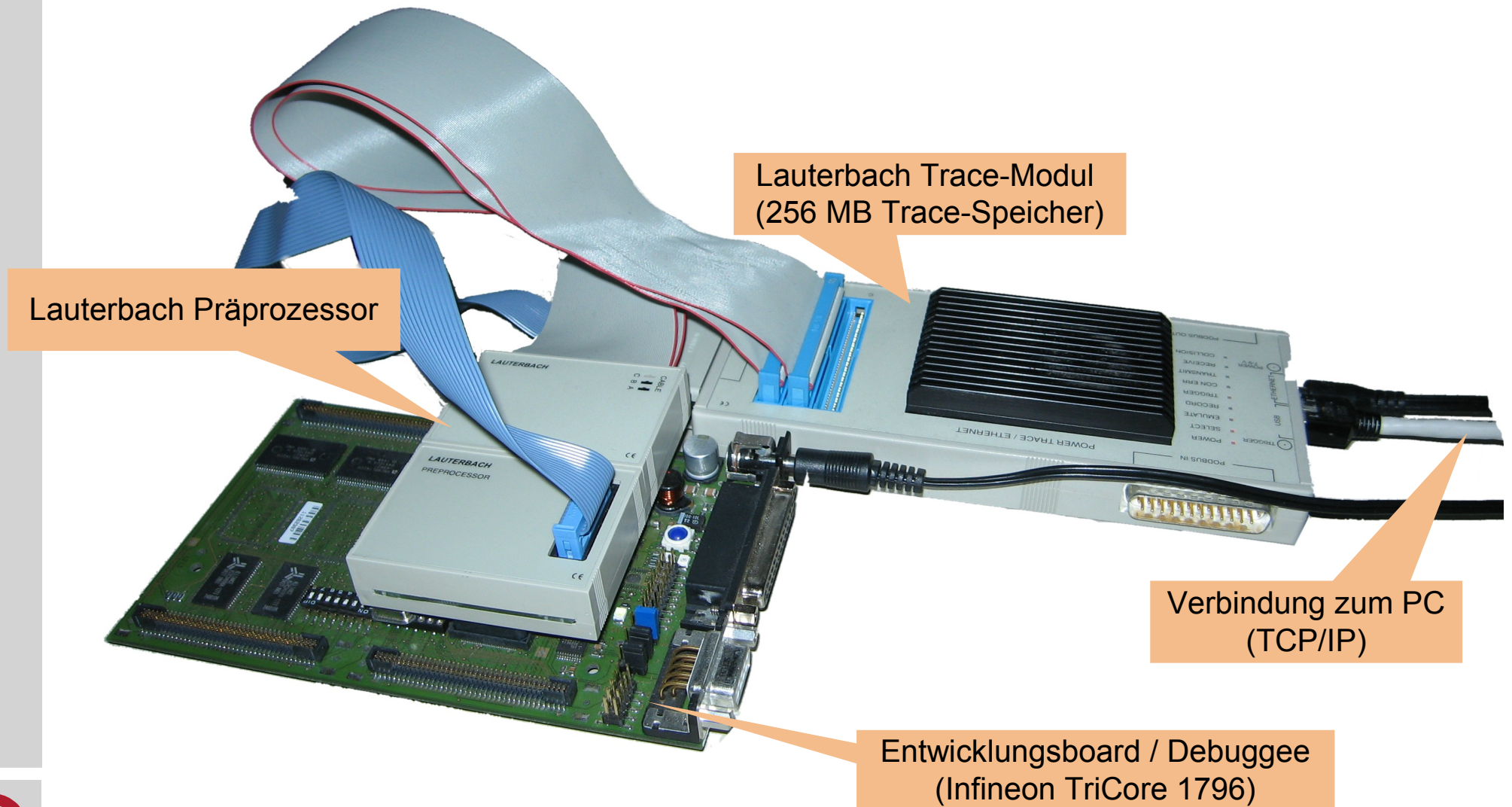
Debugging Deluxe – Beispiel BDM

- *"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

The screenshot displays the TRACE32 debugger interface, which is used for debugging embedded systems. The interface is divided into several panes:

- Top Pane:** Contains the menu bar (File, Edit, View, Var, Break, Run, CPU, Misc, Trace, Perf, Cov, TriCore, Window, Help) and a toolbar with various debugging controls like step, over, next, return, up, go, break, and mode.
- Left Pane (B::REGISTER /SPOTLIGHT):** Shows the current state of the processor registers. The PC register is highlighted, showing the address 040002F0.
- Bottom-Left Pane (B::var.watch os::krn::theTasks):** Displays the contents of the 'theTasks' variable, which is a structure containing fields like 'pri_', 'state_', 'func_', 'stack_', and 'interrupted_'.
- Right Pane (B::Data.ListAsm):** Shows the assembly code at the current PC address. The code is disassembled from memory, showing instructions like 'ld16.bu', 'j', 'ret16', 'movh.a', 'mov16', 'movh.a', 'lea', 'lea', 'ld16.w', 'insert', 'st16.w', 'call', 'mov16', 'ret16', 'call', 'call', 'call', 'call', 'j', 'mov16', 'nop16', 'j16', 'mov16', 'nop16', 'j16', 'mov16', 'nop16', 'j16', 'mov16', 'nop16', 'j16', 'mov16', 'nop16', 'j16', 'movh.a', and 'lea'.

The status bar at the bottom indicates the current address is D:0002028, the file is \\Measure\\main\\os::krn::theTasks, and the debugger is in a stopped state.



Debugger Deluxe: Lauterbach-Frontend

The screenshot displays the TRACE32 debugger interface. The main window shows a trace list with columns for record, address, ti.zero, and ti.back. The trace list is filtered to show records from 00000125 to 00000080. The assembly window on the right shows the corresponding code for these records.

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
	bisr16 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
	rslcx		
00000081	P: A1000052	6.260us	0.240us
	nop16		
00000080	P: A1000054	6.500us	0.240us
	rfe16		

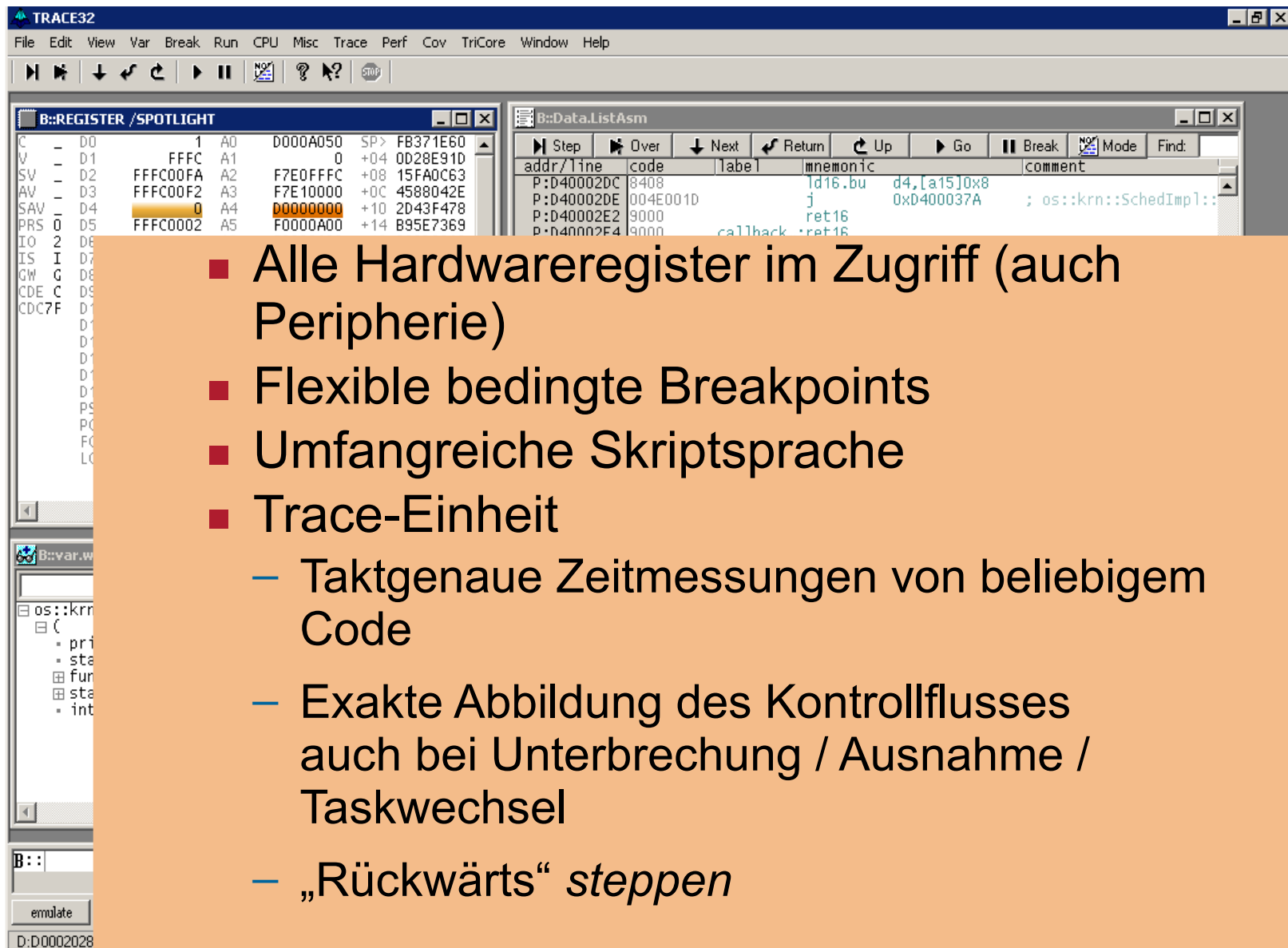
The assembly window on the right shows the following code:

```
16.bu d4,[a15]0x8  
0xD400037A ; os::krm::SchedImpl::  
16  
16  
16.h.a a15,0xF000  
16 d4,0x0  
16.h.a a4,0xD000  
16 a15,[a15]0x2FC  
16 a4,[a4]0x2170  
16.w d15,[a15]  
16.w d15,d15,0x0F,0x0C,0x1  
16 [a15],d15  
16 0xD4000428 ; os::krm::OSControl::  
16 d2,0x0 ; return,0  
16  
16 0xD4000762 ; hw::tc::unlock_wdtc  
16 0xD400070E ; hw::init()  
16 0xD4000798 ; hw::tc::lock_wdtcon  
16 0xD40006EC ; hw::hal::init()  
16 0xD4000354 ; os::init()  
16 d2,0x1  
16  
16 a11  
16 d2,0x1  
16  
16 a11  
16 d2,0x1  
16  
16 a11  
16 d2,0x1  
16  
16 a11 ; _tc_reg_A11  
16 d2,0x1  
16  
16 a11 ; _tc_reg_A11  
16.h.a a4,0xD000  
16 a4,[a4]0x2150
```

„TRACE32“ erlaubt das Aufzeichnen von Programmabläufen. Zeiten werden mit hoher Auflösung protokolliert.



Debugger Deluxe: Lauterbach-Frontend



The screenshot shows the TRACE32 debugger interface. The main window is titled 'TRACE32' and contains a menu bar (File, Edit, View, Var, Break, Run, CPU, Misc, Trace, Perf, Cov, TriCore, Window, Help) and a toolbar with various icons. Below the toolbar, there are several panes. The 'B::REGISTER /SPOTLIGHT' pane on the left shows a list of registers (C, V, SV, AV, SAV, PRS, IO, IS, GW, CDE, CDC7F) and their values. The 'B::Data.ListAsm' pane on the right shows assembly code with columns for address/line, code, label, mnemonic, and comment. The assembly code includes instructions like 'ld16.bu', 'j', 'ret16', and 'callback *ret16'. The 'B::var.w' pane at the bottom left shows a list of variables (os::krm, pri, sta, fun, sta, int) and their values. The 'B::' pane at the bottom right shows the current address (D:00002028) and the 'emulate' button.

- 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



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

