

Betriebssysteme (BS)

VL 2 – Einstieg in die Betriebssystementwicklung

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

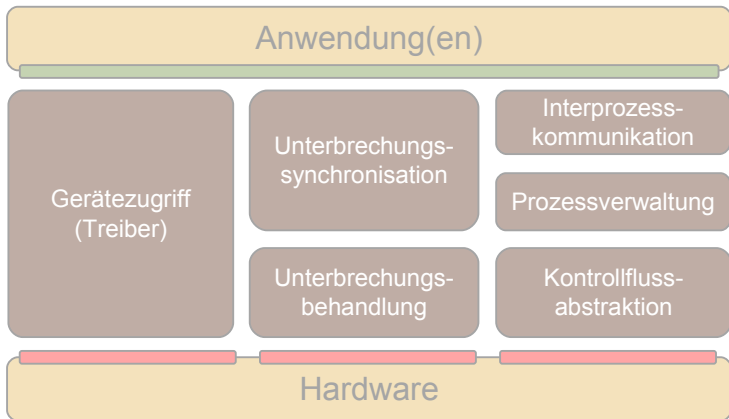
Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 21 – 25. Oktober 2021



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

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

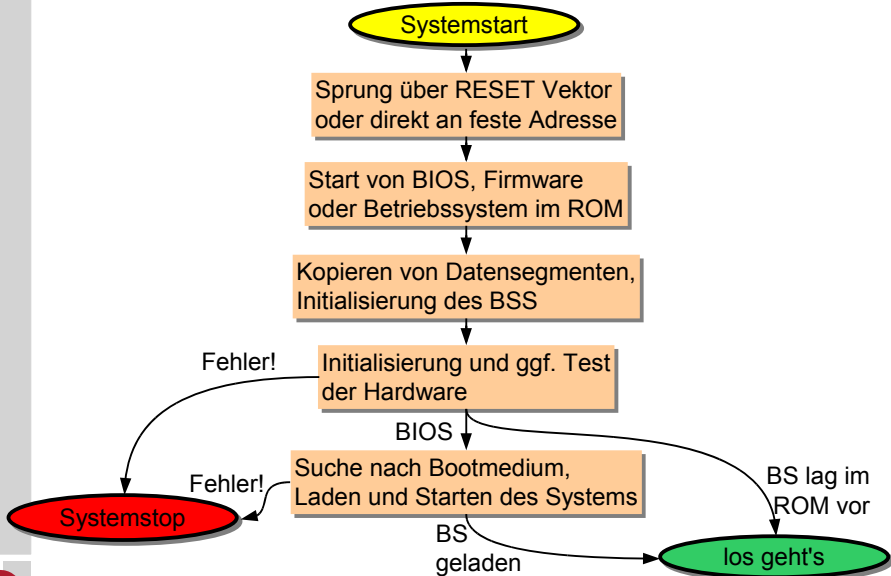
Einordnung
Übersetzen und Linken
Booten
Debugging
Zusammenfassung



„**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; (ebx90)
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> ...
0x01fe	0xaa55



- 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

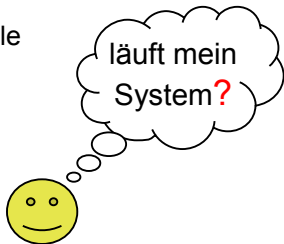
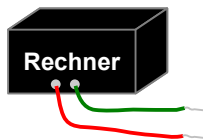


Debugging



„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@fai48:~> 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

Abbruch-Ereignis

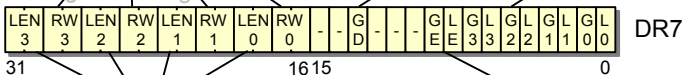
- 00: Befehlsausführung
- 01: Schreiben
- 10: I/O (ab Pentium)
- 11: Schreiben/Lesen



Zugriffssperre
für DR0-7

Breakpoint-Freigabe
(lokal, global)

Debug Steuerregister



Länge des überwachten
Speicherbereichs

exakter Daten-Breakpoint
(lokal, global)

Debugging – Funktionsweise (3)

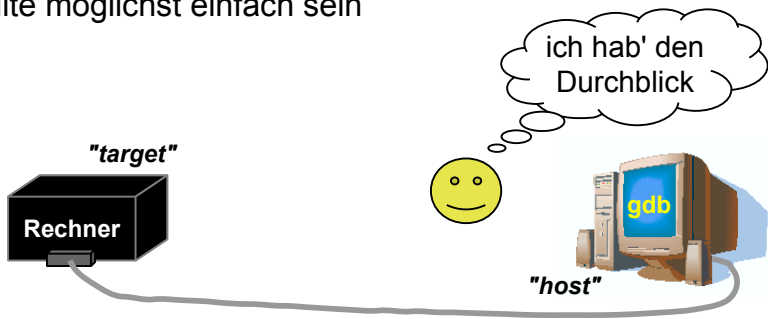
- besonders effektiv wird Debugging, wenn das Programm im Quelltext visualisiert wird (*source-level debugging*)
 - erfordert Zugriff auf den Quellcode und Debug-Informationen
 - muss durch den Übersetzer unterstützt werden

```
lohmann@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
```

```
lohmann@fau148:~>
```

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)

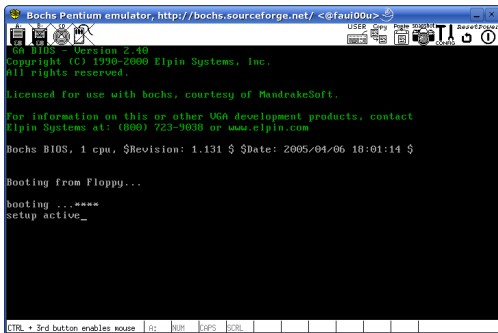
- das Kommunikationsprotokoll – kompletter Umfang
 - Register- und Speicherbefehle
 - lese/schreibe alle Register
 - lese/schreibe einzelnes Register
 - lese/schreibe Speicherbereich
 - Steuerung der Programmausführung
 - letzte Unterbrechungsursache abfragen
 - Einzelschritt
 - mit Ausführung fortfahren
 - 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 content displays the following text:

```
GR BIOS - Version 2.48  
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) 723-9030 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 icons: A:, NUM, CAPS, SCRL, and several empty boxes.

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



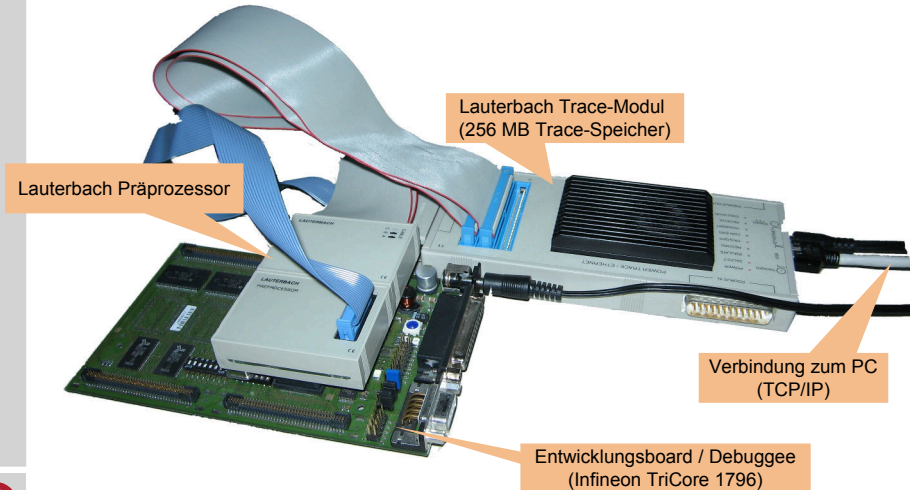
Debugging Deluxe – Beispiel BDM

- *"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 with the following components:

- TRACE32 Title Bar:** File Edit View Var Break Run CPU Misc Trace Perf Cov TrCore Window Help
- Navigation Bar:** Includes icons for back, forward, search, and other navigation functions.
- B:REGISTER / SPOTLIGHT:** A table showing register values and addresses.

Register	Value	Address	Comment
C	- D0	1 A0	D000A050 SP> F8371E50
V	- D1	FFFC A1	0 +04 0D28E91D
SV	- D2	FFFC00FA A2	F7E0FFFC +08 15FA0C63
AV	- D3	FFFC00F2 A3	F7E10000 +0C 4588042E
SAV	- D4	0 A4	00000000 +10 2043F478
PRS	D 05	0 A5	F000A000 +14 895E7369
IO	Z D6	8 A6	F000A000 +18 000000A6
IS	I D7	FFFC0002 A7	F000A000 +1C D30C362C
GW	C D8	0 A8	0 +20 E3A2ED1D
CDE	C D9	0 A9	D0000000 +24 788308BC
CDC	F D10	0 A10	D00029A8 +28 8878666C
D11	0 A11	D40000A4	+2C E3972A15
D12	0 A12	0	+30 FCC28370
D13	0 A13	0	+34 17E5863A
D14	0 A14	0	+38 C08C1270
D15	A0010622 A15	F0000000	+3C 40E1C77C
PSW	080008FF PC	D40002F0	+40 5894C304
PKCI	0040007F TSP	D00025A8	+44 0A86798E
FCX	0000007E TCR	0	+48 00237C06
LCK	00000002 BTV	D4002000	+4C 78136F75
		D4002200	+54 F4A11296
- B:Data.List.Asm:** A disassembly window showing instructions and their addresses.

Step	Over	Next	Return	Up	Go	Break	Mode	Find
addr / line	code	label	mnemon1c	comment				
P: D400020C	0400		ld16.bu	d4,[a15]0xb				
P: D400020E	004E0010		j	0xb400037A				: os::krrn::SchedImp1:
P: D40002E2	9000		ret16					
P: D40002E4	9000	callback::	ret16					
P: D40002E6	FF000031	main:	movh.a	a15,0xF000				
P: D40002EA	0482		mov16	d4,0x0				
P: D40002EC	40000091		movh.a	a4,0x0000				
P: D40002F0	8080FF03		lea	a15,[a15]0x2FC				
P: D40002F4	52304409		lea	a4,[a4]0x2170				
P: D40002F8	F04C		ld16.v	d15,[a15]				
P: D40002FA	F601FFB7		insert	d15,d15,0x0F,0x0C,0x1				
P: D40002FE	0F68		st16.v	[a15],d15				
P: D4000300	00940060		call	0x04000428				: os::krrn::OSControl:
P: D4000304	1292		mov16	d2,0x0				: return,0
P: D4000306	9000		ret16					
P: D4000308	02200060	__main:	call	0x04000762				: hw::tc::unlock_wdct
P: D400030C	02010060		call	0x0400070E				: hw::init()
P: D4000310	02440060		call	0x04000798				: hw::tc::lock_wdctcon
P: D4000314	01E00060		call	0x040006EC				: hw::hal::init()
P: D4000318	001E0010		j	0x04000354				: os::init()
P: D400031C	1282	hw::irq::	mov16	d2,0x1				
P: D400031E	0000		nop16					
P: D4000320	080C	a11	j16					
P: D4000322	1282	hw::irq::	mov16	d2,0x1				
P: D4000324	0000		nop16					
P: D4000326	080C		j16	a11				
P: D4000328	1282	hw::irq::	mov16	d2,0x1				
P: D400032A	0000		nop16					
P: D400032C	080C		j16	a11				
P: D400032E	1282	hw::irq::	mov16	d2,0x1				
P: D4000330	0000		nop16					
P: D4000332	080C		j16	a11				: _tc_reg_A11
P: D4000334	1282	hw::irq::	mov16	d2,0x1				
P: D4000336	0000		nop16					
P: D4000338	080C	a11	j16					: _tc_reg_A11
P: D400033A	40000091	hw::irq::	movh.a	a4,0x0000				
P: D400033E	52104409		lea	a4,[a4]0x2150				
- B:var_watch os::krrn::theTasks:** A window showing the state of a task.

```
os::krrn::theTasks = (  
  + pri_ = 3,  
  + state_ = SUSPENDED,  
  + func_ = 0x04000290,  
  + stack_ = 0x00002150,  
  + interrupted_ = 0)
```
- B:::** A status bar at the bottom showing the current instruction: D:D0002028 \Measure\main:os::krrn::theTasks stopped.

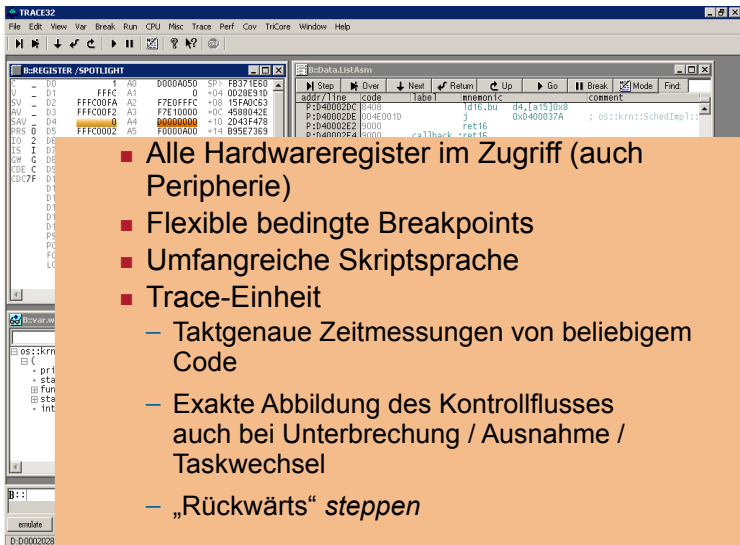
Debugger Deluxe: Lauterbach-Frontend

The screenshot displays the TRACE32 debugger interface. A window titled "B::trace.list address list.asm ti.zero ti.back" is open, showing a list of execution records. The records include addresses, instructions, and timing information for 'ti.zero' and 'ti.back'. An orange callout box points to the list with the text: „TRACE32“ erlaubt das Aufzeichnen von Programmabläufen. Zeiten werden mit hoher Auflösung protokolliert.

record	address	ti.zero	ti.back

-00000125	P: A1000040 dsync	0.000	
-00000124	P: A1000044 nop16	0.240us	0.240us
-00000123	P: A1000046 nop16	0.240us	<0.020us
-00000117	P: A1000048 bisr16 0x2	0.480us	0.240us
-00000112	P: A100004A call 0xA1000E4E	0.960us	0.480us
-00000099	P: A1000E4E ret16	3.140us	2.180us
-00000085	P: A100004E rsicx	6.020us	2.880us
-00000081	P: A1000052 nop16	6.260us	0.240us
-00000080	P: A1000054 rfe16	6.500us	0.240us

Debugger Deluxe: Lauterbach-Frontend



The screenshot displays the TRACE32 debugger interface. The main window is titled 'TRACE32' and contains several panes:

- B:REGISTER / SPOTLIGHT:** Shows a list of registers (C, V, SV, AV, SAV, PRS, IO, IS, CDE, CDC7F) with their current values and addresses. For example, register C contains 0000A050.
- B:Data.ListAsm:** Shows assembly code with columns for address/line, code, table, mnemonic, and comment. The current instruction is 'P:0400020C 0400 1d16_bu d4,[a15]0xb'.
- Variable Watch:** Shows a list of variables being monitored, including 'os::krm'.

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



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

