Betriebssysteme (BS)

VL 2 – Einstieg in die Betriebssystementwicklung

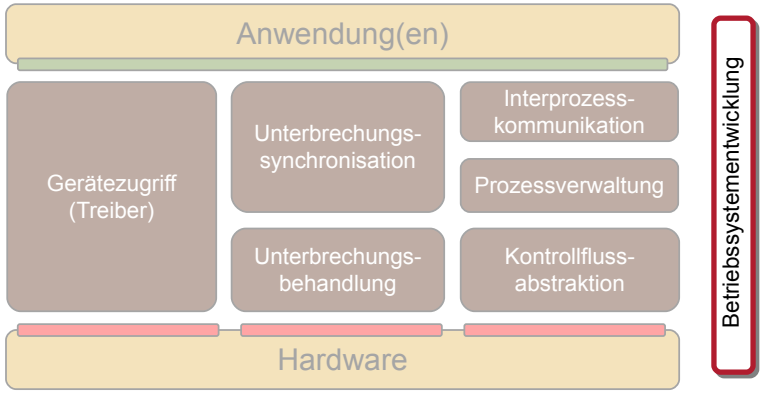
Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 11 – 25. Oktober 2011

Überblick: Einordnung dieser VL



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 debugging*
  - Hardwareunterstützung

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

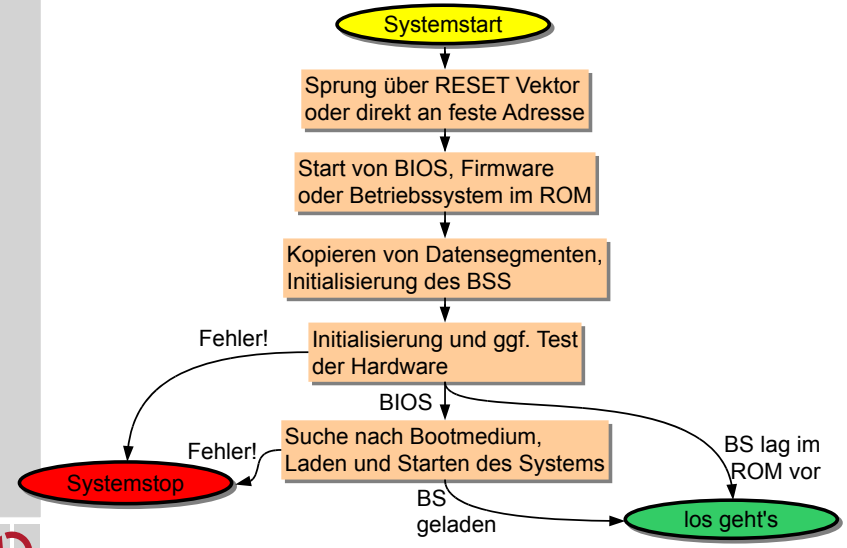
Bootvorgang

> *„**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.“*

wikipedia.de

Bootvorgang



Bootvorgang beim PC – Boot Sektor

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

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 – Boot Sektor

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

Alternative
(OO-StuBS):

Offset	Inhalt
0x0000	`jmp boot;`
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	`boot:` `...`
0x01fe	**0xaa55**

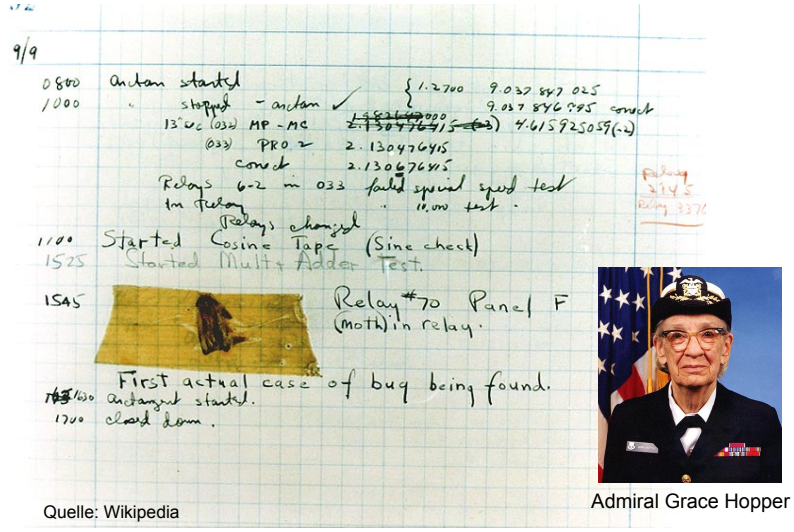
Wichtig ist eigentlich nur der Start und die „Signatur“ (0xaa55) am Ende. Alles weitere benutzt der ***Boot Loader***, um das eigentliche System zu laden.

Bootvorgang beim PC – *Boot Loader*

- 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 Boot Blocks nach 0x7c00 und Start

Debugging

Der erste dokumentierte „Bug“



Quelle: Wikipedia

Admiral Grace Hopper

„printf - Debugging“

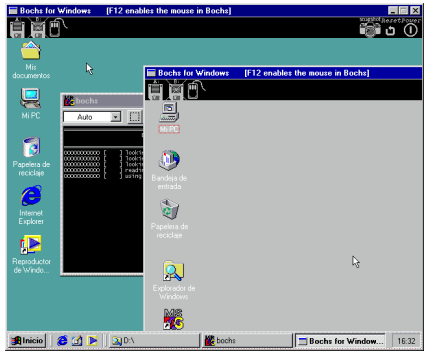
- gar nicht so einfach, da es `printf()` per se nicht gibt!
  - oftmals gibt es nicht mal einen Bildschirm
- `printf()` ändert oft auch das Verhalten des *debuggee*
  - mit printf() tritt der Fehler nicht plötzlich nicht mehr / anders auf
  - das gilt gerade auch bei der Betriebssystementwicklung
- Strohhalme
  - 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
  - Multiprozessoremulation
- 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 Software-Fehlern 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@faui48:~> 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

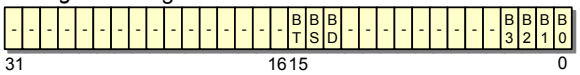
Debugging – Funktionsweise (2)

die Debug Register des 80386

Breakpoint Register

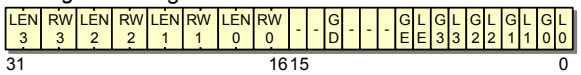
breakpoint 0: lineare Adresse	DR0
breakpoint 1: lineare Adresse	DR1
breakpoint 2: lineare Adresse	DR2
breakpoint 3: lineare Adresse	DR3
reserviert	DR4
reserviert	DR5

Debug Statusregister



DR6

Debug Steuerregister



DR7

Debugging – Funktionsweise (2)

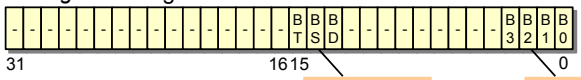
die Debug Register des 80386

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

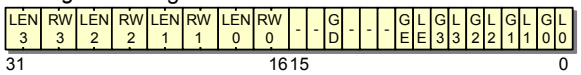


DR6

Einzelschritt

Breakpoint 0-3

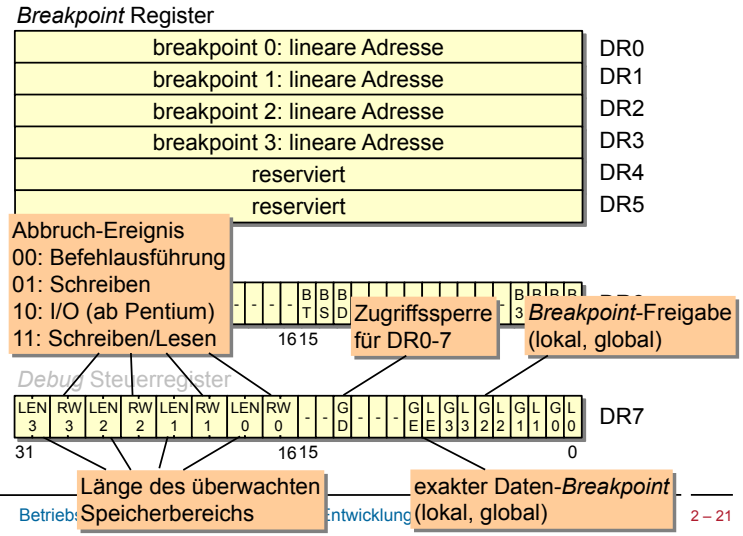
Debug Steuerregister



DR7

Debugging – Funktionsweise (2)

die Debug Register des 80386



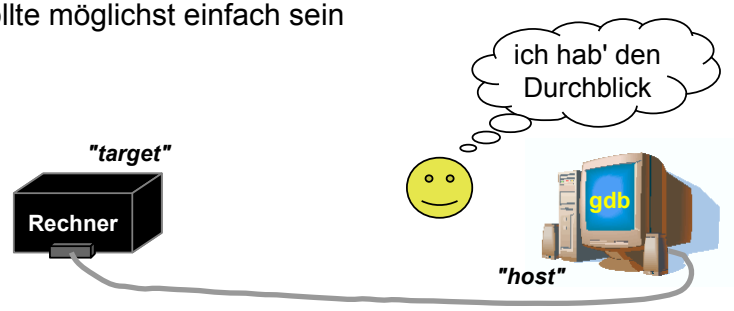
Debugging – Funktionsweise (3)

- 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@faui48:~> g++ -o hello -g hello.cc
lohmann@faui48:~> 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@faui48:~>
```


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

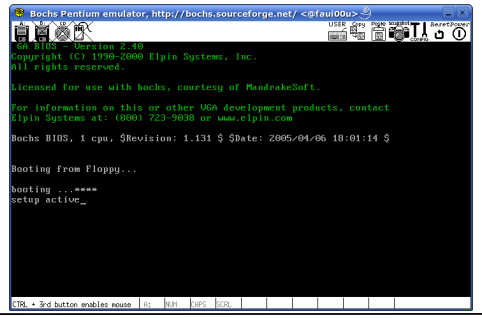
Remote Debugging – Beispiel gdb (2)

- 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.
```

Debugging Deluxe

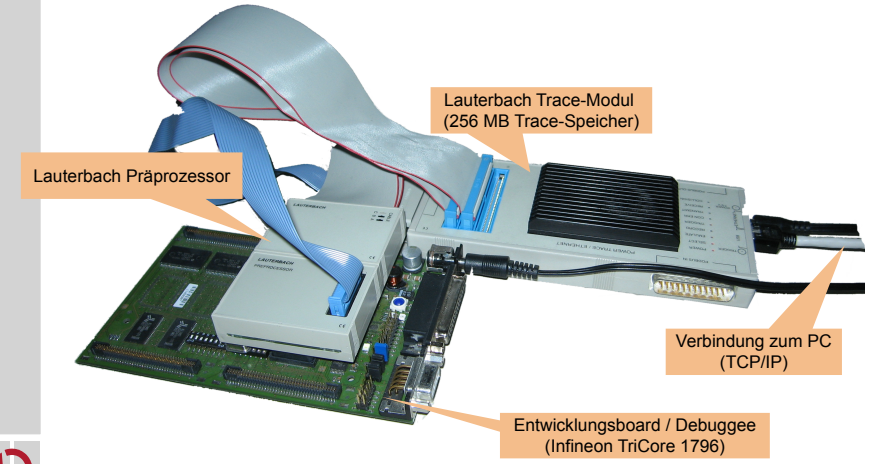
- viele Prozessorhersteller integrieren heute Hardwareunterstützung für *Debugging* auf ihren Chips (*OCDS – On Chip Debug System*)
  - BDM, OnCE, MPD, JTAG
- i.d.R. einfaches serielles Protokoll zwischen *Debugging*-Einheit und externem *Debugger* (Pins sparen!)
- Vorteile:
  - der *Debug Monitor* (z.B. 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

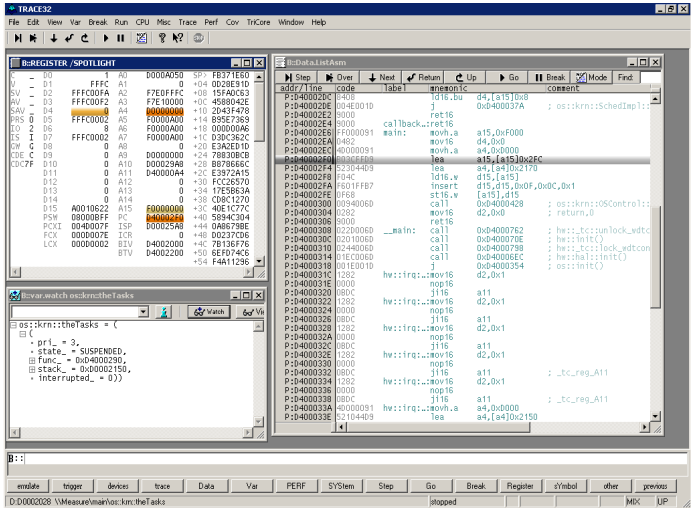
- *"Background Debug Mode"* - eine *on-chip debug* Lösung von Motorola
- serielle Kommunikation über drei Leitungen (DSI, DSO, DSCLK)
- BDM Kommandos der 68k und ColdFire Prozessoren
  - RAREG/RDREG – Read Register
    - lese bestimmtes Daten- oder Adressregister
  - WAREG/WDREG – Write Register
    - 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

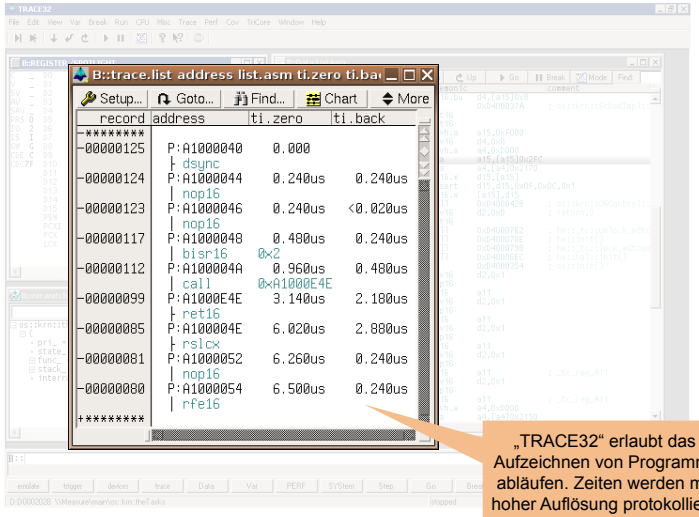
- Lauterbach Hardware-Debugger



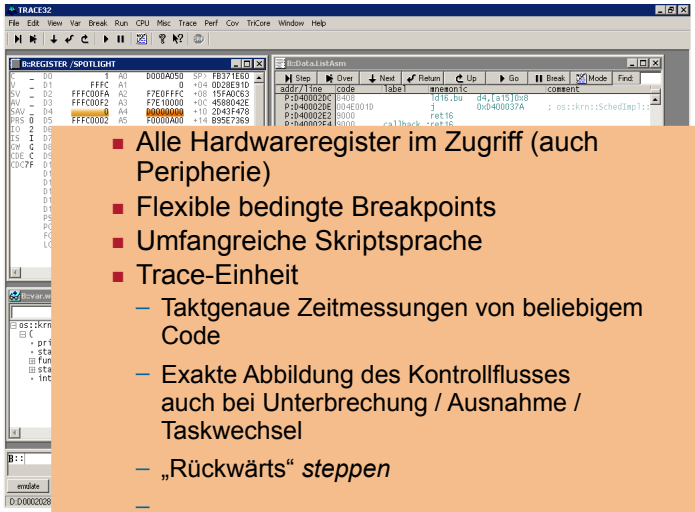
Debugger Deluxe: Lauterbach-Frontend



Debugger Deluxe: Lauterbach-Frontend



Debugger Deluxe: Lauterbach-Frontend



- 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*
  - ...

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