TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

OPERATING-SYSTEM CONSTRUCTION

Material based on slides by Olaf Spinczyk, Universität Osnabrück

Operating-System Development 101

https://tud.de/inf/os/studium/vorlesungen/betriebssystembau

HORST SCHIRMEIER

OS Development (Not Always Comfy)

- **First Steps**

  How to get your OS onto the target hardware?
  - Compilation/Linking
  - Boot process
- **Testing and Debugging**

  What to do if your system doesn't respond?
  - "printf debugging"
  - Emulators, virtual machines
  - Debuggers
  - Remote Debugging
  - Hardware support

OS Development (Not Always Comfy)

- **First Steps**

  How to get your OS onto the target hardware?

  - Compilation/Linking
  - Boot process
- **Testing and Debugging**

  What to do if your system doesn't respond?

  - "printf debugging"
  - Emulators, virtual machines
  - Debuggers
  - Remote Debugging
  - Hardware support

Compilation/Linking – *Hello, World*

```
#include <iostream>

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

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

- Assumption:
  - Development system runs an x86 Linux
  - Target system also is a PC
- Does this program also run on **bare metal**?
- Is OS development in a **high-level programming language** possible at all?

Compilation/Linking – Problems and Solutions

- No dynamic linker available
  - ➔ link all necessary libraries statically
- libstdc++ and libc use Linux system calls
  (e.g., write)
  - ➔ We **cannot use** regular C/C++ runtime libraries.
    (We usually don’t have alternatives either.)
- Generated addresses refer to *virtual* memory
  ("nm hello | grep main" yields "0000000000404745 T main")
  - ➔ We cannot use standard linker settings but **need a custom linker config**.
- High-level language code: environment expectations
  (CPU-register usage, address mapping, runtime environment, stack, ...)
  - ➔ Own **startup code** (written in assembler) must prepare high-level language code execution.

Booting

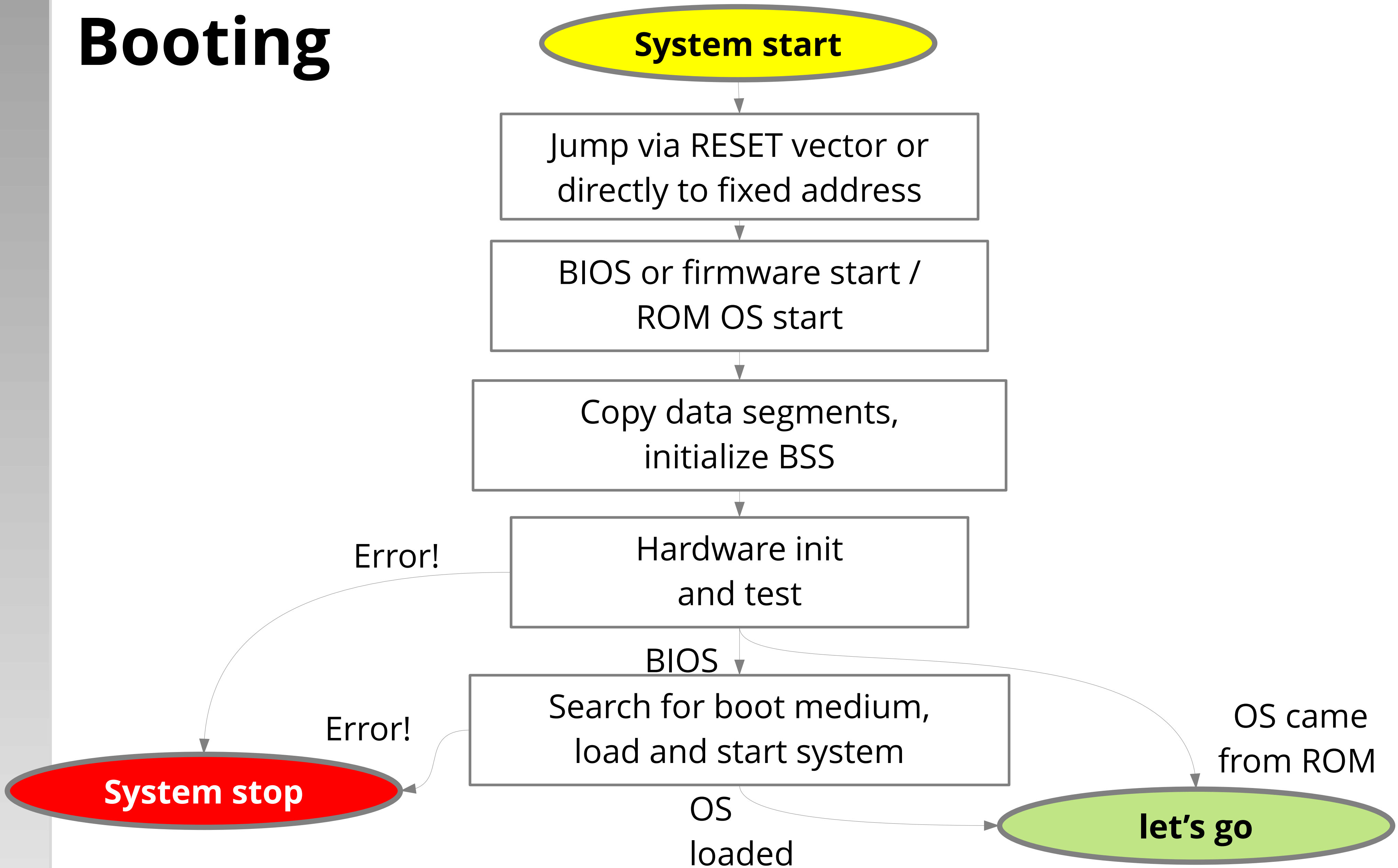
*"**Boot** is short for **bootstrap** or **bootstrap load** and derives from the phrase **to pull oneself up by one's bootstraps**."*

*"**Booting** is the process of starting a computer, specifically with regard to starting its software. The process involves a chain of stages, in which at each stage, **a smaller, simpler program** loads and then executes the **larger, more complicated program** of the next stage."*

The term is sometimes attributed to a story in Rudolf Erich Raspe's The Surprising Adventures of Baron Munchausen, *but in that story Baron Munchausen pulls himself (and his horse) out of a swamp by his hair (specifically, his pigtail), not by his bootstraps – and no explicit reference to bootstraps has been found elsewhere in the various versions of the Munchausen tales.*

en.wikipedia.org

Booting


System start
Jump via RESET vector or directly to fixed address
BIOS or firmware start / ROM OS start
Copy data segments, initialize BSS
Hardware init and test
Error!
BIOS
Search for boot medium, load and start system
Error!
System stop
OS loaded
OS came from ROM
let's go

PC Booting – Boot Sector

- PC BIOS loads 1st block (512 bytes) of boot drive at address 0x7c00 and jumps there (“blindly”)
- Boot-sector layout

FAT disk (DOS/Windows)

Offset	Inhalt
`0x0000`	`jmp boot; nop;` (ebxx90)
`0x0003`	System name and version
`0x000b`	Bytes per sector
`0x000d`	Sectors per cluster
`0x000e`	reserved sectors (for boot record)
`0x0010`	number of FATs
`0x0011`	number of root-directory entries
`0x0013`	number of logical sectors
`0x0015`	media descriptor byte
`0x0016`	sectors per FAT
`0x001a`	number of heads
`0x001c`	number of hidden sectors
`0x001e`	`boot:` `...`
`0x01fe`	0xaa55

PC Booting – Boot Sector

- PC BIOS loads 1st block (512 bytes) of boot drive at address 0x7c00 and jumps there ("blindly")
- Boot-sector layout

Alternative (OOStuBS)

Offset	Inhalt
0x0000	`jmp boot; nop;` (ebxx90)
0x0003	System name and version
0x000b	Bytes per sector
0x000d	Sectors per cluster
0x000e	reserved sectors (for boot record)
0x0010	number of FATs
0x0011	number of root-directory entries
0x0013	number of logical sectors
0x0015	media descriptor byte
0x0016	sectors per FAT
0x001a	number of heads
0x001c	number of hidden sectors
0x001e	`boot:` `...`
0x01fe	0xaa55

In fact, only the beginning and the **"signature" (0xaa55)** at the end matters. Everything else is used by the **boot loader** to load the actual system.

PC Booting – Boot Loader

- Simple, **system-specific** boot loaders
  - Define hardware/software state
  - If necessary: Load further blocks with boot-loader code
  - Pinpoint the actual system on the boot media
  - Load the system (via BIOS functions)
  - Jump into loaded system
- Boot loader on disks not flagged as “bootable”
  - Error message, halt / reboot
- Boot loader with **boot menu** (e.g., GRUB)
  (for example in the **Master Boot Record** of a HDD)
  - Display a menu
  - Emulate BIOS when booting the selected system
    (load boot block to 0x7c00, jump)

OS Development (Not Always Comfy)

- **First Steps**

  How to get your OS onto the target hardware?

  - Compilation/Linking
  - Boot process

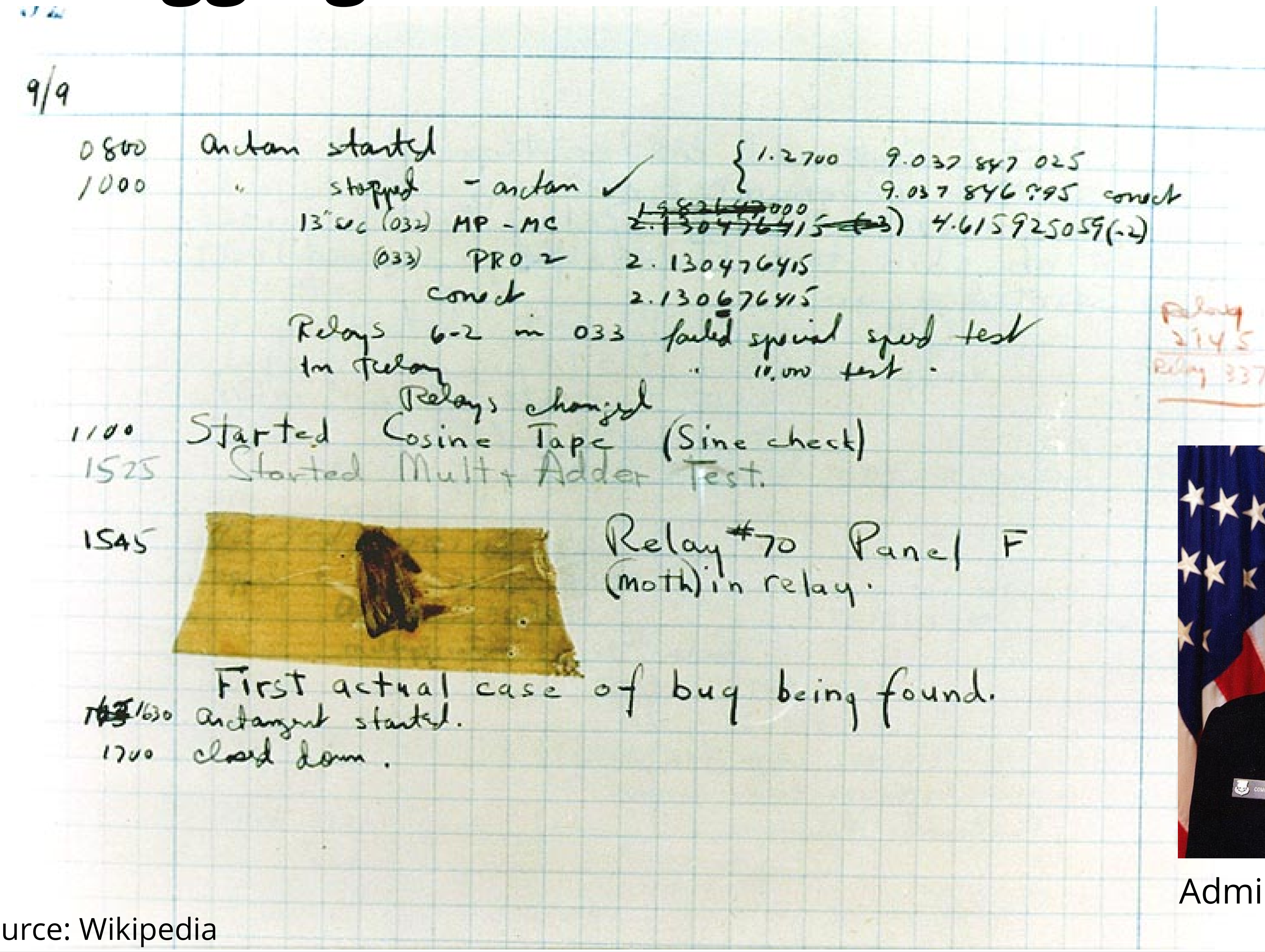
- **Testing and Debugging**

  What to do if your system doesn't respond?

  - "printf debugging"
  - Emulators, virtual machines
  - Debuggers
  - Remote Debugging
  - Hardware support

Debugging

1947



Source: Wikipedia

Admiral Grace Hopper

"printf Debugging"

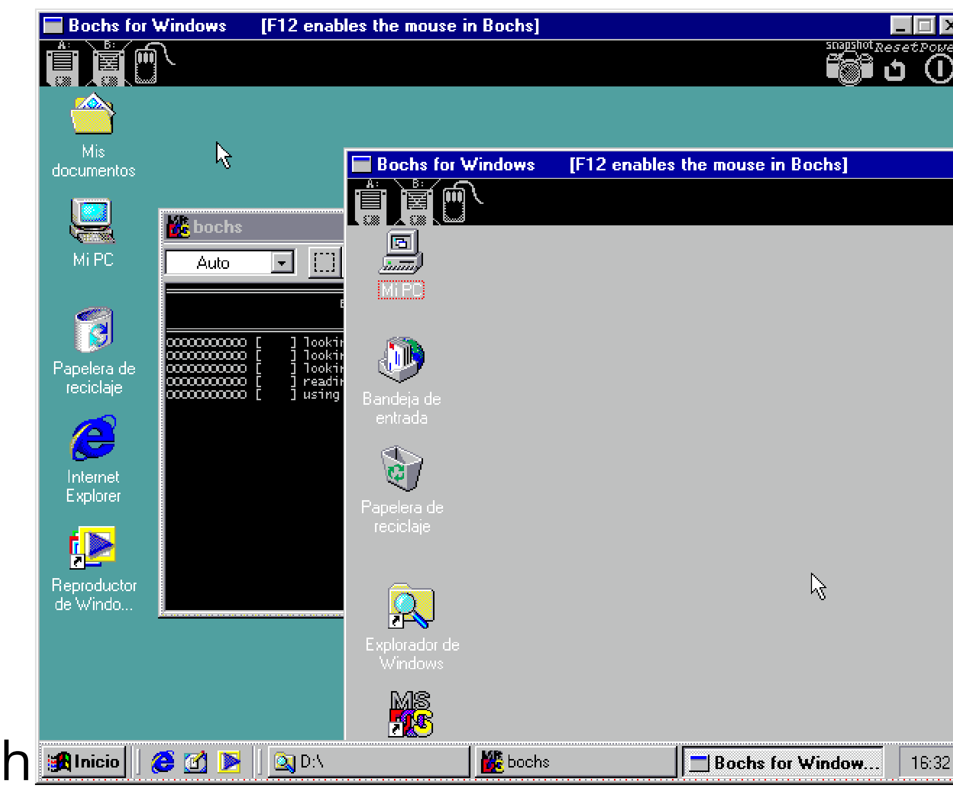
- Not that simple – if you don't have a (working) printf
  - Often you don't even have a display.
- printf() often changes the debuggee's behavior
  - Problem vanishes / changes symptoms
  - Unfortunately particularly true for OS development
- Last resort:
  - blinking LED
  - serial interface

(Software) Emulators

- Emulate real hardware in software
  - Simplifies debugging
    (Emulation software usually more communicative than real HW)
  - Shorter development cycles
- **Careful:** In the end, the system must run on real hardware!
  - Emulator and real hardware may differ in details!
  - Harder to find bugs in a complete system than during incremental development
- Emulation: a special case of **virtualization**
  - Provides a **virtual resource Y** (e.g., an Arm CPU)
    based on a resource X (e.g., the systems x86-64 host CPU)

Emulators – Example "Bochs"

- Emulates i386, …, Pentium, x86-64 (interpreter loop)
  - plus MMX, SSE–SSE4, 3DNow! instructions
  - Multiprocessor emulation
- Emulates a complete PC
  - Memory, devices (including sound, networking, …)
  - Capable to run Windows, Linux
- Implemented in C++
- Development support
  - Logs helpful info, e.g. from crash
  - Built-in debugger (*GDB stub*)



Bochs in Bochs

Debugging

- Debugger helps locating software bugs by tracing/controlling the debuggee:
  - **Single-step mode**
  - **Breakpoints:** trigger when reaching a particular machine instruction
  - **Watchpoints:** trigger when a particular data element is accessed
- **Careful:** Bug-hunting might take *longer* when using a debugger
  - Taking a break and thinking about the problem can be more time-efficent
    - Single-stepping costs a lot of time
    - Often no way back in case you miss the problematic instruction
  - “printf debugging” allows better control over output format
  - Synchronization / race-condition bugs are impractical to debug with a debugger
- helpful: “Core dump” analysis
  - but of little relevance during OS development :-(

Debugging – Example Session

Setting a breakpoint

Running the program

Single-stepping

Continuing

```
$ g++ -static -g -o hello hello.cc
$ gdb hello
GNU gdb (Ubuntu 11.1-0ubuntu2) 11.1
...
(gdb) break main
Breakpoint 1 at 0x40474d: file hello.cc, line 4.
(gdb) run
Starting program: hello

Breakpoint 1, main () at hello.cc:4
4               std::cout << "Hello, World" << std::endl
(gdb) next
Hello, World
5       }
(gdb) next
0x00000000004a7f4a in __libc_start_call_main ()
(gdb) continue
Continuing.
[Inferior 1 (process 663394) exited normally]
(gdb) quit
```

Debugging – Technical Background (1)

- Practically all CPUs support debugging
- Example: Intel x86
  - **INT3** instruction triggers a *"breakpoint interrupt"* (in fact a *trap*)
    - User "sets breakpoint", debugger (at runtime) replaces program instruction with INT3 (and saves the original instruction)
    - Trap handler redirects control flow to debugger
  - **enabled Trap Flag (TF)** in status register (EFLAGS / RFLAGS): trigger *"debug interrupt"* after every instruction
    - Can be used for implementing single-stepping in the debugger
    - Trap handler itself is, of course, *not* executed in single-stepping mode
  - **Debug Registers DR0–DR7** can monitor up to 4 breakpoints or watchpoints
    - No code manipulation necessary: breakpoints in ROM/FLASH or read-only memory segments (e.g. *shared libraries*!)
    - Efficient watchpoints only possible through this mechanism

TECHNISCHE UNIVERSITÄT DRESDEN

Debugging – Technical Background (2)

80386 Debug Registers

Breakpoint Register

breakpoint 0: linear address	DR0
breakpoint 1: linear address	DR1
breakpoint 2: linear address	DR2
breakpoint 3: linear address	DR3
reserved	DR4
reserved	DR5

Debug Status register

31															16	15															0	
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	BT	BS	BD	-	-	-	-	-	-	-	-	-	B3	B2	B1	B0	DR6

Debug Control register

31							16	15															0	
LEN 3	RW 3	LEN 2	RW 2	LEN 1	RW 1	LEN 0	RW 0	-	-	GD	-	-	-	GE	LE	G3	L3	G2	L2	G1	L1	G0	L0	DR7

Debugging – Technical Background (2)

80386 Debug Registers

Breakpoint Register

breakpoint 0: linear address	DR0
breakpoint 1: linear address	DR1
breakpoint 2: linear address	DR2
breakpoint 3: linear address	DR3
reserved	DR4
res	DR5

De

- - - -

31

Aborting event
00: Instr. execution
01: Memory write
10: I/O (>=Pentium)
11: Mem. write or read

Access control for DR0-7

Breakpoint enable (local, global)

Debug Control register

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

31 … 16 15 … 0

Length of monitored memory area

exact data breakpoint (local, global)

Debugging – Technical Background (2)

80386 Debug Registers

Breakpoint Register

breakpoint 0: linear address	DR0
breakpoint 1: linear address	DR1
2: linear address	DR2
3: linear address	DR3
	DR4
reserved	DR5

Provides information about trigger event to the trap handler

Debug Status register

31															16	15															0	
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	BT	BS	BD	-	-	-	-	-	-	-	-	-	B3	B2	B1	B0	DR6

Single step

Breakpoint 0-3

Debug Control register

31							16	15															0	
LEN 3	RW 3	LEN 2	RW 2	LEN 1	RW 1	LEN 0	RW 0	-	-	GD	-	-	-	GE	LE	G3	L3	G2	L2	G1	L1	G0	L0	DR7

Debugging – Technical Background (3)

- For debugging **regular user-space applications**, the OS must provide an interface
  - e.g. Linux: **ptrace (2)**

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

Request (PTRACE_...)	Semantics
TRACEME	Indicate that this process is to be traced by its parent
ATTACH, DETACH	Seize control over another process (alt. to TRACEME)
PEEKTEXT, PEEKDATA, PEEKUSER	Read data from debuggee's address space
POKETEXT, POKEDATA, POKEUSER	Change data in debuggee's address space
SYSCALL, CONT	Monitor system calls and continue
SINGLESTEP	Single-stepping mode (machine instruction granularity)
KILL	Abort debuggee

Debugging – Technical Background (4)

```
int main(void) {
  long long counter = 0;  /* machine instruction counter */
  int wait_val;           /* child's return value        */
  int pid;                /* child's process id          */

  puts("Please wait");
  pid = fork();           /* create child process */
  if (pid == -1)          /* failed to create child process */
    perror("fork");
  else if (pid == 0) {    /* child process starts */
    ptrace(PTRACE_TRACEME, 0, 0, 0); /* allow parent to control child */
    execl("/bin/ls", "ls", NULL);    /* run child program (ls) and terminate*/
  }
  else {                  /* parent process starts */
    /* wait for SIGTRAP */
    while (wait(&wait_val) != 1 && WIFSTOPPED(wait_val) && WSTOPSIG(wait_val)) {
      counter++;
      if (ptrace(PTRACE_SINGLESTEP, pid, 0, 0) != 0) { /* enable single step mode */
        perror("ptrace");
        break;
      }
    }
    printf("Number of machine instructions : %lld\n", counter);
    return 0;
} }
```

ptrace(2) example

Debugging – Technical Background (5)

- User expects **source-code** visualization: *source-level debugging*
  - **Prerequisites:** access to sources, (compiler-generated) *debug information*

```
$ g++ -g -o hello hello.cc
$ objdump --section-headers hello
hello:     file format elf64-x86-64
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
...
 24 .data         00000010  0000000000004000  0000000000004000  00003000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 25 .bss          00000118  0000000000004040  0000000000004040  00003010  2**6
                  ALLOC
 26 .comment      00000025  0000000000000000  0000000000000000  00003010  2**0
                  CONTENTS, READONLY
 27 .debug_aranges 00000030  0000000000000000  0000000000000000  00003035  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 28 .debug_info   000023bb  0000000000000000  0000000000000000  00003065  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 29 .debug_abbrev 0000059b  0000000000000000  0000000000000000  00005420  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 30 .debug_line   0000014a  0000000000000000  0000000000000000  000059bb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 31 .debug_str    0000120b  0000000000000000  0000000000000000  00005b05  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 32 .debug_line_str 0000028b  0000000000000000  0000000000000000  00006d10  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

Remote Debugging

- Allows debugging programs on platforms we cannot (yet) work on interactively
  - Requires **communications link** (serial, Ethernet, ...)
  - ... which in turn necessitates a **device driver**
  - Target “device” can also be an emulator (e.g., QEMU)
- Debugging component on the target system (“stub”) should be as simple as possible

Remote Debugging – Example GDB (1)

- Communication protocol
  ("*GDB Remote Serial Protocol*" – RSP)
  - Reflects requirements on GDB *stub*
  - Based on transferring ASCII strings
  - Message format: **$**<*command or reply*>**#**<*checksum*>
  - Messages are directly acknowledged with **+** (OK) or **–** (error)
- Examples:
  - **$**g**#**67 ►Read contents of all registers
    - Reply: **+ $**123456789abcdef0...**#**... ► Reg. 1 = 0x12345678, 2 = 0x9...
  - **$**G123456789abcdef0...**#**... ► Set register contents
    - Reply: **+ $**OK**#**9a ► Success
  - **$**m4015bc,2**#**5a ► Read 2 bytes starting at address 0x4015bc
    - Reply: **+ $**2f86**#**06 ► Value 0x2f86

Remote Debugging – Example GDB (2)

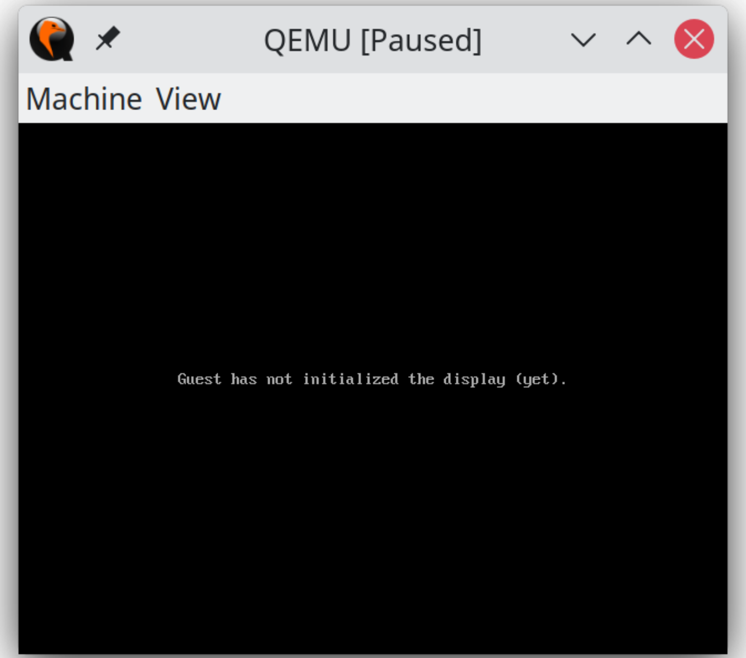
- Communication protocol – all command categories:
  - Register and memory commands
    - read/write all registers
    - **read/write single register**
    - **read/write memory area**
  - Controlling program execution
    - request reason for latest interruption
    - single-step
    - **continue execution**
  - Miscellaneous
    - Print to debug console
    - Error messages

Minimum stub functionality

Remote Debugging – with QEMU

- With the right command-line parameters, QEMU offers a GDB stub communicating via TCP

```
$ make qemu-gdb
...
```



Remote Debugging – with Bochs

```
$ gdb build/system
GNU gdb (Ubuntu 11.1-0ubuntu2) 11.1
...
Reading symbols from build/system...
(gdb) break main
Breakpoint 1 at 0x10167f: file main.cc, line 11.
(gdb) target remote localhost:2024
Remote debugging using localhost:2024
0x000000000000fff0 in ?? ()
(gdb) continue
Continuing.

Breakpoint 1, main () at main.cc:4
4       {
(gdb) next
11              return 0;
(gdb) continue
Continuing.
```

Automated in OOStuBS Makefile to prevent TCP-port collisions:

make gdb

(and skip the `target remote` … step)

Debugging *Deluxe*

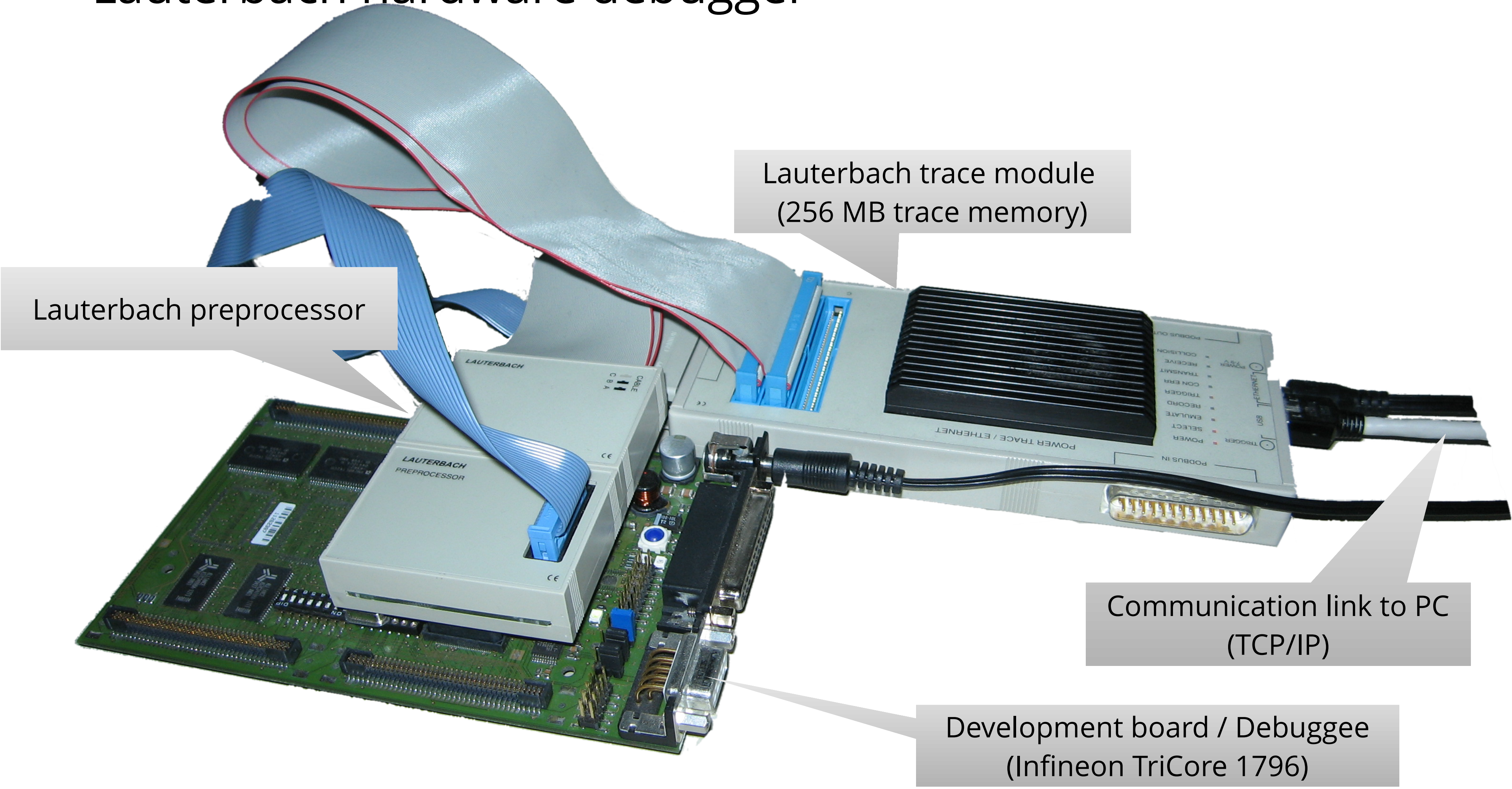
- Many chip manufacturers integrate **hardware support for debugging** (OCDS – *On Chip Debug System*)
  - BDM, OnCE, MPD, JTAG
- Usually simple serial protocols between debugging unit and external debugger (save chip pins!)
- Advantages:
  - *Debug Monitor* (e.g. gdb stub) does not use any application memory
  - Debug Monitor implementation unnecessary
  - ROM/FLASH breakpoints using hardware breakpoints
  - Concurrent access to memory and CPU registers
  - Specialized hardware partially allows to record a control-flow trace (ex post analysis)

Debugging *Deluxe* – Example BDM

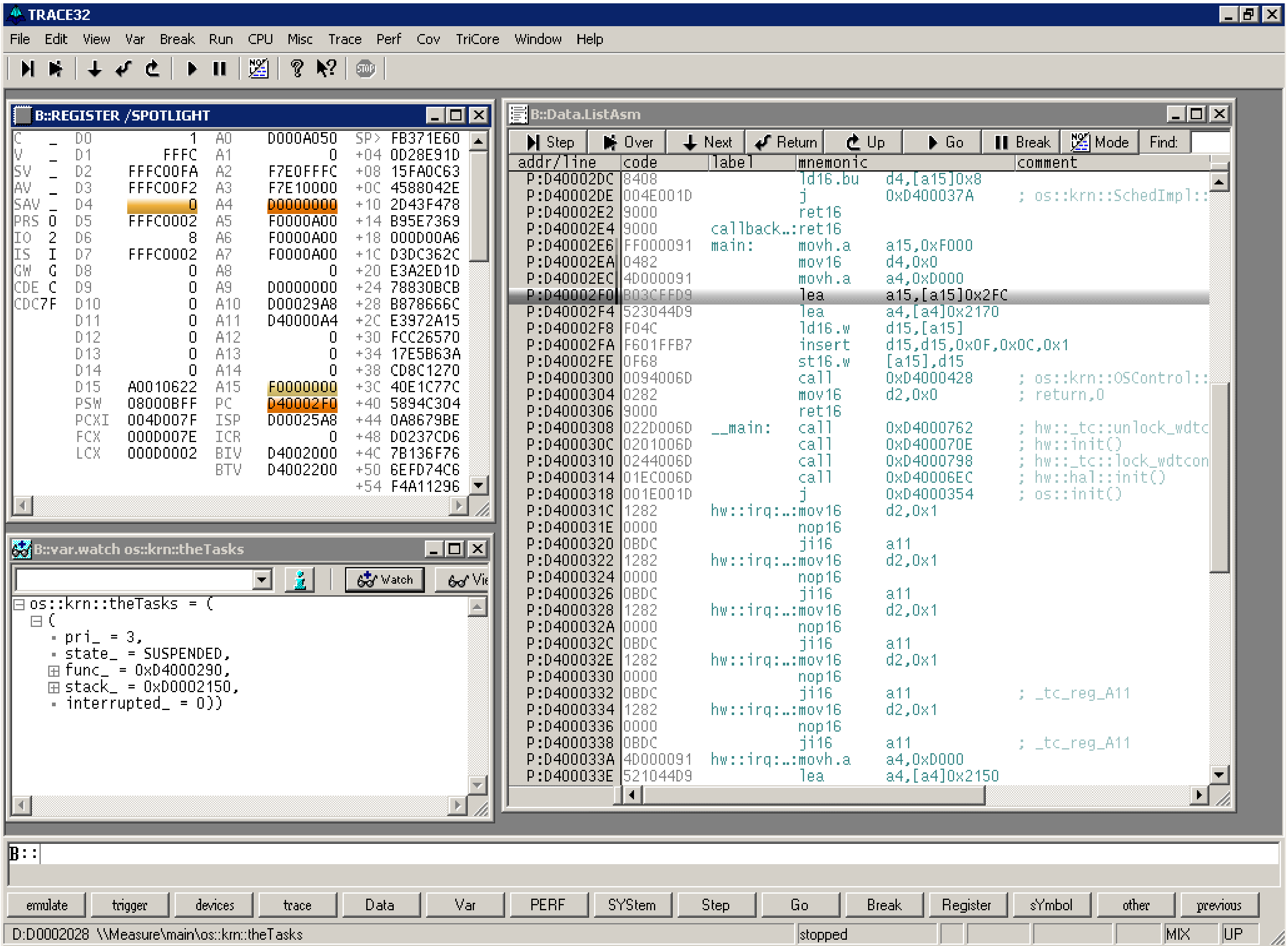
- "Background Debug Mode" – on-chip debug solution by Motorola
- Serial communication via 3 lines (DSI, DSO, DSCLK)
- BDM commands of 68k and ColdFire processors:
  - RAREG/RDREG – Read Register
    - read particular data or address register
  - WAREG/WDREG – Write Register
    - write particular data or address register
  - READ/WRITE – Read Memory/Write Memory
    - read/write specific memory location
  - DUMP/FILL – Dump Memory/Fill Memory
    - read/fill block of memory
  - BGND/GO – Enter BDM/Resume
    - stop/continue execution

Debugging *Deluxe* – Hardware Solution

- Lauterbach hardware debugger

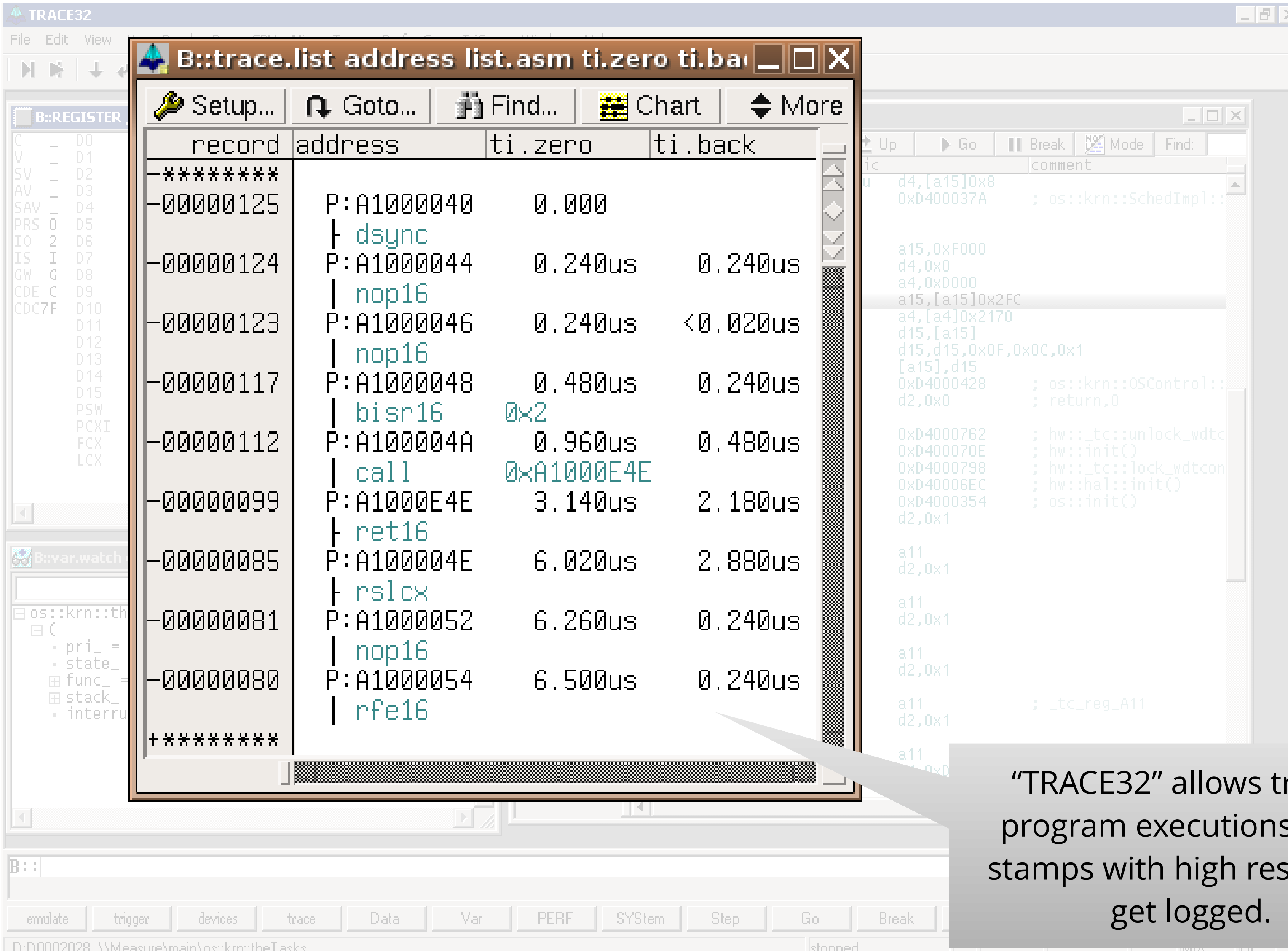


Debugging *Deluxe* – Lauterbach Frontend



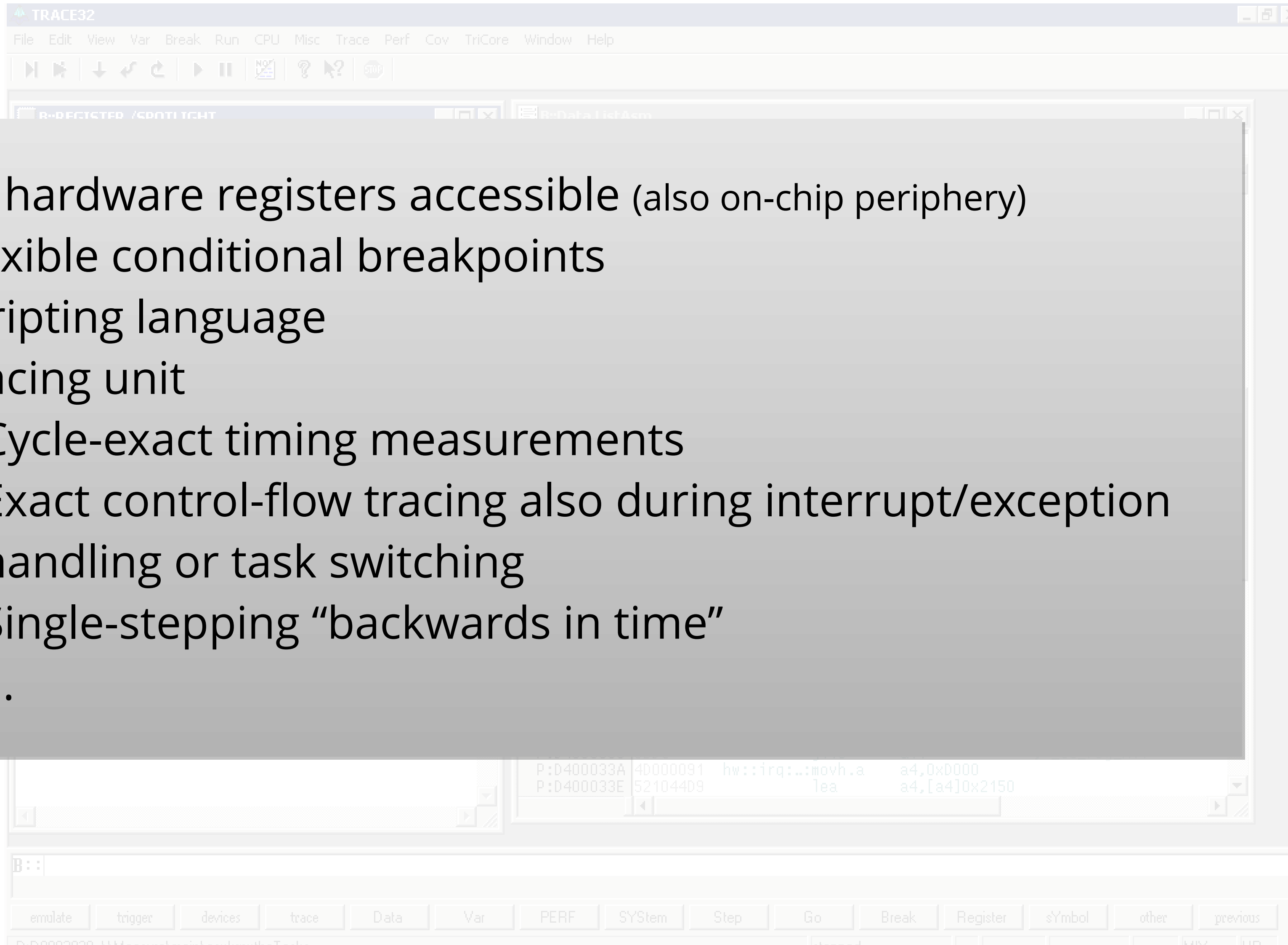
TECHNISCHE UNIVERSITÄT DRESDEN

Debugging *Deluxe* – Lauterbach Frontend



"TRACE32" allows tracing program executions. Time stamps with high resolution get logged.

Debugging *Deluxe* – Lauterbach Frontend



- All hardware registers accessible (also on-chip periphery)
- Flexible conditional breakpoints
- Scripting language
- Tracing unit
  - Cycle-exact timing measurements
  - Exact control-flow tracing also during interrupt/exception handling or task switching
  - Single-stepping “backwards in time”
  - …

Summary

- Operating-system development differs significantly from regular application development:
  - No libraries
  - Bare metal is the basis we build upon
- The first steps are often the hardest
  - Compilation/linking, booting, system initialization
- Comfortable bug hunting necessitates infrastructure
  - Device drivers for “printf debugging”
  - Stub and communication link/driver for remote debugging
  - Hardware debugging support like with BDM
  - Ideal: Professional hardware debuggers (e.g. Lauterbach)