Operating System Concepts and Implementations

Betriebssystemkonzepte und -implementierungen

II. Operating-System Development

October 13, 2026

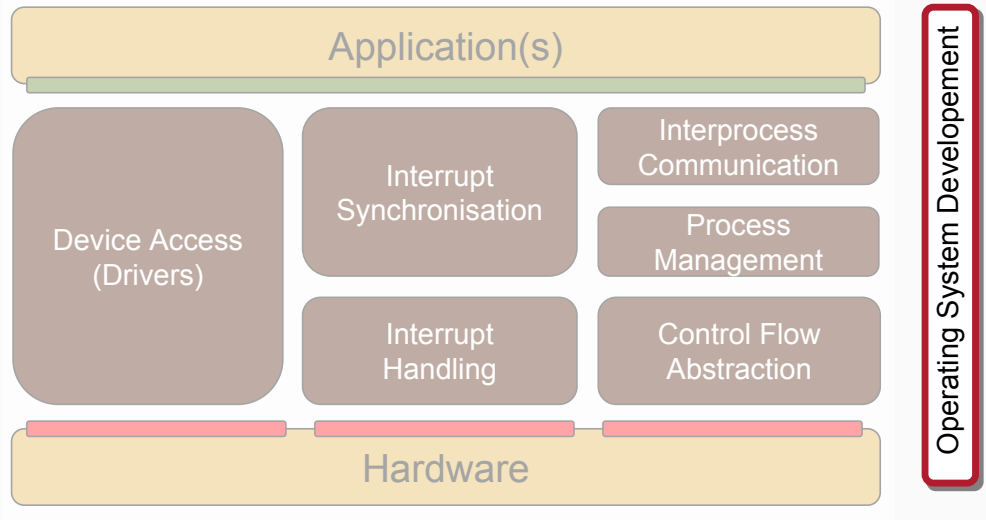
Benedict Herzog

Material based on *Chair of Computer Science 4 (System Software)*
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

RUHR
UNIVERSITÄT
BOCHUM

RUB

Overview: Today's Focus



OS Development (Often an Uphill Battle)

- **First steps** – How do you get your system onto the target hardware?
  - Compilation
  - Boot process
- **Testing and debugging** – What do you do when the system stops responding?
  - `printf()` debugging
  - Emulators
  - Debuggers
  - Remote debuggers
  - Hardware support

Agenda

Context

Compilation / Linking

Booting

Debugging

Summary

Agenda

Context

Compilation / Linking

Booting

Debugging

Summary

Compilation – *Hello, World?*

```
#include <iostream>

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

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

- **Assumptions:**
  - The development system runs Linux/x86
  - The target system is also a PC
- Does this program also run on **bare metal**?
- Can operating systems be developed in a **high-level language** at all?

Compilation – Problems and Solutions

- **No dynamic linker** available
  - ⇒ **Statically link** all required **libraries**
- `libstdc++` and `libc` use **Linux system calls** (e.g. `write()`)
  - ⇒ Standard C/C++ **runtime libraries cannot be used**. Alternatives are (usually) not available.
- Generated addresses refer to **virtual memory**!
  (“`nm hello | grep main`” returns “`0804846c T main`)”
  - ⇒ Linker’s default settings cannot be used. **Custom linker configuration** is required.
- High-level language code has **environment expectations**
  (register usage, address mapping, runtime environment, stack, ...)
  - ⇒ Custom **startup code** (written in assembly) must prepare high-level code execution

Agenda

Context
Compilation / Linking
Booting
Debugging
Summary

Booting

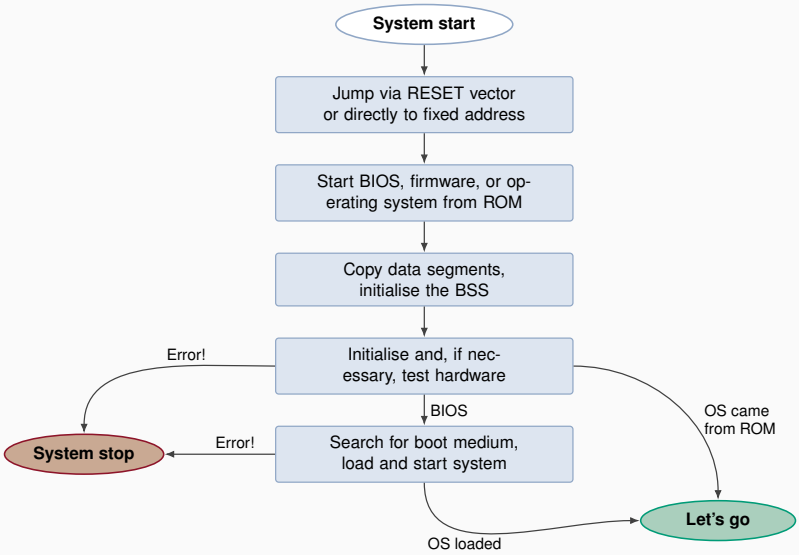
"Boot is short for **bootstrap** or **bootstrap load** and derives from the phrase to pull oneself up by one's bootstraps."

"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
Start BIOS, firmware, or operating system from ROM
Copy data segments, initialise the BSS
Initialise and, if necessary, test hardware
Error!
BIOS
OS came from ROM
System stop
Error!
Search for boot medium, load and start system
Let's go
OS loaded

PC Boot Process – Boot Sector

- The PC BIOS loads the first block (512 bytes) of the boot drive at address `0x7c00` and jumps there (blindly!)
- Boot sector loading:

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`	Numer 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 Boot Process – Boot Sector

- The PC BIOS loads the first block (512 bytes) of the boot drive at address `0x7c00` and jumps there (blindly!)
- Boot sector loading:

Only the start and the **“signature” (`0xaa55`)** at the end are actually important. Everything else is used by the **boot loader** to load the actual system.

Alternative (OOStuBS):

Offset	Inhalt
0x0000	`jmp boot;`
0x0004	Number of tracks
0x0006	Number of heads
0x0008	Number of sectors
0x000a	Reserved sectors (setup code)
0x000c	Reserved sectors (system)
0x000e	BIOS device code
0x000f	Start track of disk/partition
0x0010	Start head of disk/partition
0x0011	Start sector of disk/partition
0x0010	`boot:` `...`
0x01fe	**`0xaa55`**

PC Boot Process – Boot Loader

- Simple, **system-specific** boot loaders
  - Establish a defined initial state of the hardware and software
  - If necessary: load additional blocks containing boot-loader code
  - Locate the actual system on the boot medium
  - Load the system (using BIOS functions)
  - Jump to the loaded system
- “Boot loader” on non-bootable floppy disks
  - Display an error message and reboot
- Boot loader with **boot menu** (e.g. GRUB)
  (e.g. in the Master Boot Record of a hard disk)
  - Display a selection menu
  - Emulate the BIOS behaviour when booting the selected system
    - Load the corresponding boot block at `0x7c00` and start it

Agenda

Context

Compilation / Linking

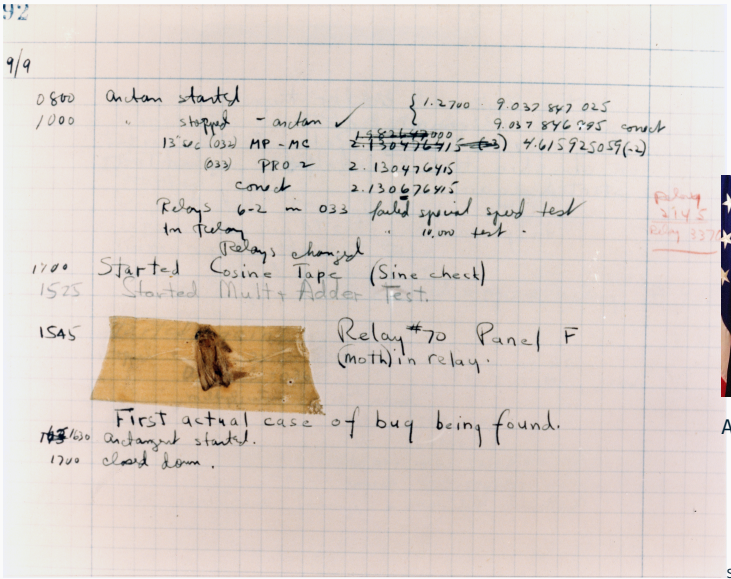
Booting

Debugging

- `printf()` Debugging
- Software Emulators
- Debugger
- Source-Level Debugging
- Remote Debugging
- Hardware Support

Summary

The First Documented “Bug”



Admiral Grace Hopper

Source: Wikipedia

“`printf()` Debugging”

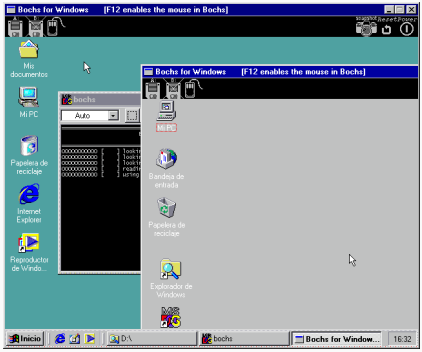
- Not that simple: if you do not have a (working) `printf()`
  - Often there is not even a screen
- `printf()` often changes the behaviour of the program being debugged
  - With `printf()`, the bug may suddenly disappear or manifest itself differently
  - This is particularly true in operating-system development
- Last resorts:
  - A blinking LED
  - A serial interface

(Software) Emulators

- Reproduce real hardware in software
  - Simplifies debugging
    - Emulation software is generally more communicative than real hardware
  - Shorter development cycles
- **Caution**: ultimately, the system must run on real hardware!
  - Emulators and real hardware may differ in subtle details
  - Bugs are harder to find in a finished system than in a system developed incrementally
- By the way: “virtual machines” and “emulators” are **not** synonymous
  - In VMware, for example, an x86 processor is not emulated; instead, the physical processor executes machine code from the VM directly

Emulators – Example: “Bochs”

- Emulates i386, ..., Pentium, AMD64 (interpreter)
  - Optional MMX, SSE, SSE2, and 3DNow! instructions
  - Multiprocessor emulation
- Emulates complete PC
  - Memory and devices (including sound, networking, ...)
  - Capable to run Windows and Linux
- Implemented in C++
- Development support
  - Diagnostic information (e.g. from crash)
  - Built-in debugger (GDB stub)

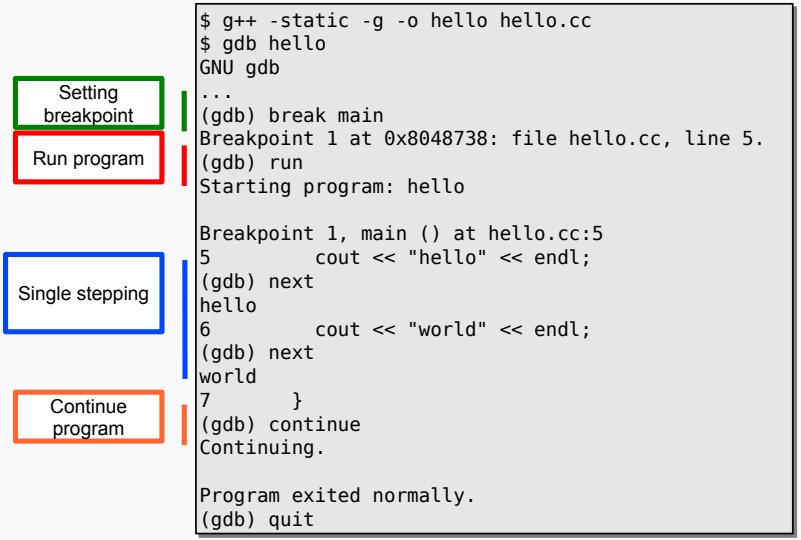


Bochs in Bochs

Debugging

- A debugger is used to locate software errors by tracing program execution
  - **Single-step mode**: one instruction at a time
  - **Breakpoints**: when a particular instruction is reached
  - **Watchpoints**: when a particular data element is accessed
- **Caution**: sometimes debugging with a debugger takes longer than necessary
  - Taking a break and careful reasoning often can be quicker
    - Single-stepping takes a lot of time
    - There is no going back if the point of interest is accidentally missed
  - With `printf` debugging, output can be prepared and presented more effectively
  - Synchronisation/race conditions bugs are impractical to debug with a debugger
- Useful: analysis of “core dumps”
  - Less relevant in operating-system development

Debugging - Example Session



Debugging – How It Works (1)

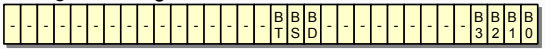
- Virtually all CPUs provide hardware support for debugging
- Example: Intel x86 CPUs
  - `INT3` instruction triggers a “breakpoint interrupt” (a trap)
    - Deliberately inserted into the code by debugger
    - Trap handler transfers control flow to debugger
  - Setting **Trap Flag (TF)** in status register (EFLAGS): triggers a “debug interrupt” after **each** instruction
    - Can be used to implement single-step mode
    - Trap handler itself is not executed in single-step mode
  - **Debug registers DR0–DR7** can monitor up to four 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

Debugging – How It Works (1)

Breakpoint registers

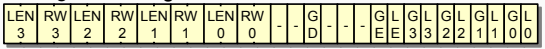
breakpoint 0: linear address	DR0
breakpoint 1: linear address	DR1
breakpoint 2: linear address	DR2
breakpoint 3: linear address	DR3
reserved	DR4
reserved	DR5

Debug status register



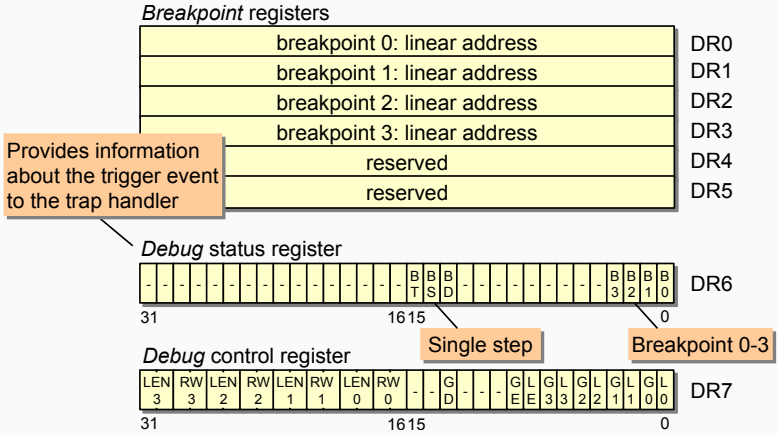
DR6

Debug control register

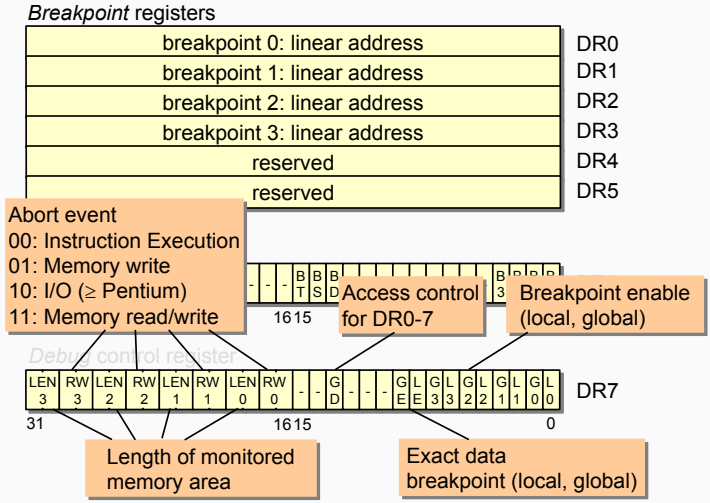


DR7

Debugging – How It Works (1)



Debugging – How It Works (1)



Debugging – How It Works (3)

- Debugging is particularly effective when program execution is visualised in the source code (source-level debugging)
  - Requires access to the source code and debugging information
  - Must be supported by the compiler

```
> 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
```

Remote Debugging

- Allows debugging programs on platforms we cannot (yet) work on interactively
  - Requires a **communication link** (serial, Ethernet, ...)
    - ... which in turn requires a device driver
  - The target machine may 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 transmission of ASCII strings
  - Message format: `$<command or reply>#<checksum>`
  - Messages are directly acknowledged with `+` (OK) or `-` (error)
- Examples:
  - `$g#67` ⇒ read all register contents
    - Reply: `+$123456789abcdef0...#...` ⇒ Reg. 1 is `0x12345678, 2= 0x9...`
  - `$G123456789abcdef0...#...` ⇒ set register contents
    - Reply: `+$OK#9a` ⇒ successful
  - `$m4015bc,2#5a` ⇒ read 2 bytes starting at address `0x4015bc`
    - Reply: `+$2f86#06` ⇒ value is `0x2f86`

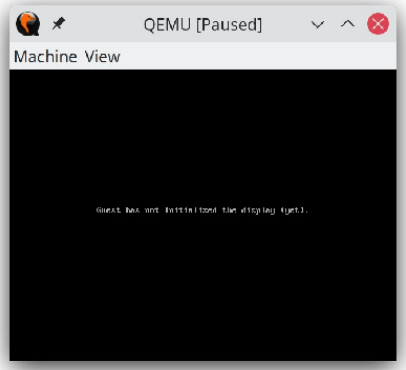
Remote Debugging – Example: GDB (2)

- Communication protocol – all operations
  - Register and memory operations
    - Read/write all registers
    - **Read/write an individual register**
    - **Read/write a memory region**
  - Control of program execution
    - Query the cause of the most recent interruption
    - Single-step
    - **Continue execution**
  - Miscellaneous
    - Print to debug console
    - Error messages
- **Blue Operations**: minimal stub implementation

Remote Debugging – with QEMU

- With the appropriate command-line parameters, QEMU provides a GDB stub that communicates via TCP

```
> make qemu-gdb
...
```



Remote Debugging – with QEMU

Automated in OSCI:

- Makefile to prevent TCP-port collisions: `make gdb`
- skip `target remote` step ...

```
> 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.
```

Debugging Deluxe

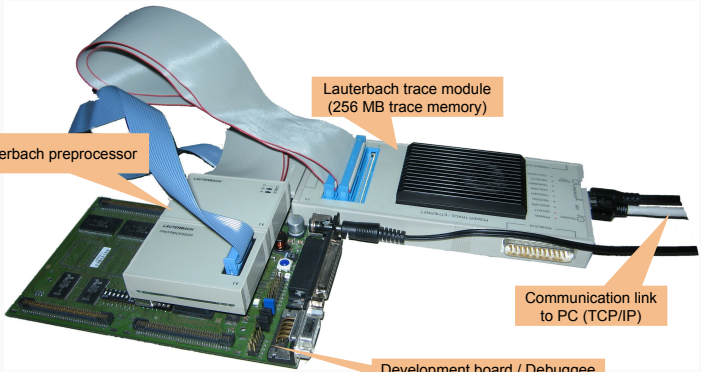
- Many processor manufacturers now integrate **hardware support for debugging** into their chips (OCDS – On-Chip Debug System)
  - BDM, OnCE, MPD, JTAG
  - Usually simple serial protocol between on-chip debugging unit and external debugger
- Some chip manufacturers add **on-chip tracing/debugging features** to their processors
  - Intel PT and Arm CoreSight
  - Instructions are recorded into a trace buffer in memory
- Advantages:
  - *Debug monitor* (e.g. GDB stub) occupies no memory
  - *Debug monitor* implementation unnecessary
  - Breakpoints in ROM/FLASH using hardware breakpoints
  - Concurrent access to memory and CPU registers
  - Specialised hardware partially allows to record control-flow trace

Debugging Deluxe – Example: BDM

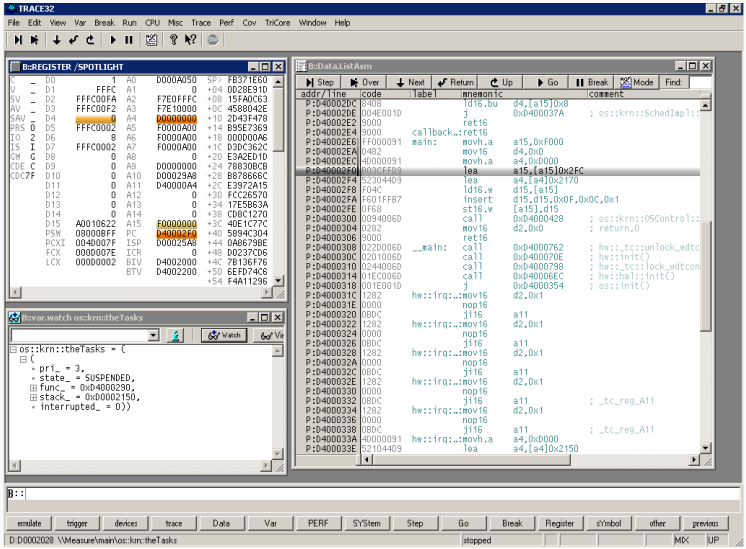
- “Background Debug Mode” – on-chip debugging solution by Motorola
- Serial communication via three lines (DSI, DSO, DSCLK)
- BDM commands of the 68k and ColdFire processors
  - RAREG/RDREG – Read Register
    - Read specific data or address register
  - WAREG/WDREG – Write Register
    - Write specific data or address register
  - READ/WRITE – Read Memory/Write Memory
    - Read/write specific memory location
  - DUMP/FILL – Dump Memory/Fill Memory
    - Read/fill an entire memory block
  - BGND/GO – Enter BDM/Resume
    - Stop/resume execution

Debugging Deluxe: Hardware Solution

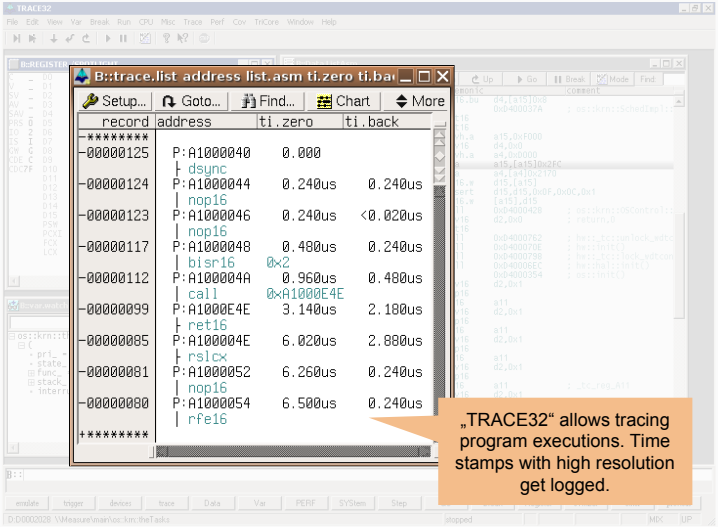
- Lauterbach hardware debugger



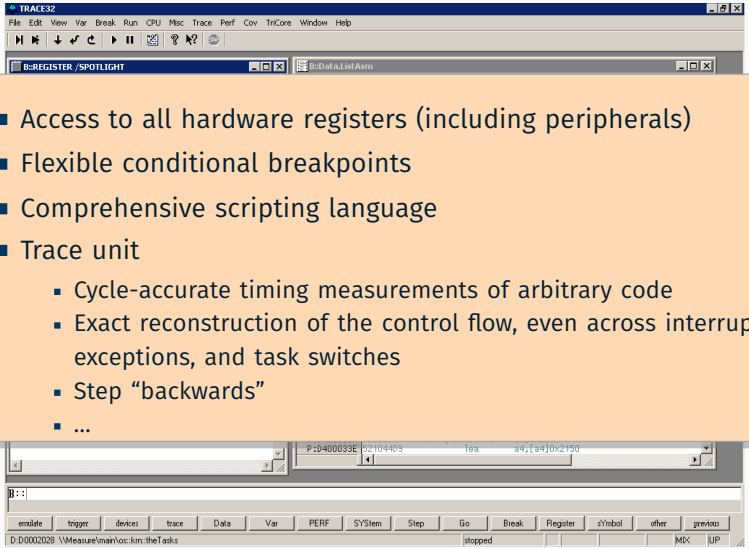
Debugging Deluxe: Lauterbach Frontend



Debugging Deluxe: Lauterbach Frontend



Debugging Deluxe: Lauterbach Frontend



- Access to all hardware registers (including peripherals)
- Flexible conditional breakpoints
- Comprehensive scripting language
- Trace unit
  - Cycle-accurate timing measurements of arbitrary code
  - Exact reconstruction of the control flow, even across interrupts, exceptions, and task switches
  - Step “backwards”
  - …

Agenda

Context
Compilation / Linking
Booting
Debugging
Summary

Summary

- Operating-system development differs significantly from ordinary application development
  - Libraries are unavailable
  - “Bare” metal is the basis we build upon
- First steps are often the hardest
  - Compilation
  - Boot process
  - System initialisation
- Convenient debugging requires supporting infrastructure
  - Device drivers for `printf()` debugging
  - Stub and connection/driver for remote debugging
  - Hardware debugging support such as BDM
  - Ideally: professional hardware debugger such as Lauterbach