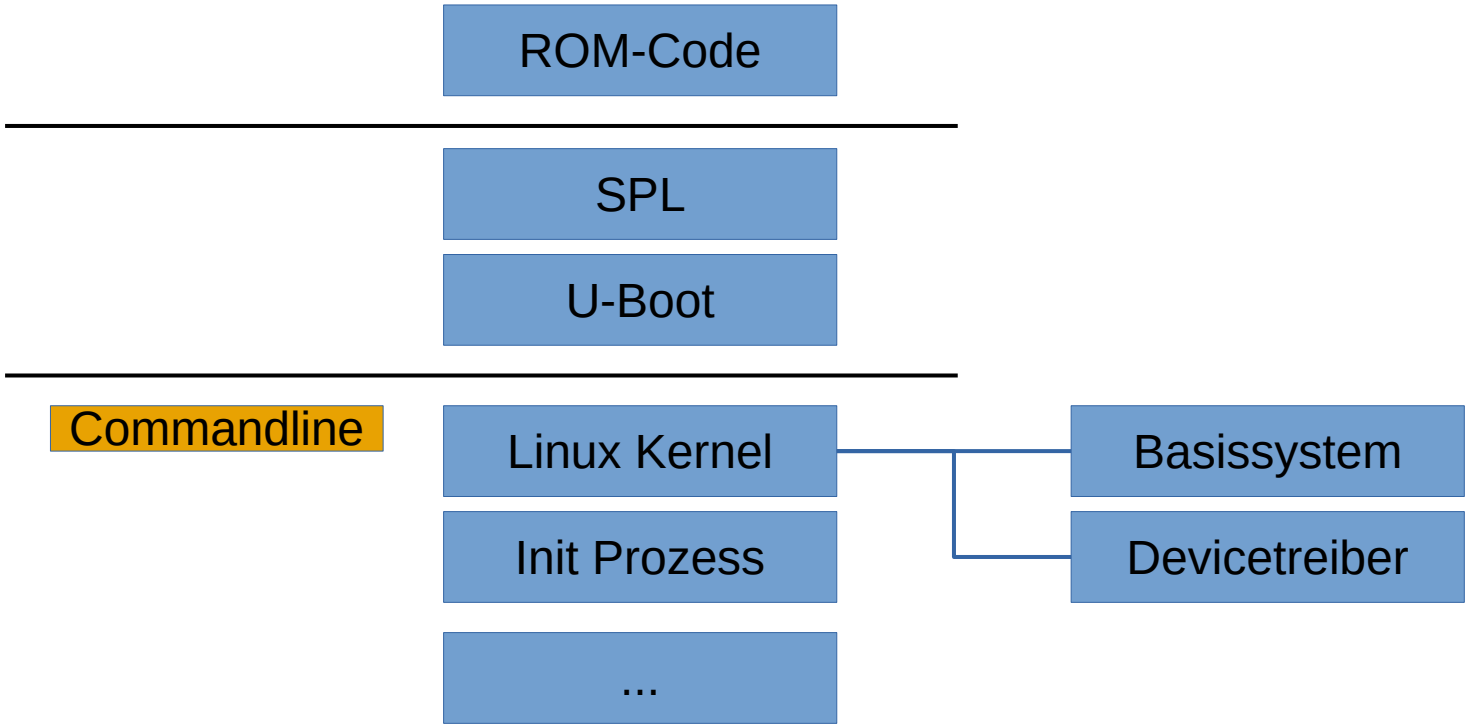
Linux Debugging

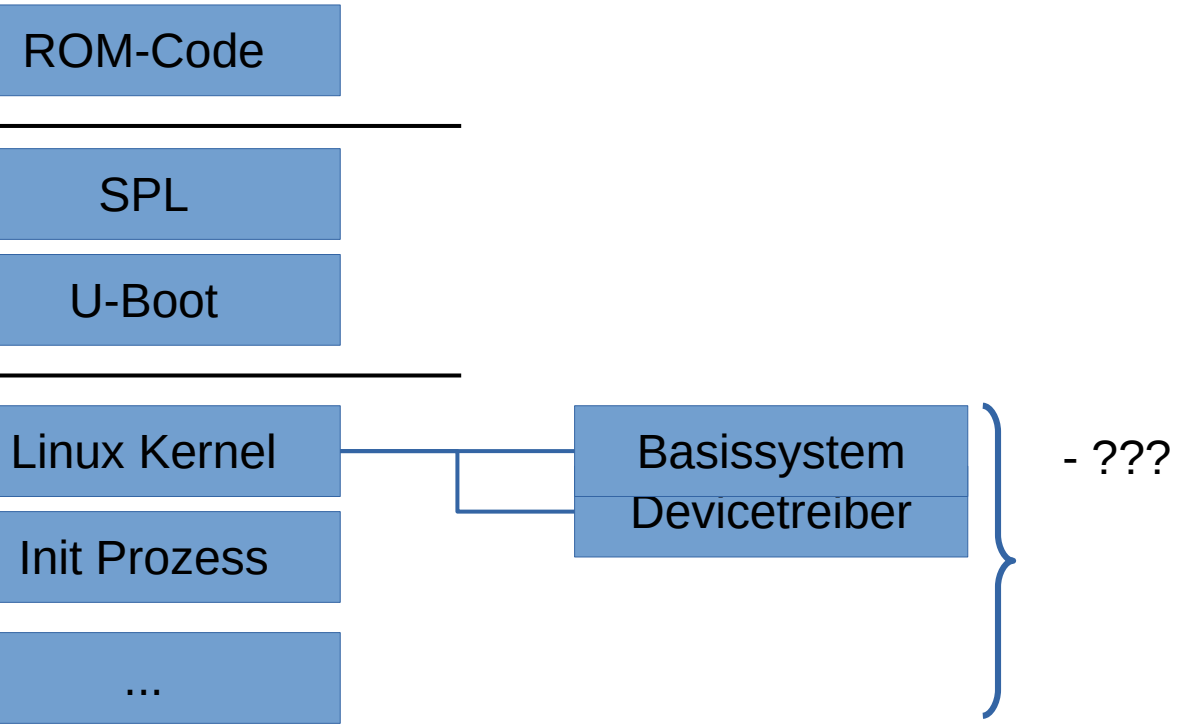
EIM/INFM

Frank Erdrich
frank.erdrich@emtrion.de

Aktueller Stand



Aktueller Stand



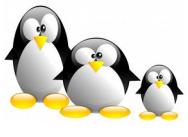
Debugging Linux

- Wie Fehler in Kernel(-treibern) und Userspace-Programmen finden?

Debugging Linux

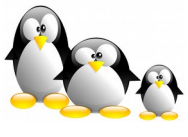
- printk()
  - Simpel und einfach zu verwenden
  - Unerlässlich, um den Status innerhalb eines Treibers auszugeben
  - Einfach zu implementieren
  - Schnelles Ergebnis

- Aber...!?!?



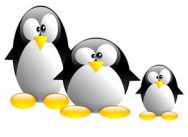
Debugging Linux

- printk()
  - benötigt Ausführungszeit:
    - Verlangsamt den entsprechenden Codeteil
    - zeitabhängige Fehler
      - entweder nicht mehr sichtbar
      - erscheinen durch printk()
  - Ohne serielle Schnittstelle oder Display kein printk() → **was tun?**



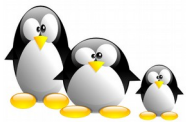
Debugging Linux

- Recap:
  - Wo können printk-Meldungen immer nachgelesen werden?
  - Welche Levels existieren?
  - Wie/Wo kann der minimal auszugebende Level gesteuert werden?
  - Wie kann nach Levels gefiltert werden?



Debugging Linux

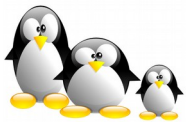
- Wo können printk-Meldungen immer nachgelesen werden?
  - Kernel Logging Buffer (abhängig vom LogLevel)
    - dmesg
    - /var/log/messages (systemabhängig)
    - journalctl (systemabhängig → systemd)
    - Direkt im Speicher → JTAG



Debugging Linux

- Welche Levels existieren?

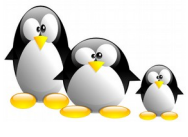
Level	Name	Bedeutung
0	KERN_EMERG	system is unusable
1	KERN_ALERT	action must be taken immediately
2	KERN_KRIT	critical conditions
3	KERN_ERR	error conditions
4	KERN_WARNING	warning conditions
5	KERN_NOTICE	normal but significant condition
6	KERN_INFO	informational
7	KERN_DEBUG	debug-level messages



Debugging Linux

- Wie/Wo kann der minimal auszugebende Level gesteuert werden?
  - Kernel cmdline → „loglevel=level“
  - echo "3" > /proc/sys/kernel/printk

    → alle Level kleiner *loglevel* werden ausgegeben



Debugging Linux

- Wie kann nach Levels gefiltert werden?
  - Siehe „man dmesg“:
    - `-l, --level list`

      `Restrict output to the given (comma-separated) list of levels.`

      `For example:`

      `dmesg --level=err,warn`

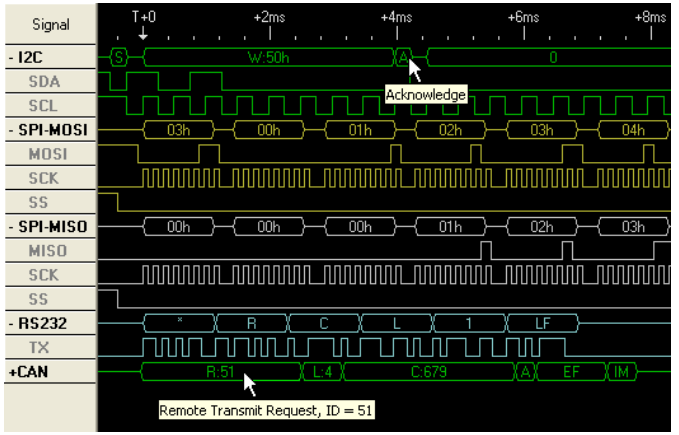
Debugging Linux

- Debugging mit GPIOs
  - GPIO ein- bzw. ausschalten
  - Schnell → zeitliche Abhängigkeiten werden vermieden
  - Zusammenhänge und Abläufe besser deutbar darstellbar

Als Trigger für analoge Signalmessung verwendbar!

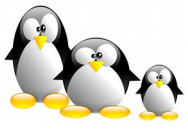
Debugging Linux

- Debugging mit GPIOs
  - Oszilloskop notwendig (LED auch möglich, falls Events langsam genug)
  - Alternativ: Logic-Analyser
  - Freie GPIO-Pins notwendig (nicht immer gegeben)



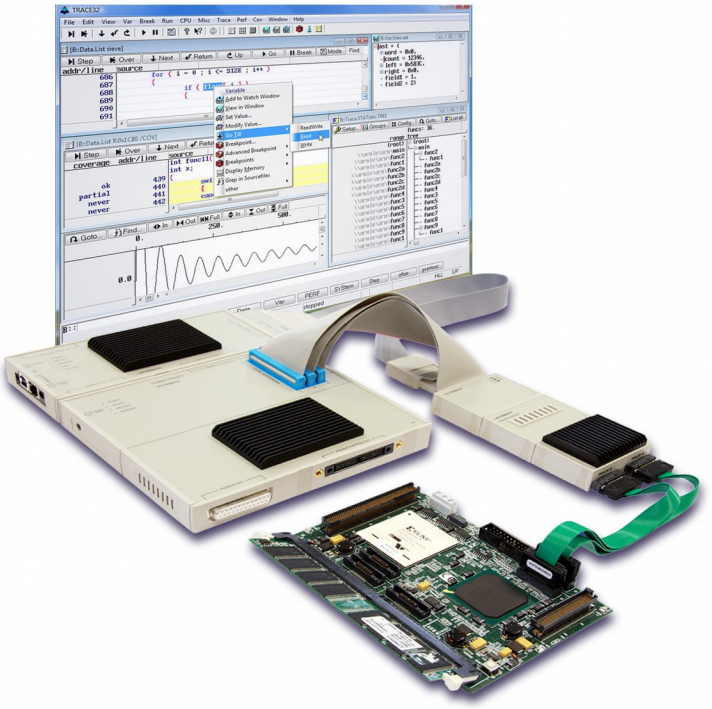
Debugging Linux

- JTAG-Debugger
  - Notwendig für Systemdebugging auf Hardwareebene (→ Registerinhalte einsehen, Speicherdumps, ...)
  - Kann den aktuellen Stand der CPU „einfrieren“
  - Kontrolle von Registerinhalten über eine dritte Instanz auslesen

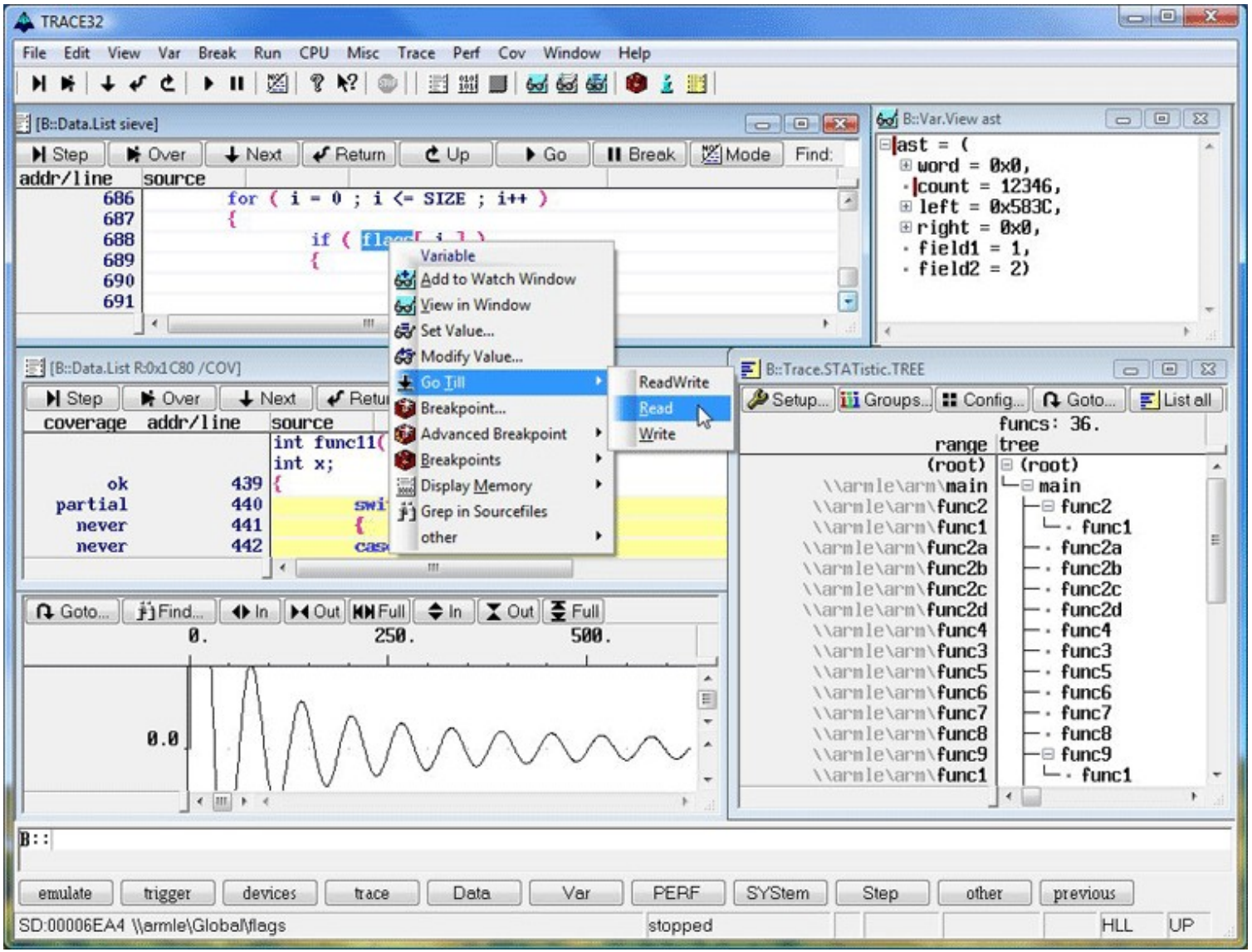


Debugging Linux

- JTAG-Debugger
  - Zusätzliche Hardware notwendig, z. B.:
    - Lauterbach TRACE32
    - Segger JLINK
    - ARM DS-5 mit DSTREAM

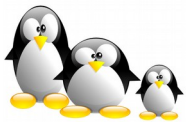


Debugging Linux



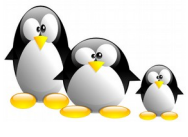
Debugging Linux

- JTAG-Debugger
  - Benötigt Debug-Port auf der CPU
    - JTAG mit z.B. CoreSight Trace (ARM)
  - CoreSight Trace
    - → ETM (Embedded Trace Macrocell)
    - → PTM (Program Flow Trace Macrocell)
    - → ITM (Instrumentation Trace Macrocell)



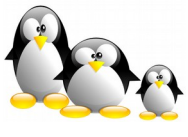
Debugging Linux

- ETM (Embedded Trace Macrocell)
  - Instruction und Data Trace (auch interessant für Code Coverage)
  - Trigger und Filter vorhanden
  - Für Details siehe
    https://static.docs.arm.com/ddi0468/a/DDI0468A_coresight_etm_a7_trm.pdf



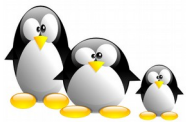
Debugging Linux

- PTM (Program Flow Trace Macrocell)
  - Traced verschiedene, sogenannte Waypoints → branches, eceptions, barriers, context changes, ...
  - Kann auch globale Timestamps und Taktzyklen zwischen Waypoints aufzeichnen



Debugging Linux

- ITM (Instrumentation Trace Macrocell)
  - Erlaubt Tracing durch User → printf-ähnliche Anweisung im Code
  - Erweiterte Code-Coverage



Debugging Linux

- GDB
  - GNU Project Debugger
  - Open Source
  - Für verschiedene Architekturen verfügbar
    - ARM, x86
    - Teilweise spezielle architekturspezifische Erweiterungen

      → VFP Register lesen auf ARM

Debugging Linux

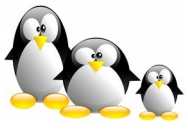
- GDB einfaches Beispiel

```
$ gdb helloworld

(gdb) break main   # set breakpoint on main
(gdb) run          # run program
(gdb) step         # step in program
(gdb) print var    # print value of var
```

Debugging Linux

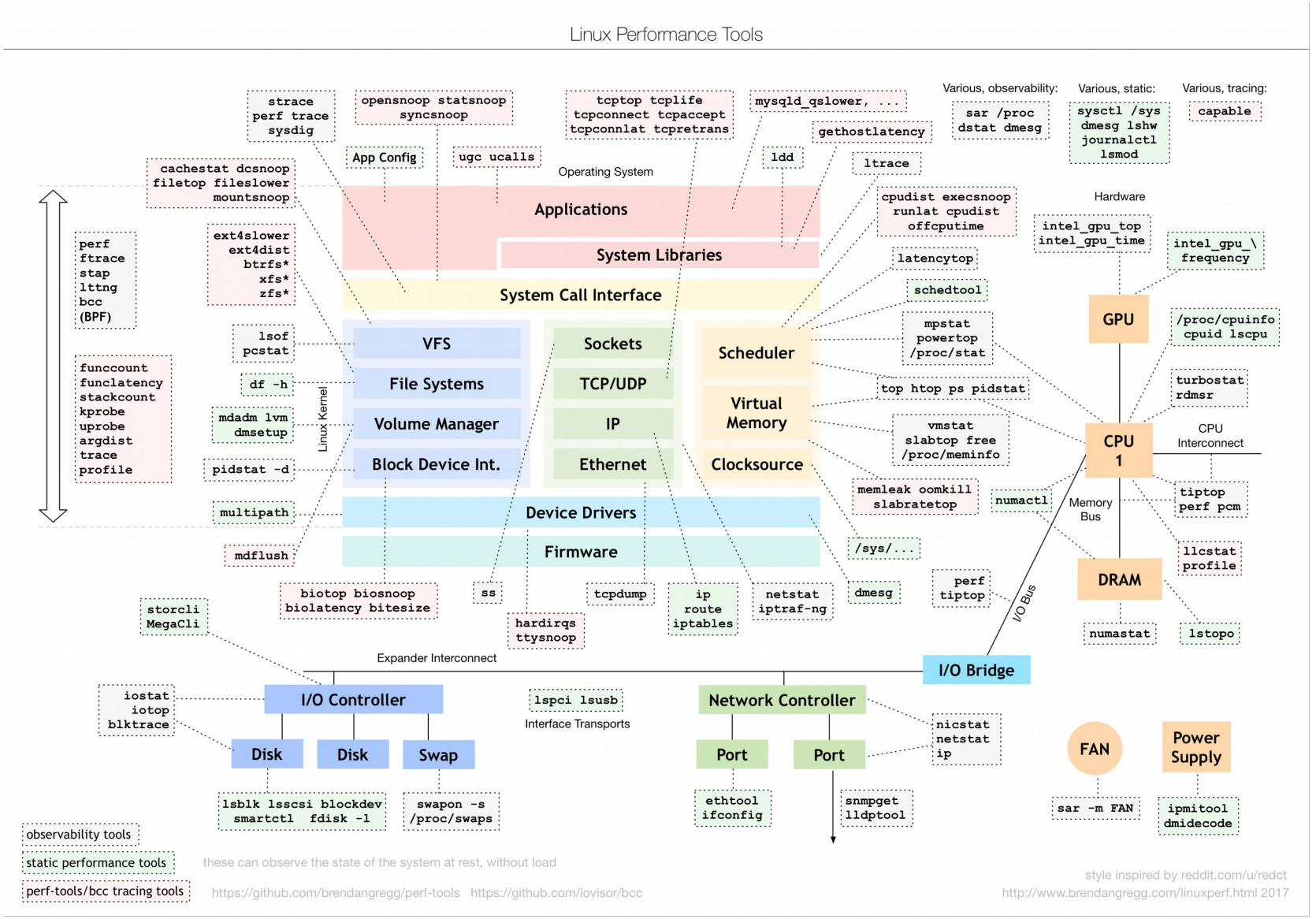
- Lokale oder Remote Debugging Sessions
- Unterstützung für Breakpoints
- Ansehen und Manipulieren von Speicher
- Aufrufen von Funktionen
- Tracing
- Reverse-Execution (je nach Plattform)



Performance Measurement

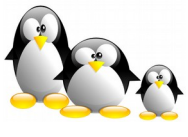
- Warum Performance Measurement?

Performance Measurement



Performance Measurement

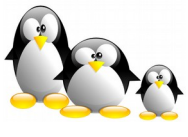
- Hardwaregestützte Performance Measurement
  - PMU (Performance Measurement Unit)
  - Auf ARM und x86 verfügbar
  - Zählt diverse Events, etwa Cache-Hits und Misses, Anzahl Exceptions und Interrupts, Anzahl ausgeführter Instructions pro Taktzyklus



Performance Measurement

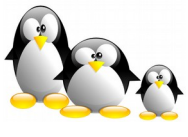
- Hardwaregestützte Performance Measurement
  - Sehr hardwareabhängig, teilweise erweitert durch CPU-Fertiger wie Qualcomm
  - Sicherheit bzw. Security?

https://www.usenix.org/system/files/conference/woot16/woot16-paper-spisak.pdf



Performance Measurement

- ftrace
  - Kernel Function Trace Interface
  - Kernelspace Debugging
  - Latenz- und Performance-Probleme
  - Aufrufreihenfolge von Funktionen
  - Eventtracing
  - Trigger (Tracing nur, wenn Trigger aktiviert wurde)



Performance Measurement

- ftrace
  - Virtuelles Dateisystem
  - Meistens eingebunden unter /sys/kernel/tracing

    → `mount -t tracefs nodev /sys/kernel/tracing`

Performance Measurement

- ftrace – wichtige Dateien
  - current_tracer
  - available_tracers
  - tracing_on
  - trace
  - buffer_size_kb
  - trace_options

siehe https://www.kernel.org/doc/html/latest/trace/ftrace.html

Performance Measurement

- ftrace
  - Kann mit Userspace synchonisiert werden
    - trace_fd = open("trace_marker", WR_ONLY);
    - write(trace_fd, buf, n);
    - Kann Trigger auslösen

      → ftrace Aufzeichnung aus Applikation heraus starten

Performance Measurement

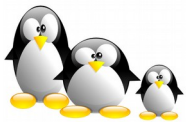
- ftrace
  - Verfügbare (wichtige) Tracer
    - function → function entry probe
    - function_graph → entry and exit probe
    - (siehe $ cat available_tracers)

Performance Measurement

- Beispiel

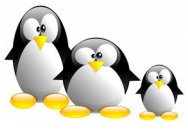
```
$ echo function_graph > current_tracer
$ echo 1 > tracing_on
$ echo 0 > tracing_on
$ cat trace              # alternativ $ less trace
$ echo funcgraph-abstime > trace_options
$ cat trace

Filter:
$ echo 'hrtimer_*' > set_ftrace_filter
$ echo 1 > tracing_on
$ echo 0 > tracing_on
$ cat trace
```



Performance Measurement

- Perf
  - Linux Performance Events Subsystem
  - Hardwarebasiert -> Special Purpouse Registers
  - Hardware und Software Events
  - Geringerer Overhead im Vergleich zur Instumentierung
  - Keine Instrumentierung notwendig
  - Viele Beispiele auf http://www.brendangregg.com/perf.html



Performance Measurement

- strace
  - Userspace-Tool, welches die syscalls eines Prozesses aufzeigt
  - Zeigt z.B. Bibliothekssuchpfade an
  - Jeder syscall wird angezeigt, ebenso der return-Wert des Calls
  - Kann sich auf einen laufenden Prozess aufschalten
    - strace -p PID