Remote Debugging via Firewire

Verteidigung der Diplomarbeit

Julian Stecklina
jsteckli@os.inf.tu-dresden.de

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

19. Mai 2009

Overview

Gliederung

Keine Software ohne Bugs

Die lange Version

Wie schreibt man fehlerfreien Code? (Auswahl)

- Passende Programmiersprache
- Formale Verifikation
- Regression/Unit Testing
- ...

Keine Software ohne Bugs

Die lange Version

Wie schreibt man fehlerfreien Code? (Auswahl)

- Wir benutzen C/C++
- ~~Formale Verifikation~~
- Regression/Unit Testing
- Trial&Error mit einem Debugger

Keine Software ohne Bugs

Die kurze Version

There are two ways to write error-free programs;
only the third one works.

Alan J. Perlis

Der Ausgangspunkt

Situation: NOVA

- Microhypervisor in Entwicklung
- kein Debugger verfügbar

Was sind die Optionen?

`printf`

Option #1

`printf` Statements benutzen.

Pro

- extrem simpel

`printf`

Option #1

`printf` Statements benutzen.

Contra

- zeitaufwändig
- Einweglösung

In-Kernel Debugger

Option #2

Den kompletten Debugger im Kern selbst implementieren (z.B. Fiascos jdb oder FreeBSDs ddd).

Pro

- integriert sich vollständig in den Kern
- hoch–spezialisierte Tools (wie dtrace)

In-Kernel Debugger

Option #2

Den kompletten Debugger im Kern selbst implementieren (z.B. Fiascos jdb oder FreeBSDs ddd).

Contra

- nicht (einfach) wiederverwendbar
- keine klare Linie zwischen Kern und Debugger
- Bloat (jdb $= \frac{1}{3}$ von Fiascos Quellcode)
- hoher Entwicklungsaufwand

Remote Debugging

Option #3

Ein externer Debugger (GDB) kommuniziert mit einem „Stub” im Kern.

Pro

- mächtiger Debugger
- grafische Frontends
- eine gute Lösung für Embedded Systems
- geringer Entwicklungsaufwand

Remote Debugging

Option #3

Ein externer Debugger (GDB) kommuniziert mit einem „Stub” im Kern.

Contra

- wenig Integration in den Kern
- serielle Verbindung (*langsam*) oder TCP/IP (*N/A)*

VM mit Debugging Features

Option #4

Eine virtuelle Maschine mit Debugging-Erweiterungen.

Pro

- kein extra Testrechner nötig
- Zugang zu kritischen Codepfaden
- keine Unterstützung vom Kern nötig

VM mit Debugging Features

Option #4

Eine virtuelle Maschine mit Debugging-Erweiterungen.

Contra

- keine echte Hardware
- wenig Integration in den Kern

Zielsetzung

Wir wollen Remote Debugging mit GDB, aber

- mit minimalen Änderungen am Zielkern
- schnell auf andere Systeme anpassbar
- mit schneller Verbindung

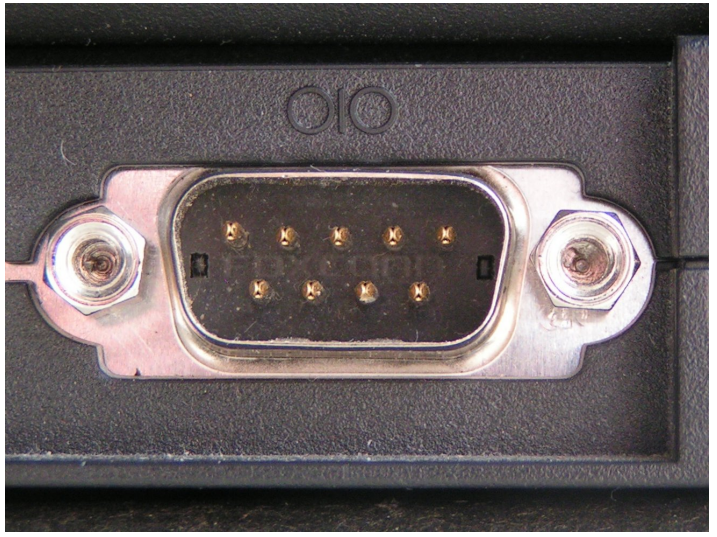
Kommen wir ohne Stub aus?

Gliederung

Die serielle Schnittstelle

Serielle Schnittstelle (RS-232)

- serielle Verbindung zwischen Geräten
- bis vor kurzem auf fast jedem PC



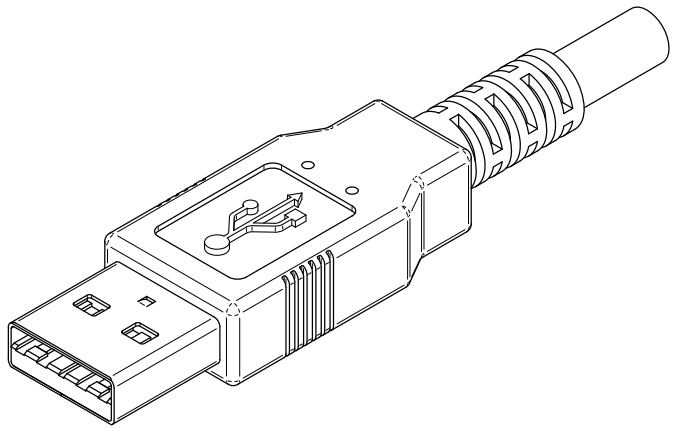
Eigenschaften

- Punkt–zu–Punkt Verbindung
- auf PCs üblicherweise $\leq$ 115.200 Bit/s Übertragungsgeschwindigkeit
- einfach programmierbar

USB

Universal Serial Bus

- serieller Bus zwischen PC und Peripherie
- ersetzt diverse „Legacy” Anschlüsse (RS-232, LPT, Game port)
- $\leq$ 480 MBit/s (USB 2)



Eigenschaften

- Stern-Topologie
- strikte Master–Slave–Philosophie
- benötigt Softwarestack zum Betrieb

USB Debug Port

USB Debug Port

- optionales Feature des EHCI Standards
- bietet bidirektionalen Kanal zwischen 2 PCs
- kein USB Softwarestack benötigt

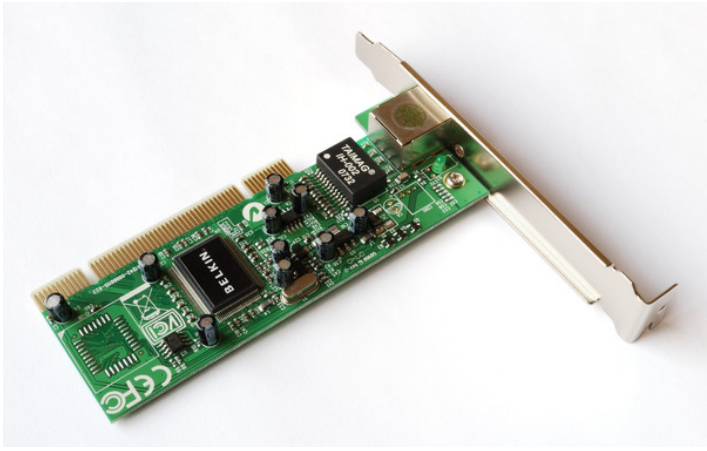
Nachteile

- nicht von jedem Controller implementiert
- praktisch schnelle serielle Schnittstelle (0,5 MBit/s)
- extra „Debug Device” benötigt (83 $)

PCI

PCI

- Bus für PC Erweiterungskarten
- $\sim$ 1 GBit/s (32-Bit 33 MHz)



Eigenschaften

- drei 32-Bit/64-Bit Adressräume (Configuration, Memory, I/O)
- Zugriff via `read`/`write` Transaktionen
- Controller bildet Memory Adressraum auf RAM ab

Firewire

Firewire (IEEE 1394)

- serieller Bus (400-3200 MBit/s)
- standardisierter Controller (OHCI)

Eigenschaften

- Baum-Topologie
- Peer-to-Peer Kommunikation
- alle Knoten (Nodes) in einem 64-Bit-Adressraum dargestellt

Bus Transaktionen

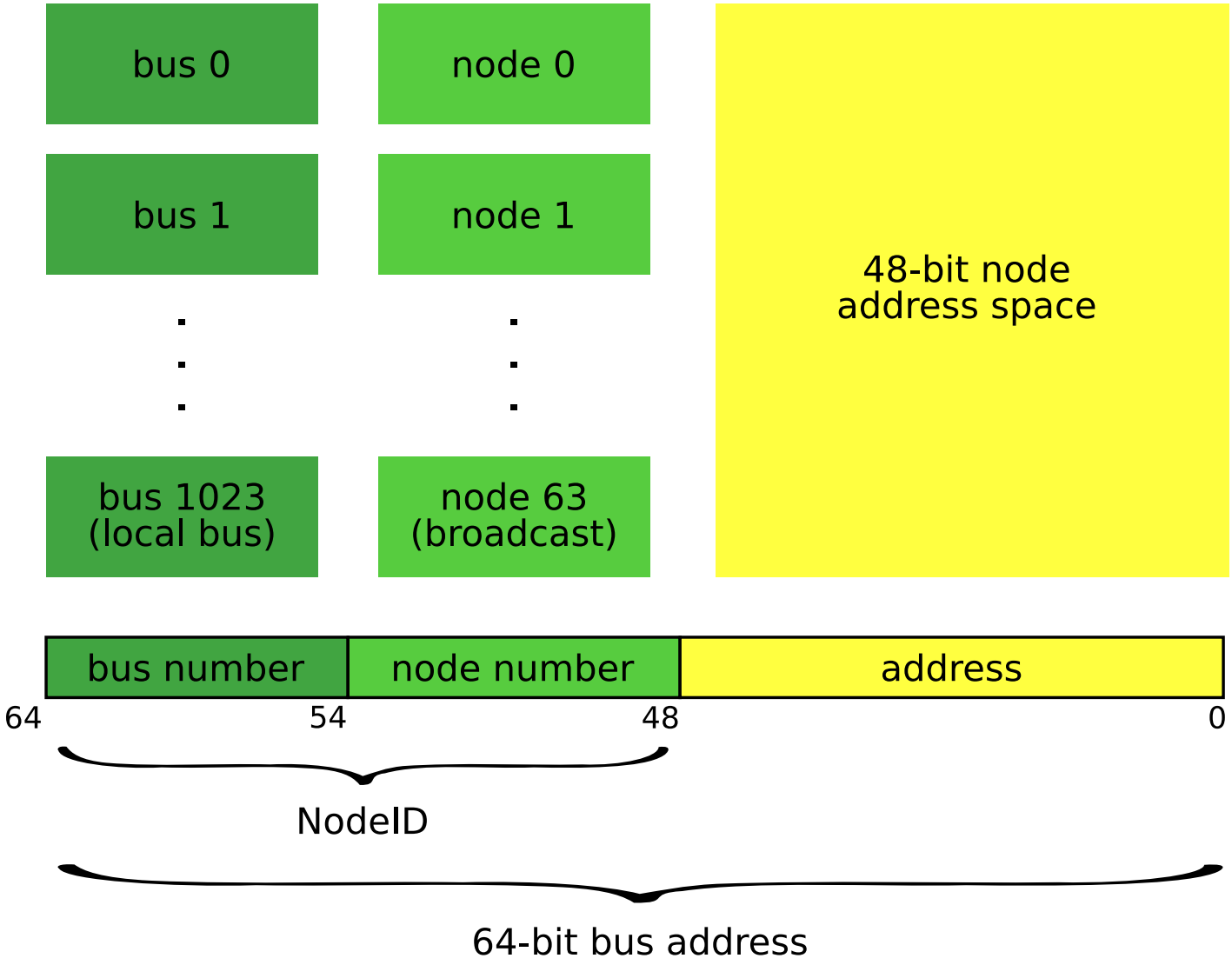
Asynchron

- **read**, **write**
- **lock** (Fetch–Add/Compare–Swap)
- zuverlässig (Bestätigung und Prüfsumme)

Isochron

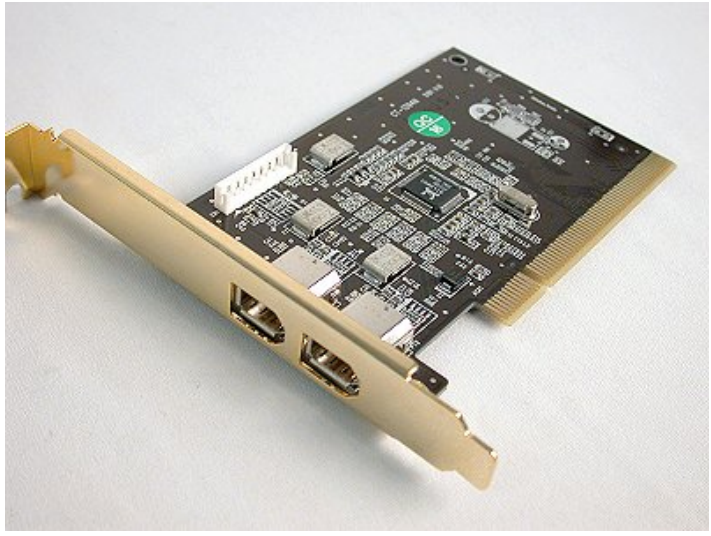
- Kanäle mit garantierter Bandbreite

Firewire Adressraum



Open Host Controller Interface

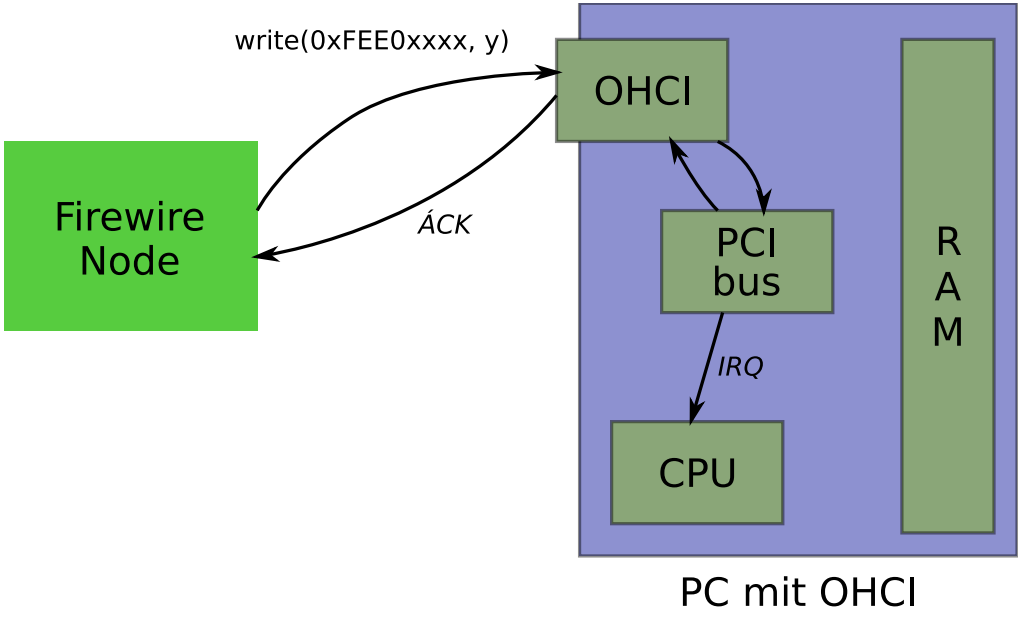
- standardisierte (PCI) Schnittstelle
- von praktisch allen Controllern implementiert



Physical DMA

- Firewire Transaktionen werden auf PCI Bus weitergereicht
- „entblößt” RAM auf dem Firewire Bus
- benutzt für Speicher-basierte Kommunikation zwischen Geräten

Interrupts über Firewire



PCI Message-Signaled Interrupts (MSIs)

- spezielle `write` Transaktion auf PCI Bus
- Art und Ziel des Interrupts in Adresse und Daten kodiert
- vom Chipsatz an Ziel-CPU weitergeleitet
- über Firewire möglich!

Gliederung

Ein Schritt Zurück

Ein Remote Debugger muss

- Zustand (Register, Speicher) auslesen/verändern
- Ausführung kontrollieren

Wieviel Unterstützung vom Zielbetriebssystem nötig?
Abhängig vom Medium!

Das Optimale Medium

	RS-232
Zugriff auf Register	-
Zugriff auf Speicher	-
Ausführungskontrolle	-
Verfügbarkeit	✓

Das Optimale Medium

	RS-232	Debug Port
Zugriff auf Register	-	-
Zugriff auf Speicher	-	-
Ausführungskontrolle	-	-
Verfügbarkeit	✓	-

Das Optimale Medium

	RS-232	Debug Port	Firewire
Zugriff auf Register	-	-	-
Zugriff auf Speicher	-	-	✓
Ausführungskontrolle	-	-	**Interrupt**
Verfügbarkeit	✓	-	✓

Firewire

- keine spezielle Hardware nötig
- Off–The–Shelf Komponenten

Das Optimale Medium

	RS-232	**Debug Port**	**Firewire**	**PCI**
Zugriff auf Register	-	-	-	-
Zugriff auf Speicher	-	-	✓	✓
Ausführungskontrolle	-	-	**Interrupt**	Interrupt
Verfügbarkeit	✓	-	✓	-

Firewire

- keine spezielle Hardware nötig
- Off–The–Shelf Komponenten

Das Optimale Medium

	RS-232	**Debug Port**	**Firewire**	**PCI**
Zugriff auf Register	-	-	-	-
Zugriff auf Speicher	-	-	✓	✓
Ausführungskontrolle	-	-	**Interrupt**	Interrupt
Verfügbarkeit	✓	-	✓	-

Firewire

- keine spezielle Hardware nötig
- Off–The–Shelf Komponenten

Stub muss nur fehlende Funktionalität bieten.

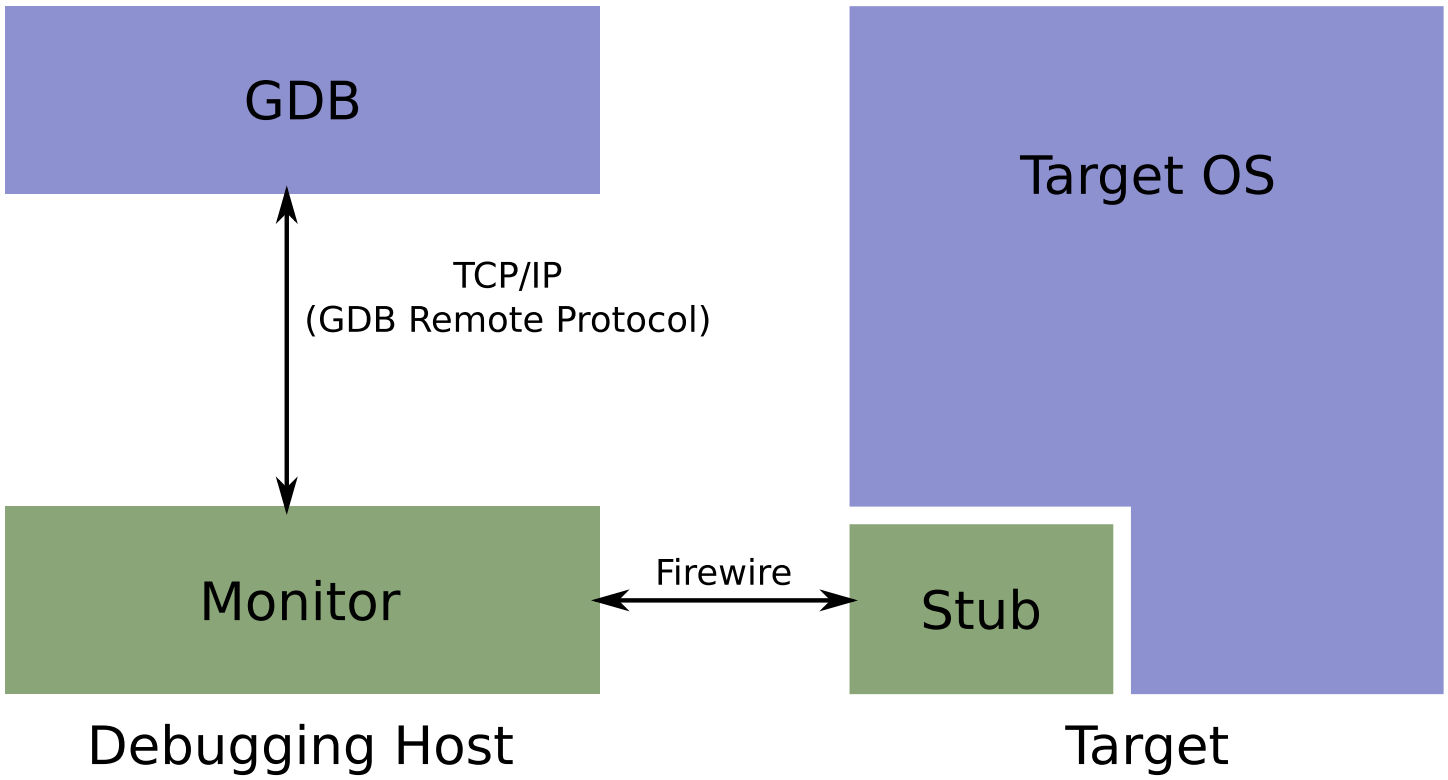
GDB

- bekannte Benutzerschittstelle
- Disassembling
- Debug-Information auswerten
- guter C++–Support
- ...

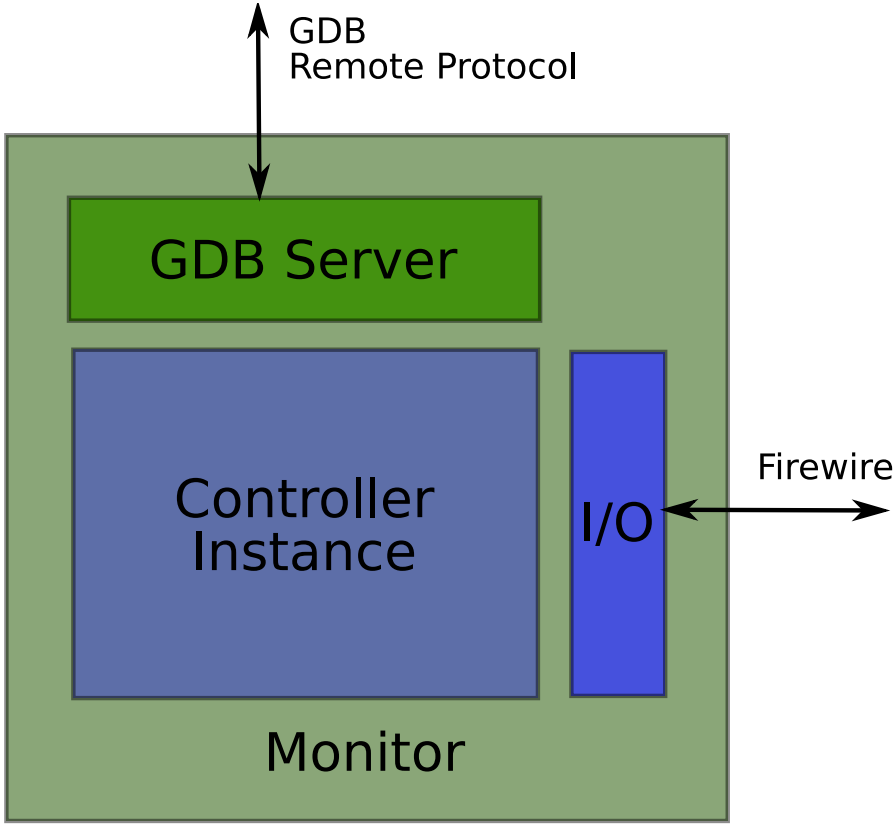
GDB unverändert verwenden.
Neue Funktionalität in *Monitor* implementieren.

Überblick

Vogelperspektive



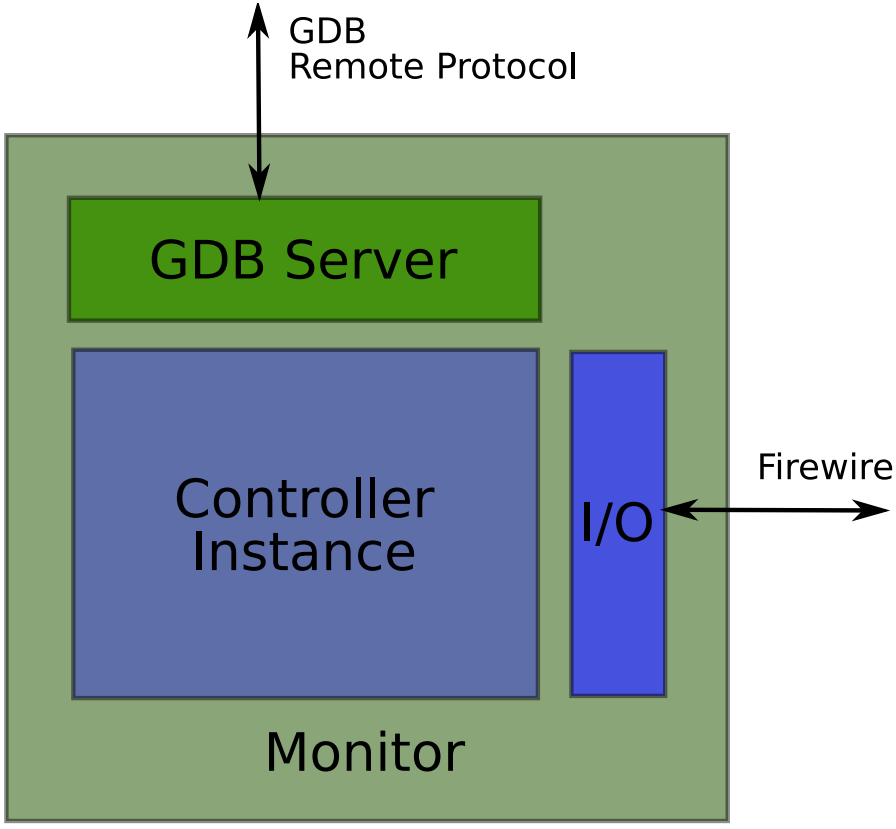
Der Monitor



GDB Server

- implementiert GDB Protokoll
- GDB Anfragen auf Controllermethoden abgebildet

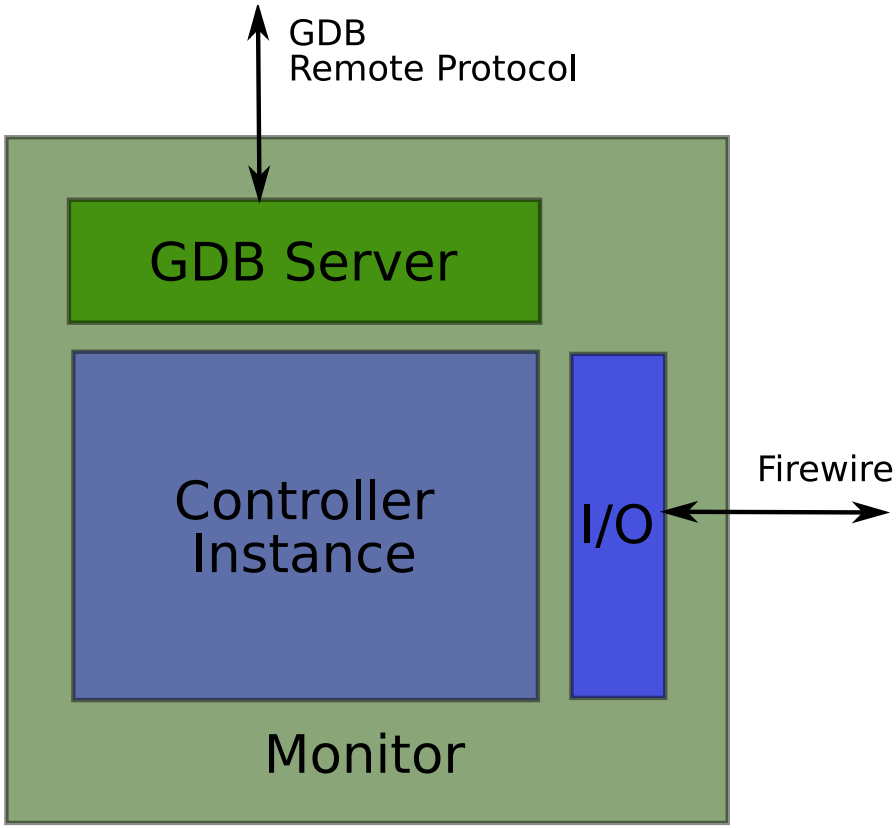
Der Monitor



Controller

- fragt Status vom Ziel ab
- löst virtuelle Adressen auf
- abstrahiert Kernelstubinterface

Der Monitor



I/O

- Zugriff auf phys. Speicher
- Interrupts

Der Monitor (2)

Modular

- andere Debugger
- andere Kommunikationsmedien
- andere CPU-Architekturen

Nicht Zielkernel–spezifisch!

Was tut der Stub?

Monitor kann

- Interrupts senden
- auf physischen Speicher zugreifen

Der Stub

- wickelt Interrupts ab
- entblößt CPU–Register

Kontexte im Zielkern

Kontext

`run`	normale Ausführung
`interrupt`	Abarbeitung eines Monitorinterrupts (nicht unterbrechbar)
`halt`	Ziel angehalten

Kontext wechselt bei

- Debugereignissen (Breakpoints) → `halt`
- Monitorinterrupts → `interrupt`

Kommunikation: Command Space

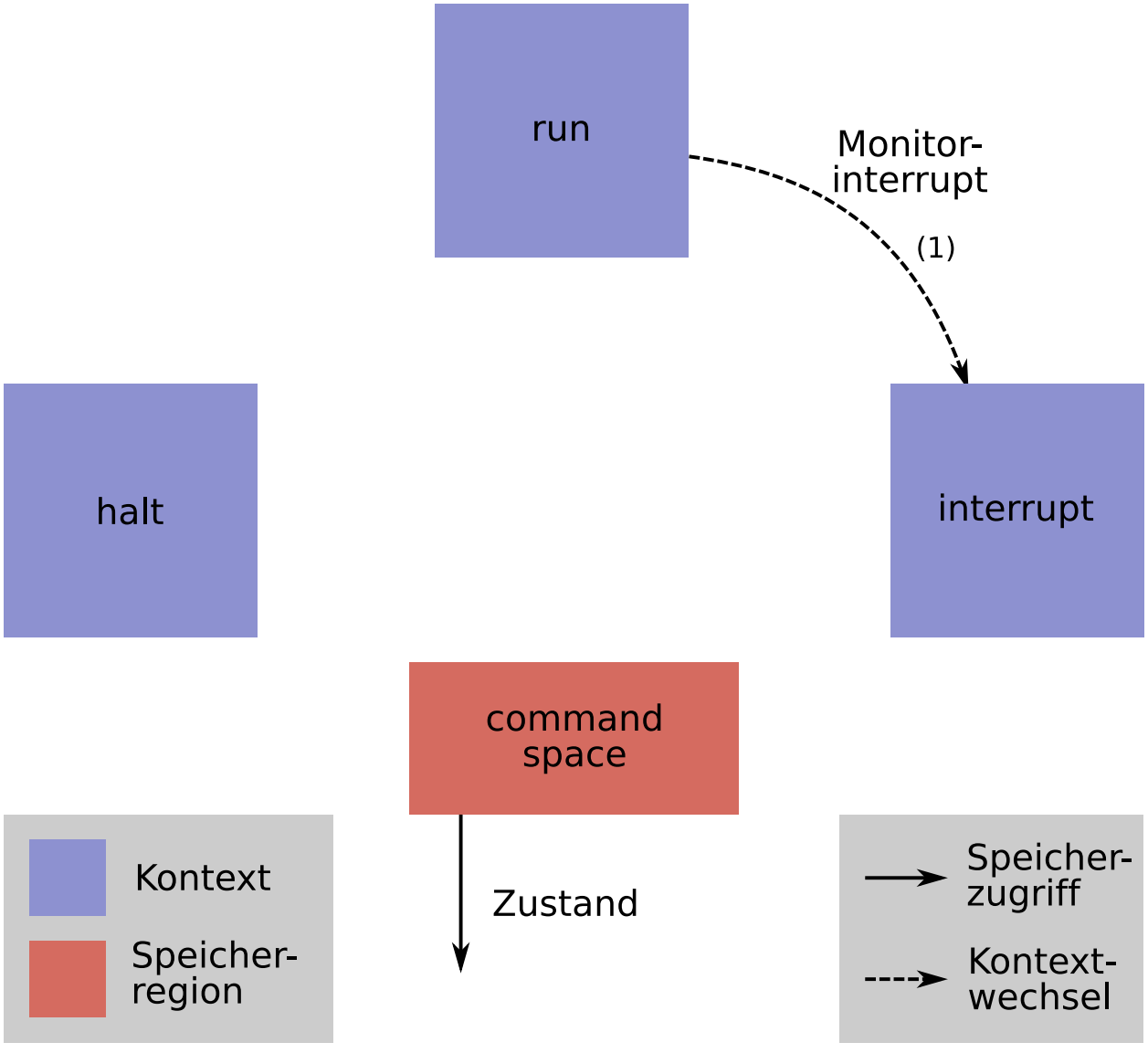
Zustand des Ziels

- aktualisiert im `interrupt` Kontext
- enthält CPU–Zustand bei Verlassen des `run` Kontexts
- vom Monitor veränderbar

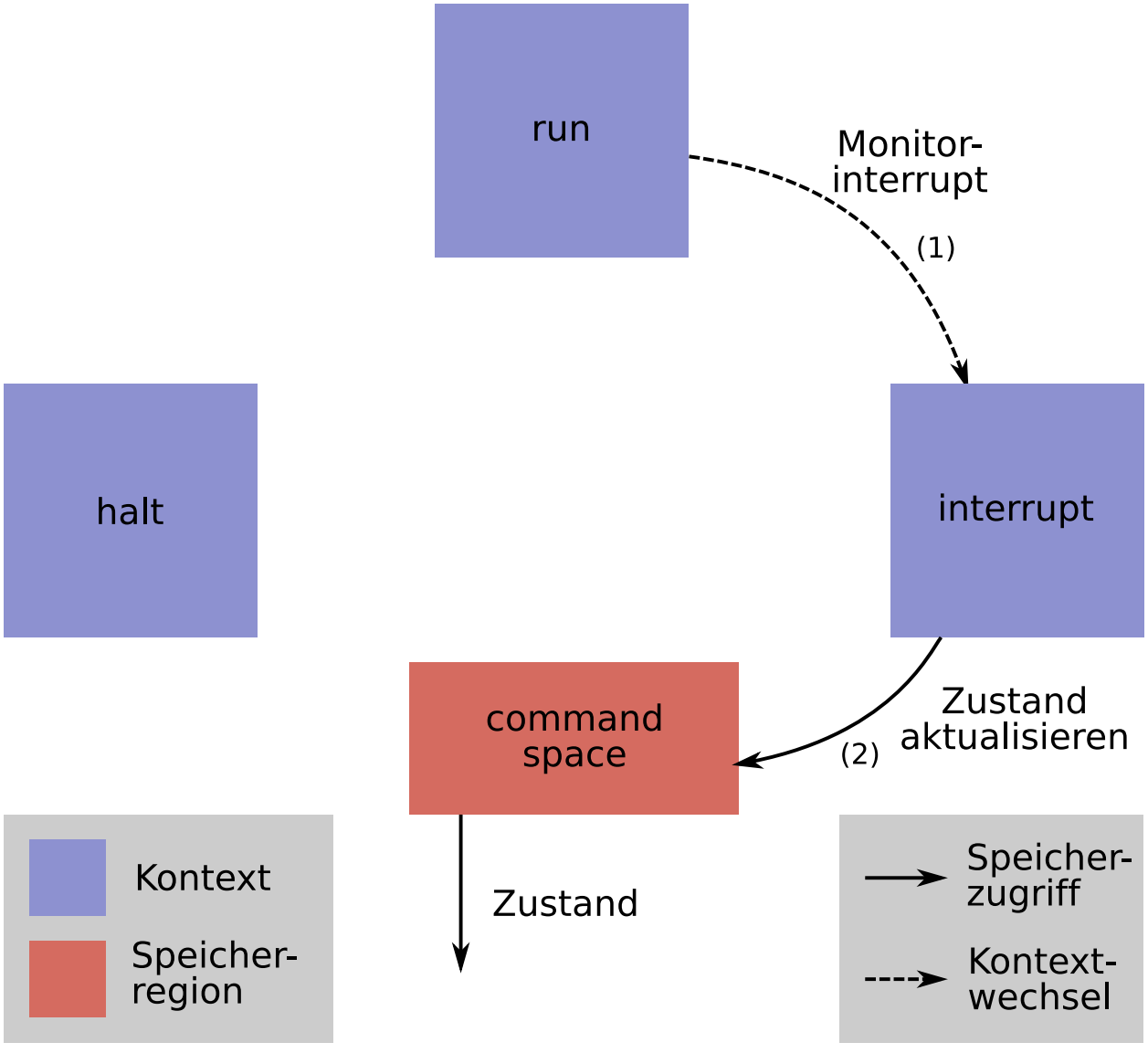
Aufgaben für Stub

- vom Monitor geschrieben
- „halt” → `halt`
- „run” → `run`

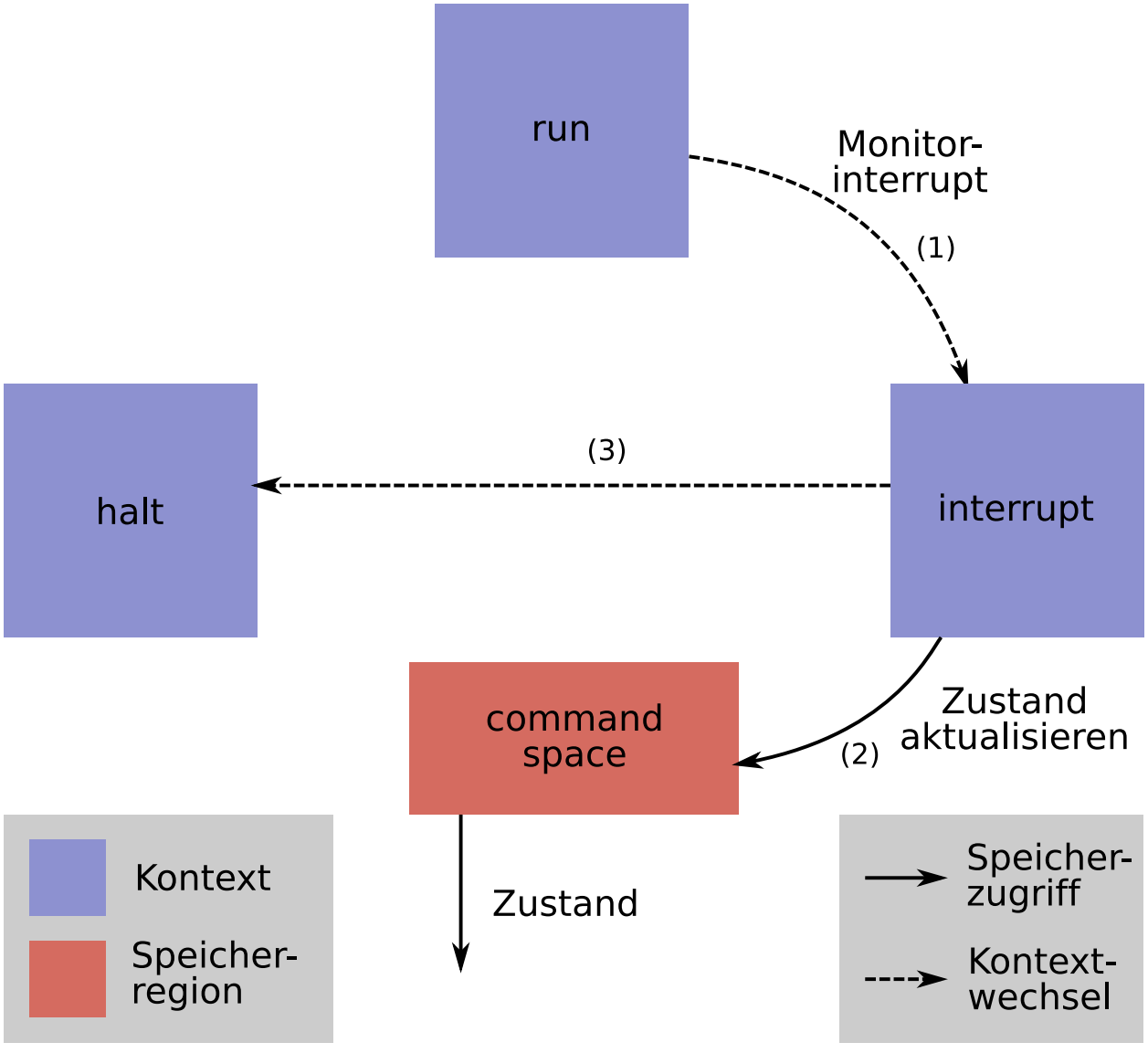
Ausführung anhalten



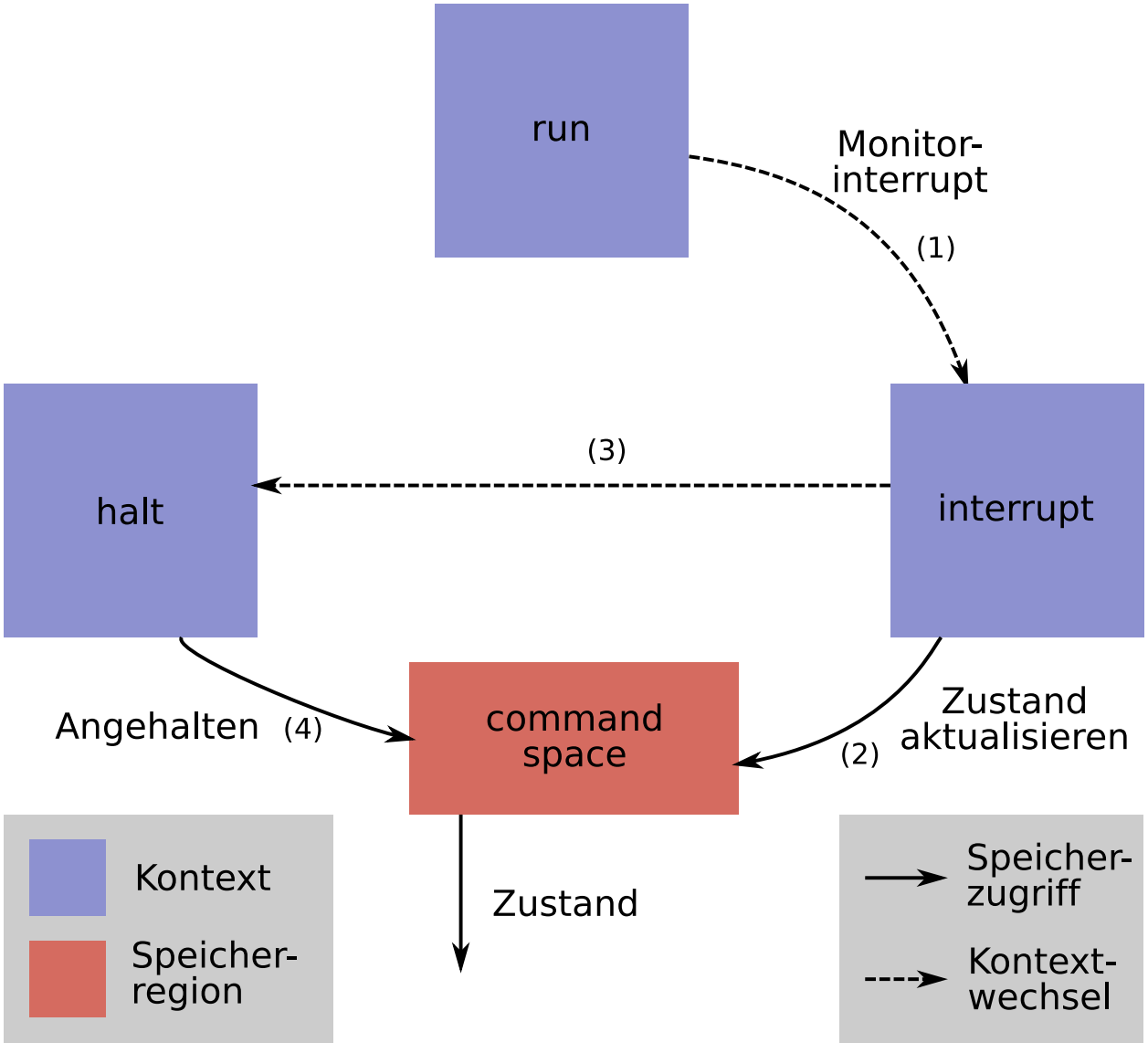
Ausführung anhalten



Ausführung anhalten



Ausführung anhalten



Breakpoints

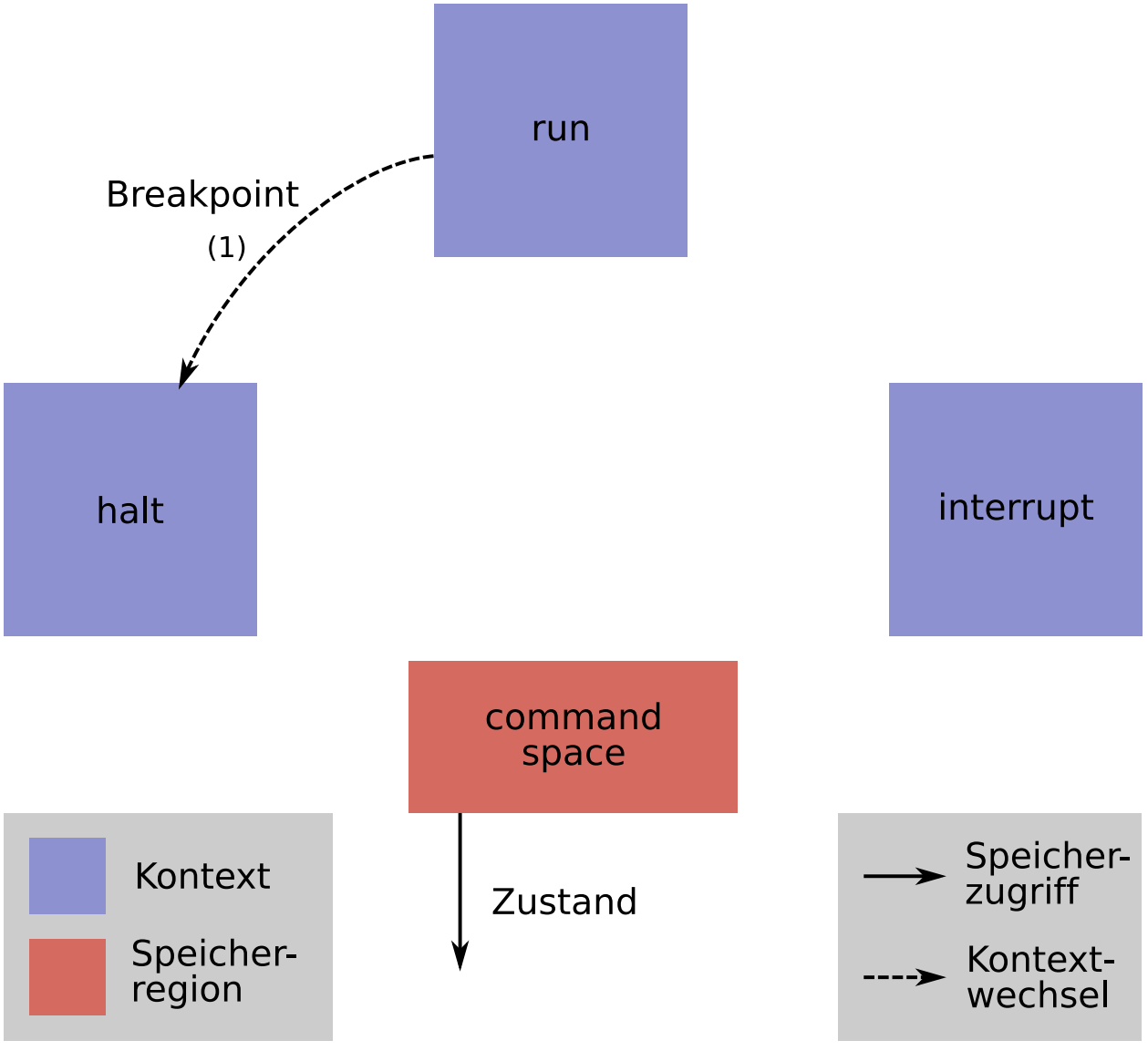
Software–Breakpoints

- Instruktion mit INT3 überschreiben
- direkt vom Monitor möglich
- kein Support im Stub nötig

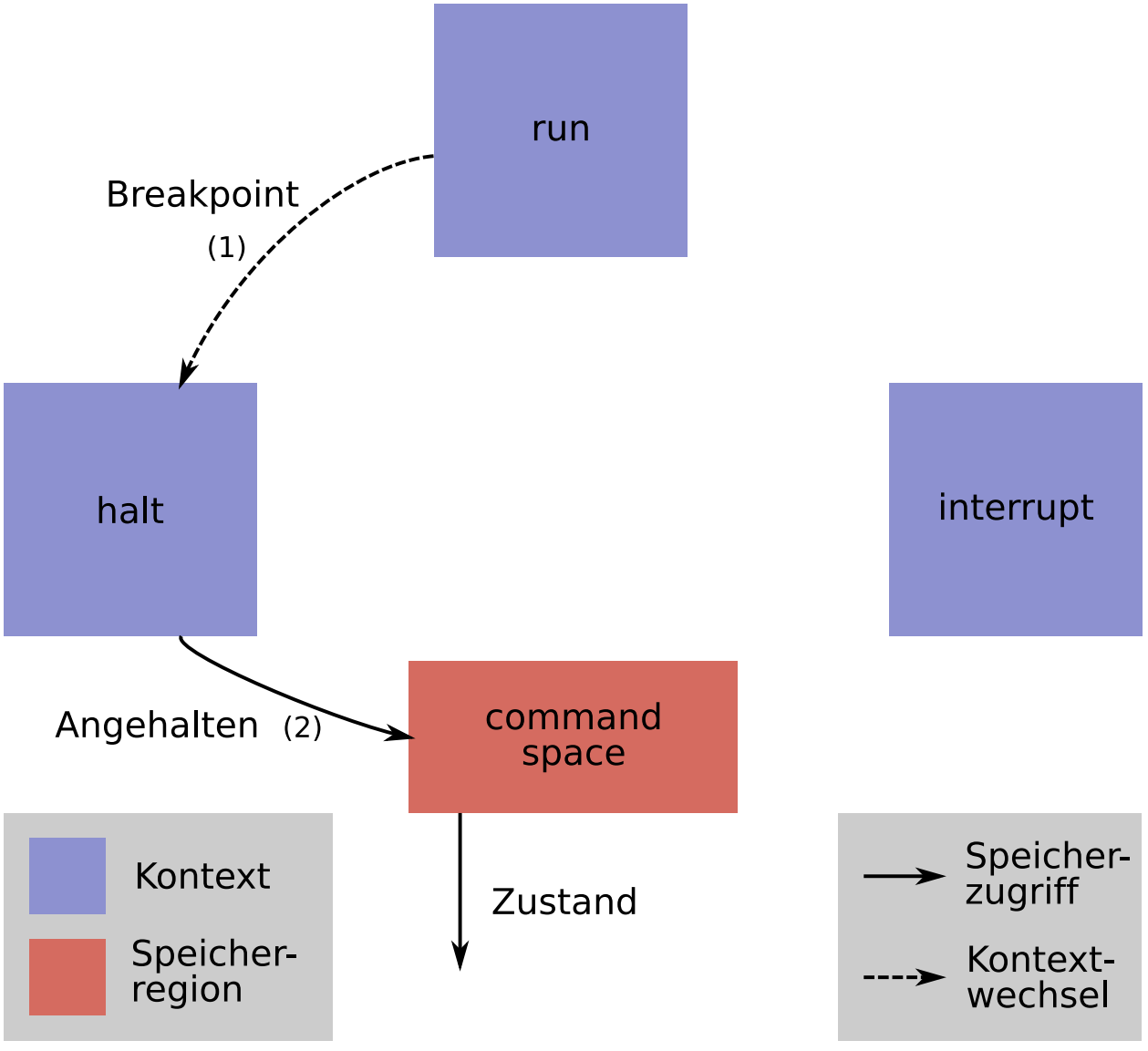
Hardware–Breakpoints

- werden über spezielle Debugregister gesetzt
- Monitor schreibt neuen Inhalt in Command Space
- Stub kopiert Inhalt in Debugregister

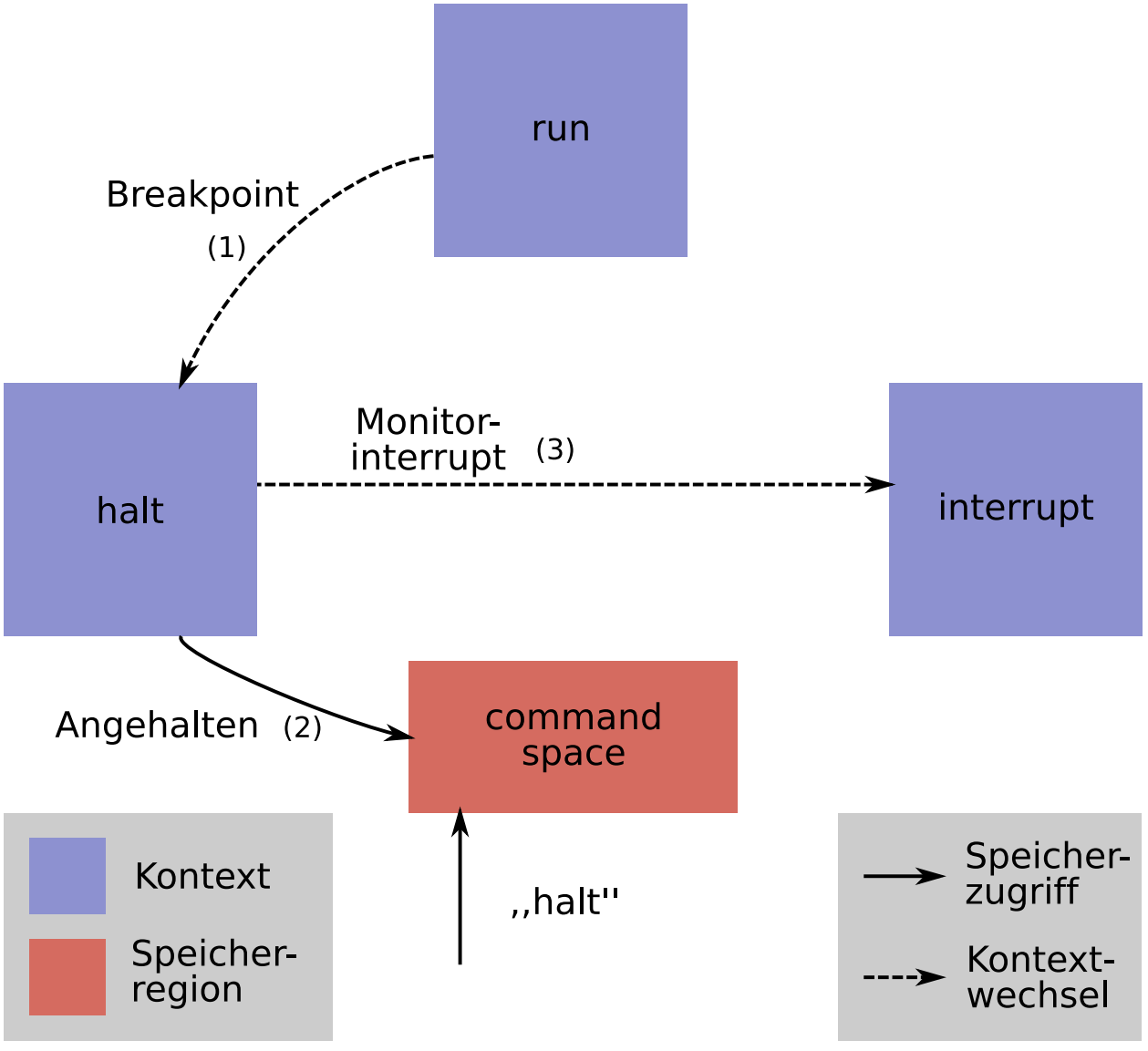
Breakpoints behandeln



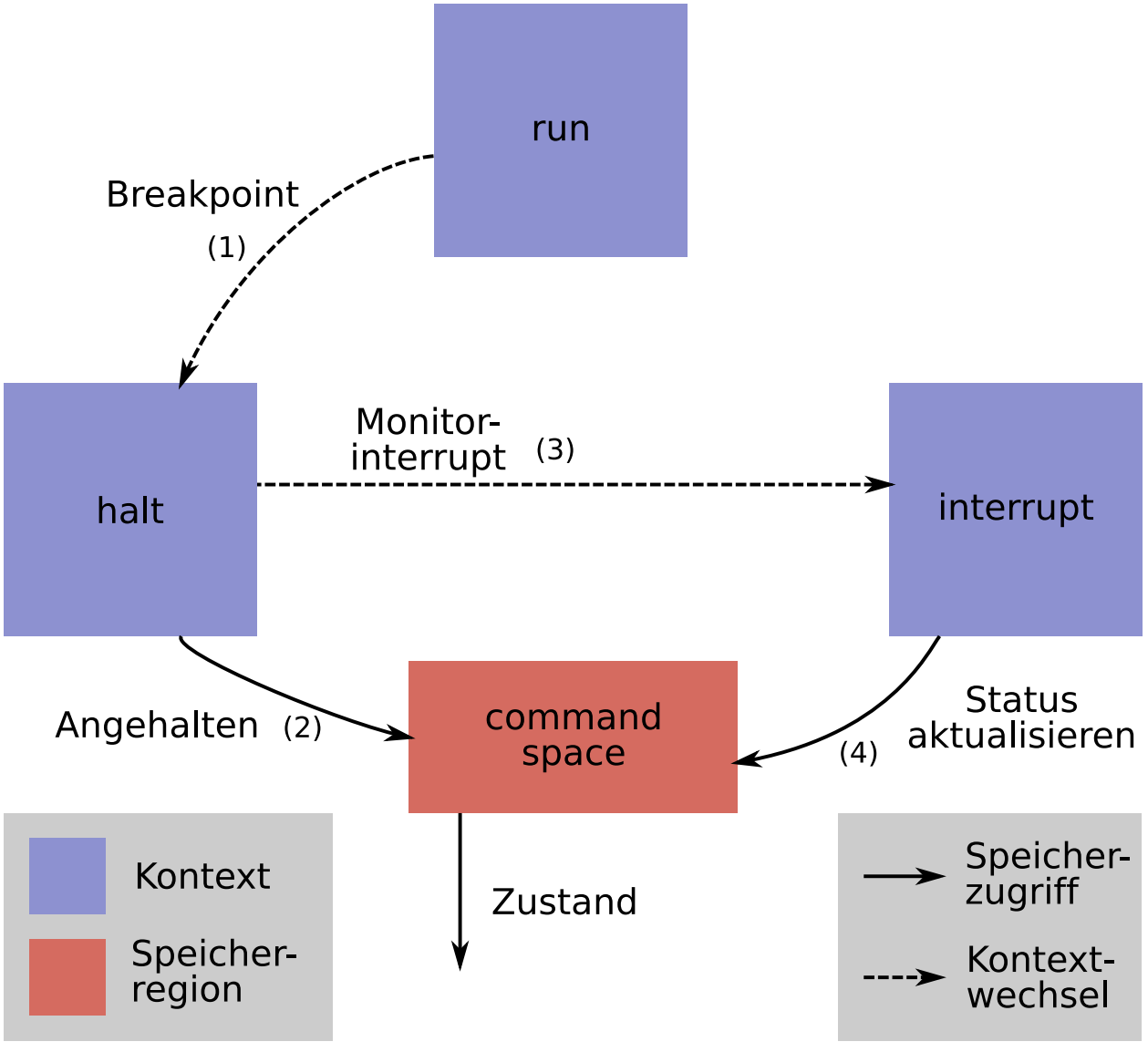
Breakpoints behandeln



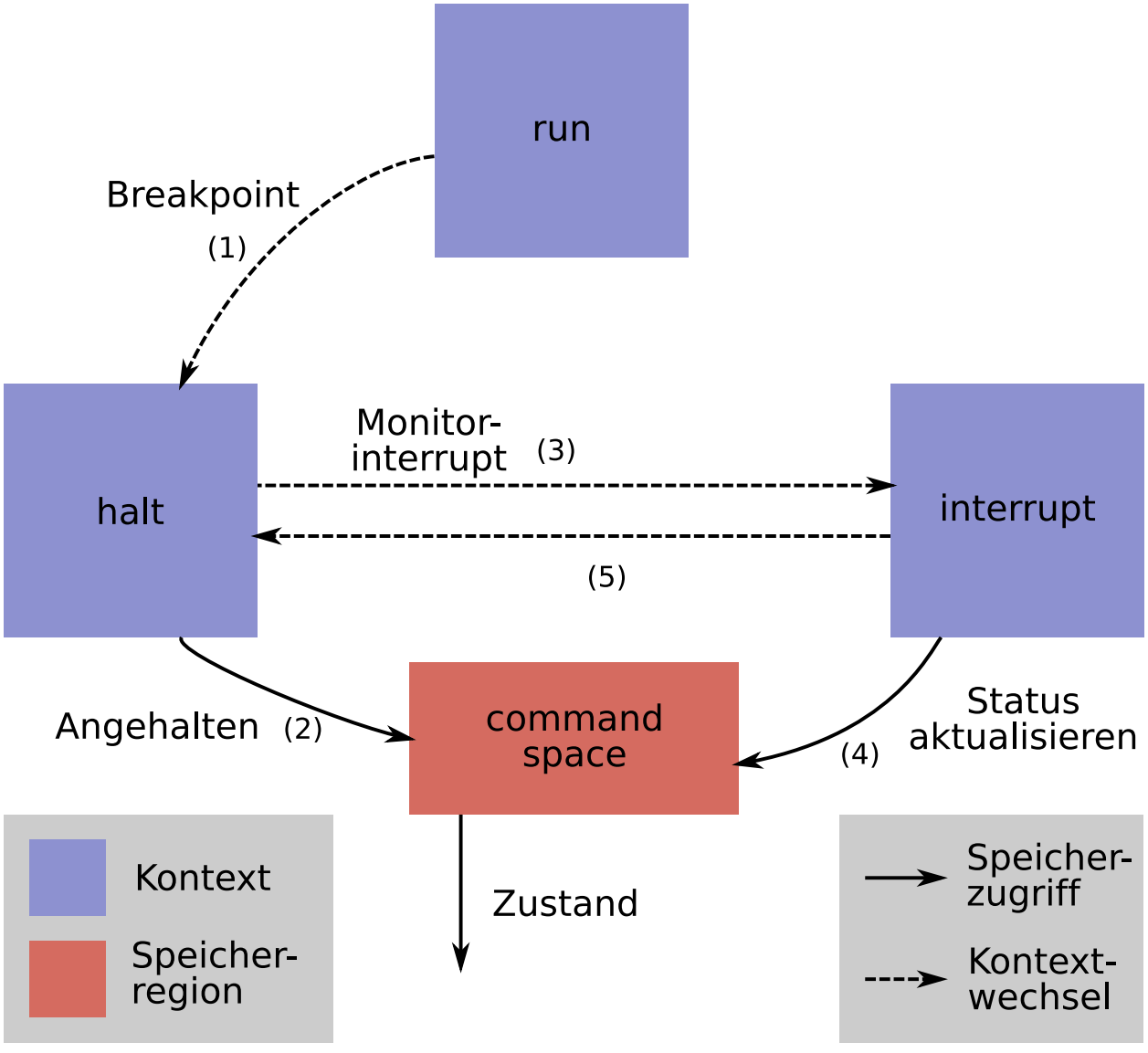
Breakpoints behandeln



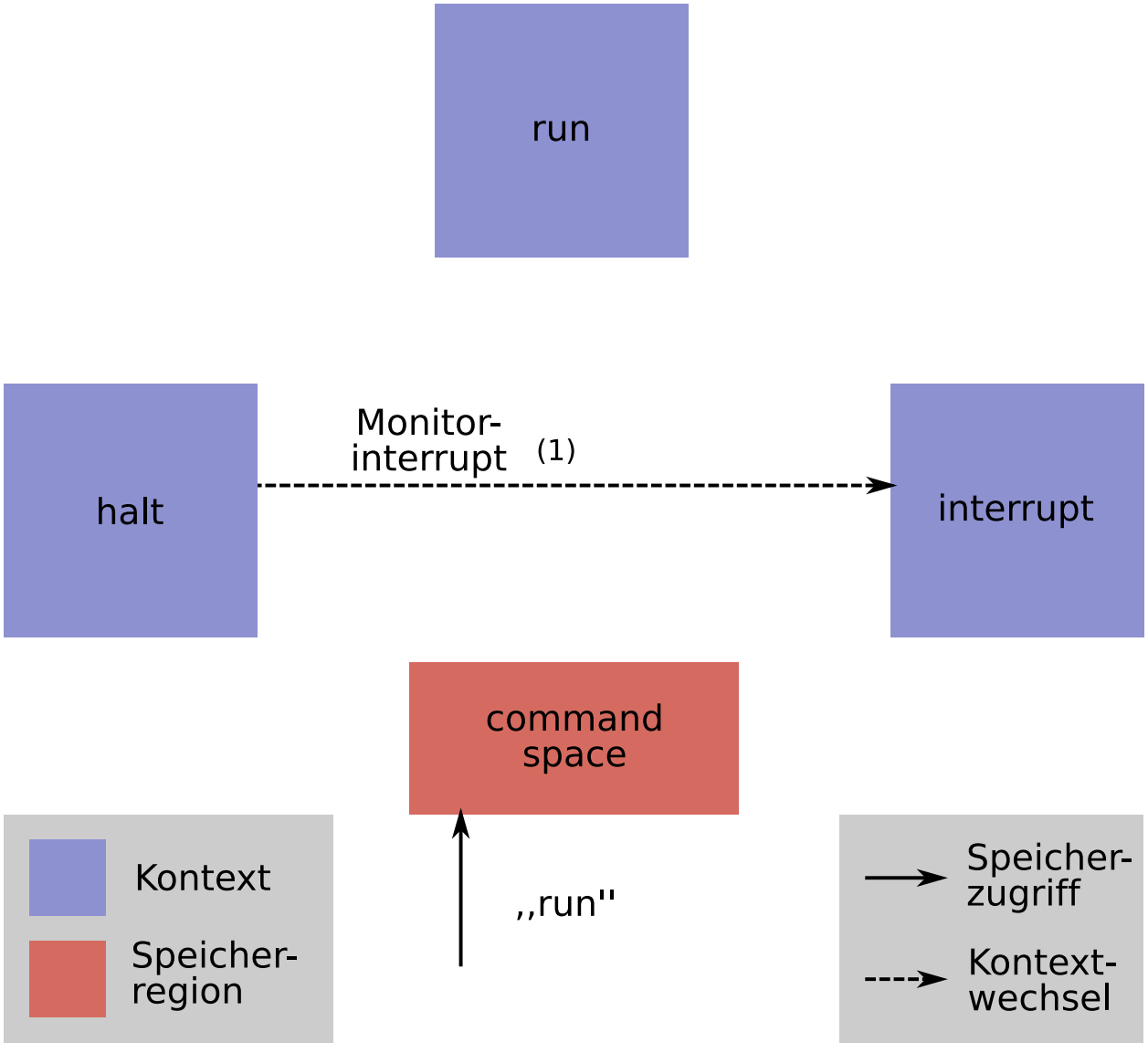
Breakpoints behandeln



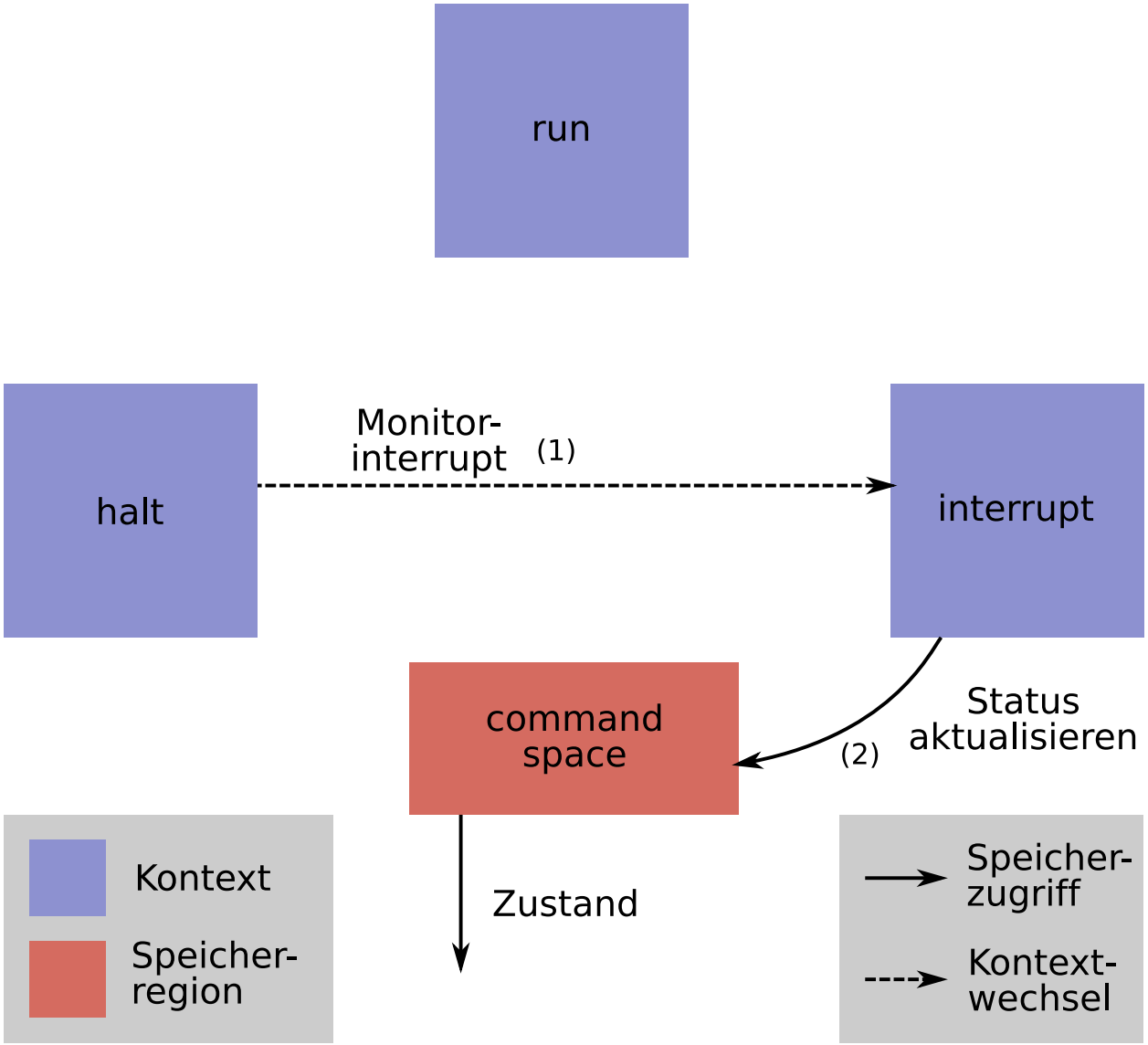
Breakpoints behandeln



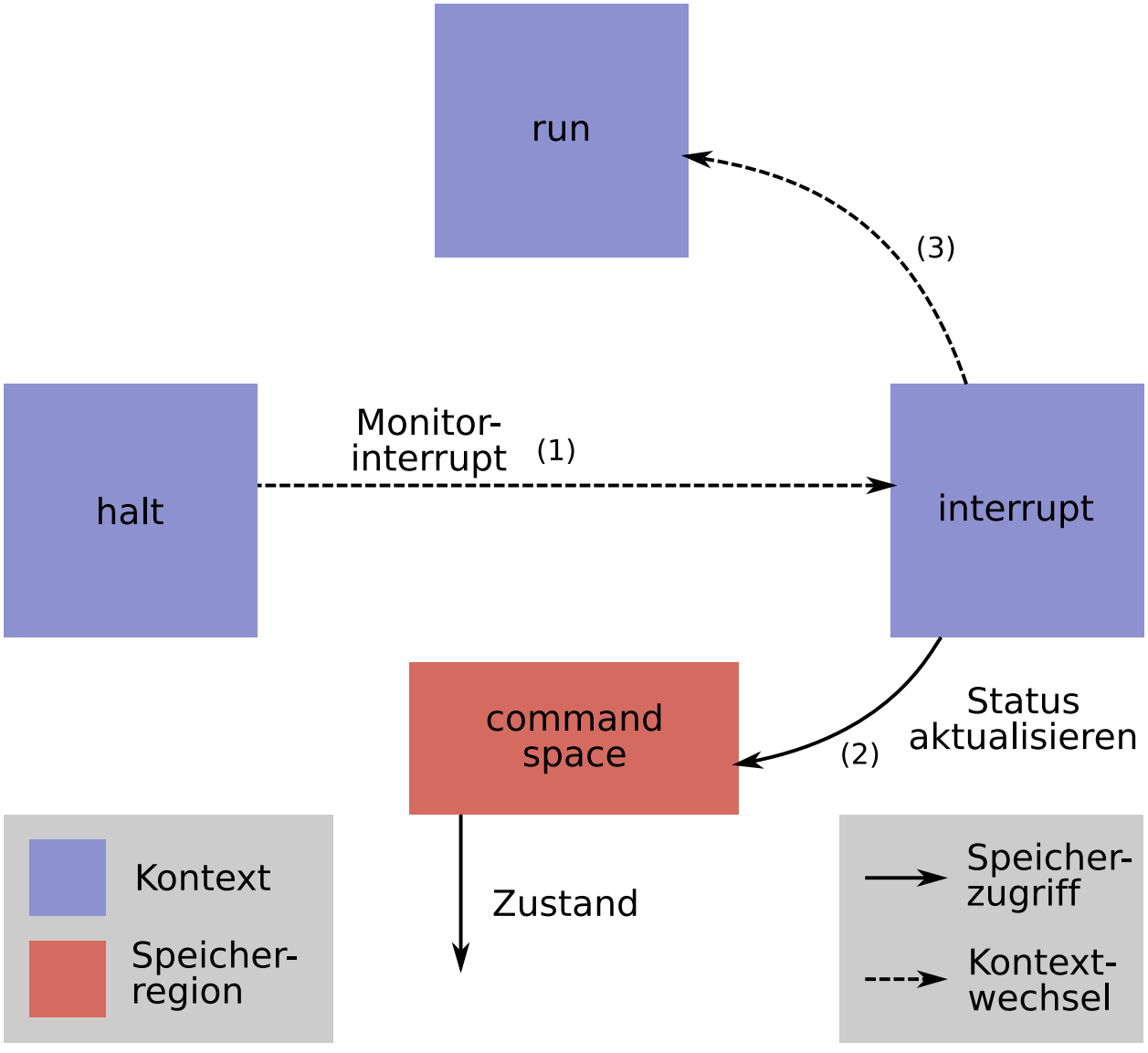
Ausführung fortsetzen



Ausführung fortsetzen



Ausführung fortsetzen



Gliederung

OHCI

Einfacher Stub

- keine Firewire–Unterstützung im Stub (gewollt)
- OHCI programmieren, bevor NOVA bootet

Annahmen

- Controller wird von NOVA in Ruhe gelassen
- NOVA schränkt den Controller nicht ein (z.B. IO-MMU)
- keine Busresets

OHCI (2)

Morbo

- läd vor NOVA
- initialisiert OHCI

Beispiel: GRUB-Config

```
title NOVA + OHCI Init
kernel /morbo
module /nova
...
```

NOVA #1

Stub als Teil des Kerns

Stub: $\sim$ 150 LOC

- Handler für Monitorinterrupt
- Handler für Breakpoints
- Setup

- NOVA bleibt weitestgehend unberührt
- keine Laufzeitbeeinträchtigung, wenn unbenutzt

Code Injection?

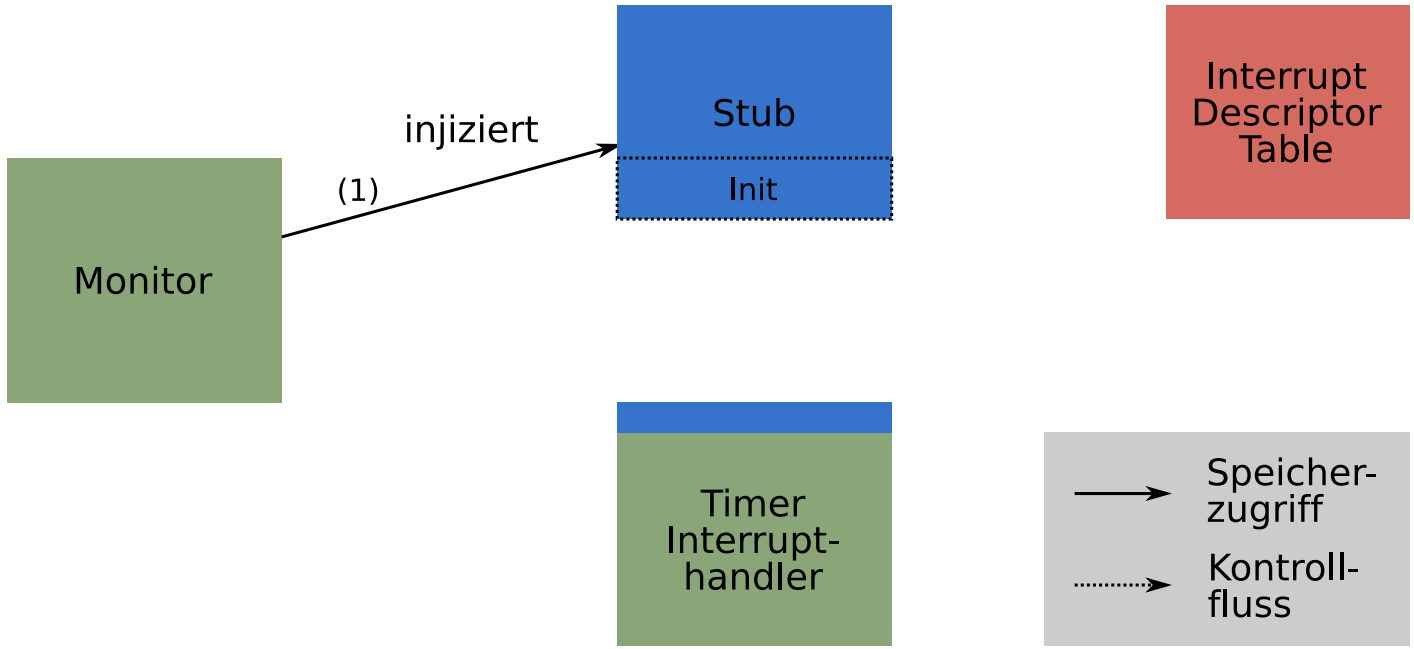
Überlegungen

- Monitor kann Code im Ziel zur Laufzeit ändern
- Stub zur Laufzeit einbringen?

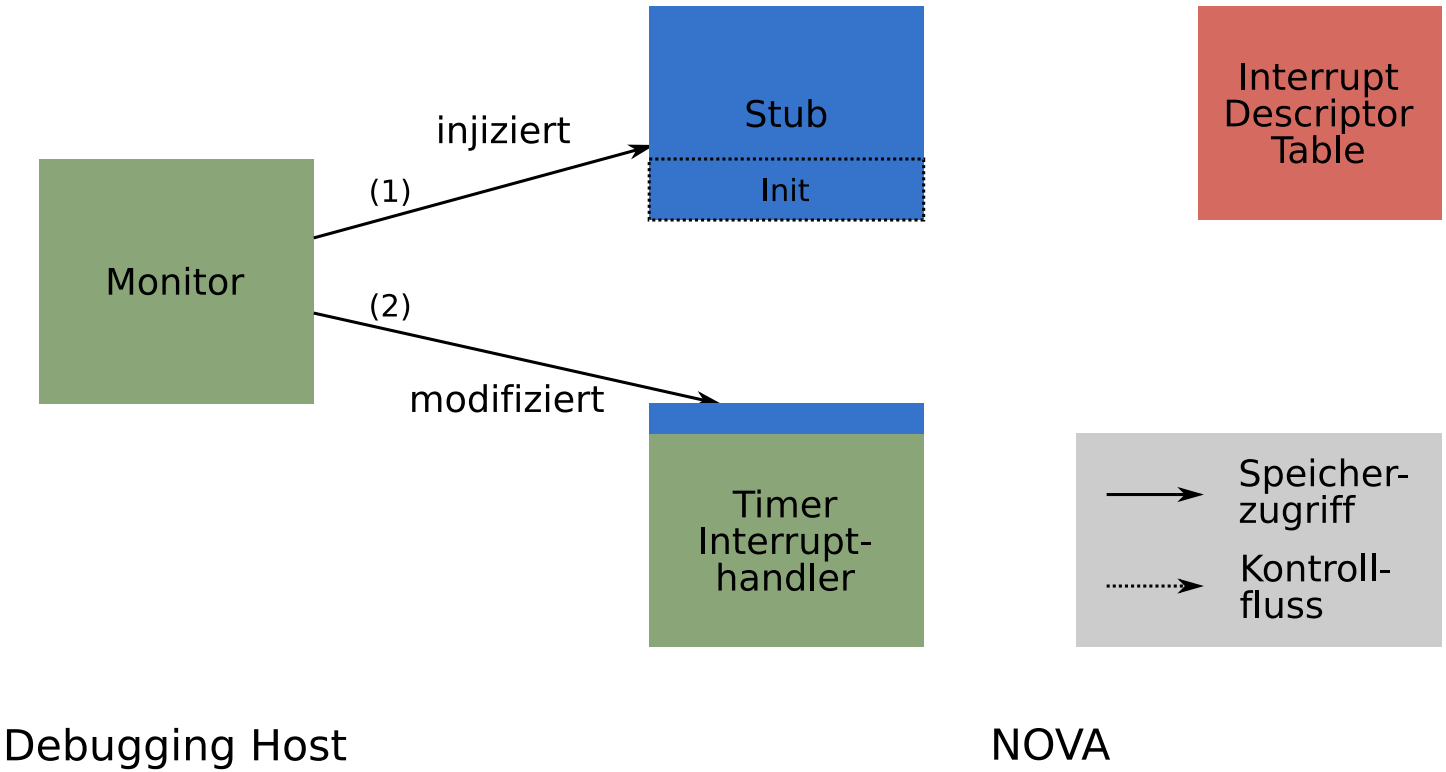
Voraussetzungen

- freier Speicher im Kernadressraum
- freie Einträge in GDT

Hijacking NOVA



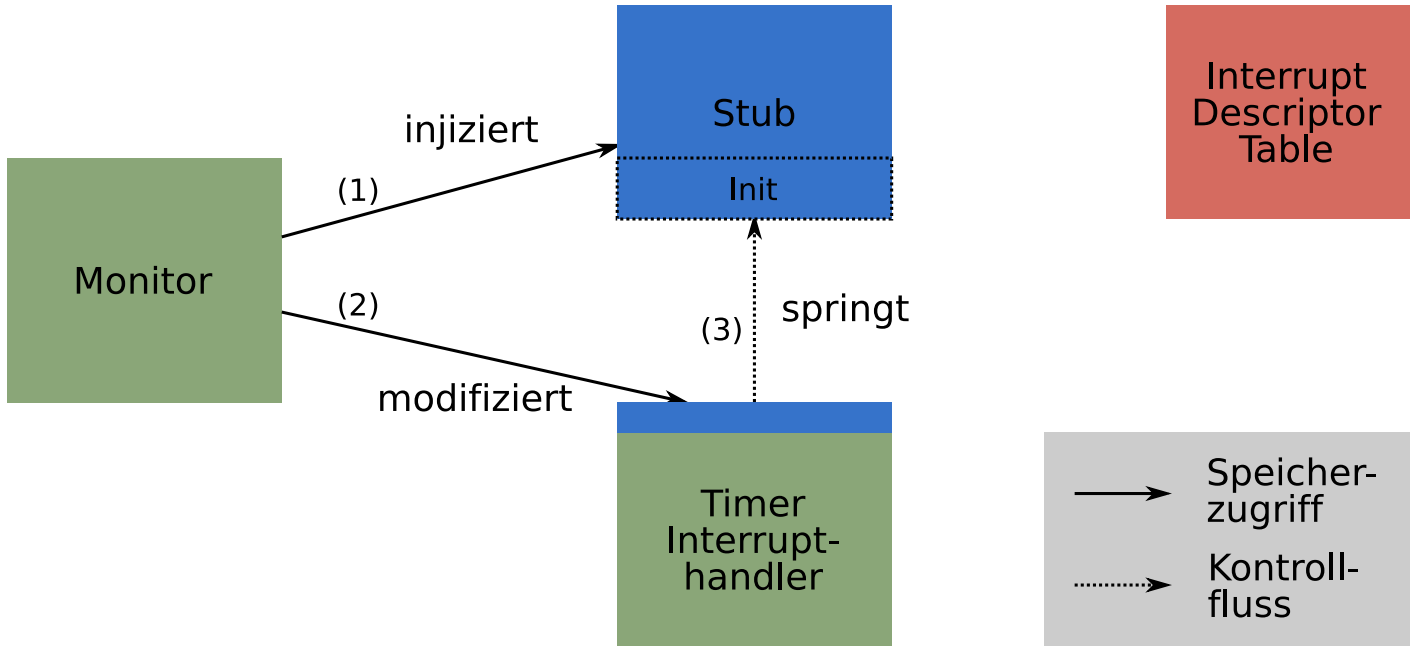
Hijacking NOVA



Debugging Host

NOVA

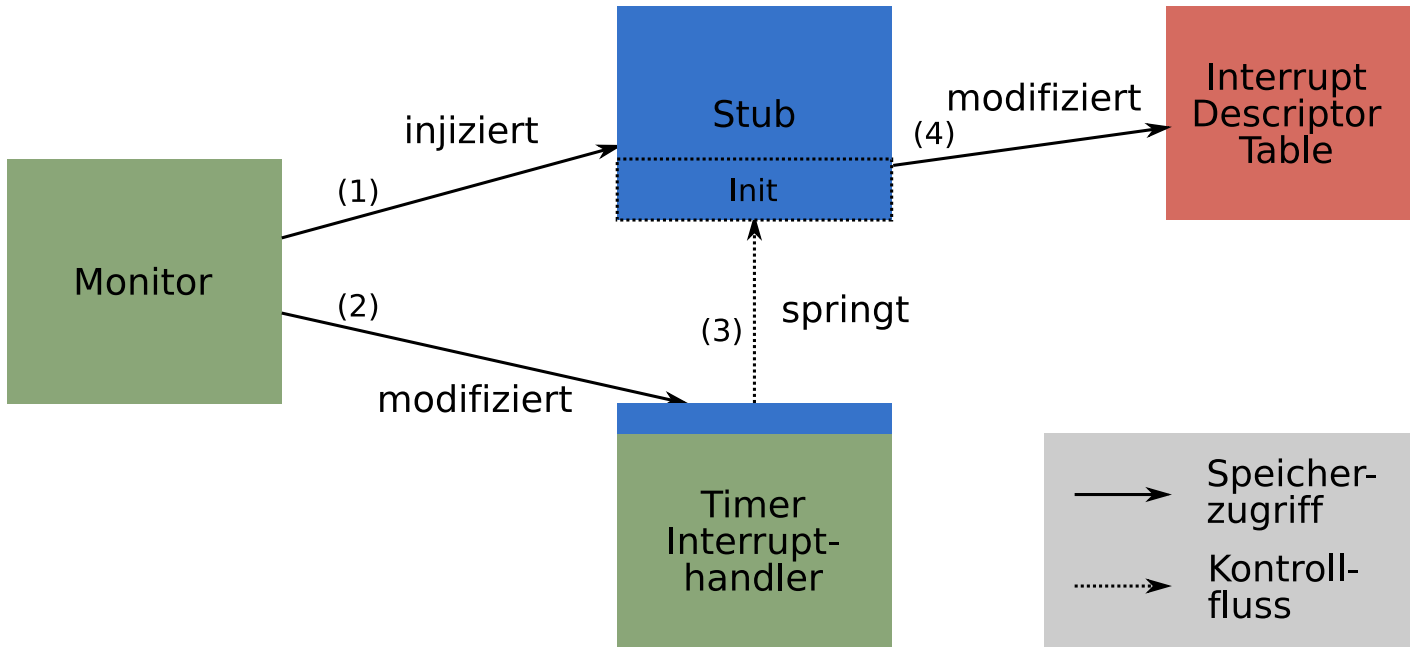
Hijacking NOVA



Debugging Host

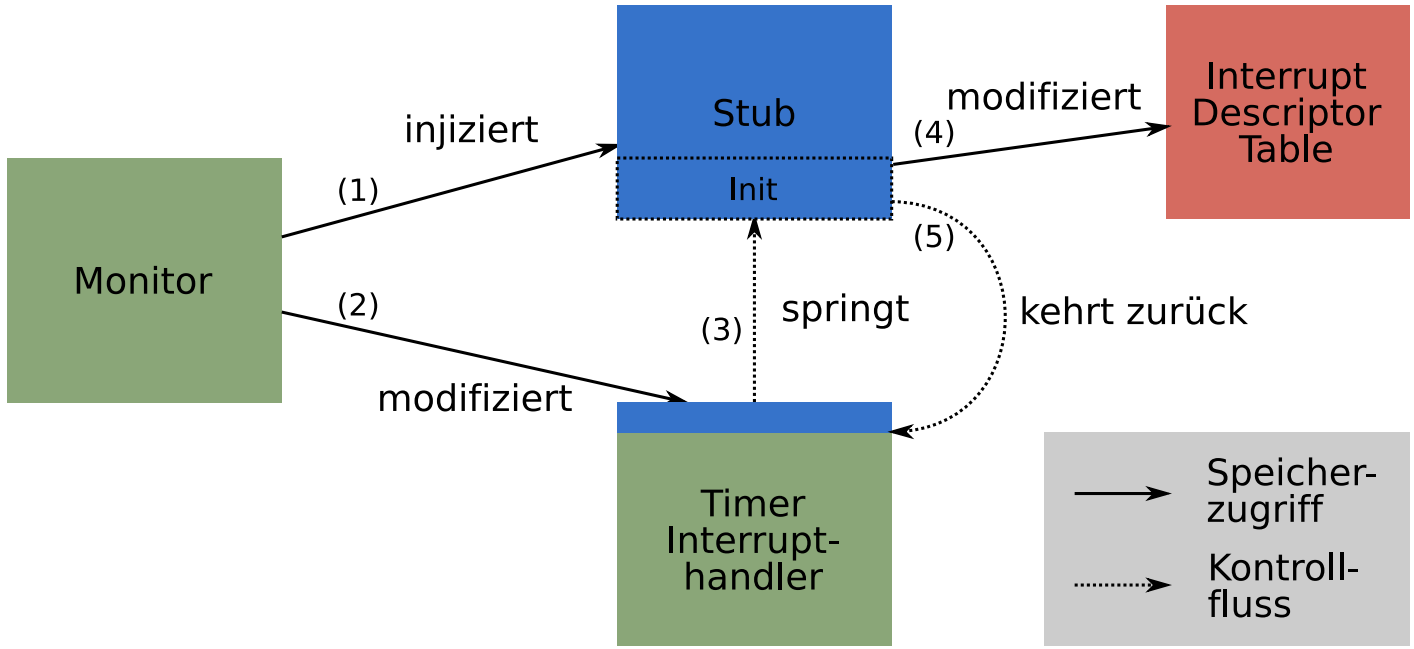
NOVA

Hijacking NOVA



Debugging Host NOVA

Hijacking NOVA



Debugging Host NOVA

NOVA #2

Injektion des Stubs zur Laufzeit

- Stub benötigt Adressen von Kernstrukturen
- halbautomatisch per Skript erzeugt

Aufwand

NOVA 6 LOC (Linkerscript, GDT Konstanten)

Stub 246 LOC Assembler
(155 LOC Initialisierung, 7 LOC NOVA–spezifisch)

Linux

Machbarkeitsstudie

- ausgehend von NOVA Stub #1
- insgesamt etwa eine knappe Woche Aufwand (busybox . . .)

Probleme

- Low–Level–Code über unzählige Dateien verteilt
- Injektion nicht praktisch

Es funktioniert!

Features

- Kernel Debugging auf Quellcode–Ebene
- Breakpoints
- Watchpoints
- Single-Stepping

Probleme

- GDB stürzt **oft** ab
- GDB Scripting ist stark eingeschränkt (Python–Scripting in Entwicklung)

Es funktioniert!

```
    }
}

void Gsi::vector (unsigned vector)
{
    unsigned gsi = vector - VEC_GSI;

    if (gsi == Keyb::gsi)
        Keyb::interrupt();

    if (gsi_table[gsi].irt & TRG_LEVEL)
        mask (gsi);

    eoi (gsi);

--:--  gsi.cpp          81% L117  Git-master  (C++/l Abbrev
#0  *gsi_vector (vector=41) at ../src/gsi.cpp:122
#1  0xc00001f0 in entry_gsi () at ../src/entry.S:127
#2  0x00000000 in ?? ()
```

Kein Testrechner verfügbar?

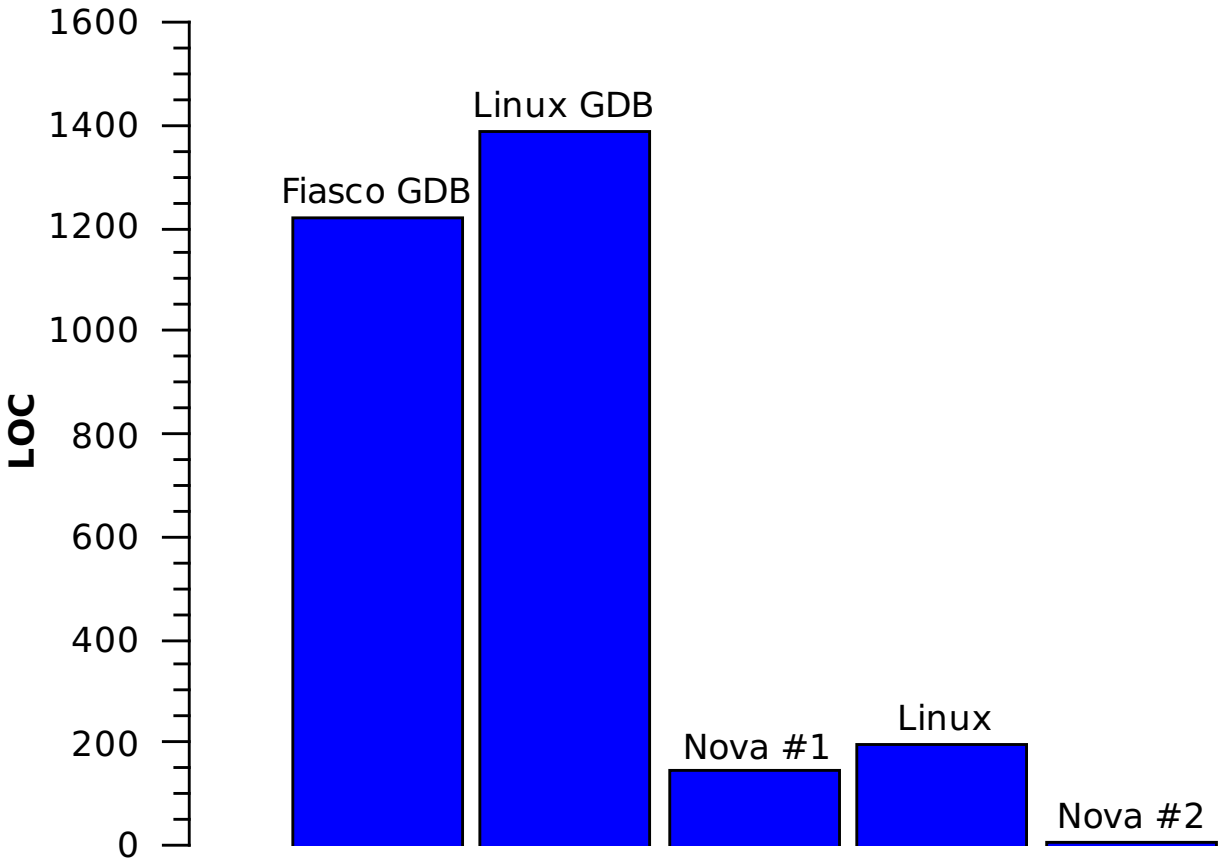
I/O Backend für Qemu/KVM

- Testrechner als Qemu-Prozesse starten
- benutzt Qemus Debugschnittstelle
- recht langsam trotz Caching

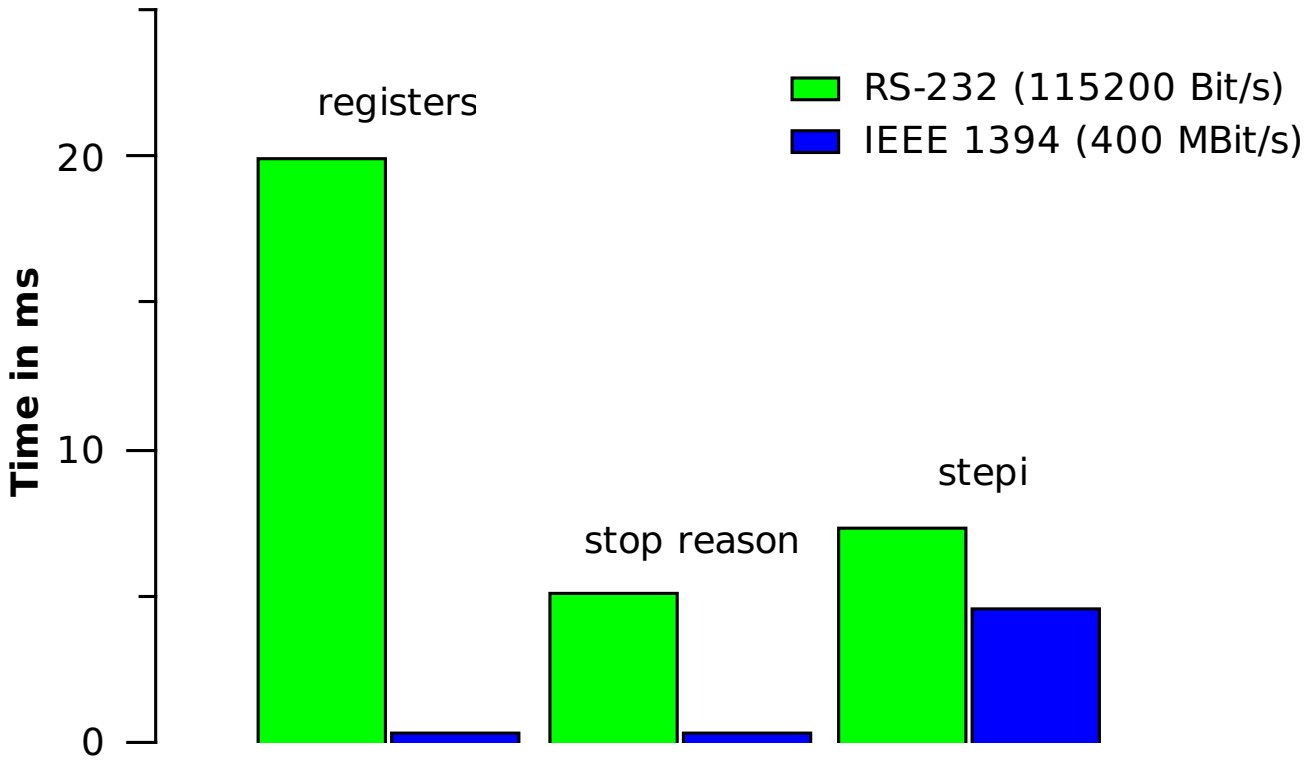
Große Hilfe bei der Entwicklung des Monitors!

Gliederung

Änderungen am Zielkern



Schneller als RS-232?



Warum ist `stepi` „langsam”?

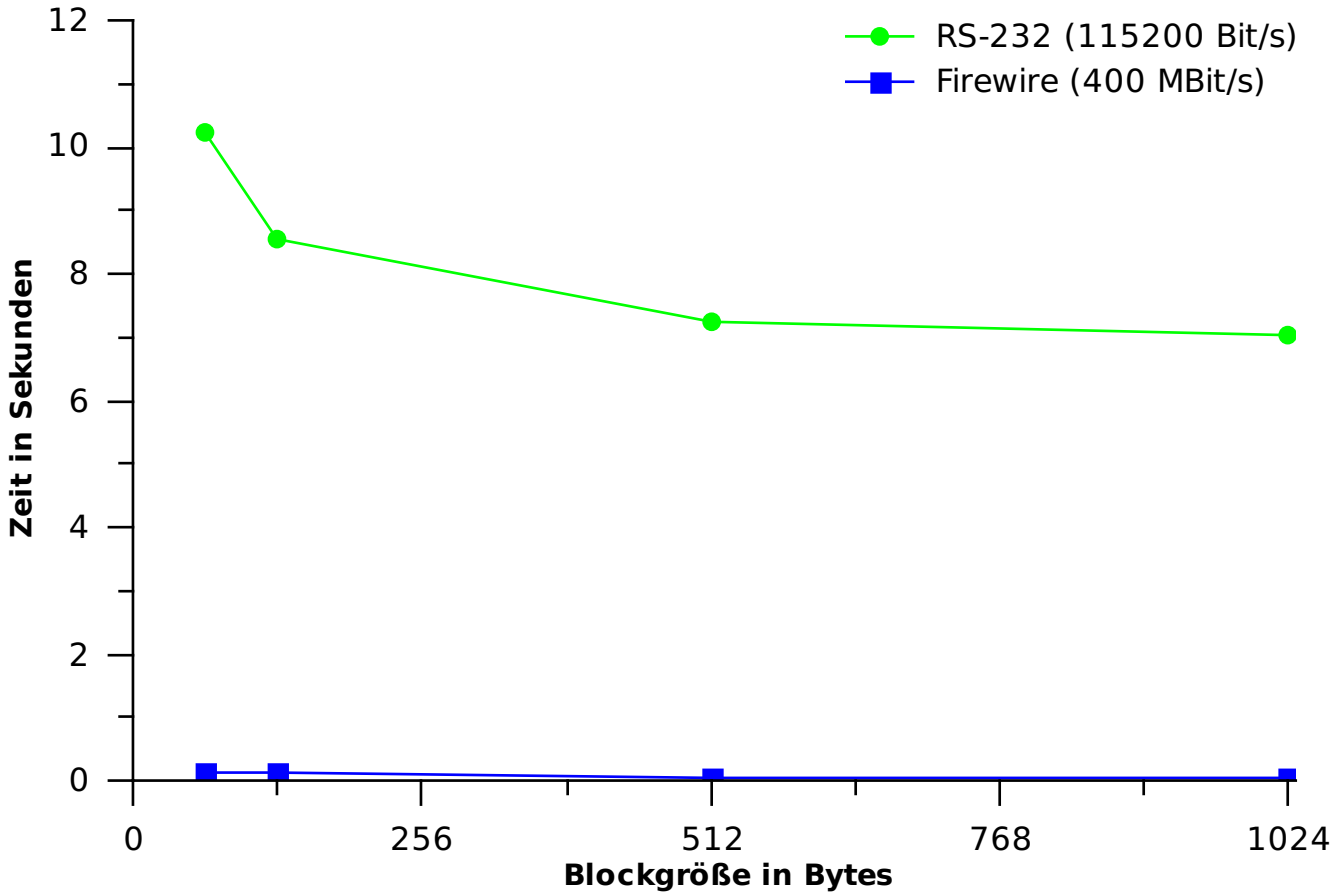
RS-232

- ein Roundtrip zum Stub

Firewire

- ein Roundtrip zum Monitor
- zwei Monitorinterrupts
- Speicherzugriffe

32KB Lesen


RS-232 (115200 Bit/s)
Firewire (400 MBit/s)
Zeit in Sekunden
0
2
4
6
8
10
12
0
256
512
768
1024
Blockgröße in Bytes

Statistisches Profiling

Kosten eines Monitorinterrupts

Monitor

- liest Instruction Pointer periodisch
- oft vorkommende Werte $\rightarrow$ Hotspots
- 24 LOC (primitiv . . .)

Interruptkosten

- 5420 Takte bzw. 4 µs auf Zielsystem (P4)
- 300 µs auf Host (3 Bustransaktionen)

$\sim$ 1% Profiling–Overhead bei 3000 Samples/s!

Gliederung

Erkenntnis #1

Kerneldebugging kommt (fast) ohne Anpassungen am Ziel aus.

- benötigter Stub wird zur Laufzeit eingebracht
- wenig Zielkernel-spezifischer Code
- Debugger schnell auf andere Betriebssysteme adaptierbar

Erkenntnis #2

GDB ist ungeeignet für Systemdebugging.

Keine Unterstützung für

- mehrere Adressräume
- mehrere Prozessoren
- spezifische Register (MSRs, Control Register, . . .)
- . . .

Zusammenfassung

- funktionierender Remote Debugger via Firewire
- minimale Änderungen an NOVA
- weiterer Fortschritt behindert durch GDB

Fragen?