RTAI: Linux im harten Echtzeiteinsatz

Robert Baumgartl
Technische Universität Chemnitz
Juniorprofessur Echtzeitsysteme

25. 01. 2005

Abstract

Der Vortrag diskutiert einführend RTAI, das Real-Time Application Interface für Linux, sowohl aus Anwender- als auch Systemsicht. Dabei stehen folgende Aspekte im Vordergrund:

- Konzept: Virtualisierung der Interrupt-Hardware
- Überblick über das Realtime Application Interface (RTAI)
- Einführung in die Programmierung von RTAI-gestützten Applikationen
- IPC zu non-RT-Applikationen
- Subprojekte: Xenomai, RTAI/fusion
- Alternativen: RTLinux, Microkernel, ...

Geschichtliches

- März 1996: Idee der Virtualisierung der Interrupts (Michael Barabanov/Viktor Yodaiken: *Real-Time Linux*, Linux Journal)
- etwa ab 1998 DIAPM ("Dipartimento di Ingegneria Aerospaziale – Politecnico di Milano") RTAI ("Real-Time Application Interface") – Konkurrenzentwicklung unter Paolo Mantegazza
- 30. 11. 1999 US-Patent: "Adding real-time support to general purpose operating systems" (US 5,995,745)

  heftig umstritten, weil ...

  > "... even if the patent owner offers a license on friendly terms, usually the project will be restricted in some way or other and the intentions of the developers to create real free software will be betrayed ..."

- ⇒ Gründung der FSMLabs, Inc. (1999)
- Dezember 2003: RTAI ersetzt patentgefährdeten RTHAL durch ADEOS

Grundprinzipien

- Virtualisierung der Interrupt-Hardware: Interrupts werden in Nachrichten umgesetzt, die zielgerichtet zugestellt werden
- Linux-System (Kern und Nutzerprozesse) ist eine RT-Task mit niedrigster Priorität
- alle RTAI-Komponenten (HAL, Module, Applikationen) sowie der Linux-Kern im Kernel-Mode abgearbeitet
  - \+ Modul-Schnittstelle des Kerns nutzbar
  - \+ keine Adreßraumwechsel zwischen RT-Applikationen ($\rightarrow$ keine TLB-Flushes, kein Paging)
  - \+ gesamtes System modular aufgebaut
  - \+ direkter Zugriff auf Hardware (da im Ring 0 ausgeführt)
  - − Sicherheit?
- Schnittstelle zum Linux-Kern für zeitunkritische Aktionen

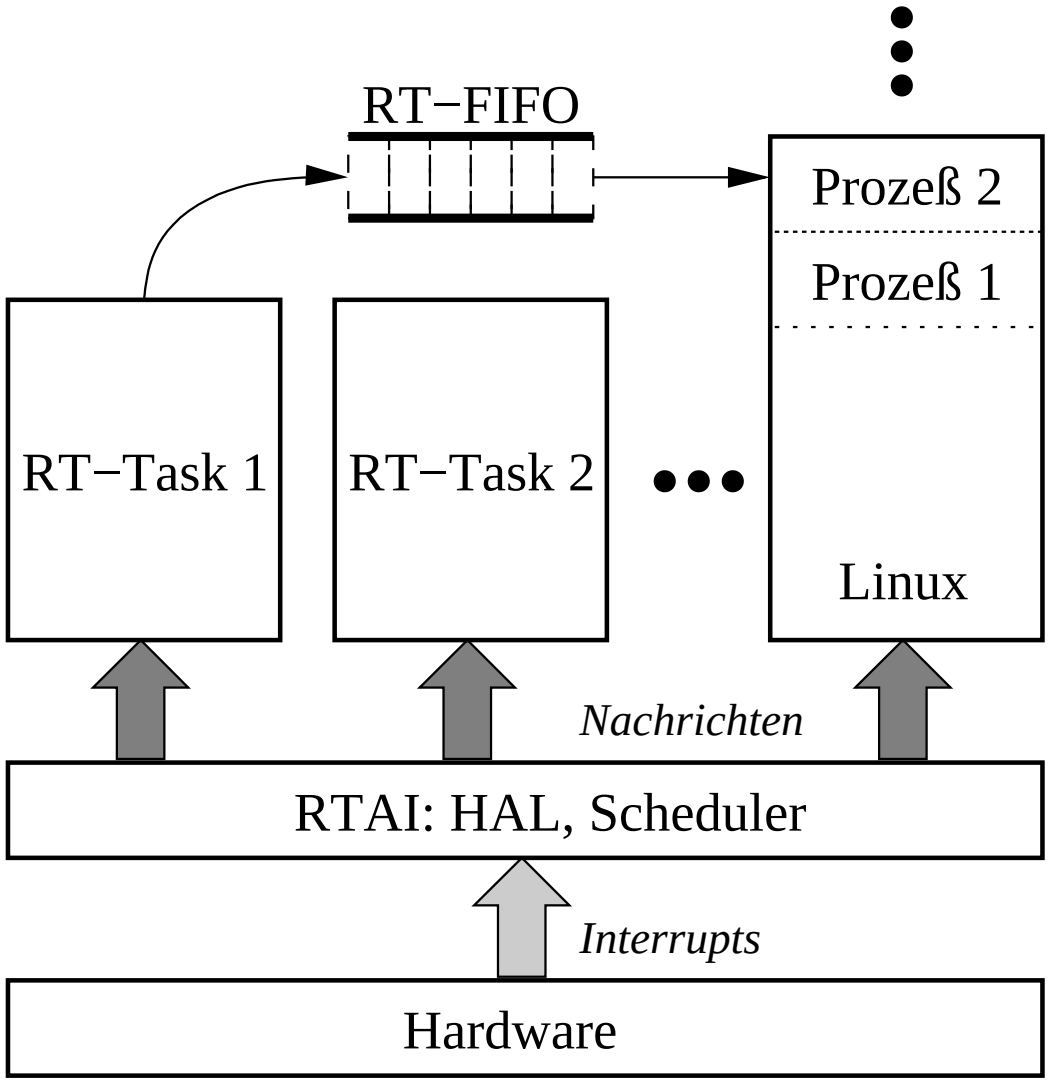

Abbildung 1: Struktur eines RT-Linux-Systems

Virtualisierung der Interrupts

- RT-Applikationen können sich für Interrupts registrieren (`rt_request_global_irq()`)
- Eintreffen eines Interrupts in RT-HAL:
  - für RT-Applikation: Zustellung des Interrupts an beanspruchende RT-Applikation (Aufruf des installierten Handlers)
  - ansonsten: Zustellung an Linux-Kern (wenn dieser Interrupts temporär verboten → Warteschlange)
- `cli`- und `sti`-Operationen des Linux-Kerns werden nicht mehr auf der Hardware durchgesetzt, sondern betreffen nur das Linux-Subsystem
- Möglichkeit, dem Linux-Subsystem eigene Interrupt-Handler „unterzuschieben“

Installation

1. Standard-Kern-Quellen und RTAI-Quellen herunterladen, installieren
2. Kern patchen (rtai-core/arch/$ARCH/patches/)
   - entweder (legacy) RT-HAL
   - oder ADEOS
3. Kern übersetzen, installieren
4. RTAI konfigurieren: `make {x|g|menu}config`
5. RTAI übersetzen, installieren: `make install`

- IMHO problemlos
- SW-Voraussetzungen: Qt oder GTK für config-Tool (nicht zwingend), Doxygen für Dokumentation
- Crossentwicklung über Variablen `ARCH` und `CROSS_COMPILE`

Applikationsschnittstelle

- Sammlung von (momentan) 21 Kernelmodulen, rasant wachsende Anzahl
- Core Module: `rtai.o`
- RT-Scheduler: `rtai_sched.o`
  - präemptiv
  - feste Prioritäten
  - verschiedene Implementierungsvarianten (UP, SMP)
- RT-FIFOs: `rtai_fifo.o`
- Shared Memory: `rtai_shm.o`
- ...

Eine einfache Echtzeit-Applikation

- Programmierung wie ein Kernmodul (gcc-Switches: `-D__KERNEL__ -DMODULE`)
- vorgeschriebene Funktionen: `init_module()`, `cleanup_module()`
- Wir sind im Kerneladreßraum → defensiv programmieren!
- Vorsicht: Linux-Subsystem hat *kleinste* Priorität
  - entweder definiertes Ende aller Tasks,
  - oder ausreichend Luft für Linux lassen (periodisches Abarbeitungsmodell)
- `insmod` aller benötigten RTAI-Module, *danach* der RT-Applikationen
- `init_module()`
  - Timer kreieren, starten
  - IPC-Ressourcen anfordern
  - Task(s) kreieren und starten
- `cleanup_module()`
  - „Rückbau“ der Infrastruktur

IPC: RT-FIFOs

- Pendant zu *named pipes* unter UNIX
- unidirektional, beschränkte Größe (→ Überlauf möglich)
- sowohl für RT-Tasks als auch Linux-Prozesse (beidseitig!) nutzbar
- Operationen von RT-Seite sind stets nichtblockierend
- Linux-Seite: Character Device, `/dev/rtf0...63`

Semantik	*RT*	*Linux*
Zugriff erlangen	`rtf_create()`	`mknod()`, `open()`
Löschung	`rtf_destroy()`	`close()`, `unlink()`
Lesen	`rtf_get()`	`read()`
Schreiben	`rtf_put()`	`write()`

Tabelle 1: FIFO-API

Weitere Aspekte

- Scheduling: neben harten Prioritäten (FEP) und Round Robin ebenfalls ratenmonotones Scheduling (RMS) und *Earliest Deadline First* (EDF) möglich
- Shared-Memory-Segmente zwischen RTAI- und/oder Linux-Applikationen
- Mailboxen
- Semaphore: Prioritätsvererbung und Priority Ceiling möglich (Verhinderung Prioritätsinversion)
- RT-pthreads
- RTAI/fusion: "RTAI-API im User-Space"
- ...

Eigene (subjektive) Einschätzung/Erfahrungen

- rapide Entwicklung (WWW: 1.3 → 1.4 → 3.0 → 3.1r2 (aktuell))
- Unterstützung durch Firmen:
  - Denx Software Engineering, Gröbenzell
  - Sysgo AG
  - Metrowerks, Inc.
- Dokumentation hinkt stark hinterher, typisch Linux (inkonsistent, "Read the source, Luke"-Hilfen etc.)
- relativ viel Legacy-Funktionalität erschwert Einarbeitung
- Einsatz an JP EZS:
  - Experimente im Rahmen des Praktikums *Embedded Programming*
  - Diplomarbeit zur Portierung der zentralen Module auf eine DSP-Architektur
  - Untersuchung der Eignung für autonome mobile Roboter (DA in Zusammenarbeit mit der Professur Prozeßautomatisierung)
- bisher nur 2.4er Kern genutzt; 2.6 wird ebenfalls unterstützt

(Bisher) unterstützte Architekturen

- x86 (mit und ohne TSC, FP-Unterstützung)
- PowerPC (z. B. Motorola MPC8xx)
- ARM (StrongARM; ARM7: clps711x-family, Cirrus Logic EP7xxx, CS89712, PXA25x)
- MIPS
- Motorola ColdFire (basierend auf *ucLinux*)

Momentane Lizenzsituation (AFAIK)

RTAI:

- LGPL2 (d.h., es können auch proprietäre Module gelinkt werden)
- Core: GPL

RT-LINUX:

- "Open RTLinux Patent License": kostenlose Nutzung bei GPL
- Kommerzielle Lizenz, wenn Ergebnis der Arbeit nicht GPL
- U.S.-Patent No. 5,995,745 momentan nicht (aber bald?) in EU durchsetzbar

Anwendungen

Abbildung 2: Lasergesteuerte Schneidemaschine

(http://www.primaindustrie.com/pr_platino.html)

Alternativen

- RTLinux: gleiches Prinzip, technisch gleichwertig, umstrittene Lizenz
- Microkernels:
  - elementare BS-Funktionalität im Kern: IPC, Adreßräume, Threads
  - alles andere im User Space (Dateisysteme, Gerätetreiber, ...)
  - bessere Separation der Komponenten
  - Beispiele: Mach, Exokernel, Chorus, L4/Fiasco
  - weitaus weniger verbreitet

Ausblick

> "TAI continues to grow and mature through the combined contributions of the DIAPM development team and those across the Internet who are encouraged by RTAI's flexible architecture, high performance and feature set."
> (P. Mantegazza)

- Linux dringt in *embedded*-Markt ein
- kontrovers diskutiert
- nachteilig:
  - relativ großer Memory Footprint
  - ungenügender Schutz

Weitere Arbeitsziele:

- "VxWorks and pSOS compatibility libraries" → Xenomai
- Linux Tracing Toolkit (LTT)

Weiterführende Informationen

- Heimatseite RTAI: http://www.aero.polimi.it/~rtai
- Heimatseite RTLinux: http://www.fsmlabs.com/index.html
- RTAI 3.0 Documentation Page: http://www.fdn.fr/~brouchou/rtai/rtai-doc-prj/
- Patent US05995745: http://www.delphion.com/details?pn=US05995745__
- Jerry Eplin: *A Developer's Perspective on the RTLinux Patent*, 2001
  http://www.linuxdevices.com/articles/AT2094189920.html
- RTLinux vs. RTAI: http://bernia.disca.upv.es/rtportal/comparative/rtl_vs_rtai.html
- Adeos Nanokernel: http://home.gna.org/adeos/
- GPL-Version von RT-Linux: http://www.rtlinux-gpl.org/