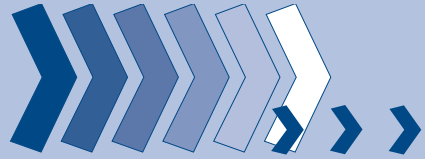
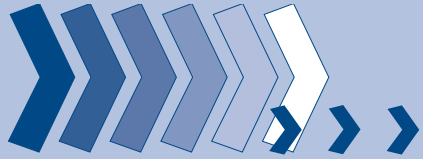


# Betriebssysteme: rtLinux

Dipl.-Inf. J. Richling  
Wintersemester 2003/2004



- Fünfeinhalb Vorlesungen:
  - Embedded- und RT-Betriebssysteme (19.1.04)
  - Beispiel: Windows CE (22.1.04)
  - Beispiel: Windows XP embedded (Selbststudium mit Material auf CD)
  - Beispiel: RT-Linux (heute)
  - Beispiel: PalmOS (29.1.04)
  - Beispiel: OSEK und PURE (2.2.04)

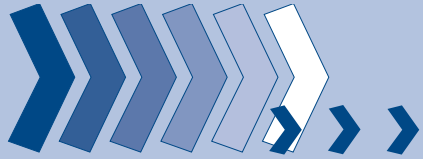


## *Real-Time Linux (rtLinux)*

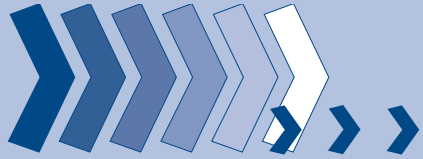
### Real-Time auf Basis von Linux - warum?

- Linux ist open source — Beliebig genaues Wissen über Abläufe im Systemkern ist möglich
- Linux ist frei verfügbar — breites Einsatzfeld in der Forschung und der Lehre (und Industrie)
- Linux läuft auf kostengünstiger Hardware
- Linux ist ein “full-featured” OS — vorhandenes kann benutzt werden



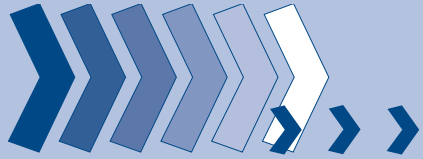


- 1995/1996: Master Thesis von Michael Barabanov am New Mexico Institute of Technology unter Leitung von Assist. Prof. Victor Yodaiken
- 1998: Gründung von FSMLabs (Finite State Machine Labs) zur Weiterentwicklung und Vermarktung
- 1999: Entwicklung zu Posix-Threads und Kompatibilität zu POSIX 1003.13 “minimal realtime operating system” Standard
- Heute:
  - Version 3
  - Umstrittene Lizenzierung, aber Kerntechnologie frei von Rechten dritter



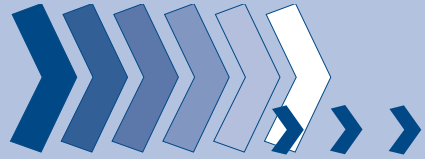
- Harte Echtzeittasks
  - Periodisch
  - Interrupt-driven (aperiodisch)
- Zugriff auf alle Tools und Dienste des Linux-Betriebssystems
  - Entwicklungstools
  - Graphische Oberfläche
  - Analyse und Anzeige von Daten
  - Netzwerk
  - . . .

. . . auf dem gleichen Rechner zur gleichen Zeit!

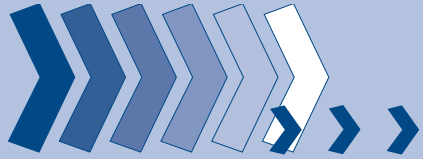


Ziel ist es, zwei inkompatible Eigenschaften in einem Betriebssystem zu mischen:

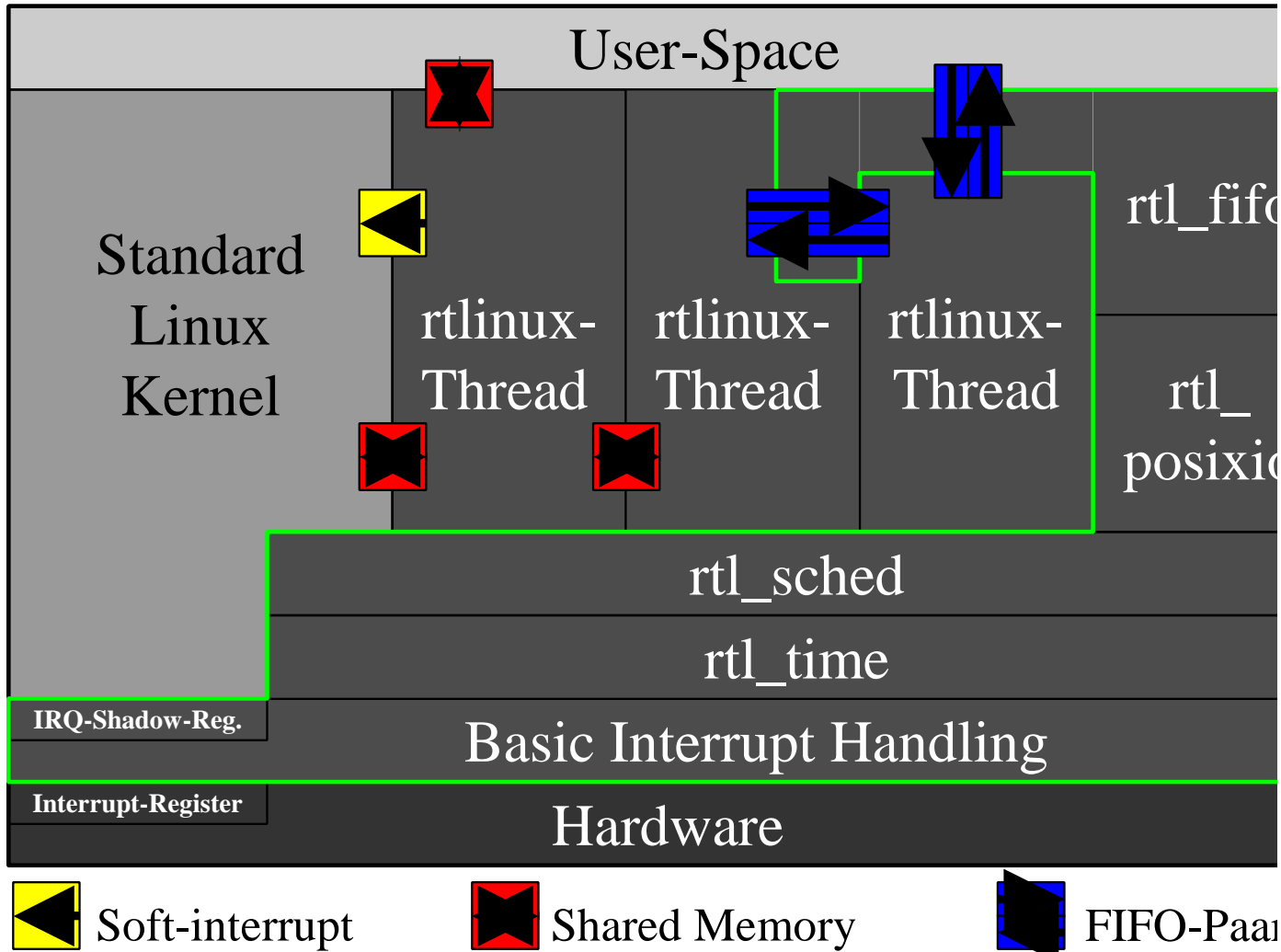
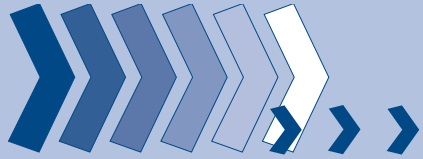
- Harte Echtzeit
  - vorhersagbar
  - “schnell”
  - geringe Latenz
  - einfacher Scheduler
- Standard-OS-Dienste (POSIX)
  - GUI
  - TCP/IP
  - NFS
  - Compiler
  - Webserver
  - . . .

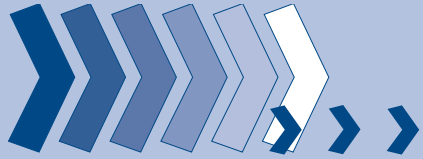


- Umwandlung des Linux-Codes für Interruptbehandlung in Behandlung **weicher** Interrupts
- **Harte** Interrupts werden durch die Echtzeit-Executive aufgefangen
  - Werden an Linux als weiche Interrupts weitergegeben, wenn sie für Linux bestimmt sind und Linux die Interrupts aktiviert hat
- Interrupts an RT-Tasks (u.a. Timer) werden von RT-Executive behandelt und können vom Linux nicht abgeschaltet werden

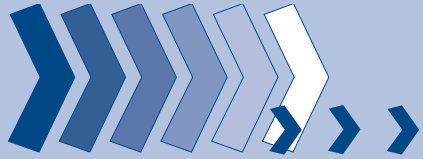


- Mini-Kernel mit uneingeschränktem Hardware-Zugriff
  - Sporadische Tasks (Interrupts)
  - Zyklische Tasks (zeitgesteuert per Scheduler)
  - Linux-Kern ist Idle-Task des Mini-Kerns
  - Ressourcenverwaltung teilweise durch Linux-Kern
- Interprozeßkommunikation
  - Zwischen RT-Tasks:
    - \* Queues
    - \* Semaphore
    - \* Mutexes
    - \* Shared Memory
  - Zwischen RT-Tasks und Linux-Tasks
    - \* RT-FIFOs
    - \* Shared Memory



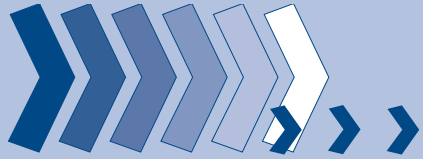


- Scheduling (beliebig per Modul)
  - Preemptives Prioritäten-Scheduling (SMP-fähig)
  - EDF-Scheduling (Nur Version 1.x)
- Taskverwaltung
  - Create, Delete, Suspend, Wakeup, Wait, Makeperiodic
- Interrupt Handler
  - Sperrt Linux-Interrupt und startet sporadische Task
  - simuliert Interrupt-Register für Linux-Kern
- Zeitverwaltung
  - Aktuelle Systemzeit ermitteln
  - Funktionen zur Schaffung eigener Scheduler



## *rtLinux — Funktionalität: Kommunikation*

- IPC (zwischen RT-Tasks)
  - Create/Delete Queue/Semaphore
  - Send/Receive Message/Semaphore
  - Mutexes
- RT-FIFOs (zwischen RT-Tasks und Linux-Prozessen)
  - Create, Delete, Resize, Clear, Put, Get
  - Kernelspace-Handler
- Shared Memory



## *rtLinux — Änderungen an Linux*

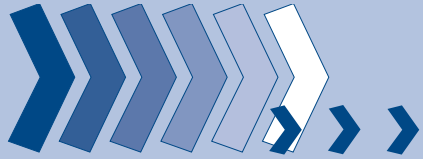
- Interrupt Handler auf unterster Ebene angepaßt an weiche Interrupts
- CLI/STI ersetzt durch S\_CLI und S\_STI
- Real-Time clock handler verwaltet Zeit
- RT-Scheduler ist ladbares Kernelmodul
- RT-Tasks sind ladbare Kernelmodule

Gerätetreiber von Linux funktionieren weiter

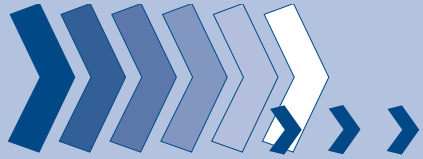


## *rtLinux — POSIX-Kompatibilität*

- Version 1 von 1995: Simple API mit vielen Beschränkungen
- Ziel: Kompatibilität zu Standards
- POSIX 1003
  - Viele Funktionen mit viel Overhead
  - z.B. Filesystem mit Links, vielen Verzeichnisebenen, usw.
- Aber: POSIX 1003.13 “minimal realtime system”
  - Anwendungsumgebung: Einzelner multithreaded POSIX-Thread auf dedizierter Hardware
  - rtLinux-Task wird zu POSIX-Thread
  - rtLinux-Interrupt-Handler wird zum Signal Handler
  - Linux läuft als Thread der niedrigsten Priorität

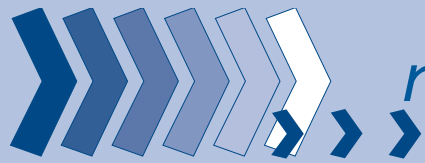


- Worst Case Interrupt Latency auf Standard-PC-Hardware: 20 Mikrosekunden
- Aber
  - PC-Hardware ist nicht für Echtzeit entwickelt
  - Einige (billige) PCI-Karten blockieren PCI-Bus für Mikrosekunden zum Füllen von internen FIFOs
- Hardware mit besseren Timern und besserer Interruptbehandlungshardware:
  - AMD Elan520 mit 133 MHz: Worst Case Interrupt Latency von 7 Mikrosekunden
  - Motorola M8260: Worst Case Interrupt Latency von 4 Mikrosekunden
  - Sorgfältig ausgewählte PC-Hardware: Akzeptabel niedrige Worst Case Interrupt Latency
- Jitter bei periodischen Tasks unter 5 Mikrosekunden (typisch: 1 Mikrosekunde)



## *rtLinux — Beispiel Tongenerator: Idee*

- Zwei periodische Real-Time Tasks implementieren einen Tongenerator
  - Eine Task setzt ein Bit des Parallelports
  - Zweite Task löscht das Bit wieder
- Linux-Prozeß erlaubt Steuerung von Frequenz und Taskverhältnis
- Kommunikation über RT-FIFOs
- Hardware:
  - Standard-PC
  - Mini-Lautsprecher + Widerstand sind am Parallelport angeschlossen (an der Leitung für niederwertigstes Bit)



## *rtLinux — Beispiel Tongenerator: RT-Modul I*

```
#define MODULE
/* diverse Includes geloescht */

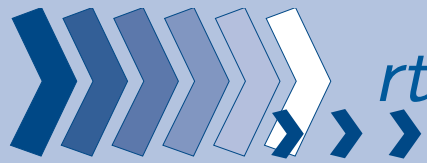
#include <linux/rt_sched.h>
#include <linux/rtf.h>

#include "control.h"

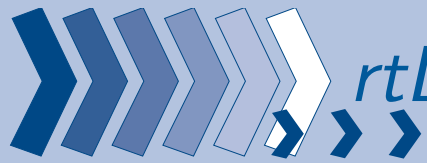
/* the address of the parallel port */

#define LPT 0x378

RT_TASK mytask;
RT_TASK mytask2;
RT_TASK control_task;
```



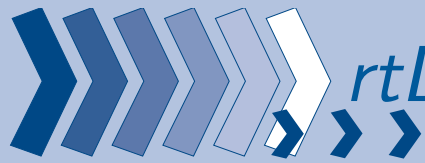
```
void fun(int t) {  
  
    while (1) {  
        outb(t, LPT); /* write to the parallel port */  
        rt_task_wait();  
    }  
}
```



```
int init_module(void)
{
    RTIME now = rt_get_time();
    rtf_create(0, 4000);

    rt_task_init(&mytask, fun, 1, 3000, 4);
    rt_task_init(&mytask2, fun, 0, 3000, 5);

    /* the 2 tasks run periodically with an offset */
    rt_task_make_periodic(&mytask, now + 3000,
        ((RT_TICKS_PER_SEC * 50000)/1000000));
    rt_task_make_periodic(&mytask2, now + 3000+
        ((RT_TICKS_PER_SEC * 25000)/1000000),
        ((RT_TICKS_PER_SEC * 50000)/1000000));
    rtf_create_handler(0, &my_handler);
    return 0;
}
```



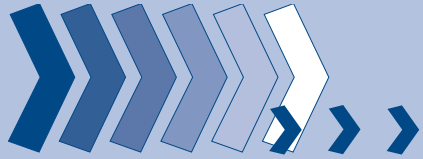
## *rtLinux — Beispiel Tongenerator: RT-Modul IV*

```
int my_handler(unsigned int fifo)
{
    struct my_msg_struct msg;
    int err;
    RTIME now;

    while ((err = rtf_get(0, &msg, sizeof(msg))) == sizeof(msg)) {
        now = rt_get_time();
        rt_task_make_periodic(&mytask, now, msg.period);
        rt_task_make_periodic(&mytask2, now + msg.offset, msg.period);
    }

    if (err != 0) return -EINVAL;

    return 0;
}
```



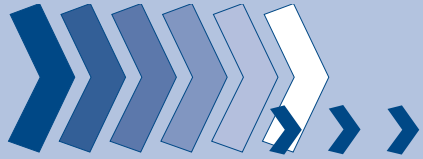
## *rtLinux — Beispiel Tongenerator: User Level*

```
char buf[BUFSIZE];
int main()
{
    int ctl;
    struct my_msg_struct msg;

    if ((ctl = open("/dev/rtof0", O_WRONLY)) < 0) {
        fprintf(stderr, "Error opening /dev/rtof0\n");
        exit(1);
    }

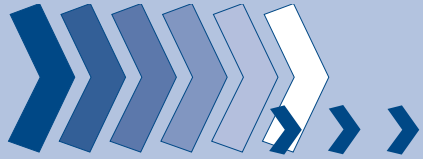
    msg.offset = ((RT_TICKS_PER_SEC * 250000)/1000000);
    msg.period = ((RT_TICKS_PER_SEC * 500000)/1000000);

    if (write(ctl, &msg, sizeof(msg)) < 0) {
        fprintf(stderr, "Can't send a command to RT-task\n");
        exit(1);
    }
}
```

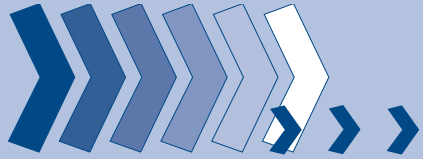


## *rtLinux — Beispiel Scheduling Server*

- Idee:
  - CPU-Partitionierung mit festen Anteilen an CPU-Zyklen
  - Analog Scheduling Server
- Umsetzung
  - Hochpriorisierte Task ( “Scheduling Server” ) läuft periodisch und regelt Prioritäten der Client-Task und suspendiert diese mit “rt\_task\_suspend”
  - Steuerung über FIFOs von Linux aus
  - Beliebige Scheduling-Algorithmen sind vorstellbar
  - Aber: Client-Task muß RT-Task sein
- Weiterentwicklung
  - Scheduling Server (RT-Task) beeinflusst Scheduling des Linux-Kerns
  - Beliebige Clienttask

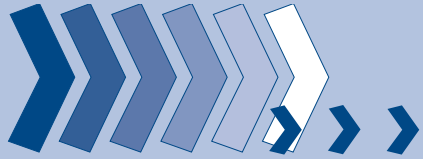


- Einfaches Beispiel für die Benutzung von POSIX-Funktionen:
  - timespec()
  - clock\_gettime()
  - clock\_nanosleep()
  - pthread\_create()
  - pthread\_join()
- Funktion:
  - Periodischer Thread
  - Liest Daten von einem Gerät
  - Schreibt Daten auf einen FIFO
  - Prozeß unter Linux kann Daten vom FIFO lesen



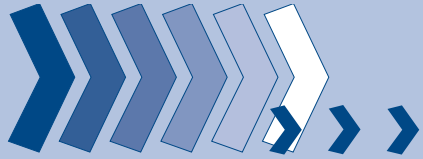
## *rtLinux — Beispiel POSIX-Threads II*

```
void *my_code(void *arg){ // argument is not used in this example.
    struct timespec t;
    struct mydata D;
    clock_gettime(CLOCK_RTL_SCHED,&t);
    while(!stop){
        copy_device_data(&D.d);
        /* ignore write fails, we just drop the data */
        write(fd,&D,sizeof(D));
        timespec_add_ns(&t,DELAY_NS);
        clock_nanosleep(CLOCK_RTL_SCHED,
                        TIMER_ABSTIME,
                        &t,
                        NULL);
    }
    return (void *)&stop;
}
```



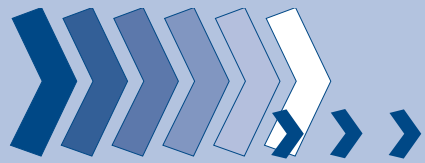
```
pthread_t T;  
int fd;  
int stop = 0;  
#define DELAY_NS 500000 // 500 microseconds
```

```
void cleanup_module(void){  
    stop = 1;  
    pthread_join(T,NULL);  
    close(fd);  
}
```



## *rtLinux — Beispiel POSIX-Threads IV*

```
int init_module(void){
    if ((fd=open("/dev/rmf0",
                O_WRONLY |
                O_NONBLOCK |
                O_CREAT ))
        < 0)
    {
        rtl_printf("Example cannot open fifo\n");
        rtl_printf("Error number is %d\n",errno);
        return -1;
    }
    if( pthread_create(&T,NULL,my_code,NULL))
    {
        close(fd);
        rtl_printf("Cannot create thread\n");
        return -1;
    }
    return 0;
}
```



## *rtLinux — Erweiterungen, Zukunft und Ports*

- Erweiterungen
  - Informationen über RT-Tasks im /proc-Filesystem
  - Treiber für serielle Schnittstelle
- Zukunft
  - rtLinux für SMP (derzeit experimentell wegen Interrupt-Routing)
  - Priority-Inheritance gegen Priority-Inversion
  - Weitere echtzeitfähige Hardware-Treiber
- Ports
  - NetBSD und FreeBSD als Idle-Task des RT-Kerns