



L I N E OTM

The Embedded Linux[®] Solutions CompanyTM

Real Time Linux

Real Time Extensions for the Linux Operating System

-An Introduction -

Bernhard Kuhn, Software Engineer, Lineo Inc.

Index

What exactly does real time mean?

What is so special about
real time operating systems?

Improving the real time
capabilities of Linux

- Nice Value, Posix Space, Kernel Space

- Scheduler extensions

- Sub-Kernel extensions

Introduction to RTL

RTAI in detail

What is Real Time?

There are several definitions

depending on the application domain

depending on the users (marketing, engineering, daytrading)

at least can mean anything

Commonly accepted definition:

The real time capability of a system can be determined by the damage that occurs depending on the event response time.

Soft vs. Hard Real Time

Soft Real Time

oftenly just means „fast“ (Daytrading)

smooth animations (console/pc games)

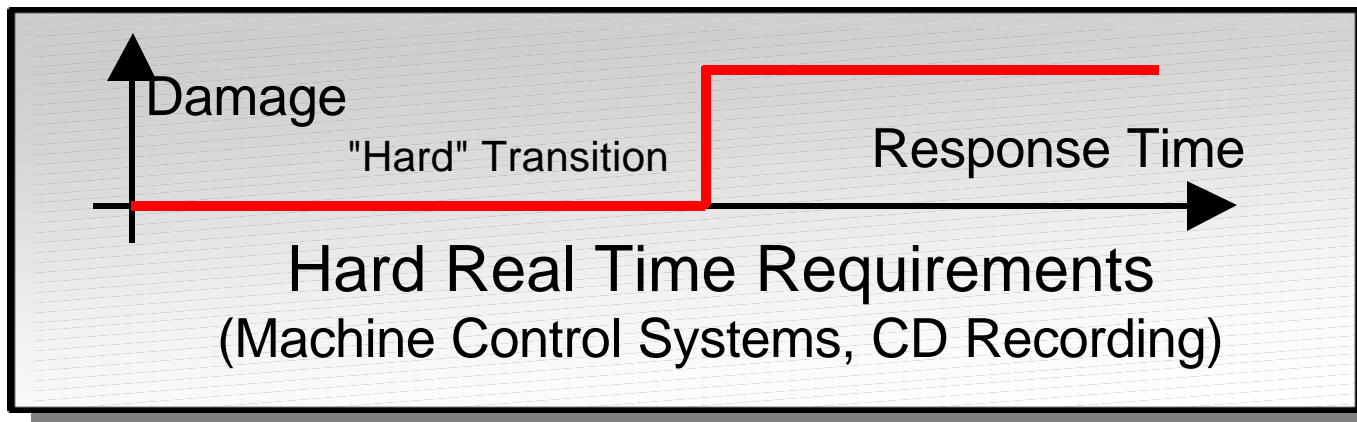
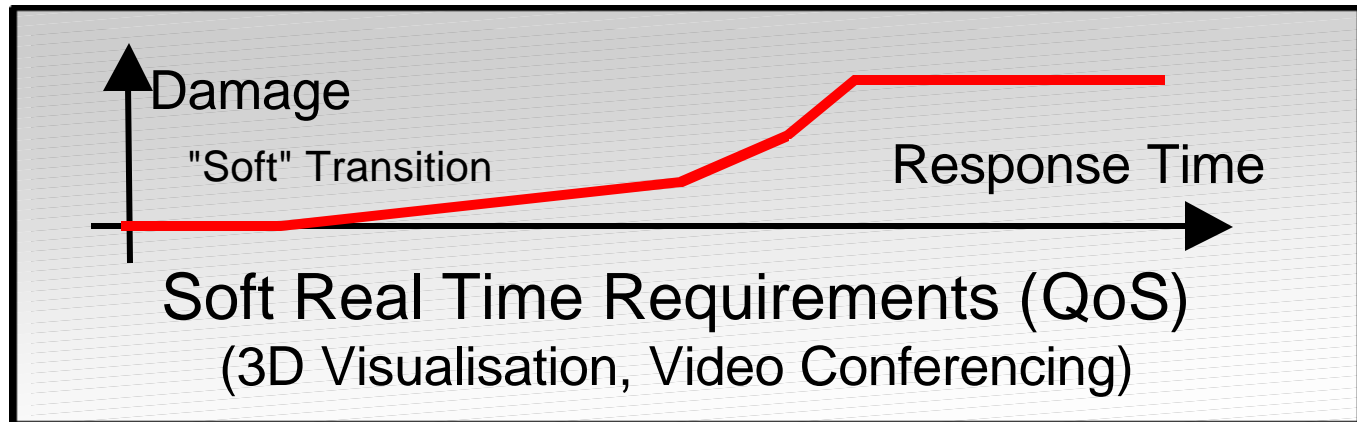
short response time on user input

Hard Real Time

guaranteed event response times

The expression „Real Time“ is strongly abused as Buzz-Word, nowadays.
Better use „Quality of Service“, if appropriate!

Damage / Response Time Ratio



Real Time vs. Server OS

Criterion \ OS	Real Time OS	Server OS
Scheduling	predictable	efficient
Optimised for	timing critical appl.	average appl.
Priority	minimum latency	maximum throughput
Offers	process+IPC API	many APIs and services

Why Linux Isn't Hard Real-time

Coarse-grained synchronization

System calls not pre-emptable

Paging

Scheduler tries to be fair”

Requests can be reordered to optimize use of hardware

Methods to Improve the Linux real time Capabilities

Classical techniques without patching the kernel

Extending the kernel scheduler for Quality of Service

Sub kernel extensions

Part 1: Classical techniques

Improoving the response times of
Linux without patching the kernel

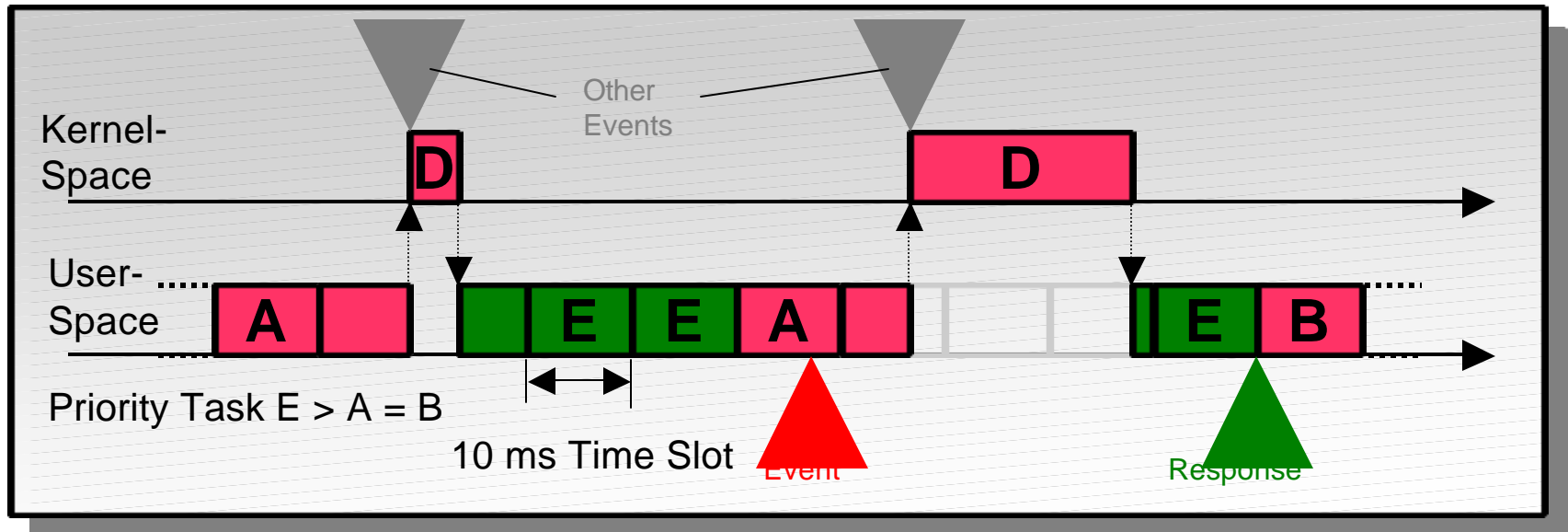
Decreasing the „Nice“ value

Posix Scheduling

Kernel Threads

Decreasing the „Nice“ value

Scheduling in time slots of 10 milliseconds each
Applications with higher „priority“ get more time slots



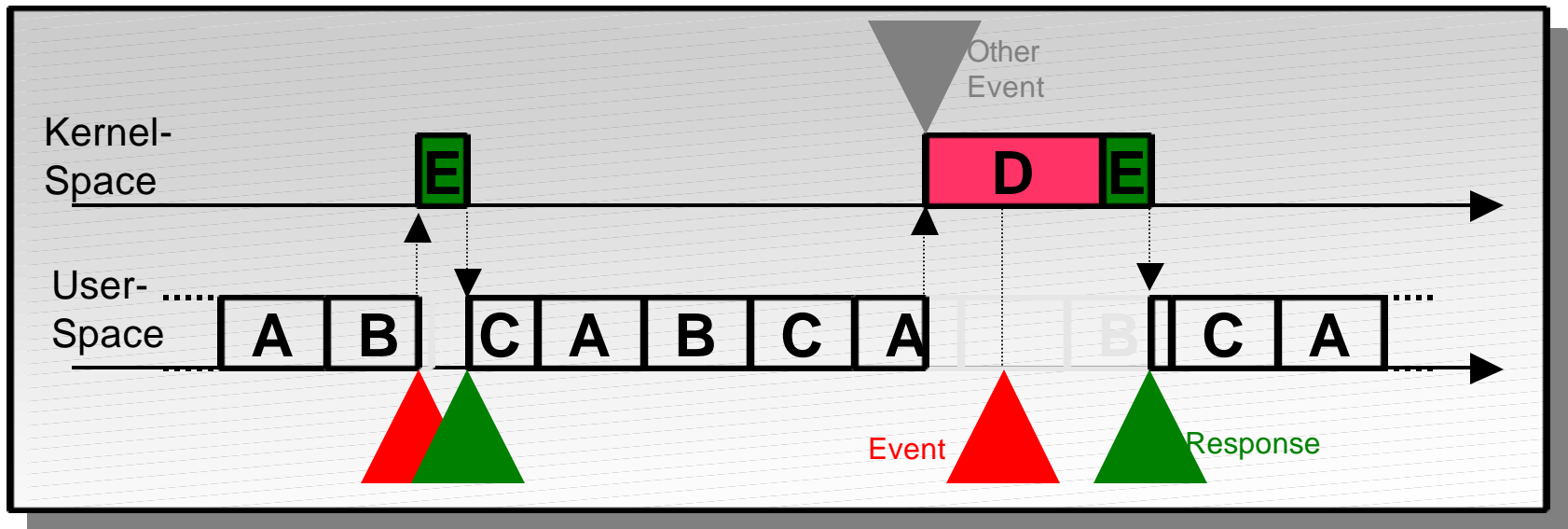
(+) Simple but robust implementation

(-) No determinisms, bad response times

Kernel Threads and Interrupts

Application is part of the Operating System

First Come First Serve scheduling (with exceptions)



(-) Only Kernel API available

(-) weak determinisms

Part 2: Scheduler Extensions

Improving the response times of
Linux by modifying the kernel scheduler

Linux Resource Kernel

Qlinux

KU-RT Linux

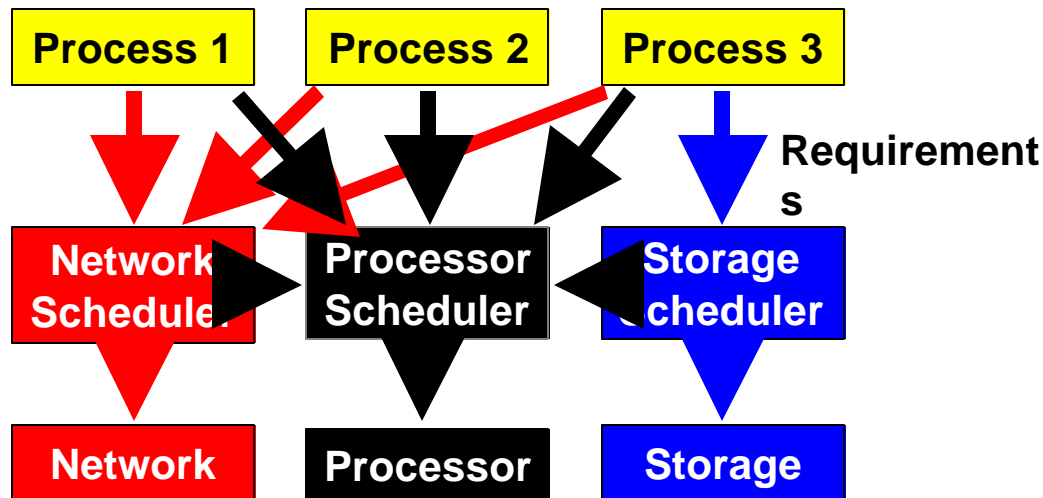
Smart Linux

Red Linux

MontaVista Scheduler

Linux Resource Kernel (Linux/RT)

**Quality of Service for Processor,
Network and Storage Devices.**

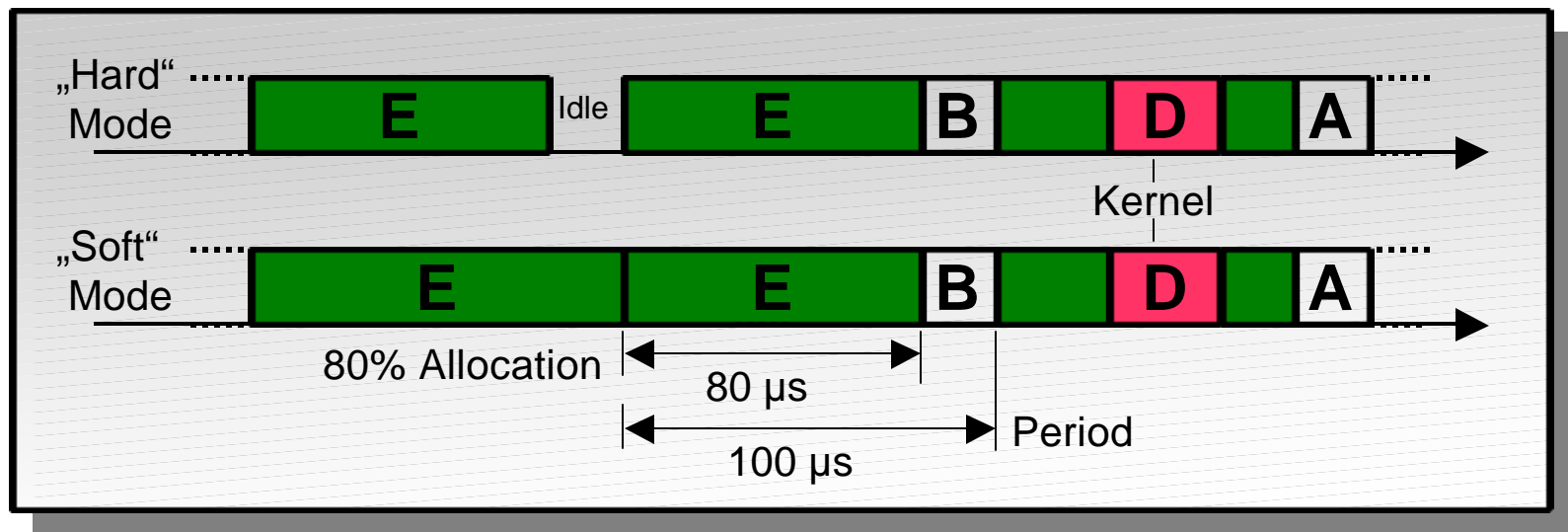


(+) User Space APIs Available

(-) Only CPU QoS implemented (as of V1.21)

Quality of Service with Linux/RT

Allocate „CPU Bandwidth“ for „Resource sets“
Adjustable Period



(+) Perfect for Game Consoles and Set Top Boxes

(-) No Hard Real Time

Part 3: Sub Kernel Extensions

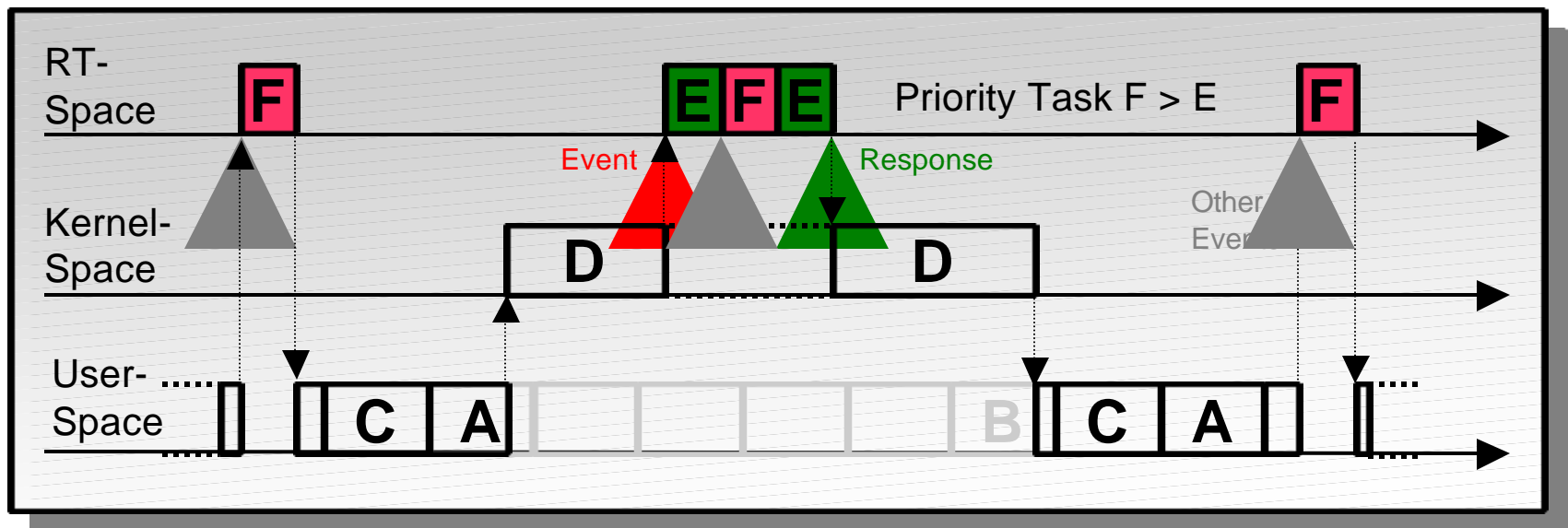
Improving the response times of Linux
by integrating a Hard Real Time Kernel

RT-Linux (RTL)

RTAI / RTHAL

RTL / RTAI

Real Time Operating System with Linux as Idle Task
Pre-emptive priority scheduling



- (+) Exceptionally well determinisms
- (-) Basicaly no Linux APIs available

Comparison of Real Time Methods

	Nice-Wert	Posix	kthreads	Sched-ext	HRT-ext
Userspace-Libs	yes	yes	no	yes	no
Kernel functions	no	no	yes	no	partly
Hard Real Time Capabilities	very bad	bad	medium	well	very well
Scheduling Priority	Timeslots	Priority	Cooperative	several	
	(RR)	Encoded	(FCFS)	Policies	Encoded
Pre-emptable by	Kernel,	Kernel,	Interrupts	Kernel	higher
	Other Tasks	Pthreads			RT-Task
typical Latencies	10ms	10ms	10ms	1ms	5 μ s
max. Latencies	>10s	>100ms	~100ms	~50ms	50 μ s

RT-Linux

Origin

Philosophy

Architecture

Functionality

Applications examples

Basic Idea:

Linux as Idle Task of a Real Time OS

RTL Origins & Philosophy

Origin

Master Thesis from Michael Barabanov directed by Assist. Prof. Victor Yodaiken, New Mexico Institute of Technology, 1996.

Philosophy

Minimum modifications to the Linux Kernel, not to put the stability at risk

Clear Architecture, simple extendable by Kernel modules

Posix compliance has a higher development priority than extending the functionality

RT-Linux is POSIX 1003.1b compliant

RT-Linux Architecture

Sub Kernel with unlimited access to the hardware

sporadic tasks (interrupt service routines)

cyclic tasks (controlled by scheduler)

Linux Kernel as Idle Task

Resources are controlled by the Linux Kernel

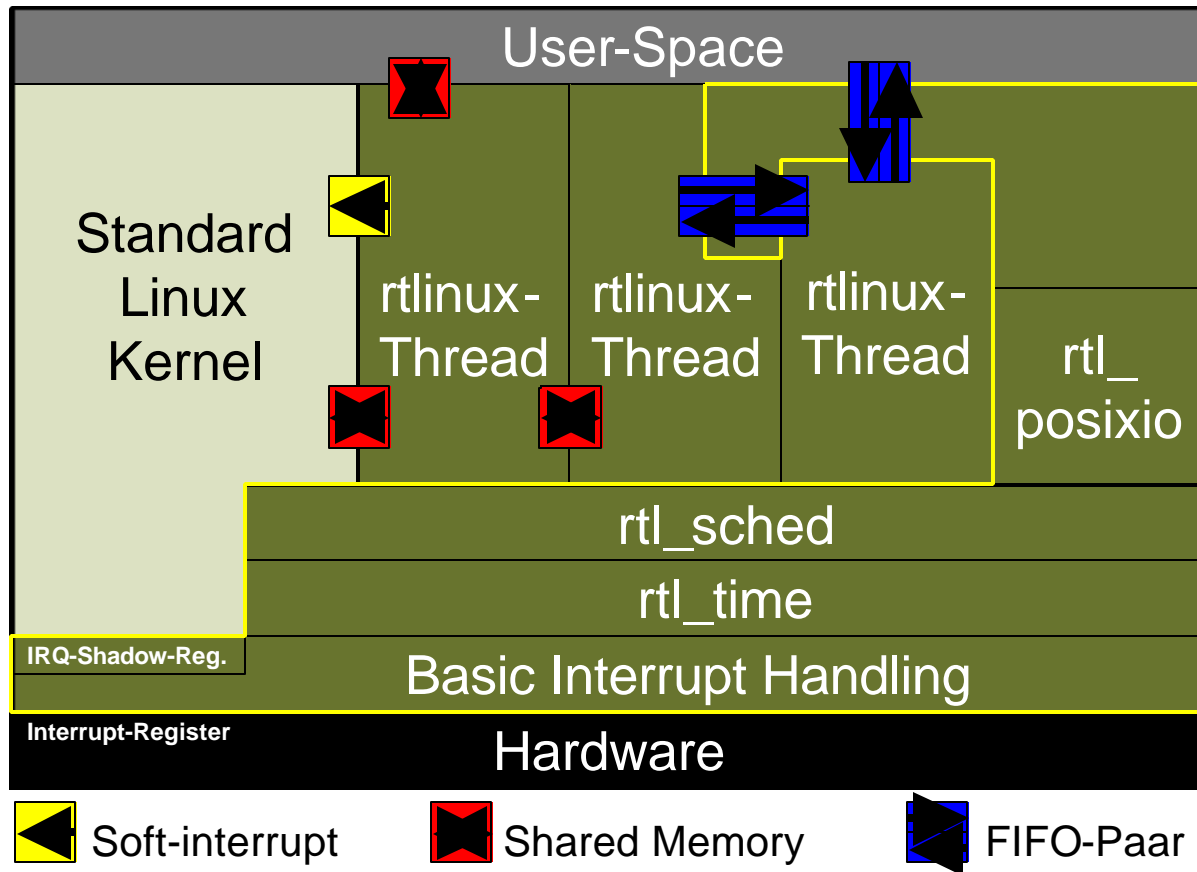
RT-Linux can't be used as stand alone Kernel

Interprocess Communication

between RT-Tasks with Queues, Semaphores,
Mutexes, Condition Variables and Shared Memory

between RT-Tasks and Linux user space processes
with RT-Fifos and Shared Memory

RT-Linux Overview



RTL Functionality / System Calls

Scheduling (runtime loadable module)

Pre-emptiv priority Scheduler (one instance per CPU)

Earliest Deadline First Scheduler (V1.x only)

Task handling

Create, Delete, Suspend, Resume, Makeperiodic, Wait

Interrupt handler

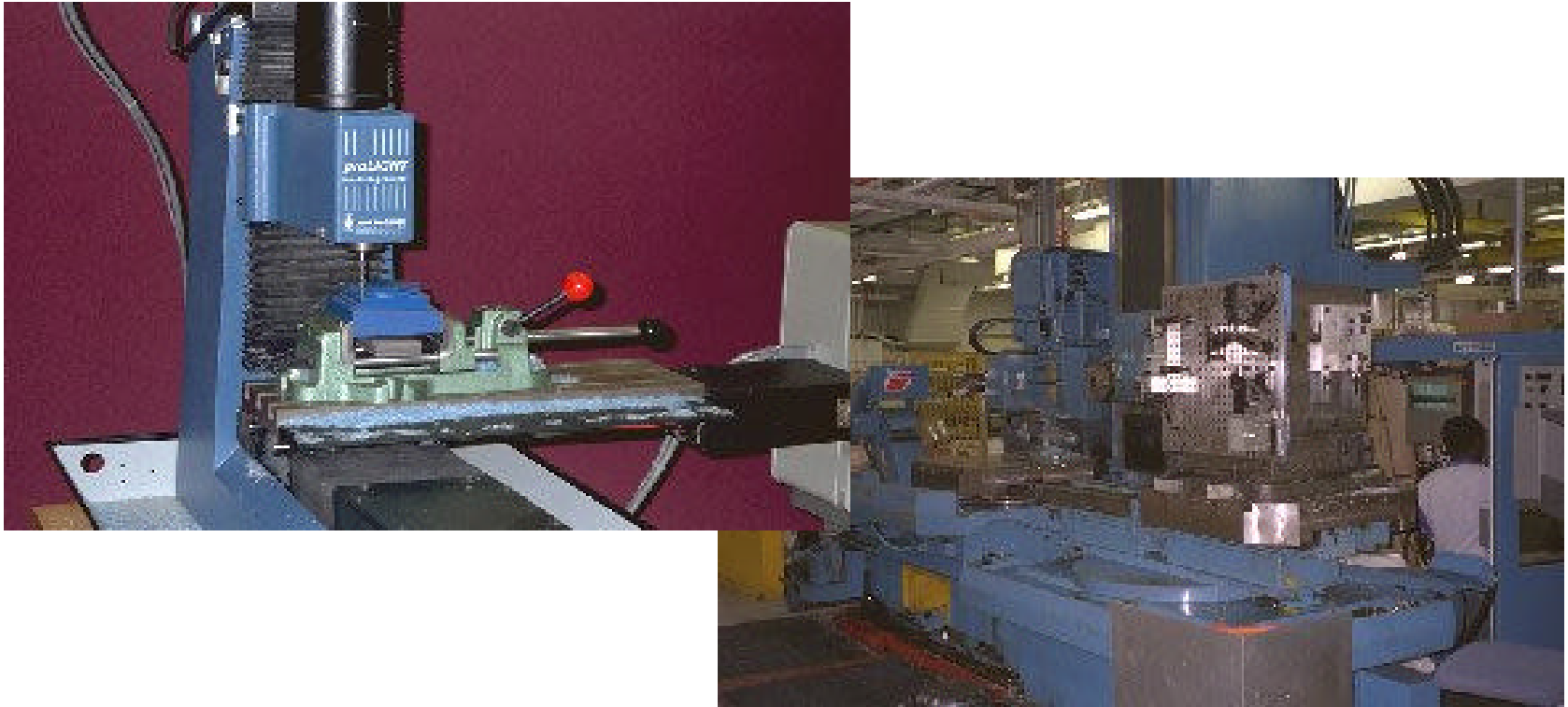
Interrupts are never blocked, but defered if of lower priority

Linux Kernel reads from a faked Interrupt Status Register instead from real Hardware

Timer functions

64 Bit timer register, resolutions:
~1 μ s on 8254 Timer, ~10ns on local APIC

RTL Application Examples



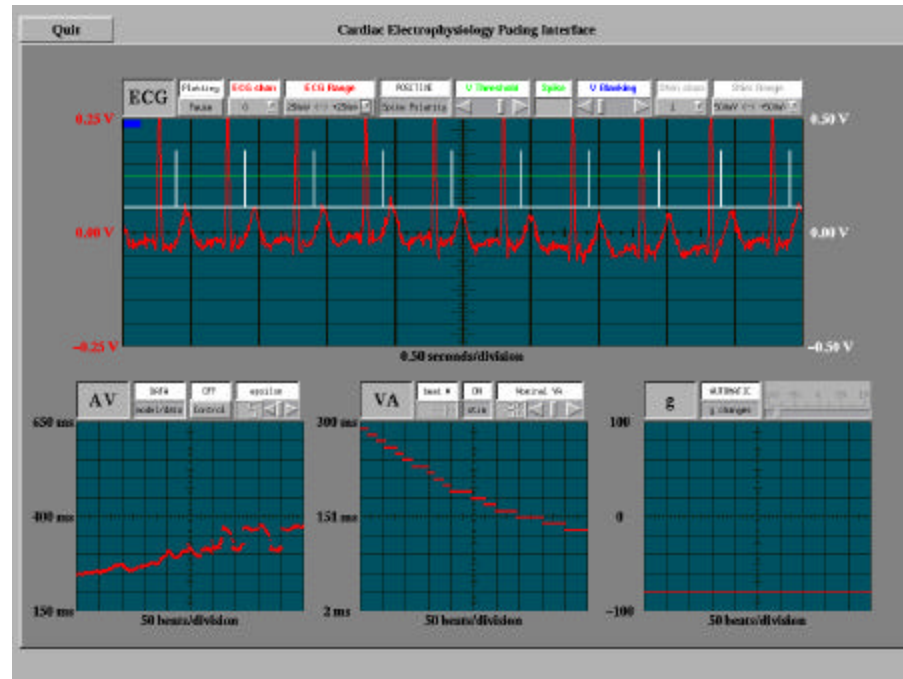
**Machine Control Systems / EMC-Projekt
(National Institute of Standards and Technology, USA)**

RTL Application Examples



**„The Hurricane Hunter“ / LIDAR
(C.W. Wright, NASA)**

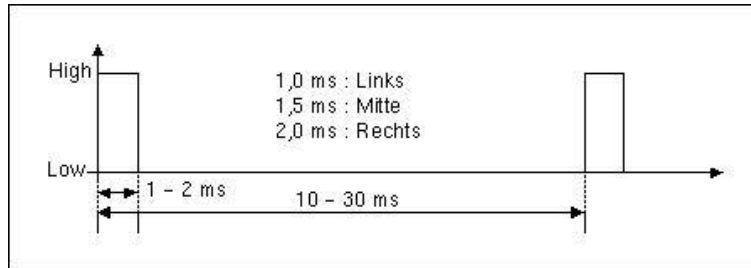
RTL Application Examples



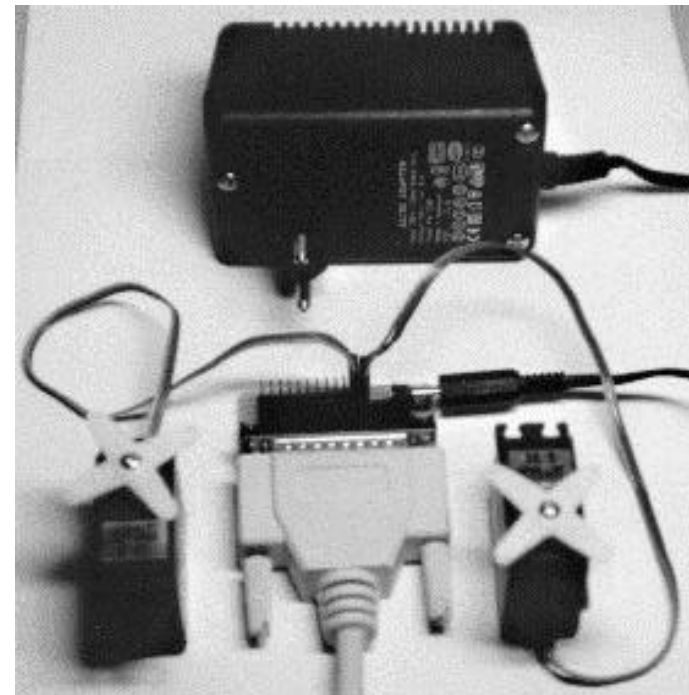
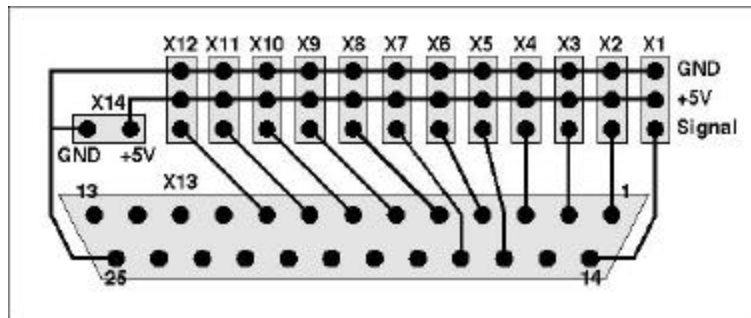
Data Acquisition / Cardiograms

(David Christini, Cornell University Medical Collage)

RTL Application Examples



timing critical signal



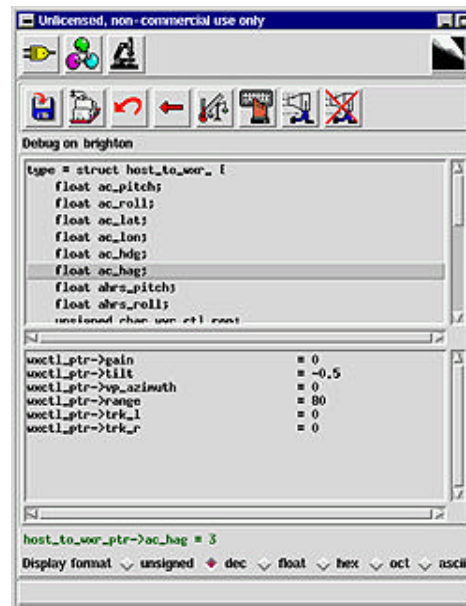
**DIY real time: signal generation for RC-Servos on the parallel port
(10 μ s equal 1 degree distortion)**

Extensions for RT-Linux

„Real Time Signals“ for User Space

Device Driver for several I/O-Boards (COMEDI)

Remote Real-time Data Debugger (R2D2)



RTL Ports and Legal Issues

Ports

i386

Alpha

PowerPC

Mips (Betatest)

M68K-no-MMU (RTL-0.9j for uClinux)

Multiple Licences / Patented Mechanism

Proprietary Licence for developing non-GPL applications necessary

controversial patent (deferred Interrupt)

RTAI/RTHAL

Origin

Philosophy

Architecture

Functionality

Applications examples

A Real Time Application Interface (RTAI) similar to RT-Linux, based on an Real Time Hardware Abstraction Layer (RTHAL)

RTAI Origins & Philosophy

Origin

DIAPM-RTOS (a DOS-TSR) ported as real time sub-kernel for Linux by Paolo Mantegazza, Dipartimento de Ingegneria Aerospaziale Politecnico di Milano)

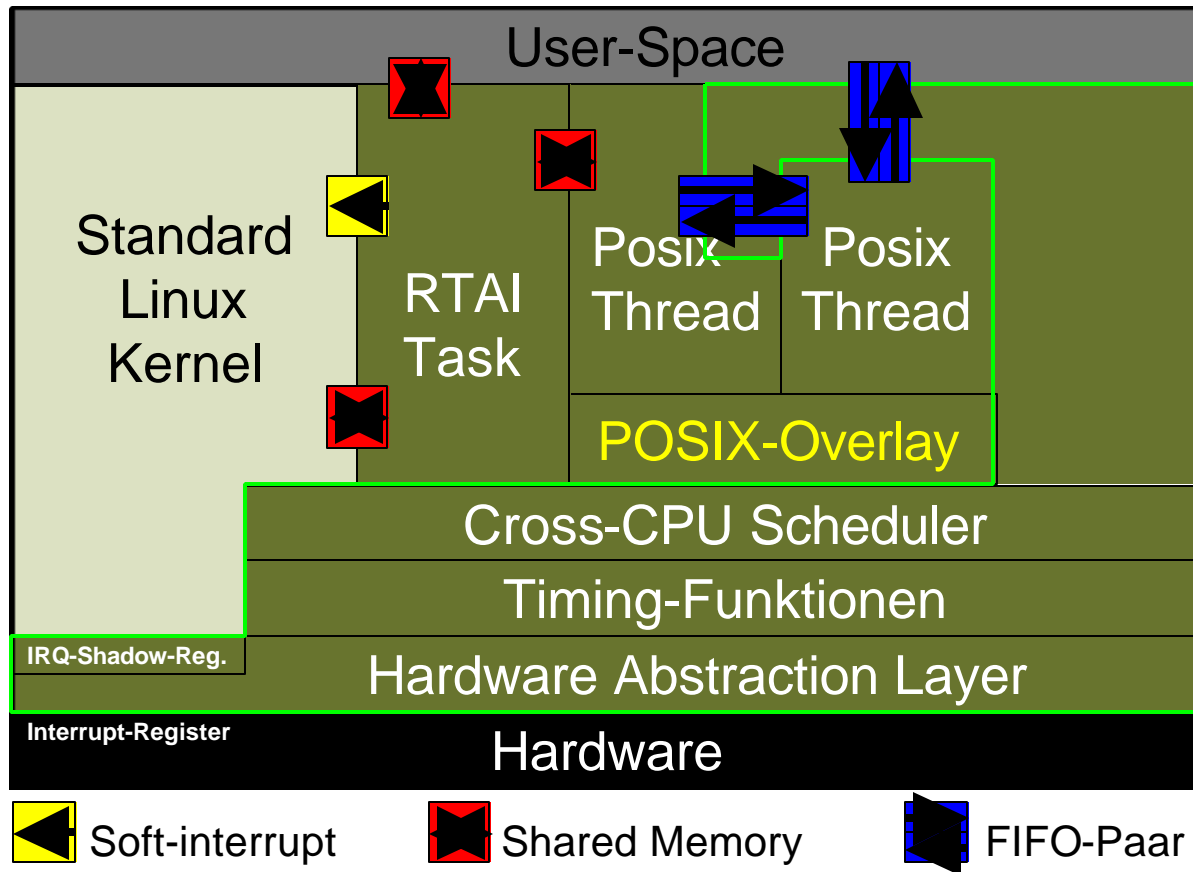
Philosophy

Hardware Abstraction Layer to simplify the sub-kernel integration

Functionality more important than posix compliance

RTAI is POSIX 1003.1b compliant by using an overlay module to the native API of RTAI

RTAI Overview



RTAI Functionality / System Calls

Scheduling (runtime loadable module)

Pre-emptiv priority cross CPU Scheduler

Task handling, Interrupts and Timer functions

Similar to RT-Linux: posix and native API

Inter Process Communication

FIFOs (RTL-Kompatibel)

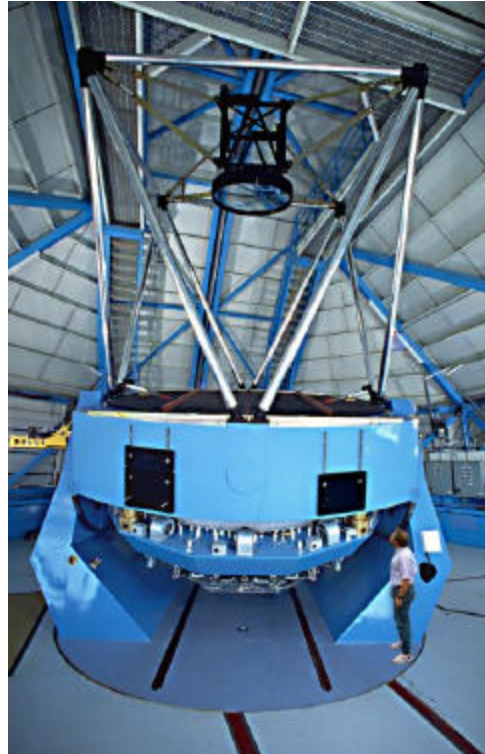
Shared-Memory (RTL/RTAI-Uniform)

Message Boxes, Queues, Semaphore, Mutexes

LXRT - Linux RealTime

RTAI-Applications in user space (Scheduler-Ext.)

RTL Application Examples



**Adjusting Mirrors of a 3.5m Telescopes
(P.N. Daly, NOAO, Tuscon)**

RTL Application Examples



**Software driven radio transmitter at 153 KHz
(via parallel port, 306.000 Interrupts per second)**

Web Resources

RTL Homepage: <http://www.rtlinux.org>

FSM-Labs (aktuelle RTL-Seiten): <http://www.fsmlabs.com>

RTAI Homepage: <http://www.rtai.org>

Real-Time Linux Homepage: <http://www.realtimelinux.org>

Linux Resource Kernel: <http://www.timesys.com>

RC Servo control: <http://www.linux-magazin.de> (11/1998 u. 03/2000)

RTAI in Detail

RTAI Features

POSIX 1003.1c compliant pthreads, mutexes and condition variables

POSIX 1003.1b compliant pqueues

Traditional RTOS IPCs--semaphores, mailboxes, FIFOs, shared memory

Uniprocessor, multiprocessor and symmetric multiprocessor schedulers

RTAI Features (continued)

Periodic and one-shot scheduling

Dynamic memory allocation from RT tasks

LXRT--Use same API from user space

/proc file interface

PERL bindings

FPU support

Hardware Abstraction Layer

RTHAL

```
struct rt_hal {
    struct desc_struct *idt_table;
    void (*disint) (void);
    void (*enint) (void);
    unsigned int (*getflags) (void);
    void (*setflags) (unsigned int flags);
    void (*mask_and_ack_8259A) (unsigned int irq);
    void (*unmask_8259A_irq) (unsigned int irq);
    void (*ack_APIC_irq) (void);
    void (*mask_IO_APIC_irq) (unsigned int irq);
    unsigned long *io_apic_irqs;
    void *irq_controller_lock;
    void *irq_desc;
    int *irq_vector;
    void *irq_2_pin;
    void *ret_from_intr;
} rthal;
```

Converting to RTAI

Linux (system.h)

```
#define __sti() __asm__ __volatile__ ("sti" : : "memory")  
#define __cli() __asm__ __volatile__ ("cli" : : "memory")
```

RTAI (system.h)

```
#define __sti()      (rthal.enint())  
#define __cli()      (rthal.disint())
```

Linux (irq.c)

```
mask_and_ack_8259A(irq);
```

RTAI (irq.c)

```
rthal.mask_and_ack_8259A(irq);
```

Basic Modules of RTAI

RTAI

Schedulers

UP - Uniprocessor

SMP - Symmetric multiprocessor

Tasks run on any processor

MUP - Multi-uniprocessor

Tasks assigned to specific processors

RT FIFO

Shared Memory

Extended RTAI Modules

LXRT

POSIX threads

POSIX queues

Dynamic memory management

Module RTAI

Base module for real-time Linux services

Takes over Linux interrupt handling

Replaces Linux entries in rthal with its own functions

Provides services for interrupt handlers
and timers

RTAI Services

```
#include <rtai.h>
```

```
void rt_mount_rtai (void);
```

```
void rt_umount_rtai (void);
```

```
int rt_request_global_irq (unsigned int irq, void (*handler)(void));
```

```
int rt_free_global_irq (unsigned int irq);
```

```
int rt_request_linux_irq (unsigned int irq,
```

```
void (*linux_handler)(int irq, void *dev_id, struct pt_regs  
regs),
```

```
char *linux_handler_id, void *dev_id);
```

```
int rt_free_linux_irq (unsigned int irq, void *dev_id);
```

```
void rt_pend_linux_irq (unsigned int irq);
```

```
int rt_request_srq (unsigned int label, void (*rtai_handler)(void),
```

```
long long (*user_handler)(unsigned int whatever))
```

```
int rt_free_srq (unsigned int srq);
```

```
void rt_pend_linux_srq (unsigned int srq);
```

```
void rt_request_timer (void (*handler)(void), int tick);
```

```
void rt_free_timer (void);
```

Module Scheduler

Manages task scheduling

Supports multiple processors

Provides services for:

- Timers

- Semaphores

- Simple task-to-task messaging

- Remote procedure calls (RPC)

- Mailboxes

Scheduler Services Tasking

```
#include <rtai_sched.h>
```

```
int rt_task_init (RT_TASK *task, void (*rt_thread)(int),  
                 int data, int stack_size, int priority,  
                 int uses_fpu, void (*signal)(void));
```

```
int rt_task_delete (RT_TASK *task);
```

```
int rt_task_get_state (RT_TASK *task);
```

```
RT_TASK *rt_whoami (void);
```

```
void rt_task_signal_handler (RT_TASK *task, void (*signal)(void));
```

```
int rt_task_use_fpu (RT_TASK *task, int yes_no);
```

```
int rt_linux_use_fpu (int yes_no);
```

```
void rt_preempt_always (int yes_no);
```

```
void rt_task_yield (void);
```

```
int rt_task_suspend (RT_TASK *task);
```

```
int rt_task_resume (RT_TASK *task);
```

```
int rt_task_make_periodic (RT_TASK *task, RTIME start, RTIME period);
```

```
void rt_set_runnable_on_cpus (RT_TASK *task, unsigned int cpu_mask);
```

Periodic vs. One-shot Timing Modes

Periodic Mode

Set timer period once. Don't reprogram at each tick interrupt.

All tasks scheduled as multiples of timer tick.

One-shot Mode

Tasks can be scheduled with arbitrary precision

Must reprogram timer at each interrupt

Scheduler Services Timing

```
void rt_set_oneshot_mode (void);
void rt_set_periodic_mode (void);
RTIME rt_start_timer (int period);
void rt_stop_timer (void);
RTIME rt_get_time (void);
RTIME rt_get_time_ns (void);
RTIME rt_get_cpu_time_ns (void);
void rt_task_wait_period (void);
RTIME rt_next_period (void);
void rt_busy_sleep (int nanosecs);
void rt_sleep (RTIME delay);
void rt_sleep_until (RTIME time);
RTIME count2nano (RTIME timercounts);
RTIME nano2count (RTIME nanosecs);
```

Scheduler

Services Semaphore

```
void rt_sem_init (SEM *sem, int value);  
int rt_sem_delete (SEM *sem);  
int rt_sem_signal (SEM *sem);  
int rt_sem_wait (SEM *sem);  
int rt_sem_wait_if (SEM *sem);  
int rt_sem_wait_until (SEM *sem, RTIME time);  
int rt_sem_wait_timed (SEM *sem, RTIME delay);
```

Scheduler

Services Messaging

```
RT_TASK *rt_send (RT_TASK *task, unsigned int msg);  
RT_TASK *rt_send_if (RT_TASK *task, unsigned int msg);  
RT_TASK *rt_send_until (RT_TASK *task, unsigned int msg, RTIME time);  
RT_TASK *rt_send_timed (RT_TASK *task, unsigned int msg, RTIME delay);  
  
RT_TASK *rt_receive (RT_TASK *task, unsigned int msg);  
RT_TASK *rt_receive_if (RT_TASK *task, unsigned int msg);  
RT_TASK *rt_receive_until (RT_TASK *task, unsigned int msg, RTIME time);  
RT_TASK *rt_receive_timed (RT_TASK *task, unsigned int msg, RTIME delay);
```

Scheduler Services RPC

```
RT_TASK *rt_rpc (RT_TASK *task, unsigned int to_do, unsigned int *result);  
RT_TASK *rt_rpc_if (RT_TASK *task, unsigned int to_do, unsigned int *result);  
RT_TASK *rt_rpc_until (RT_TASK *task, unsigned int to_do, unsigned int *result,  
    RTIME time);  
RT_TASK *rt_rpc_timed (RT_TASK *task, unsigned int to_do, unsigned int *result,  
    RTIME delay);  
int rt_isrpc (RT_TASK *task);  
RT_TASK *rt_return (RT_TASK *task, unsigned int result);
```

Scheduler Services Mailbox

```
int rt_mbx_init (MBX *mbx, int size);
```

```
int rt_mbx_delete (MBX *mbx);
```

```
int rt_mbx_send (MBX *mbx, void *msg, int msg_size);
```

```
int rt_mbx_send_wp (MBX *mbx, void *msg, int msg_size);
```

```
int rt_mbx_send_if (MBX *mbx, void *msg, int msg_size);
```

```
int rt_mbx_send_until (MBX *mbx, void *msg, int msg_size, RTIME time);
```

```
int rt_mbx_send_timed (MBX *mbx, void *msg, int msg_size, RTIME delay);
```

```
int rt_mbx_receive (MBX *mbx, void *msg, int msg_size);
```

```
int rt_mbx_receive_wp (MBX *mbx, void *msg, int msg_size);
```

```
int rt_mbx_receive_if (MBX *mbx, void *msg, int msg_size);
```

```
int rt_mbx_receive_until (MBX *mbx, void *msg, int msg_size, RTIME time);
```

```
int rt_mbx_receive_timed (MBX *mbx, void *msg, int msg_size, RTIME delay);
```

LXRT

Allows Linux Processes to use RTAI APIs

Develop your RT tasks as protected mode processes

Move to RTAI when working

Supports messaging, semaphores and mailboxes

LXRT Services

```
#include <rtai_lxrt.h>
```

Process Functions

```
LX_TASK *rt_task_init (unsigned int name, int prio, int stack_size);  
SEM *rt_sem_init (unsigned long name, int initial_count);  
MBX *rt_mbx_init (unsigned long name, int buf_size);
```

Task Functions

```
int rt_register (unsigned long name, void *adr);  
int rt_drg_on_adr (void *adr);  
int rt_drg_on_name (unsigned long name);
```

Utilities

```
void *rt_get_adr (unsigned long name);  
unsigned long rt_get_name (void *adr);  
unsigned long nam2num (char *name);
```

Hard Real-time from User Space

```
void rt_make_hard_real_time (void)
void rt_make_soft_real_time (void)
void rt_allow_nonroot_hrt (void)
void print_to_screen (const char *format, ...)
```

Module RT FIFOs

Point-to-point communication

Between RT tasks and Linux processes

Blocking for Synchronization

Processes/Tasks need not poll

Linux character devices, /dev/rtf0..63

Processes use read() and write()

RT Task uses rtf_get() and rtf_put()

RT FIFO Services

RT Task Services

```
#include <rtai_fifos.h>
```

```
int rtf_create (unsigned int fifo, int size);
```

```
int rtf_destroy (unsigned int fifo);
```

```
int rtf_reset (unsigned int fifo);
```

```
int rtf_resize (unsigned int fifo, int size);
```

```
int rtf_put (unsigned int fifo, void *buf, int count);
```

```
int rtf_get (unsigned int fifo, void *buf, int count);
```

```
int rtf_create_handler (unsigned int fifo, int (*handler)(unsigned int fifo);
```

Misc. FIFO Functions

Semaphores

```
int rtf_sem_init (unsigned int fd, int init_val)
int rtf_sem_wait (unsigned int fd)
int rtf_sem_trywait (unsigned int fd)
int rtf_sem_timed_wait (unsigned int fd, ms_delay)
int rtf_sem_post (unsigned int fd)
int rtf_sem_destroy (unsigned int fd)
```

Named FIFOs

```
int rtf_create_named (const char *name)
int rtf_get_fifo_by_name (const char *name)
```

Shared Memory

Data not queued

Application must define handshake protocol

Blocking not directly supported

Not point-to-point

Accessible by any number of processes/tasks

Supports very large data items

Shared Memory API

```
#include <rtai_shm.h>
```

Kernel Functions

```
void *rtai_kmalloc (unsigned long name, int size);  
void rtai_kfree (unsigned long name);
```

Process Functions

```
void *rtai_malloc (unsigned long name, int size);  
void rtai_free (unsigned long name, void *adr);
```

Utilities

```
unsigned long nam2num (const char *name);  
void num2nam (unsigned long num, char *name);
```

Dynamic Memory Allocation

```
void *rt_malloc (int size);  
void rt_free (void *addr);
```

/proc Support in RTAI

In directory /proc/rtai:

rtai

scheduler

fifos

memory_manager

POSIX and Real-time

1003.1b Real-time extensions

Priority scheduling

Process memory locking

Semaphores

Shared memory

Clocks & Timers

Messages

1003.1c Threads

1003.1d Further real-time extensions

POSIX Threads

Multiple *strands* of execution within one process

Creating a thread is more efficient than *fork()*

Threads share the resources of their creating process

Portable API for real-time services

POSIX and RTAI

Full Pthread implementation including:

Mutexes

Condition variables

Message Queues (1003.1b)

Scheduler

Scheduler and RT tasks = one POSIX process. Tasks = threads

Wrapper around native API

Thread API

```
int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(* start_routine) (void *),  
void *arg);  
void pthread_exit (void *retval);  
pthread_t pthread_self (void);  
int sched_yield (void);
```

```
int pthread_attr_init (pthread_attr_t *attr);  
int pthread_attr_destroy (pthread_attr_t *attr);  
int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);  
int pthread_attr_getdetachstate (const pthread_attr_t *attr, int *detachstate);  
int pthread_attr_setschedparam (pthread_attr_t *attr, const struct sched_param *param);  
int pthread_attr_getschedparam (const pthread_attr_t *attr, struct sched_param *param);  
int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy);  
int pthread_attr_getschedpolicy (const pthread_attr_t *attr, int *policy);  
int pthread_attr_setinheritsched (pthread_attr_t *attr, int inherit);  
int pthread_attr_getinheritsched (const pthread_attr_t *attr, int *inherit);  
int pthread_attr_setscope (pthread_attr_t *attr, int scope);  
int pthread_attr_getscope (const pthread_attr_t *attr, int *scope);
```

Mutex

Like Binary Semaphore

Implements Priority Inheritance

Three kinds in Linux

- Fast

- Recursive

- Error checking

Mutex API

```
int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t  
    *mutex_attr);
```

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
```

```
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
```

```
int pthread_mutexattr_setkind_np (pthread_mutexattr_t *attr, int kind);
```

```
int pthread_mutexattr_getkind_np (const pthread_mutexattr_t *attr, int *kind);
```

```
int pthread_setschedparam (pthread_t thread, int policy, const struct  
    sched_param *param);
```

```
int pthread_getschedparam (pthread_t thread, int *policy, struct sched_param  
    *param);
```

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

Conditional Variable

Like a Counting Semaphore

Threads wait for an event to occur

Another thread signals that the event has occurred

Requires a mutex to avoid race conditions

Conditional Variable API

```
int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t  
                      *cond_attr);
```

```
int pthread_cond_destroy (pthread_cond_t *cond);
```

```
int pthread_condattr_init (pthread_condattr_t *attr);
```

```
int pthread_condattr_destroy (pthread_condattr_t *attr);
```

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
```

```
int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex,  
                            const struct timespec *abstime);
```

```
int pthread_cond_signal (pthread_cond_t *cond);
```

```
int pthread_cond_broadcast (pthread_cond_t *cond);
```

Message Queues

FIFOs that queue discrete messages

Like files but have their own API

Threads block on empty or full queue unless
O_NONBLOCK is specified

Messages have a priority and are received in priority
order

Message Queue API

```
mqd_t mq_open (char *mq_name, int oflags, mode_t permissions, struct
                mq_attr *mq_attr);
size_t mq_receive (mqd_t mq, char *msg_buffer, size_t buflen,
                  unsigned int *msgprio);
int mq_send (mqd_t mq, const char *msg, size_t msglen, unsigned int
             msgprio);
int mq_close (mqd_t mq);

int mq_getattr (mqd_t mq, struct mq_attr *attrbuf);
int mq_setattr (mqd_t mq, const struct mq_attr *new_attrs, struct mq_attr
               *old_attrs);
int mq_notify (mqd_t mq, const struct sigevent *notification);
int mq_unlink (mqd_t mq);
```



L I N E OTM

The Embedded Linux[®] Solutions CompanyTM