



**RTLinux V3.0:**  
**a POSIX 1003.13 PE51 OS**  
**with a POSIX 1003.13 PE54 thread**

[yodaiken@fsmlabs.com](mailto:yodaiken@fsmlabs.com)

[www.fsmlabs.com](http://www.fsmlabs.com) *and* [www.rtlinux.com](http://www.rtlinux.com)

This talk is copyright FSMLabs, Inc.

## Realtime versus Time Shared



1. *Time sharing software*: switch between different tasks fast enough to create the illusion that all are going forward at once.
2. *Realtime software*: switch between different tasks in time to meet deadlines.



# Hard realtime

---

1. *Predictable performance* at each moment in time: not as an average.
2. *Low latency* response to events.
3. *Precise scheduling* of periodic tasks.



## Soft realtime

---

- *Good average case performance*
- *Low deviation from average case performance*



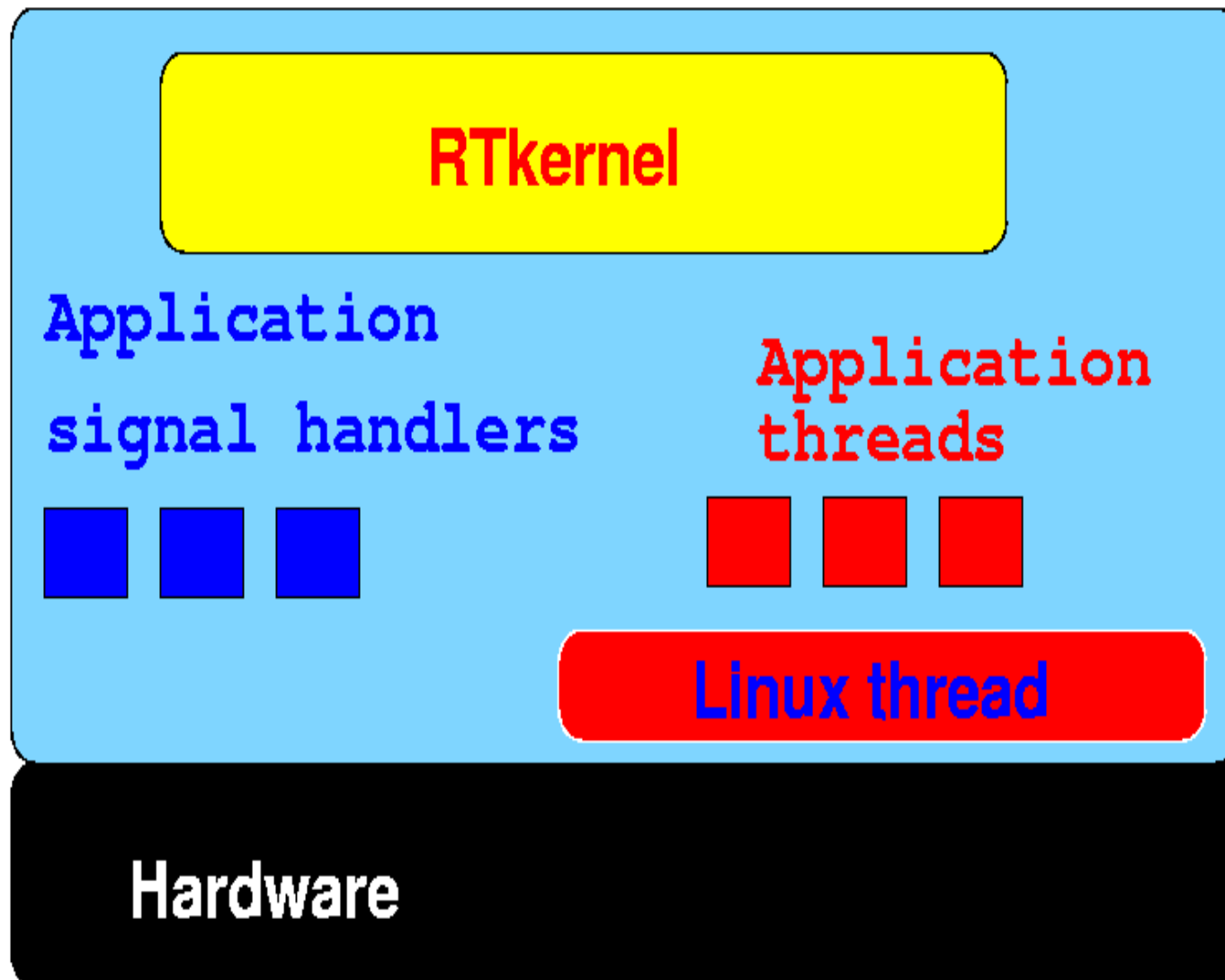
## The RTLinux operating system

---

A small hard realtime operating system that runs Linux as its lowest priority task. Used for everything from making chain-saw chains, to switching packets to animating movies.

# One view of RTLinux

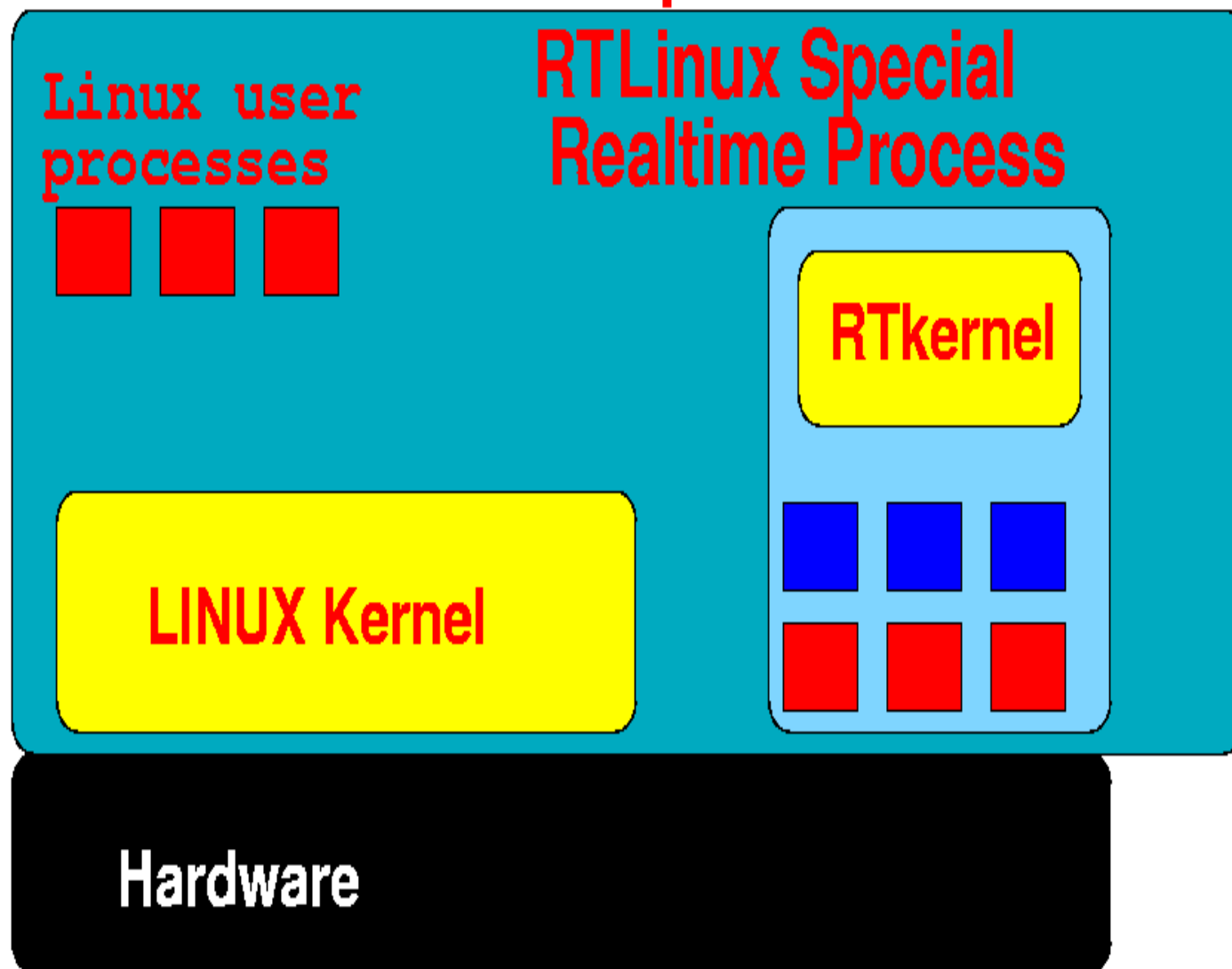
## RTLinux Realtime Minimal POSIX OS



## Another view of RTLinux

---

### Linux with Realtime process





## RTLinux is decoupled from Linux

---

RTLinux schedules itself — Linux scheduler is not involved.

Linux cannot delay or interrupt execution of RTLinux



## Hard realtime

---

Predictable performance, low latency, precise scheduling.

1. *Low latency* response to events. 5microseconds or less (AMD SC520 133MHz).
2. *Precise scheduling* of periodic tasks. 8 microseconds or less error.(AMD SC520 133MHz).
3. *On a standard PC: 15 microseconds and 20 microseconds*



## Programming model

---

1. For Realtime tasks and event handlers: POSIX threads & signals.
2. For Linux processes connection via POSIX I/O, shared memory, and signals.

As standard as possible with no efficiency loss.



## Programming view

---

1. The hard realtime component of realtime applications should run in a simple, minimal, predictable environment.
2. The non-realtime components should run in UNIX until something better is invented.
3. Hard realtime and non-realtime should be decoupled in operation.



## Why is decoupling so important?

---

1. Mars Lander almost broke because a low priority non-realtime process was able to lock a resource needed by a critical realtime task. The lock was hidden in complex code shared by all tasks, realtime and non-realtime.
2. RTLinux makes all interactions between RT and non-RT explicit and *transparent* .



## RTLinux Version 3.0

---

1. V1 was a research project.
2. V2 was the first production version.
3. V3 is the first industrial strength version.



## RTLinux Version 3.0

---

1. Production versions for IA32 (Intel,AMD ...), Alpha, PowerPC and beta version for MIPS.
2. Developed POSIX threads and signal API.
3. User space `rtlinux_sigaction` to invoke RT components.
4. SMP enhancements and RTiC Labs graphical front end.

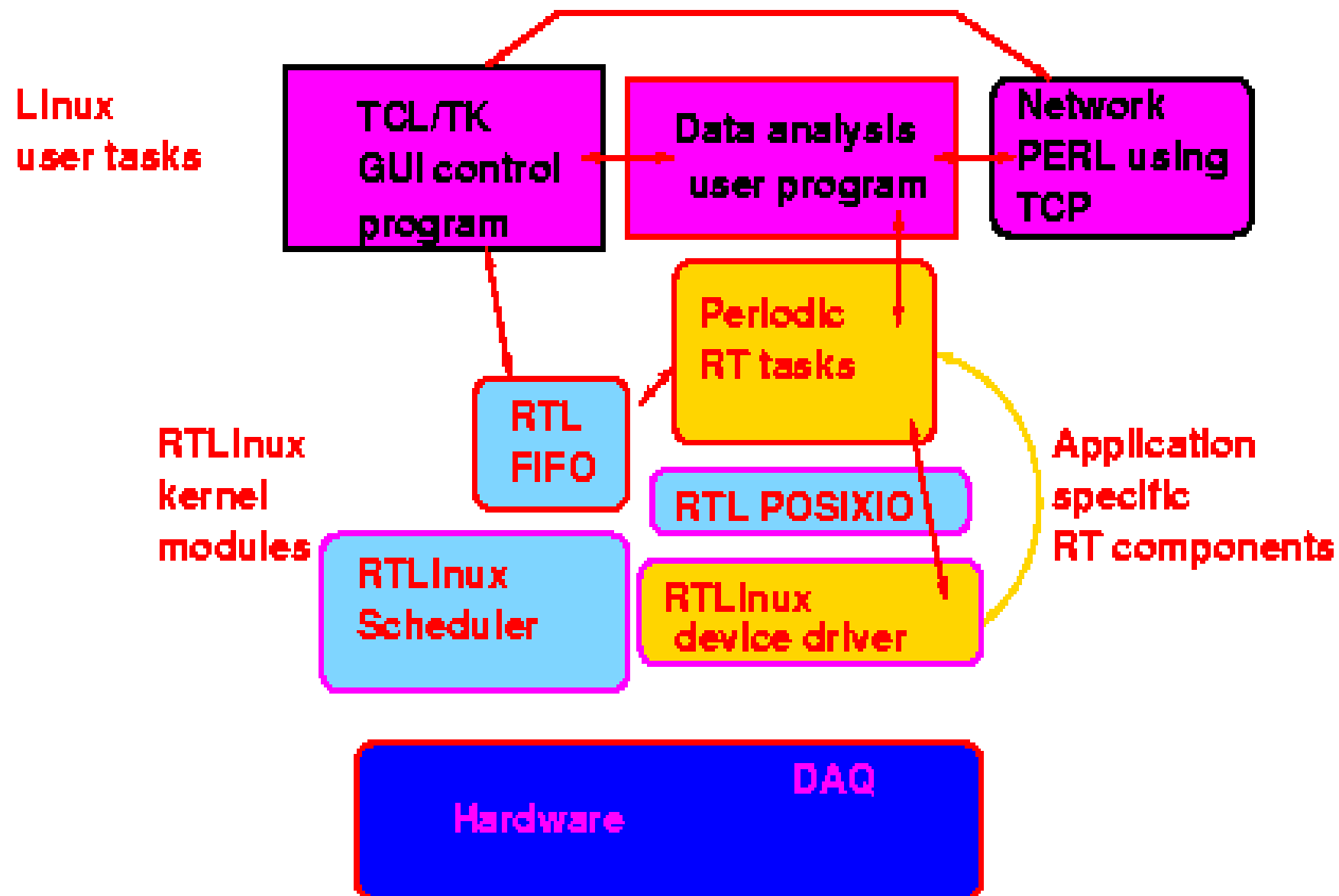


# POSIX 1003.13

---

1. PS51: minimal realtime system
2. PS52, PS53 — waffling.
3. PS54: Full POSIX with `sched_setsched` soft RT extensions.

## A typical simple application





# The simplest user side data logging program

---

```
cat < /dev/rtf0 > logfile
```



# RT modules are Linux kernel modules

---

INCLUDES

GLOBAL VARIABLES AND DEFINITIONS

INITIALIZE CODE

CLEANUP CODE

REALTIME THREAD CODE



## A RT thread

---

```
void *my_code(void *arg){
    struct timespec t; struct mydata D;
    clock_gettime(CLOCK_RTL_SCHED,&t);
    while(!stop){
        copy_device_data(&D.d);
        write(fd,&D,sizeof(D));
        timespec_add_ns(&t,DELAY_NS);
        clock_nanosleep(CLOCK_RTL_SCHED, TIMER_ABSTIME, &t,NULL);
    }
    return (void *)&stop;
}
```



## Initialization I

---

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



## Initialization II

---

```
int init_module(void) {
    fd = open("/dev/rtf0", O_WRONLY | O_NONBLOCK | O_CREAT );
    if (fd < 0)
    {
        rtl_printf("Example cannot open fifo\n");
        rtl_printf("Error number is %d\n", errno);
        return -1;
    }
}
```



## Initialization II

---

```
if( pthread_create(&T, NULL, my_code, NULL) )
{
    close(fd);
    rtl_printf("Cannot create thread\n");
    return -1;
}
return 0;
}
```



## Cleanup

---

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



## Sending signals to Linux

---

```
// Send interrupt #irq to the Linux thread  
pthread_kill(LinuxThread(),RTL_LINUX_MIN_SIGNAL+ irq);
```



## Installing an interrupt handler

---

```
// catch or ignore hardware interrupt "irq"  
sigaction(RTL_SIGIRQMIN+irq, &sig_new, &sig_old);
```



## Extension: SMP processor ids

---

```
// Determine cpu to run thread on  
pthread_attr_setcpu(thread,cpu_id);
```



## Extension: Optional floating point

---

```
// Enable FP state save/restore  
pthread_attr_setfp_np (pthread_attr_t *attr, int flag);
```



## Extension: Simplified periodic thread setup

---

```
// Run at (now+ start)+ N*period
int pthread_make_periodic_np(pthread_t p, hrtime_t start,\
    hrtime_t period);
// wait until next period.
int pthread_wait_np(void);
```



## Extension to PS54: RT signal handlers

---

```
// Run the signal function in RT mode in user address space
int rtlinux_sigaction(int, struct rtlinux_sigaction *,
                     struct rtlinux_sigaction *);
```



## Complaints: condvars and cancel

---

- Cancel slows down everything. Async cancel makes no sense anyways.

- Condvars are overdesigned.

Want: `SleepIfCondZero(Rendezvous_t *r);`



## Complaints: SMP is weakly specified.

---

- Too many extensions needed.
- BUT: optional “shared” is brilliant.



## Complaints: Periodic scheduling is too wordy.

---

- "Sporadic" is very non-simple.
- Maybe too much messing around with timespecs.



## Low end examples.

---

1. Replace a digital joystick with an analog joystick and a sound card.
2. Reduce control loop times by sampling A/D at 100 microseconds.
3. Log data over the network using Linux utilities with no software development costs.
4. Use Apache standard web server as a control interface – with no software development costs.



## Larger examples.

---

1. Multiple software "Virtual routers" operating on packets in realtime.
2. Interactive robot control with non-RT graphical interface.



## The RTLinux technical synergy.

---

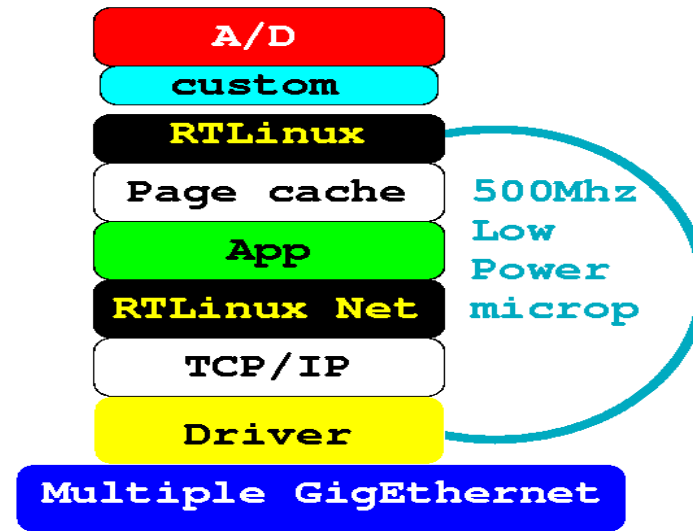
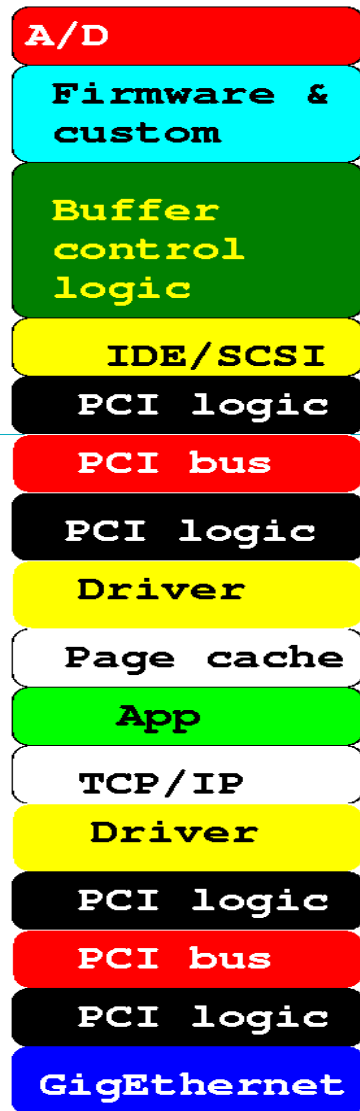
1. Highly efficient realtime.
2. Connected to a reliable and powerful networked operating system (Linux).
3. Running on standard PCs, servers, and embedded hardware.



## The RTLinux cost advantage.

---

1. The costs of maintaining general purpose software – data bases, networks, graphics, development, ... – are shared with server and desktop companies.
2. Prototype on PCs.
3. Access to source is essential for important applications.



Legacy vs. RTLinux disk drives  
Serving data to a network



# Finite State Machine Labs

---

1. Core kernel development and GPL and non-GPL RTLinux distributions.
2. Training, engineering services and product support.
3. Application software in communications and factory automation.



[www.fsmlabs.com](http://www.fsmlabs.com)

---



[www.rtlinux.com](http://www.rtlinux.com)

---