Real-Time Linux
Applications and Design

Victor Yodaiken

Michael Barabanov

New Mexico Institute of Technology

yodaiken@nmt.edu

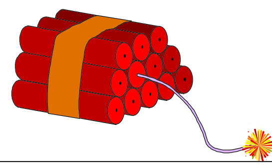
http://luz.nmt.edu/~rtlinux

Outline of the talk

- The meaning of “real-time”.
- The purpose of RT-Linux
- Writing applications for RT-Linux?
- How it works and limitations.
- Future directions.

I. What does “real-time” mean?

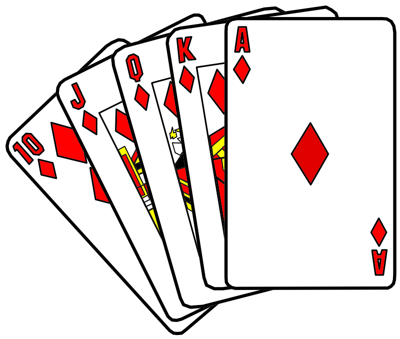
- For marketing folks real-time means **fast.**
- **Soft real-time** means that the program must ***usually*** run at some rate. For example a video player can miss frames now and then but not too often.
- **Hard-real-time** means that timing is critical and deadlines **cannot** be missed.



Hard-Real-Time requires

- Predictability: a real-time task cannot tolerate much variability in response to interrupts or in scheduling.
- Low latency (fast response)

Predictability



- If the OS can disable interrupts for critical regions, as in standard Linux and most other operating systems, then timing of tasks is not predictable.

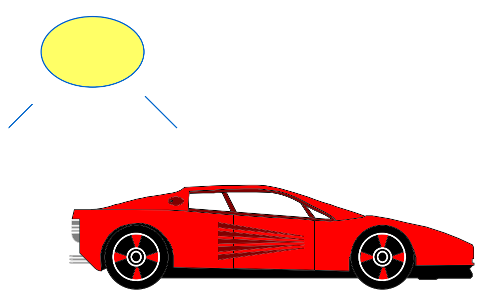
Lack of predictability

- Many A/D boards are now advertised as including a FIFO buffer so that ``most configurations'' of MS Windows will not lose samples.

II. Purpose
RT-Linux is aimed primarily at

- Lab equipment! PCs controlling instruments or sampling sensors are found in almost every science and engineering lab.
- Embedded Systems. Robots, engines, telescopes, even set-top boxes.

Example: Embedded Systems

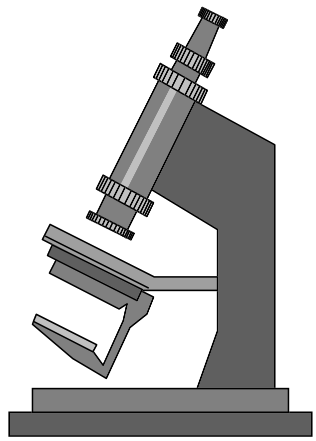


We are currently making a control system for a car that will be an entry in the solar car race this year.

This will be the first solar race car/web server

Examples: Instrumentation

- Most of the applications we have heard about have been for data acquisition.
- A physiology lab is sampling cardiological function.
- We have a slow scope and signal generator.



Linux offers

- X-windows, TCL/TK etc.
- Networking
- Compilers
- GNU utilities
- Great support and source code
  - Source code may allow validation
- Rapid development and big user base

So what we want is

- Hard-real time tasks, both periodic and interrupt driven.
- Access to all the tools and services that we have become accustomed to use on Linux so that we can develop programs, display and analyze data, and use the network.

The purpose of RT-Linux is to mix two incompatible properties

- **Hard real-time** service: predictable, fast, low latency, simple scheduler
- All the services of standard Posix: GUI, TCP/IP, NFS, compilers, web-servers, …

In the same operating system

What's incompatible?

Real-Time OS	**Full-Featured OS**
Optimize worst case	Optimize average case
Predictable schedule	Efficient schedule
Simple executive	Wide range of service
Minimize latency	Maximize throughput

Nevertheless, it works

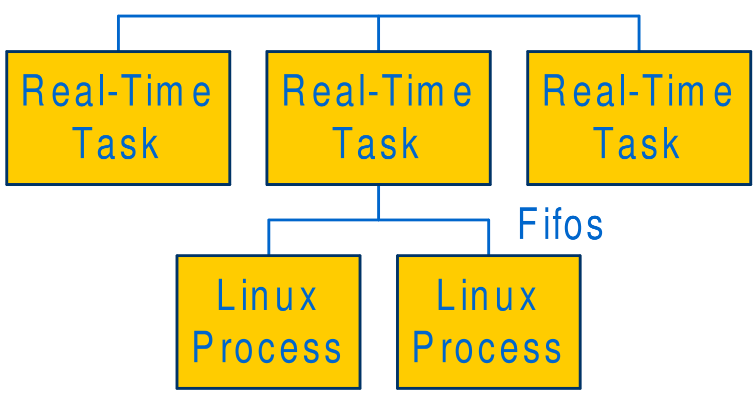
- An interrupt driven sound sampling application at 8KHz on a 486/33.
- Reported data acquistion performance better than DOS (no operating system). Sampling at 3KHz on a 33MHz/486 with a low-cost A/D board connected to the serial port --- while driving a Motif display and logging data to disk.

(application of Harald Stauss, Humbolt University)

III. Using Real-Time Linux

- RT-Linux applications are usually made up of two components.
  - A hard-real-time component that consists of 1 or more real-time tasks.
  - A non-real-time component that consists of 1 or more ordinary Linux processes or thread.
- Linux processes and real-time tasks communicate via special fifos or shared memory

Organization of a real-time application



A signal generator application
(developed by Bill Crum on a 486/33)

- Two periodic RT-tasks (period @800 μ s) Each generates points on its own D/A channel driving a ‘scope. Each task can generate a canned square, triangular, or sine wave.
- TCL/TK user programs display push-buttons used to select wave patterns. Commands are sent over fifos to the real-time tasks.

Coding

- The real-time components are coded in standard Linux loadable kernel modules.
- User processes make system calls to create, read, and write fifos. The fifos are specially designed to avoid the dreaded **priority inversion problem.**

The task module contains

- Initialization code.
  - Initialize task structures with the **rt_task_init** call. This fills in the task structure and allocates memory, stack, and FIFO.
  - Schedule task structures either by attaching to an interrupt, or by attaching to the periodic scheduler.
- Code and data for the tasks.

Example initialization

```
int init_module(void){
  RTIME now = rt_get_time();
  rt_task_init(&mytask1, wave_handler, 1, 3000, 5);
  rt_task_init(&mytask2, wave_handler, 2, 3000, 5);
  rt_task_make_periodic(&mytask2, now,993);
  rt_task_make_periodic(&mytask1, now+3000,993);
  return 0; }
```

Create two tasks with period 993 = @800µs

Task code for signal generator

- while(1) {

```
if (rt_fifo_get(t,&command,1) > 0)
   outdev(PORT, next(command));
 rt_task_wait();
}
```

- This is a much simplified version!

The user part of the signal generator

- Basic idea is to write a standard Linux application made up of a TCL/TK front end and a collection of very simply C programs that initialize the fifos and that send commands to the RT-tasks.

TCL/TK user program

```
frame .f1 -relief groove -borderwidth 3
frame .f2 -relief groove -borderwidth 3

label .f1.l1 -text " Channel 1 "
label .f2.l2 -text " Channel 2 "
button .f1.widget1 -text " sine wave " -command { exec ./sinewave 1 1}
button .f1.widget2 -text "square wave" -command { exec ./sinewave 1 3}
button .f1.widget3 -text " sawtooth  " -command { exec ./sinewave 1 2}
button .f1.widget4 -text " flatline  " -command { exec ./sinewave 1 0}
button .f1.widget5 -text "   exit    " -command { exec rmmod rt_process.o
exit }
```

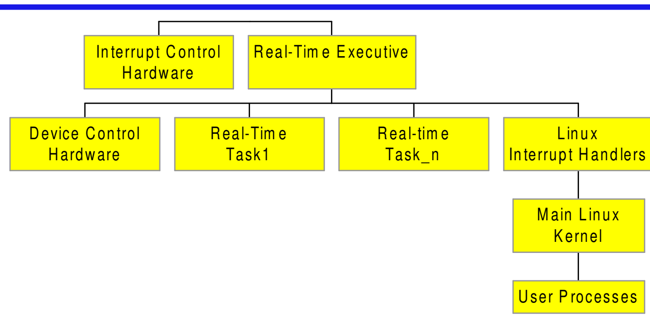
Program to send commands

```
int main(int argc, char **argv){
   char outbyte;
   int fifo;
   fifo = atoi (argv[1]);
   outbyte = (char) atoi (argv[2]);
   rt_fifo_write(fifo,&outbyte, 1);
   exit(0);   }
```

IV. How does RT-Linux work?

- The basic idea is that Linux code that disables and enables interrupts is rewritten to disable and enable **soft** interrupts.
- Hard interrupts are caught by the real-time executive. It passes these on to Linux if Linux is handling the interrupt and if Linux is enabling interrupts.
- Interrupts to real-time tasks -- and the timer -- cannot be disabled by Linux

Organization of RT-Linux



Changes to Linux

- The lowest level interrupt handlers are changed to handle soft enable/disable
- CLI/STI are replaced by S_CLI and S_STI
- Real-time clock handler tracks time.
- RT scheduler is a loadable kernel module
- RT tasks are loadable kernel modules

Linux device drivers work as usual (unless they do something they should not)

Current Operation of RT-Linux

- All resources for RT-Tasks are statically allocated. Memory, fifos, and processor time is fixed at task creation.
- Real-time tasks are interruptible and pre-emptable (RT tasks may disable interrupts)
- Communication with user tasks via non-blocking channels.

V. Future Directions

- Different scheduling algorithms.
  - Rate monotonic with automatic analysis
  - Dynamic scheduling especially for QOS
- Optimizations in the code
- Static analysis tools and testing support
- Ports to other architectures
- A large collection of libraries
- Better documentation.
- Inclusion in the Linux distribution?