The RTLinux Manifesto

Victor Yodaiken

Department of Computer Science
New Mexico Institute of Technology
Socorro NM 87801
yodaiken@cs.nmt.edu
http://www.rtlinux.org

ABSTRACT

RTLinux is the *hard realtime* variant of Linux that makes it possible to control robots, data acquisition systems, manufacturing plants, and other time-sensitive instruments and machines.

1 Introduction

Real-Time Linux (RTLinux) is a version of Linux that provides *hard real time* capability. A NASA computer running RTLinux flew into the eye of Hurricane Georges to collect data[14]; the Jim Henson Creature Shop in Hollywood is developing a RTLinux application to control "animatronic" things used in movies; RTLinux has been used for video editors, PBXs, robot controllers, machine tools, and even to stop and start hearts in medical experiments[1]

RTLinux provides the capability of running special realtime tasks and interrupt handlers on the same machine as standard Linux. These tasks and handlers execute when they need to execute no matter what Linux is doing. The worst case time between the moment a hardware interrupt is detected by the processor and the moment an interrupt handler starts to execute is under 15 microseconds on RTLinux running on a generic x86. A RTLinux periodic task runs within 35 microseconds of its scheduled time on the same hardware. These times are hardware limited, and as hardware improves RTLinux will also improve. Standard Linux takes up to 600 microseconds to start a handler and can easily be more than 20 milliseconds (20,000 microseconds) late for a periodic task[2]. As an unfair but fun comparison, an optimistic study of MS-Windows/NT didn't even bother to try to measure times under a millisecond and still found that NT numbers were essentially the same as the standard Linux numbers, while Windows/98 was up to 140 milliseconds too late on a periodic task [7]. To be fair, there are now Window-NT versions of the RTLinux method and these seem to get low level timings that are sometimes almost as good and generally not more than two times worse than RTLinux[6].

What makes RTLinux useful is that it extends the standard UNIX programming environment to realtime problems. RTLinux realtime interrupt handlers and tasks can be connected to ordinary Linux processes – either via a device interface where Linux processes read/write data, or via shared memory. A standard Linux pro-

[1] RTLinux is released, as is, with no warranty of any kind. Use at your own risk.

[2] A process using `sched_setsched`

cess, perhaps executing a shell script or a Perl program, can collect data from a realtime handler or task, process and log it and display the results on X-Windows. Using Perl scripts to to control a realtime device from an ordinary PC may seem ridiculous, but it works surprisingly well.

The rest of this paper is in five parts. Section 2 is an introduction to realtime computing. Section 3 is an explanation of why realtime computing is so hard to integrate with non-realtime computing and why obvious methods – like making the standard kernel directly support realtime – are doomed to failure. Section 4 explains how RTLinux works and section 5 shows how to write applications. The last section covers future directions and has some random acknowledgements.

2 An introduction to realtime programming.

"Realtime" is is an over-used term that can be used to mean "right away" or "fast" as in "realtime stock quotes". RTLinux is addressed at *hard realtime* systems: those with timing deadlines that cannot be missed. The traditional uses for these systems are to control or monitor some physical system or device such as a motor, an assembly line, a telescope, or an instrument. Telecommunications and networking devices often also need realtime control. Consider the difference between the response time needed for text editor, a video display, and a program controlling the shutdown sequence of a liquid fuel rocket.

- The text editor should respond quickly to user commands, but if it takes a half a second to update a display every now and then, few users will notice.
- Video displays should *almost always* keep up with the frame rate. A half a second freeze will be noticed and a couple of freezes in a minute will make the system unpleasant. If the program starts dropping frames, it can sometimes use an algorithm to hide the missing frames by interpolation.
- The rocket shutdown sequence must meet deadlines or the rocket may explode. A single freeze of a half a second at the wrong moment might have a spectacular and terminal result.

So the text editor needs to be **fast and responsive**, the video display needs to **usually meet timing deadlines** and the rocket control needs to be able to **guarantee** response times. In the CS literature, the video display would be called a *soft realtime* system, and the rocket control would be called a *hard realtime system* [13, 12].

Most hard realtime applications have failure modes that are not as spectacular as those of liquid fuel rockets, but they need guaranteed timing nonetheless. If you are controlling a servo motor through the parallel port of a cheap PC[8], each timing jitter of 10 microseconds causes an error of one degree. If you are collecting data from a scientific instrument or video frame grabber, a missed deadline may result in missed data or even a confused device. For high speed networks, a delay of a couple of microseconds may drop a packet and cause a major performance loss as the system times out and requests a retransmit. The distinctive property of hard real time systems is this requirement for guaranteed timing – an average time response of 5 milliseconds on our rocket controller will not make up for a single worst case of 100 milliseconds.

> *Hard realtime systems cannot use average case performance to compensate for worst case performance.*

Suppose that we have a board that samples analog lines and produces an 8 bit result every 100 microseconds. Most boards like this are designed with hardware buffers right now so that they can compensate for the non-realtime behavior of Microsoft Windows and NT. The analog to digital device generates samples and puts them in the buffer. If the buffer is deep enough, say with room for 512 samples, then we won't lose any data if we read the buffer at least once every 50 milliseconds. There are three problems with this scenario. The first problem is that even though 50 milliseconds is a long time, it is easy to miss the deadline in even a moderately loaded system. Ordinary Linux can't keep up, even using the so-called "realtime" POSIX extensions. A second problem is that if the hardware has to cope with software timing uncertainties, the hardware becomes more complex and costly. Finally, if we need to control a device – to react to data "in real time" – then 50 milliseconds may be too long and the hardware buffers may cause instability as they introduce delays in the control loop.

RTLinux provides support for hard realtime programs and will, in the future, offer support for some kinds of quality of soft realtime. RTLinux leaves the problem of "fast" applications to standard Linux. Making non-realtime applications run fast is incredibly complex. It turns out that much of what you can do to speed up the non-realtime performance of applications introduces unpredictable delays that are unhealthy for realtime. The obvious example is paging of virtual memory. If 99.99% of program memory references are in the page cache, and if a hit takes one time unit and a miss takes 500 time units, then over a sufficiently large time period the average access time is 1.05 time units – virtual memory is practically free. On average, virtual memory is a major performance win, because processes will be able to run as if there was a much larger amount of memory available — with close to zero cost. On the other hand, a realtime task that misses its first 10 pages will take 5000 time units to do what could otherwise be done in 10 time units and this worst case performance is what matters.

Because of these problems, back in the far past[3] realtime systems were hand made, simple contraptions designed by tough, rugged, engineering types who scorned compilers and virtual memory and thought that alphanumeric displays were unnecessary luxuries. In these systems, optimizing worst case performance was quite straightforward. In fact, many realtime systems still are designed as loops that execute on a bare machine. The program loops through a list of simple tasks and the longest time before a task will run is the sum of the execution times of all the tasks on the list.

```
counter=500;
while(1){
    if(data_on_sensor()){
        read_sensor();
        compute_output();
        counter--;
    }
    if(!counter){
    output();
    counter=500;
    }
}
```

The problem with this design is that is does not scale. As realtime applications get more complex, they need more complex support. If you want to have a realtime control program for an aircraft engine diagnostic system that monitors hundreds of sensors and also needs to display graphical data, interact with a database, connect to a network and even run a web interface, then writing a control loop system is out of

[3] Well before Dave Miller was born.

the question. The problem then becomes one of keeping the fast, predictable operation that you can get from control loop on a bare machine, while running in a much more sophisticated software environment. One solution is to add realtime support to a non-realtime kernel, but that's not as simple as it may sound.

3 You can put racing stripes on a bulldozer, but it won't go any faster.

Before describing how RTLinux works, it is worthwhile to explain why what seems like a more straightforward approach does not work. You could just take an operating system like Linux and stick realtime support into the kernel. For example, you could allow for "realtime" processes with locked memory (so there were no delays while virtual memory was swapped in) and a special scheduler to always run these processes first and in a predictable order. This approach is exemplified in the part of the POSIX 1003.13[10] standard called "Multi-Purpose Realtime System Profile" (PSE54). RTLinux does not try to meet this standard, but standard Linux does support PSE54 compliant `mlock` (memory lock) and `sched_setsched` (special schedule) system calls and POSIX RT signals.

To see how well the POSIX PSE54 approach works under standard Linux, we can write a simple program that asks to be specially scheduled and then calls `udelay` to suspend itself for, say, 50 milliseconds. On a Pentium, it is possible to read the processor cycle counter before and after this delay to get reasonably precise timing. We can set up a loop and look for average and worst case delay. On an idle standard Linux system, the task is remarkably precise: the time for 1000 trips through the loop is rock solid and the worst case deviation from average is about 100 microseconds in my test on a 333MHz dual PII system. But when I/O activity begins, the worst case deviation rapidly rises to a half a millisecond and if one starts up Netscape, the worst case deviation exceeds a couple of milliseconds. A simple test program that does `write(1,buff,BIGNUMBER)` causes the "realtime" task to experience delays of over 18 milliseconds — even though the average remains unchanged. Ordinary standard Linux processes, of course, do a lot worse. So we could not reliably use the data acquisition card discussed above if all we had to work with were standard Linux processes and POSIX PSE54. For comparison, the same test on RTLinux gives worst case difference between minimum and maximum (not between max and average) of under 25 microseconds – almost 1000 times better. Running the RTLinux task at 500 microsecond intervals gives identical precision, and Linux continues to run quite well. Running the POSIX "realtime" process at 500 microsecond intervals stops Linux completely.

No other general purpose OS does any better than standard Linux in mixing RT with standard services — all for the same reasons. The most obvious problem is that the most useful design rule for general purpose operating systems is: *optimize the common case.* Thus, Linux SMP locks are exceptionally fast when there is no contention — the common case – and pay a significant price when there is contention. To use another design would slow down general operation of the system in order to optimize something that should happen only rarely. Similarly, standard Linux interrupt handling is really fast for a general purpose operating system. In some of our measurements, standard Linux averages 2 microseconds to get to the interrupt handler on reasonably standard x86 PCs. That's impres-

sive and it is critical for making users of graphical user interfaces and interactive networks feel happy. Worst case behavior is not so impressive and some interrupts are delayed by hundreds of microseconds. The problem is that it is not so easy to figure out how to decrease worst case without increasing average case.

If you look at Linux code long enough, you will see many more fundamental contradictions with realtime requirements. A few examples should make the case.

- "Coarse grained" synchronization means that there are long intervals when one task has exclusive use of some data. This will delay the execution of a realtime task that needs the same data structures. "Fine grained" synchronization, on the other hand, will cause the system to spend a lot of time uselessly locking and unlocking data structures, slowing down all system tasks. Linux uses coarse grained scheduling around some of its core data structures because it would be stupid to slow down the whole system to reduce worst case.

- Linux will sometimes give even the most unimportant and nicest task a time-slice, even when a more important task is runnable. It would not be smart to never run a background process that cleans up log files, even if a higher priority computation is willing to use up all available processor time. But in a realtime system, the highest priority task should not wait for a lower priority task. In a realtime system, you cannot assume that low priority tasks will ever make progress – but in a general purpose operating system we do assume that low priority tasks will progress.

- Linux will reorder requests from tasks to make more efficient use of the hardware. For example, disk block reads from the lowest priority processes may take precedence over read requests from the highest priority process so to minimize disk head movement or to improve chances of error recovery.

- Linux will "batch" operations to make more efficient use of the hardware. For example, instead of freeing one page at a time when memory gets tight, Linux will run through the list of pages clearing out as many as possible — delaying execution of all processes. It would be counterproductive for Linux to run the swapper each time a page was needed, but the worst case certainly gets a lot worse.

- Linux will not preempt the execution of even the lowest priority tasks during system calls. While the RC5 process is in the middle of "fork", a message that arrives for the video display process will sit on the queue. To get around this problem, Linux would have to put in preemption points that would slow down all system calls.

- Linux will make high priority tasks wait for low priority tasks to release resources. If the RC5 program allocates the last network buffer and the robot arm controller needs to send a message to stop the robot arm, the robot arm controller will just have to wait until some other process frees a network buffer.

You could argue that since there are operating systems mixing realtime and standard services in the same kernel, the above must be wrong. Those systems work, and are substantial technical achievements, but they are complicated and quite slow. You can have deterministic worst case behavior time on Solaris RT, but the worst

case is really worse than you might be willing to tolerate. In the academic realtime literature, it is often stated that realtime does not require speed, it requires determinism (fixed worst case timings). That's true as far as it goes, but you can't do realtime video editing if your worst case behavior only allows for a frame rate of one frame every 2 seconds.

Since realtime and general purpose operating systems have contradictory design goals, it is not surprising that what is smart in one system is deadly in another. If we attempt to satisfy both design goals in the same system, we end up with something that does neither very well.

4 The RTLinux solution

About 20 years ago, researchers at Bell Labs built an experimental operating system called MERT[9]. This operating system was intended to run both realtime and general purpose applications. But instead of trying to make one operating system that could support both, MERTs designers tried to make a system in which a realtime operating system and a general purpose (time-sharing) operating system worked together. The designers wrote:

> *... the availability of a sophisticated time-sharing system in the same machine as the realtime operating system provides powerful tools which can be exploited in designing the man-machine interface to the real-time processes.*

That is, MERT's designers claimed that by decoupling the realtime OS from the non-realtime OS, they were able to allow application developers to use the services of the non-realtime OS [4]

[4]The MERT paper appeared in a famous 1978 issue of the Bell Systems Technical Journal. This issue also carried articles reporting on porting C code to multiple machines (a novel idea at the time), a new text processing system called troff, the programmers workbench, and the original papers on UNIX and C. You should read it or reread it.

RTLinux works by treating the Linux OS kernel as a task executing under a small realtime operating system. In fact, Linux is the idle task for the realtime operating system, executing only when there are no realtime tasks to run. The Linux task can never block interrupts or prevent itself from being preempted. The technical key to all this is a software emulation of interrupt control hardware. When Linux tells the hardware to disable interrupts, the realtime system intercepts the request, records it, and returns to Linux. Linux is not ever allowed to really disable hardware interrupts. No matter what state Linux is in, it cannot add latency to the realtime system interrupt response time. When an interrupt arrives, the RTLinux kernel intercepts the interrupt and and decides what to do. If there is a realtime handler for the interrupt, the handler is invoked. If there is no realtime handler, or if the handler indicates that it wants to share the interrupt with Linux, the interrupt is marked pending. If Linux has asked that interrupts be enabled any pending interrupts are emulated and the Linux handlers are invoked – with hardware interrupts re-enabled.

No matter what Linux does, whether Linux is running in kernel mode or running a user process, whether Linux is disabling interrupts or enabling interrupts, whether Linux is in the middle of a spin-lock or not, the realtime system is always able to respond to the interrupt.

> *RTLinux decouples the mechanisms of the realtime kernel from the mechanisms of the general purpose kernel so that each can be optimized independently and so that the RT kernel can be kept small and simple.*

RTLinux is designed so that the RT kernel never has to wait for the Linux side to release any resources. The RT kernel does not request memory, share spin-locks, or synchronize on any data structures — except in tightly controlled situations. For example, the communication links used to move data between RT tasks and Linux processes are *non-blocking* on the RT side: there is never a case where the RT task waits to queue or dequeue data. The near failure of the Mars Lander was caused by an interaction between a realtime task and an operating system service that assumed it was ok to de-schedule a process. One VxWorks task acquired a semaphore and a high priority task was then put to sleep when it tried to write to a pipe protected by that semaphore. RTLinux does not have any such hidden points of synchronization.

Of course, it doesn't do much good to have a realtime system that can't communicate at all with the non-realtime system, so RTLinux provides both shared memory (using a crude method right now) and also a device interface that lets Linux processes read and write to realtime tasks.

One of the key design principles of RTLinux is that the more that is done in Linux — and the less that needs to be done on the RT side — the better. Linux takes care of system and device initialization and of any blocking dynamic resource allocation. Device initialization can be left to Linux. There cannot be any realtime constraints at boot time, so there is no need for the RT system to be involved. Blocking dynamic resource allocation is left to Linux. Any thread of execution that is willing to be blocked when there are no available resources cannot have hard realtime constraints. For example, there is no way for a RT task to call *malloc* or *kmalloc* or any other memory allocator. If the task does not statically allocate memory, it does not have access to that memory. Finally, RTLinux relies on the Linux loadable kernel module mechanism to install components of the RT system and to keep the RT system modular and extensible. Loading a RT module is not a realtime operation and it can also be safely left to Linux. The job of the RT kernel is to provide direct access to the raw hardware for realtime tasks so that they can have minimal latency and maximal processing when they need it. Anything else just gets in the way.

So RTLinux is a variation, a better variation in my humble opinion, of the basic idea found in MERT. Around the same time that we developed and released RTLinux, Greg Bollella [4, 3] was working on putting a realtime kernel on the same machine as the general purpose IBM MicroKernel and a little later two companies produced systems in which a realtime kernel shares the machine with Windows NT [2, 5]. All of these systems use some variation on the technique of putting a virtual machine layer between the general purpose OS and the interrupt hardware. The low level times measured by the Radisys authors [6] and by Bollella are similar to the RTLinux times: reflecting mostly the limitations of PC motherboard design. It's irritating to observe an out-of-order execution, deeply pipelined, highly cached, 400MHz processor, turn into an expensive space heater as it negotiates a 8 bit ISA bus path to its legacy original-PC system timer.

5 Using RTLinux

RTLinux is very much module oriented. To use RTLinux, you load a modules that implement whatever RT capabilities you need. Two of the core modules are the scheduler and the module that implements RT-fifos. If the services provided by these modules don't meet the requirements of the application, they can be replaced by other modules. For example, there are two

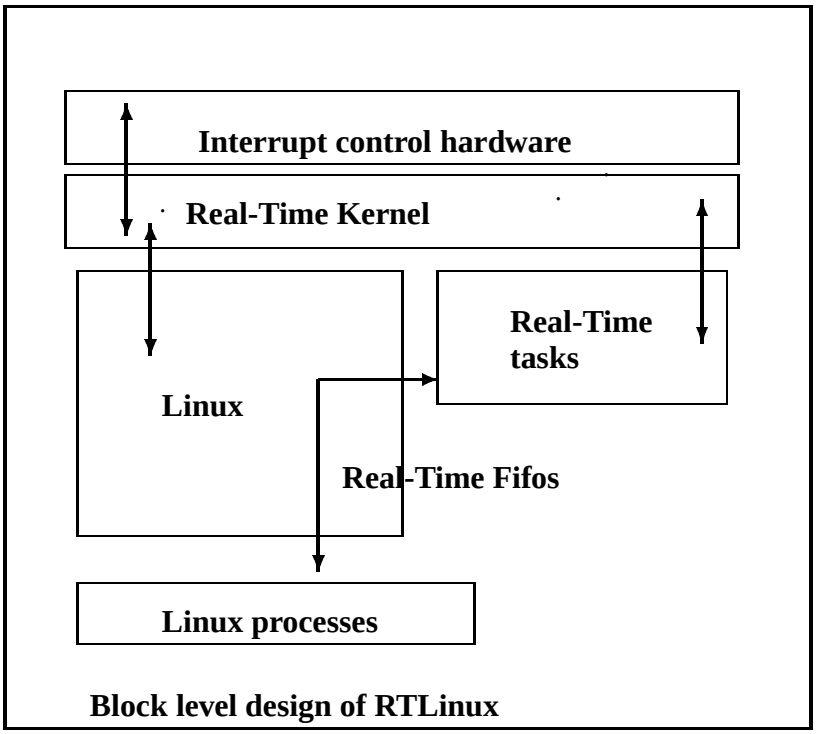

Figure 1: Flow of data and control

alternative scheduling modules — a "earliest deadline first" scheduler implemented by Ismael Rippol[11] and a rate-monotonic scheduler implemented by Oleg Subbotin (see the rtlinux.org web page)[5]. The basic scheduler simply runs the highest priority ready RT task until that task suspends itself or until a higher priority task becomes ready.

The original RTLinux scheduler (written by Michael Barabanov) used the timer in "one shot" mode so that it could easily handle a collection of tasks with periods that had a small common divisor. For example, if one task must run every 331 time units and the other runs every 1027 time units, there is no good choice for a timer period. In one-shot mode, the clock would be set first to generate an interrupt after 331 time units and then reprogrammed after the interrupt to generate a second interrupt in another 691 time units (minus the time needed to reprogram the clock). The price we pay is that we reprogram the clock on every interrupt. For x86 generic motherboards, reprogramming the clock is relatively slow. It turns out, however, that many applications don't need the generality of a one-shot timer and can avoid the expense of reprogramming. Professor Paolo Mantegazza of the Aerospace Engineering Department in *Politecnico di Milano* wrote a scheduler that demonstrated the utility of periodic mode and encouraged us to put it into the standard scheduler[6]. The current RTLinux scheduler of-

[5]The academic CS literature is deeply concerned with the right way to schedule RT tasks. My theory is that nobody knows yet, and that the OS should not make the choice.

[6]Professor Mantegazza is also responsible for debugging floating point support in RTLinux.

fers both periodic and "one shot" modes. On SMP systems the problem gets simpler because there is an on-processor high frequency timer chip that is available to the RTL system.

5.1 The API

The standard API for Version1 (based on the 2.0 Linux kernel) is as follows.

- `rtl_request_irq` and `rtl_free_irq`. These activate and deactivate interrupt handlers.
- `rt_get_time` returns the time in "ticks".
- `rt_task_delete` destroys a task and frees its resources.
- `rtl_task_init` sets up, but does not schedule a task.
- `rt_task_make_periodic` asks the periodic scheduler to the run task at a fixed period (given as a parameter).
- `rt_task_suspend` takes the task off the run queue.
- `rt_task_wait` yields the processor until the next time slice for this task.
- `rt_task_wakeup` wakes up a suspended task.
- `rt_use_fp` allows the task to use floating point operations.
- `rtf_create` creates a fifo.
- `rtf_create_handler` attaches a routine that runs under the Linux kernel to a fifo so that user processes can be made runnable when there is data available.
- `rtf_destroy` frees a fifo.
- `rtf_get` is the non-blocking *read* operation for realtime tasks.
- `rtf_put` is the non-blocking *write* operation for realtime tasks.
- `rtf_resize` changes the size of data in the fifo.
- `rtl_set_periodic_mode` optimizes the system for running a collection of tasks that share a common fundamental period.
- `rtl_set_oneshot_mode` optimizes the system for cases where periodic mode is not appropriate.

In the Version2 (based on Linux 2.2) many of these calls have an alternate form with an additional parameter for cpu identifier. On a SMP system, a task is associated with a particular cpu and is only scheduled by the scheduler on that cpu. Also, on SMP systems, some interrupts are local and need to be given handlers per/cpu.

5.2 Examples

5.2.1 Squares

Figure 2 shows how a program to produce a square wave on the parallel port output would be written.

This program would be compiled as a module, and `insmod` is used to start it. When the module starts, it runs initialization code that constructs a single task using `rt_task_init` and then asks the realtime scheduler to run the task every 450 ticks of the clock. On a SMP x86 system, we have a more sophisticated timer that used the processor clock, but on the standard motherboards we are still stuck with the antique i8353 – a vestige of the original IBM PC. In spite of this embarrassment, this task is never more than about 40 microseconds late or early,

```
/* Module to toggle output on the parallel port */
RT_TASK my_task;
#define STACK_SIZE 3000

void code_for_rtl_task(unsigned int pin) {
 static unsigned char bits = 0;
 while(1){
   if(bits)bits = 0;
   else  bits = (1<< pin);
   /* write on the parallel port */
   outb(bits, LPT_PORT);
   /* wait till next period */
   rt_task_wait();
   }
}

int init_module(void)
{
 RTIME now = rt_get_time();
 /* Initialize a task with code code_for_rtl_task,
    pin 3 ,stack size STACK_SIZE and priority 1 */
 rtl_task_init(&my_task, code_for_rtl_task, 3, STACK_SIZE, 1);
 /* run every 450 8253/4 ticks
    (about 50 milliseconds)*/
 rtl_task_make_periodic(&mytask, now, 450 );
 return 0;
}
```

Figure 2: Square wave RT program

on anything from an old 386 to a 500Mhz PIII, no matter what load is running in the system.

5.2.2 Collecting data

To make something like the analog/digital converter discussed above work, we would write two pieces of code. The first would be a realtime module that would poll the device and then put data into a realtime fifo. The other piece of code would read the fifo and would run as a Linux user process. The major change from the parallel port toggling example would be in the main loop of the function `code_for_rtl_task` which might be written something like this:

```
while(1){
 read_data_from_hardware;
 rtf_put(FIFO_ID,data,size);
 /* wait till next period */
 rt_task_wait();
}
```

On the Linux side, the realtime fifos are devices: `rtf0,rtf1 ....` The Linux side task could be a one line shell script:

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

5.2.3 Interrupt handlers

Instead of using the calls to the scheduler module – `rtl_task_wait`, `rtl_task_init`, and `rtl_task_make_periodic` – we could simply attach our function to a timer. Using similar task code, we could change the initializing code as follows.

```
int init_module(void)
{
 rtl_request_irq(CMOS_CLOCK,\
                 handler_ptr);
 INIT_CMOS_CLOCK(FIFTYMS);
 return 0;
}
```

Our cleanup code for the module would call `rtl_free_irq`. The code for the handler would look like the code for task, except instead of calling `rtl_task_wait` it would reset the CMOS clock to allow further interrupts.

6 What next and acknowledgments

A 2.2 version of RTLinux was released in January 1999 with some missing features, but with improved support of SMP. A fully functional version will, I hope, be released in March. Ports are underway to Alpha and to PowerPC. SMP with larger numbers of processors is a key goal over the next few months. We are also rewriting the scheduler to support the minimal POSIX RT standard (**not** from the PSE54 standard mentioned earlier) and are looking at how to support QOS (quality of service) assurances for soft realtime tasks [12]. Other developments are several commercial packages for RTLinux under development. See `http://www.rtlinux.com` for commercial links. Linus Torvalds once said that the RTLinux core would become integrated with the standard kernel in 2.3, but the availability of pre-patched kernels makes this a less pressing issue. News and code can always be found at `http://www.rtlinux.org`.

At New Mexico Tech, RTLinux development has been funded mostly by USENIX (`http://www.usenix.org`). There is (GPL) RTLinux development now being carried out on a commercial basis and this is being financed by industrial contracts. RTLinux was first implemented by Michael “FZ” Barabanov[1]. As with all open-source

projects, RTLinux is a collaborative effort. Thanks to the Linux kernel developers for such a useful idle task and thanks to the RTLinux users for being so enthusiastic and brave and for contributing code, ideas, and interesting stories.

References

[1] Michael Barabanov and Victor Yodaiken. Real-time linux. *Linux journal*, February 1997.

[2] Nick Vasilatos Bill Carpenter, Mark Roman and Myron Zimmerman. Rtx real-time subsystem for windows nt. In *Windows NT System Engineering Workshop*. USENIX, August 1997.

[3] Greg Bollella. *Slotted Priorities: Supporting Real-Time Computing Within General-Purpose Operating Systems*. PhD thesis, University of North Carolina, 1997.

[4] Greg Bollella and Kevein Jeffay. Supporting co-resident operating systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 4–14, May 1995.

[5] Radisys Corporation. Intime kernel. Technical report, Radisys Corporation, `http://www.radisys.com` 1997.

[6] Radisys Corporation. Intime interrupt latency report. Technical report, Radisys Corporation, `http://www.radisys.com` 1998.

[7] James P. Held Erik Cota-Robles. A comparison on windows driver model latency performancea on windows nt and windows 98. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI99)*, pages 159–172, Boston, MA, Feb 1998. USENIX.

[8] Bernhard Kuhn. Servomotorensteuerung mit rt-linux. *Linux Magazine (germany)*, December 1998.

[9] H. Lycklama and D. L. Bayer. The MERT operating system. *Bell System Technical Journal*, 57(6):2049–2086, 1978.

[10] The Portable Application Standards Committee of the IEEE Computer Society. P1003.13 draft standard for information technology — standardized application environment profile — posix realtime application support (aep). Technical report, IEEE, 1998.

[11] Ismael Ripoll. Earliest deadline first scheduler. Technical report, University of Valencia (Spain), `http://bernia.disca.upv.es/` 1998.

[12] John A. Stankovic. Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4):751–763, Dec 1996.

[13] John A. Stankovic and Krithi Ramamritham. *Hard Real-Time Systems*, volume 819 of *IEEE Tutorials*. IEEE, 1988.

[14] C. Wayne Wright and Edward J. Walsh. Hurricane hunting. *Linux Journal*, (58), Feb 1999.