Overview of Real Time Operating Systems
Fall 2012 Independent Study

Ian O'Hara

February 23, 2013

At UPenn's Modlab (GRASP Subsidiary)
Advisor: Dr. Mark Yim

1 INTRODUCTION

When writing software for hardware, the ability to control the timing of events is crucial. With conventional operating systems meant for generic user programs, this is not possible. A class of operating systems called Real Time Operation Systems aims to solve this problem. This document will outline first what Real Time Operating Systems are, why they are needed, and then will go into the details of getting one working on a Gumsitx [24]. From there, the performance of this specific RTOS and hardware combination will be characterized, and details of how to take advantage of the RTOS capabilities will be laid out.

2 DEFINITIONS

The basic terminology of real time systems is taken from [17]. Unless otherwise noted, assume these definitions can be found there.
This section contains definitions of terms used throughout the document. It is the equivelant of a glossary, but at the document start.

System A mapping of a set of inputs into a set of outputs

Response Time The time a system takes to map a set of inputs to the corresponding set of outputs

Real Time System (RTS) A system where there is a bound set on the response time of a system. When this bound is not met, the system is considered in a failed state.

Hard Real Time System (HRTS) A RTS in which failure to meet the response time bound leads to catastrophic failure. IE: Response times must be deterministically met.

Soft Real Time System (SRTS) A RTS in which the response time bounds must be met the majority of the time. IE: Performance is degraded, but not destroyed, when response times are not met.

Process Preemption A process preemption is the suspension of a process so that another higher priority process can run. This can occur for a number of reasons, and is essential in RTS.

3 WHAT IS REAL TIME

The definition of real time is ambiguous in scope. Almost any problem can be casted as a "real time" problem (eg: shipping a package across the US can be considered real time if you need it to be there in 5 days).
In the context of robots, real time demands are typically associated with sensor input and control loops. EG: Read this sensor every 1ms and run a control loop at 100us.
It is importantant to note that while RTS often deal with "fast" times, a RTS does not need to be fast. It simply needs to have explicit bounds on response times. For example, a task that must run once a year 4.000 ± 0.001 seconds after the new year is real time.
Another phrasing from [2] is that a Real Time Application is one in which there are operational deadlines between some event being triggered and the applications response to that event. The use of a Real Time Operating System (RTOS) gives the programmer calculated (hard real time) or measurement predicted (softer real time) response times.

4 WHY

Why should we bother with Real Time Operating Systems (RTOS)? This needs to be answered and justified, and can be done by looking at the different ways in which we can program, in this case, robots.
On the one hand we have high level generic operating systems, like Linux, which provides a mechanism for:

a. Running many independent processes

b. Memory management and safety. One process in user space cannot corrupt the memory of another process or the kernel (as long as the system has an MMU, and most systems running Linux do).

c. Interfacing with many different pieces of hardware without having to write low level device specific code (i2c bus, spi bus, etc all are just simple entities we can use)

d. Developing in-situ

e. Using many different programming languages

f. Taking advantage of open source code without worrying about it being specific to your hardware

However, along with the benefits of abstracting away all of the low level details for us, a generic operating system does not normally care about meeting timing deadlines. For instance, a desktop user does not care if their mouse position is updated exactly once every 1 ms. If the update time fluctuates between 100ms and 100us, the user will not notice a difference. The same goes for most typical uses of generic operating systems; scientific number crunching, web browsing, text editing, and many others do not need consistent timing. The "real time" general use computers face is video games which require consistent 30-60hz update rates. However, even in this case, the failure to meet the deadlines does not result in catastrophic failure; just lag.
On the opposite side of the spectrum is the microcontroller approach. In this scenario, the programmer himself "writes" the operating system. He is responsible for coordinating and scheduling everything that happens. This can be nice because it allows for absolute control of what happens when - timing is in the programmer's hands, and has the resolution of the microcontroller's clock.
However, the advantages of the absolute control of a microcontroller are accompanied by many disadvantages:

a. You must develop offboard, cross compile, and then use a binary loading mechanism to load code onto the microcontroller.

b. The programmer is responsible for everything.

c. Device interfaces are at the lowest level. Most microcontrollers have basic libraries in the wild, but the platform specific nature of microcontroller programming makes for many platform specific libraries that accomplish the same funcionality. This results in a lot of un-maintained dead libraries.

d. The programmer must know the minutia of any microcontroller used.

e. Switching hardware often involes re-writting significant portions of the code since it is very hardware specific.

f. There is no memory management - an errant memory access can change anything. This includes mux registers, timer registers, or literally anything that has a memory address.

The middle ground is the RTOS which provides the advantages of having an operating system and the advantages of having the guaranteed timing of a microcontroller. However, this comes with the added complexity of having to be, to some level, aware of how your operating system works. Also, writing code must be done in such a manner that it can be real time.

5 LINUX

5.1 Operating System

An Operating System is a piece of software that does two things:

a. Provides an abstracted interface to the underlying hardware

b. Manages the execution of a users programs (ie: sharing the CPU between all users' processes)

An operating system does its best to give each process the impression that the resources of the entire computer are at its disposal at all times. In the absence of inter-process communication, the operating system does its best to make each process think it is the only think running on a computer.
The main functions of a kernel (the core part of an OS that is always in memory and running) are[15]:

a. memory management

b. process scheduling

c. interfacing to the hardware

d. file management

e. communication with external devices and networks.

5.2 Task Scheduling

Linux uses `nice()` to control task priority. Priority levels are -20 (high priority) to 19 (low priority). You can see tasks as a column in `top` output (Column "NI" column)
The `SCHED_FF`, `SCHED_RR`, `SCHED_OTHER`, and `SCHED_FIFO` scheduling policies define how the kernel schedules each individual system process. The `SCHED_OTHER` option is default.
`SCHED_FIFO` should be used for real time tasks. Reading through all of the the Linux documentation on the scheduler [28], particularly `sched-rt-group.txt`, is a good idea.
Each process is given one of the scheduling policies listed above. For a full discussion of how to use process scheduling (watch out, it might be slightly outdated) see Chapter 10 of Understanding the Linux Kernel.[29] Also, check the `syscalls` manpage for system calls of the form `sched_*` for a good starter list of relevant system calls for getting and setting scheduling related settings for a process.
The important real time scheduling settings `/proc/sys/kernel/sched_rt_period_us` and `/proc/sys/kernel/sched_rt_runtime_us` define how much time is given to realtime processes in your system. They are given detailed descriptions in [28] so, again, go read it!

5.3 Context switching and Preemption

Context switching occurs when the kernel changes the "current" task, be it to another kernel task or to a user space process. This occurs often behind the scenes to make it look as if all of the processes running on a system are running simultaneously. In reality, only one task runs at a time per cpu (or core).
Preemption is the suspension of a current running task in favor of another higher priority one.
See [14] for an excellent explanation of the different ways linux knows it is safe to switch context, and how different tasks (including kernel tasks) can be preempted.

5.4 User Control: Sleep

Users can take advantage of the sleep() system call (and others of the same flavor such as usleep() and nanosleep()) to signify to the kernel that the user process is done what it needs to do for some time. POSIX compliance of an OS stipulates that the sleep() functions return control the process at or after the specified time. The "or after" specification is important because an OS does not need to make any guarantees on when it returns control to the process as long as it is after the specified sleep time. This is an issue when trying to write realtime programs.

5.5 User Control: Scheduling priority

A realtime process needs to modify its scheduling policy to tell the operating system that it is realtime and should not be preempted. This is done by using the `sched_` system calls. An example of how to do this can be found at src/schedulerTest.c.
In Linux, users can configure the scheduling priority within their scheduling policy. This is done with the setpriority() system call. Mentioned in this document is the nice() scheduling priority mechanism.

5.6 User Control: `sched_yield`

If your process or thread is done what it needs to do, but does not need to sleep for a specified time, you can yield control to the kernel with `sched_yield`.

5.7 User Control: Timers (High Resolution Timers)

Timers provide a mechanism for telling the kernel that a realtime process wants control after a certain interval of time has passed (either repeatedly, or once depending on how a timer is setup). For a realtime process running with higher scheduling priority than any other process, its timer triggering should result in regaining control (typically) in the average latency of the system.
An example of using hrtimers can be found in hrtimerTest.c. It is self documenting, so go read it.

Useful resources for learning more about Timers/hrtimers are:

a. hrtimers.txt in the kernel documentation [33]

b. High resolution timer API on LWN [34]

c. Original article on changing to the hrtimers [35]

6 REAL TIME LINUX

The traditional linux kernel allows one process to preempt other processes in limited cases. Specifically, preemption is controlled by the kernel and occurs only when:

a. User-Mode code is running (Kernel preempts it)

b. Kernel returns from a system call or an interrupt back to user space

c. Kernel code blocks on a mutex or explicitly yields

These points are from the FAQ at [2].
In 2002 the National Institute of Standards and Technology (NIST) had Aeolean Inc. write a report detailing the viability of using Real-Time Linux for control applications. [15] In it, they note that standard linux is alright for soft realtime applications where scheduling on the order of milliseconds (with several hundred MHz system clock) and occasional missed deadlines are accepted. The report then goes on to say that hard realtime linux can achieve 10 to 100 microsecond timing with no misses. It is important to note that dedicated RTOS can do better than this, but they come without the established (and growing!) Linux community. Linux is active, has thousands of developers contributing, is used in government and universities, and is not going anywhere. This is a major reason for trying our best to make realtime Linux to work for us.

6.1 Considerations

The RT Linux community came up with a list of quality-of-service metrics that they wanted to consider. [13] They are:

a. List of services for which realtime response is supported

b. Probability of meeting a deadline in absensce of hardware failure

c. Allowable deadline (response time) of a task

d. Performance and Scalability (Ambiguous?)

In addition to quality of service, they list 5 other qualities that are desirable in a full real time operating system. All 6 are:

a. Quality of Service

b. Amount of code that must be inspected to assure the quality of service

c. API is provided

d. Minimal added complexity of applications that take advantage of real time

e. Fault isolation: If non-RT code fails, does it affect RT code?

f. What hardware and software is suppored?

The general consensus was that a POSIX conforming api should be provided.

6.2 Development Info

It looks like [6] is highly involved with the Real Time Linux development process. Their Real Time Linux project page [7] cites both Ingo Molnar and Thomas Gleixner as being the lead developers. After spending time on the real time linux users list (linux-rt-users) it appears that as of October 31, 2012 Steven Rostedt (rostedt@goodmis.org) is the current manager of patches/releases to real time linux development.
The OSADL site cites a few important locations for downloading RTL requirements:

a. The Linux Kernel at kernel.org

b. The Real Time Preempt Patch at [8]

c. The Real Time Linux Kernel git repo [30] (active and current October 31, 2012). Also referenced in the README in [8]

6.3 Performance

There are many qualitative (and somewhat subjective) performance metrics used, some of which are mentioned above. However, the two objective quantitative measures of performance for a RTOS are:

a. Event Latency

b. Periodic Jitter

Event Latency is the time it takes a request to be fulfilled. In other words, if my real time process requests control but does not actual get control of the CPU for 10 micro seconds, then the latency is 10 micro seconds.
Periodic Jitter is the variation in time of a periodic repetitive task, or the variance of latency. IE: If I have a task that needs control once every 100 micro seconds, what is the variance of the actual time between each successive resumption of control.

6.4 Testing

There is a set of testing tools on kernel.org that is maintained by Clark Williams at [16].

6.4.1 cyclictest

Cyclictest measures the latency and jitter of executing timer based events on a system. It spawns a number of threads, and then sets a timer for each thread that triggers at a certain interval. An example of a call to cyclictest is:

```
root# cyclictest -n -p80 -i0 -l5000 -v -t10 -d10
```

which shows results in nanoseconds (-n), sets thread 1 to priority 80 (-p80), spaces the interval of each thread by 0 nanoseconds (-i0), runs 5000 loops (-l5000), in verbose mode which is good for automated post processing (-v), runs 10 threads (-t10), where each thread's timer is set at a 10 microsecond interval (-d10).
The rt.wiki.kernel.org wiki page for Cyclictest has some good other examples of "in the wild" uses. [23]
This section outlines how to use them, and some specific results for the gumstix.

6.4.2 Results of cyclictest

Table 1 and Table 2 give some basic, not necessarily rigorous, results characterizing the real time abilities of the different kernels I have compiled for the gumstix.

Image	Cycles	Interval μs	Timer Event Response Latency		
			Min μs	Avg μs	Max μs
linux-rt, No Preempt, No High Res Timers[25]	5000	10000	7679	11446	15240
Sakamon, No Preempt, High Res Timers[26]	50000	1000	14	236	426
linux-rt, Full Preempt, High Rest Timers[27]	50000	1000	30	156	332

Table 1: Unloaded system Result of running `# cyclictest -n -p80 -l<Count> -i<Interval>` with test kernels running on a gumstix. The uImages corresponding to all of these kernels can be found through their citations.

Image	Cycles	Interval μs	Timer Event Response Latency		
			Min μs	Avg μs	Max μs
linux-rt, No Preempt, No High Res Timers[25]	5000	10000	7868	11663	15429
Sakamon, No Preempt, High Res Timers[26]	50000	1000	0	138	271
linux-rt, Full Preempt, High Rest Timers[27]	50000	1000	32	158	302

Table 2: Result, with loaded system, of running `# cyclictest -n -p80 -l<Count> -i<Interval>` with test kernels running on a gumstix. The uImages corresponding to all of these kernels can be found through their citations. The command `stress -c 2` was used to stress the system. [31]

6.5 On The Gumstix

The thread on the gumstix-users mailing list at [21] is the most comprehensive discussion of real time gumstix I have found. That specific response by Philipp Lutz mimicks and confirms a lot of the work I have done so far. Early in the thread, Xenomai [22] is also mentioned. **TODO: Research Xenomai**

7 U-Boot

The Das U-Boot is a "Universal Boot Loader" [18] developed in Denmark. The gumstix uses U-Boot as its boot loader, and so in order to change our boot routine we need to know how it works. U-Boot's documentation can be found at [19]. U-Boot's source code can be found at [20].

7.1 Building Kernels for UBoot

To build a kernel meant for booting with UBoot, we need a uImage kernel image. This requires that the "mkimage" tool is available to the kernel build tools. On a Debian based system, this can be done with "apt-get install uboot-mkimage".

8 Building the RT Kernel

With a clean checkout of the real time kernel [30], you can build it by changing directory to the root of the git checkout and doing:

```
user$ make config
```

or:

```
user$ make menuconfig
```

if you have ncurses and want to navigate through a menu to setup your kernel. This step is essential, because you'll configure what your compiled kernel will act like and what it will contain. To turn on full real time capability, you should make sure that the following options are set to 'y':

a. CONFIG_PREEMPT

b. CONFIG_PREEMPT_RT_BASE

c. CONFIG_PREEMPT_RT_FULL

d. CONFIG_HIGH_RES_TIMERS

Also note that it is important to configure the "CONFIG_CMDLINE" option so that once your kernel boots it knows where to get its root filesystem and what console to output to. An example of a working version of this option comes from Sakoman's kernel:

```
CONFIG_CMDLINE="root=/dev/mmcblk0p2 rootwait console=tty02,115200"
```

If you have a config file you want to use as a basis for a kernel build, put it in the linux root as .config and then run

```
user$ make oldconfig
```
Once configuration is done, you can build the kernel by
```
user$ make prepare
```
and then
```
user$ make
```
This might prompt for a few missed config options, and then will head off building the whole thing. If compiling on the gumstix, this can take 40 minutes or more.
Once the build succeeds, you need to make a uImage version of the new kernel which can be done with
```
user$ make uImage
```
This is a packaged version of the kernel that is specifically meant to be used with the U-Boot boot loader. The new uImage will be in "arch/arm/boot/".
If you're using a gumstix and want to put the new kernel in place, you first need to mount the boot partition on the micro SD card. This should be a FAT partition on the micro SD card, and if you're using the CONFIG_CMDLINE specified above (which specifies that the root filesystem is at /dev/mmcblk0p2, or the second partition on the mmcblk device) then you can mount the boot partition with
```
root# mount -t vfat /dev/mmcblk0p1 /media/card
```
which will mount the boot partition to /media/card.
Then just copy over the new uImage to /media/card. Since you have already booted from this micro SD card, the current running kernel will be at /media/card/uImage . So, before you copy over the new kernel it is probably a good idea to back it up. If the new kernel does not boot, you can access the micro SD card from another computer and replae the original kernel.

9 Using Real Time Linux

Once you have a working system running real time linux, reading through this section should serve as a solid first step toward taking advantage of the real time capabilities.

9.1 Permissions

A realtime application wants to run with its scheduling policy set to `SCHED_FIFO`. Permission is needed to do this, which means either being `root` or have special permission to do so. Granting special permission is done by modifying the `/etc/security/limits.conf` configuration file which configures different system limits for specific users and groups. For example, to enable the `rtian` user to set process scheduling to `SCHED_FIFO` and then increase their rtprio priority level up to a max of 80 (You should use the `sched_*` system calls to get the max in code), the following line can be added to `limits.conf`:
```
rtian - rtprio 80
```
The `limits.conf` manpage has more details, and to check your current user's

security limits you can use:

```
user$ ulimit -a
```

Please be warry of the fact that `SCHED_FIFO` processes with high priority can kill your system by hogging all of the CPU time. You should know how long the process will ask to run for and make sure that it leaves computation time for the system to do its thing. **TODO: This needs to be more specific. Its the essence of writing real time applications** **TODO: Look up and explain CAP_SYS_NICE. It is important** **TODO: Look up writing "0" to /dev/cpu_dma_latency and its effect. clark and JackWinter had convo about it on #linux-rt on 11-14-2012 at 12:30pm. See** `http://www.embedded-linux.co.uk/downloads/ESC-5.5-power-saving-slides.pdf` **on pg 14**

9.2 High Resolution Timers

Documentation for High Resolution Timers can be found in the kernel documentation at [33].
There are a number of ways to use timers in practice, but all of them involve using the linux signal system. Namely, the operating system lets your process, or a thread within your process, know when a timer has run out (fired, or triggered) by sending it a signal. What signal, and how the signal is received is up to you - there are a few choices:

a. Use a single thread and an asynchronous signal handler (See src/hrtimerTest.c for an example of this)

b. Use a single thread, block the signal, and receive it synchronously with sigwait

c. Use multple threads and tell the signal to target a specific thread with an asynchronous signal handler

d. Use multiple threads and tell the signal to target a specific thread. Block the signal in each thread, and use sigwait to receive the thread specific signal synchronously in each thread (this is what cyclictest in rt-tests does.)

TODO: linuxsymposium pg 333 hr timer paper. Look into "clock event source registration"
High resolution timer articles on lwn: [34], [35]

10 NOTES ON OUR USE

We are using the rt Linux release at [8]. We are using the 3.2 kernel, so we are using the "v3.2-rt" branch on the repo.

11 NOTES

This section is full of random votes that don't fit anywhere else.

a. Real time software can't have it memory paged out, so in linux a call to mlockall() needs to be made. This makes sure all of a program's memory stays in RAM. (FAQ at [2])

b. NUMA is mentioned throughout the rt-test code, and there is an `rt_numa.h` header file with functions that provide some custom memory management functionality. NUMA, or Non-Uniform Memoery Architecture, pertains to multiprocessor systems so we can ignore it on the gumstix. See the NUMA manpage [32] for details.

References

[1] Laplante, Phillip A. Real Time System Design and Analysis, 3rd Ed. Pscataway, NJ: IEEE Press, 2004. Print.

[2] Real-Time Linux Wiki, `https://rt.wiki.kernel.org/index.php/Main_Page`, 2012-09-18

[3] Yaghmour, K.; Masters, J; Ben-Yossef, G; and Gerem, P. Building Embedded Linux Systems, 2nd Ed. Sebastopol, CA: O'Reilly Media, Inc. 2008. Print.

[4] Gleixner, T. "[ANNOUNCE] 3.0-rc7-rt0". `https://lkml.org/lkml/2011/7/19/309`. Email

[5] `http://www.h-online.com/open/features/Kernel-Log-real-time-kernel-goes-Linux-3-0-1382791.html`

[6] Open Source Software for Automation and Other Industries. `https://www.osadl.org/Home.1.0.html`

[7] OSDL Real Time Linux Project, `https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html`

[8] Real Time Linux Latest Patches, `http://www.kernel.org/pub/linux/kernel/projects/rt/`

[9] Real Time Linux Installation, `https://www.osadl.org/Realtime-Preempt-Kernel.kernel-rt.0.html`

[10] Real Time Linux Users mailing list, `http://vger.kernel.org/vger-lists.html#linux-rt-users`

[11] The Linux Kernel Archives, `http://kernel.org/`

[12] McKenney, Paul. A Real Time Preemption Overview. http://lwn.net/Articles/146861/

[13] McKenney, Paul. Real Time Patch Acceptance Summary. http://lwn.net/Articles/143323/

[14] Love, R. Linux Kernel Development: Preemption and Context Switching, http://www.informit.com/articles/article.aspx?p=101760&seqNum=3

[15] Aeolean Inc., Introduction to Linux for Real-Time Control, 2002, http://tornasol.datsi.fi.upm.es/ciclope-old/doc/rtos/cache/doc/realtimelinuxreport.pdf

[16] Real Time Linux Testing Tools. git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git

[17] Laplante, Phillip A. Real Time System Design and Analysis, 3rd Ed. Pscataway, NJ: IEEE Press, 2004. Print.

[18] Das U-Boot – the Universal Boot Loader. http://www.denx.de/wiki/U-Boot. October 7, 2012.

[19] Das UBoot Bootloader Documentation, http://www.denx.de/wiki/view/DULG/LinuxConfiguration

[20] Das U-Boot official git repository. git://git.denx.de/u-boot.git . October 7, 2012.

[21] Real Time Gumstix discussion on gumstix-users (Re: Realtime on gumstix), http://gumstix.8.n6.nabble.com/Realtime-on-gumstix-tp4964556p4964587.html . October 16, 2012

[22] Xenomai: Real-Time Framework for Linux, http://www.xenomai.org/ . October 16 2012

[23] Cyclictest page on rt.wiki.kernel.org , https://rt.wiki.kernel.org/index.php/Cyclictest, October 16, 2012.

[24] Gumstix small open source hardware, http://www.gumstix.com, October 23, 2012

[25] Kernel Image from linux-rt source with no Preemption or High-Res Timers, http://svn.modabupenn.org/personal/iano/realtime_gumstix/bootImages/10-14-2012/

[26] Kernel Image from linux-rt source with no Preemption, High-Res Timers enabled, http://svn.modlabupenn.org/personal/iano/realtime_gumstix/bootImages/sakoman_default/

[27] Kernel Image from linux-rt source with Full Preemption, High-Res Timers enabled, http://svn.modlabupenn.org/personal/iano/realtime_gumstix/bootImages/10-11-2012/

[28] Kernel Documentation, Scheduler. https://github.com/torvalds/linux/blob/master/Documentation/scheduler. October 31, 2012.

[29] Understanding the Linux Kernel: Chapter 10: Process Scheduling. http://oreilly.com/catalog/linuxkernel/chapter/ch10.html, October 31, 2012

[30] Git repository for actively developed real time linux, git://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git, October 31, 2012.

[31] stress command. http://weather.ou.edu/~apw/projects/stress/. October 31, 2012.

[32] Non-Uniform Memory Architecture Manpage, http://www.kernel.org/doc/man-pages/online/pages/man7/numa.7.html, November 5, 2012.

[33] High resolution timers and dynamic ticks design notes, http://www.kernel.org/doc/Documentation/timers/highres.txt, November 8, 2012.

[34] The high-resolution timer API, Jonathan Corbet, http://lwn.net/Articles/167897/, Posted January 16, 2006. Viewed November 8, 2012.

[35] A new approach to kernel timers, Jonathan Corbet, http://lwn.net/Articles/152436/, Posted September 20, 2005. Viewed November 8, 2012.