MINIRTL – HARD REAL TIME LINUX FOR EMBEDDED SYSTEMS

Nicholas Mc Guire
University Vienna, Institute for Material Physics
Computational Material Science Group, Prof. Dr. J. Hafner
Sensengasse 8 Vienna, AUSTRIA, A-1090
`der.herr@hofr.at, http://www.hofr.at`

Abstract

Embedded system development is moving away from single-task/user applications running on dedicated target platforms towards a reduced general purpose OS running on PC-like hardware. This move is well represented by Linux based embedded projects and specialized minimum GNU/Linux distributions. As a sample system and a basis for development of hard-real time embedded systems MiniRTL was developed at the University of Vienna and at FSMLabs New Mexico. A system that is designed around three major tenets: bootable off a 1.44MB media (currently simply a floppy), hard-real time capabilities, based on FSMLabs' current RTLinux development, maximum compatibility to a "standard" desktop development system running an up-to-date kernel. In this paper strategies to reach these goals will be described and the successful implementation in a running system, MiniRTL, shown.

1 Introduction

System software used in embedded systems covers a range spreading from micro-kernels starting at a size of 40 KB up to full-wedged operating systems such as Linux with a run time kernel in MB range. In this paper we are concerned with the high end of the embedded systems domain. More precisely, we deal here with embedded systems based on GNU/Linux, referred to as Linux embedded user system, which try to provide an as close to complete user environment on the embedded system as possible.

Expressing the size of such systems in terms of memory is hard. They start somewhere in the range of 2 MB disk and 2 MB RAM requirements for a 1.0.9-ELF kernel (Linux lite) and 2 MB disk with 4 MB RAM for 2.0.x or 2.2.x kernels. If these systems are to do more than the absolute minimum, however, then a 4 MB disk and 4 MB RAM mark the bottom line at which such a system can be identified as Linux embedded user system. In particular, these systems are appropriate for mobile units, a small series of special devices or systems that need extensive maintenance and remote monitoring.

Traditionally, system software for embedded systems has been developed independently of general purpose operating systems. So why take mainstream Linux as a basis and drop it to an embedded system? The answer is actually quite simple: GNU/Linux is a stable desktop system which provides a complete set of development tools. Thus, it makes life very much easier as it offers the possibility of doing development, debugging, optimizing and testing on the desktop system, dropping the results to the embedded platform then. There is no need for any special development environments and no need for proprietary "tools". The decision to implement the current kernels, although they are quite large, is based upon:

- The kernel's features, especially in the networking and hardware support area.
- It is hard, if not impossible, to maintain an independent kernel development track without tremendous manpower behind it.
- The availability of well-documented open source kernel eases development for embedded systems greatly

Sticking with the mainstream kernel makes it easy to ensure compatibility with the standard desktop GNU/Linux system. Real time Linux development is also tightly coupled to the current kernel development in many essential aspects. Notably, full SMP support requires current RTLinux kernel/patches,

and support for flash-disk and disk-on-chip are just emerging in 2.4.0, although it might not always be necessary to have the newest version, for a particular application.
RTLinux for embedded platforms is in general intended for high-end projects and for projects that want to utilize the advantage of using commodity-component PCs. Even if a midi-tower equipped with an "old" i386/i486 is not recognized as an embedded system by most people, from a functional standpoint this is often a very interesting alternative for low-cost and prototyping systems.

2 Features of MiniRTL

MiniRTL was originally based on the Linux router project. Naturally, it is a little archaic and you should not expect GNU/emacs as the system default editor. The main features available on this minimum system are not really specific to this system but rather simply the standard GNU/Linux features. We list them here because these capabilities might not have been taken into account when considering embedded system design:

- FSMLabs hard real-time Linux kernel 2.2.14 RTLinux 2.3.
- glibc-2.0.7 (at time of writing 2.1.3 is at work).
- Full support of the x86 chips from 386 upward.
- SMP capable (if you really need power).
- Low latency on disk access and high disk bandwidth with RAM-disk usage.
- Support for most standard PC hardware (embedded commodity components PC).
- Full network support (inetd, SSHD, HTTP, DNS, NFS, dial-in connectivity).
- Shell (ash) access at the console and via the network, via telnet (for those who like insecure connections...)
- Secure access via SSH/SCP (...for those who don't).
- Capable of off-site logging via syslog, for error diagnostics and accounting.
- mini_httpd with full cgi-bin support for simple monitoring.
- No specialized software required for developing your own projects.
- Source availability easing development of your own concepts.
- Modular structure, eases adding your own package as a tar.gz archive.
- No specialist required for administration, it "standard" GNU/Linux.

You might not need all this, and you might be missing something, but Mini-RTL should give you a good idea of what can be squeezed onto 1.44 MB! Considering this, a 2 MB flash-disk based system gives you lots of space to play with.

3 Designing a Minimum System

Minimizing disk usage in a RAM-disk based system is performance-critical. Most embedded systems are low-memory systems, at least by desktop standards. A 2.2.14 kernel will boot and run in 4 MB, but there is little room for applications in this setup. Linux is quite sensitive to low-RAM setups. We might be exaggerating a little by stating that doubling RAM on low memory systems will increase overall performance just as much as doubling CPU speed.
A RAM-disk resides in buffer cache, so increasing disk usage will reduce available memory. This implies that we should review strategies for optimizing performance, and also makes clear why dynamic disk usage during boot-up is not really a critical point (as long as you never try to exceed available total ram, which can happen if you try to put the entire system into a single archive file). So, there is little necessity to reduce the image size of the booting system, as only run-time size is relevant.

3.1 Reduce the system size

Designing the system requires the most time, because it is hard to tell what one can drop and what one really needs in a minimum system. Fortunately, there are some measures that can be taken to reduce system size, as discussed in the following.

3.1.1 Exploiting redundancy

Linux is very redundant when it comes to executables: you can do the job of displaying a file on the console with cat, tail, more, less, grep and even dd. So finding the redundant executables and defining the scope really required is the first step. Some questions to ask here are:

- What do we really need to do on-site? What can I move off-site?
- Do we need all options available with ls, tar and so on? (e.g. ls of bussybox)
- Can we substitute some executables by simple shell-scripts? (e.g. /bin/grep on MiniRTL)
- Can we substitute system-services such as tftp for ftp?
- What do we need only at boot-up? (network modules , boot-media support)
- What can be added at runtime, either by uploading or by mounting it? (most admin tools)

3.1.2 Deletion

One point here is probably unique to minimum systems, and so deserves a little more detail: A normal system has its executables in place, and these will be used at the next system boot. Embedded systems that run in RAM-disk are running off a decompressed image copy, so they don't need to preserve files if they are not required for the system's operation. They can simply be removed. Candidates for removal are:
The kernel image file! Once it's loaded, we don't need it. Initialization scripts. Modules that we will not need to reload until a system reboot. Executables that only do initialization (sshd key generation).

3.1.3 Compression

Many executables/modules may reside on the file system in compressed form, because they will only be needed for specific purposes during maintenance. It is essential, though, to make sure that these compressed executables are not called from scripts without decompressing them first. This sounds obvious, but the dependencies in even a 2MB system can be quite complex. It is in general not a trivial task to ensure this. Candidates for this are all executables that will only be used by administrative personnel and not called by scripts (after boot up that is!), such as editor(s), maintenance programs (such as ifconfig, ping, rmmod) and file utilities (such as ctar, ae).
All these measures can reduce system runtime size by 30 to 40 percent without any black magic. A 40 percent file-system size reduction means a substantial increase in RAM available at run-time.

3.2 Reduce run time sizes

The above size reduction was achieved by using compression, which is is limited to the file system size of executables. The run time size is not affected in the same manner, so the next step, naturally, is to reduce run time size as well. This does not always correlate with a reduction in file system size!

3.2.1 Restricting kernel resource

There are lots of provisions made in the regular Linux kernel that assume operation as a full multiuser multi-tasking system, with no real limits to the number of processes and users. As a consequence, the more resources are allocated than a minimal system needs. As a first step, therefore, optimization of the Linux kernel for minimum systems consists of no more than throwing out some of these resources (pty's, tty's, hdX and lots of hooks in /dev).

Real optimization, in the sense of modifying kernel behavior to better use specific resources, is not intended (and probably not that easy either). It's also counterproductive, due to the fact that there is a new Linux kernel out there every few weeks. Removing resources is simple and requires no real kernel hacking. Starting points may be found in include/linux/*.h. Be careful with the limits defined there, though, for not all may be reduced without side-effects. If you modify any #define statement, you will need to grep through the kernel tree to find if it is referenced anywhere else, and thus create side-effects if modified.

As an example, there will rarely be a need for 63 console devices on an embedded system, so the maximum number of consoles in include/linux/tty.h can be reduced to 3 (giving you four consoles). Other good candidates are device files. You will not need to access all hda* to hdh*, as it is difficult to imagine an embedded system running with eight IDE devices hooked up.

A significant drawback to compile-time modifications is that you will have to dig around in the kernel source every time a new kernel comes out. This means that you will have a job to do every few weeks if you want to keep up with the current kernel. Putting minimization flags in the config process would be a nice solution, but I doubt that it will find its way into the main stream kernel.

3.2.2 Shrinking executables

Many executables that you use on a day-to-day basis on your desktop system have many options that you will rarely need and could either spare completely or substitute with a generic command. Going through the executables you designated for your minimum system and stripping out options, error messages and built-in help is an easy way of reducing such files.
Also, consider compile-time options, as they also influence size. On desktop systems this isn't a big deal, so often the Makefiles will provide the safest options, not the most efficient with respect to size. As a general way to reduce size, don't include the default libs, but explicitly include the libs you need. To do so, use the -nostdlib and -nostartfiles flags with gcc, and include libc with -lc explicitly along with any other libs your app needs.

3.2.3 Compressing output

Whenever programs on a minimum system produce data, you might consider outputting it in a compressed format from the beginning. This will not only save scarce disk-space, downloading to a server or pushing data over an NFS-mount can also be substantially improved if a compressed format is used. Alternatively, data files and log-files may be compressed periodically by a simple cronjob, especially for local log-files on embedded systems where in many cases only recent events will be relevant.

3.2.4 multi-call binaries

An interesting concept to pack lots of functionality on a minimum system is the so-called busybox concept. It is a monolithic collection of base functions in a single executable with function selection by the name of the calling process. This is handled by symbolically linking functions names to the busybox executable.
The advantage is that there is little overhead in the executable for argument handling and the like, since there is only one parser for all linked executables. Also in the busybox, functionality is reduced to a minimum, which comes at a price: some standard functions behave a little differently than one might expect, with some options missing and others reduced in scope. Adding functions to the busybox is quite simple, but it requires recompiling the entire suite and modifications to the system initialization to set up the required links appropriately.

3.2.5 The Busybox vs. Standard Tools

First, let us look at the file size of the busybox (0.28) when compared to the sum of the sizes of the corresponding standard Linux tools. The busybox occupies 65KB, whereas the sum of the Linux tools occupies 608KB. Even the stripped Linux tools still require 592KB (egcs-2.91.66, glibc 2.0.7).
This comparison might not seem fair, since many options available in the standard tools are not available in the busybox, but the essence is that you can pack lots of functionality into a tiny executable, if reduced to the minimum needs of administration. The real challenge is deciding precisely what is needed and what can be removed. With all the dependencies within an OS like Linux (and the use of shell-scripts for many jobs), this is not easily decided. This leads us back to one of the motivations for such a minimum system: reduced system complexity, which eases understanding of such dependencies.
The second comparison is the hello world program in C compiled with gcc (egcs-2.91.66) as an ELF-executable and the size difference of a hello world function added to busybox called via a link named hello. The ELF-executable comes in at 33KB raw and 7KB stripped. The increase in size of the busybox, however, is a nominal 161 bytes.
This shows how efficiently the overhead reduction is achieved by having one main routine and calling everything else as a function, via the name of which busybox is called.

3.3 Libraries

Some of the library problems were mentioned above, glibc is a very large and powerful library, but for minimum systems it's a problem since it is very resource consuming. Nevertheless we stick with glibc, because reducing its size is not only complicated (you must figure out all function calls that are unused and remove them), but also because it poses a compatibility problem. If you try to optimize by modifying libraries, you lose compatibility with your desktop system. At the same time, it means maintaining a private version of the lib, and you don't want to maintain your own libc track!
Stripped libraries are dramatically smaller, and since debugging can comfortably be done on the desktop-system, there is no need to include debug symbols on MiniRTL. The same holds for executables that can be stripped, thereby massively reducing size. To reduce the number of required libraries, it is best to define a set of libraries for the minimum system and then strictly build on those. This is not such a big problem, due to the vast amount of software/sources on the Internet, it is quite easy to find editors, scripting languages and the like that will not need any special libraries. Naturally, the system will have a little bit of an archaic touch, but that's ok; you're not expected to work full time with ash and ae as

your shell and editor. For administrative jobs, you can get used to it.

3.3.1 The minimum list of libraries

for glibc-2.0.7pre6 assuming network support is

```
ld-2.0.7.so
libc-2.0.7.so
libcrypt-2.0.7.so
libdl-2.0.7.so
libncurses.so.4
libnsl-2.0.7.so
libnss_db-2.0.7.so
libnss_dns-2.0.7.so
libnss_files-2.0.7.so
libresolv-2.0.7.so
libss.so.2.0
libutil-2.0.7.so
libuuid.so.1.1
```

3.4 Run-Time Optimization Strategies

The optimization strategies presented here are only applicable in very special situations and might seem a little weird at first glance. Yet consider that many embedded systems run for months without any human intervention. For these systems, optimization strategies that are repaid by an increase in administrative complexity might be acceptable.

3.4.1 Kernel Resource Optimization

The modular structure of the Linux kernel permits optimization of run time size. Modules for network-support may be loaded, enabling the embedded system to drop log/data-files to a central server system, and then be removed from the kernel until needed again. The same holds for file system support. The size reduction may be performance-relevant on low-memory systems.
A further kernel optimization may result from reducing allocated resources, either by kernel parameters at boot-up or by reducing resources in the kernel source. This does not really require kernel hacking, merely going through kernel source files and stripping down resources that you will not need for embedded systems (e.g., by setting the maximum number of IDE devices to 2 instead of 8, reducing number of consoles to 4 and so on). These modifications can be done without going very deep into the kernel and are not too hard to apply again when a new kernel comes out. This stripping of resources will not substantially reduce the compressed kernel image but it will reduce the run time kernel memory trace.
Many kernel resources can be tuned on your desktop system via the sysctl interface operating via /proc. By tuning system settings on the full system you can find an optimized setting for your kernel. On a minimum system sysctl would be a real waste since it is very large, so the optimized values for free pages page-cache etc. must be set in the kernel source and the kernel recompiled. This is not as bad as it sounds, since optimization is rarely so application-specific that it would change within a running system.

3.4.2 File Systems

Many file systems are designed for very big systems. Embedded systems rarely need a directory depth of 1024 directories or maximum filename length of 1800 characters. Memory can be saved if the correct file systems are used with the appropriate options (compare msdos/minixfs/ext2fs/reiserfs...). Many file-systems have options to reduce the number of inodes provided or reduce filename length but keep in mind that every restriction on the file system (DOS 8.3 filename length or mkfs.minix -n14 giving you 14-character maximum filename length) is paid for by having to be careful of file conversions when moving files around (notably, msdos-fs is a real limitation), this easily can lead to hard-to-detect side-effects.

3.4.3 On-demand Loading

Not all executables may be necessarily resident on the system. You can have administrative tools like an editor or some additional kernel modules only required for administration on the remote site and simply upload them on demand, removing them when the tasks have completed. This is performance-critical since, as noted above, the file system resides in buffer cache. Alternatively, you can have modules with nfs support on the embedded system, which can temporarily increase disk space availability tremendously. You can even swap via nfs, but this is not suitable via a modem-line! Of course, nfs has security implications that need to be considered if running such a system outside a secure network.
The point of all this is that there are very few resources really needed on-site. Much can be moved off-site as long as there are mechanisms available to hook them up on demand (via scripts, cron etc.).

4 Kernel modifications

The kernel modifications described here were not done as part of this project, but were rather the basis for it. Describing them all here in detail would not be possible, so we will only give a brief overview of the concepts involved.

4.1 initrd

To boot off the initial RAM-disk, you need to modify the standard kernel a bit. Basically, it is no more difficult than replacing the hook in init/main.c that points to the regular root device and make it point to the initial RAM-disk. Using raw images this would be all that is required to run a system, yet running raw images is not very comfortable during development and also quite messy to modify. Since raw images are required to tell the kernel the exact hexadecimal address where to jump to, raw images have no file system hook. So, to permit the usage of standard tar.gz archives, some additional modifications are added to create an initial minix file system on the RAM-disk. The kernel then unpacks all it needs into the file system.
One problem with this is that the initial console does not exist at the time when the kernel actually is ready to boot up the system, because it has not been created yet. This is solved in a very pragmatic way by having a special device packed on the disk (called linuxrc.tty) which is a minimalist tty, only used at boot time.

4.2 mkminixfs and untar

In drivers/block/rd.c, the RAM-disks are created by the kernel at boot time. This is where the code for creating the minix file system on /dev/ram0 and unpacking the root archive from tar.gz form is inserted. This is essential because now the root image is no longer a raw image, but simply a standard tar.gz file, so manipulation is comfortably done. Due to this add-on, it is possible for any Linux user to create the modified archive files. All you need to do is pack everything you would like to have in a tar archive (provided you can get it on a 1.44 MB floppy) and gzip it, then tell linuxrc to unpack it via the "command-line options" in syslinux.cfg. The setup script (linuxrc) can access these command-line options via /proc/cmdline, this is a quite simple way to pass any string from the syslinux.cfg (the syslinux configuration file on the floppy) to the running linux system.

4.3 FSMLabs rt-linux

Besides the above modifications, the normal RTLinux patches version 2.3 (kernel 2.2.14) were applied, with no modifications. The front-end application interfaces are restricted in the libs available on the Mini-RTL system and compile-time options where introduced to reduce the size of executables. For the standard demo-apps, no modifications were required, but application design for a minimum system must take the lib-restrictions into account.

4.3.1 rt-linux concept

FSMLabs rtlinux is based on a very small layer that is put beneath the actual Linux kernel. It is dropped in as a kernel patch and permits a system to utilize all the capabilities of a regular GNU/Linux desk-top system. Building these hard real-time capabilities as modules allows for runtime insertion, reducing the kernel changes to a minimum. This layer incorporates the following basic functionalities:

- interrupt interception ,no real interrupt ever reaches Linux, all interrupts destined for a non-rt task are flagged and passed to Linux when no rt-task is ready to run.
- timing precession in the nano-second range (rtl_time.o)
- scheduling of real time tasks independently of the Linux scheduler (rtl_sched.o), rtlinux is running a loadable scheduler beneath the non-rt level, this give great flexibility in design and optimization of the rt-scheduling policy.
- simple communications with the non-rt side via rtl_printk ,rt-fifos (rtl_fifo.o) and shared memory (mbuff.o).
- process synchronization via mutex and mmap functions (mutex.o mmap.o)
- runs Linux as its low-priority idle-task.

5 Example programs on MiniRTL-V2.3

The following list of example real-time linux programs are available on MiniRTL-V2.3 ,kernel modules listed as .o , front-end programs without extensions:

```
hello.o
sound.o
read_lpt / rectangle.o
read_lpt / sched_toggle.o
lpt_irq + monitor / rt_irq_gen.o
frank_app / frank_module.o
monitor / oneshot_test.o
monitor / periodic_test.o
monitor / rtc_fifo_test.o
rtc_test.o
multitask.o
monitor / rt_process.o
mutex.o
rtpu / rtp.o
```

In addition to these examples the basic rtlinux modules are also on-disk:

```
mbuff.o
rt_com.o
rtl_fifo.o
rtl_posixio.o
rtl_sched.o
rtl_time.o
```

More information on rtlinux, writing code and understanding it can be found at http://www.rtlinux.org. The following is a brief description of some of the examples, these show the basic real-time program structures, interrupt service routine directly mapped to a hardware interrupt (sound.c), single scheduler managed real time task (hello.o) , multiple real-time task managed by the real-time scheduler (rectangle.o) and external signal triggered real-time task (rt_irq_gen.o).

5.1 "Hello World": hello.o

One of the first C-programs to master is "hello world", a simple program that will print "hello world" on your screen. In the real-time linux world its not much different, this simple "real-time hello world" will write hello world from the rt-side to the non-rt-side of your running rtlinux box. To see the messages hello.o is writing to you, you must call the Linux command dmesg, which will print the kernel messages to stdout. This is because on the real-time side of your box you can't directly access the console, so hello.o drops a message to the linux kernel and that in turn prints it via printk.

hello.c close to the simples module possible, init_module, cleanup_module as with all modules, and start_routine, the actual task, which is only a periodic rtl_printk of a message and any arguments received. After inserting hello.o with insmod, simply execute dmesg, and it will talk to you.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>

pthread_t thread;

void * start_routine(void *arg)
{
   struct sched_param p;
   p . sched_priority = 1;
   pthread_setschedparam (pthread_self(),
      SCHED_FIFO, &p);

   pthread_make_periodic_np (pthread_self(),
      gethrtime(), 500000000);

   while (1) {
      pthread_wait_np ();
      rtl_printf("I'm here; my arg is %x\n",
         (unsigned) arg);
   }
   return 0;
}

int init_module(void) {
   return pthread_create (&thread, NULL,
      start_routine, 0);
}

void cleanup_module(void) {
   pthread_delete_np (thread);
}
```

5.2 Sound with a pc-speaker: sound.o

The PC-speaker is driven by a programable timer/counter chip in your PC, it will generate a periodic signal, that can be thought of as a "sine-wave" or simply a periodic signal of some shape or simply a beep. If this beep is turned on and off in a reasonably fast and precisely timed fashion then one can produce "sound". This condition is what makes it clear why such a speaker driver would not properly work when executed from regular non-real-time linux, on an unloaded box it might work more or less, but with increasing load distortion would become unacceptable.

The data used to control the "on" or "off" state of the speaker is 8-bit mu-law encoded audio data (regular .au file). It is reduced to 1-bit and this is then used to turn the speaker on or off , depending on what was left of the 8-bit after running it through a simple filter function.

The sound.c is a little different from other sample programs, in that it is not scheduled and thus managed by the real-time scheduler, but it is simply assigned as the interrupt handler of interrupt 8. This is done for performance reasons, but is restricted to a one-task situation. init module will save the CMOS settings then assigned the "sound" routine as the interrupt handler for interrupt 8, program the CMOS clock to generate interrupts at 8KHz and then exit.

So this is basically what you would do on a DOS box to run a routine as interrupt service routine directly. This interrupt service routine then reads the data in from the fifo and filters it from 8-bit to 1-bit in filter() then sets the speaker active of deactivates it according to the filter output. cleanup_module will reset the state of the CMOS clock , remove the fifo and free the RTC interrupt.

```
#include <linux/mc146818rtc.h>
#include <rtl_fifo.h>
#include <rtl_core.h>
#include <rtl_time.h>
#include <rtl.h>
#define FIFO_NO 3

static int filter(int x){
   static int oldx;
   int ret;

   if (x & 0x80) {
      x = 382 - x;
   }
   ret = x > oldx;
   oldx = x;
   return ret;
}

unsigned int intr_handler
(unsigned int irq, struct pt_regs *regs) {
   char data;
   char temp;
   /* clear IRQ */
   (void) CMOS_READ(RTC_REG\_C);
   if (rtf_get(FIFO_NO, &data, 1) > 0){
      data = filter(data);
      temp = inb(0x61);
      temp &= 0xfc;
      if (data) {
         temp |= 3;
         }
      outb(temp,0x61);
      }
   rtl_hard_enable_irq (8);
   return 0;
}

char save_cmos_A;
char save_cmos_B;

int init_module(void)
{
   char ctemp;
   rtf_create(FIFO_NO, 4000);

   /* this is just to ensure that the */
   /* output of the counter is 1 */

   /* binary,mode 0,LSB/MSB,ch 2*/
   outb_p(0xb0, 0x43);
   outb_p(0x1, 0x42);
   outb_p(0x0, 0x42);

   rtl_request_irq (8, intr_handler);

   /* program the RTC to interrupt at 8kHz */
   save_cmos_A = CMOS_READ(RTC_REG_A);
   save_cmos_B = CMOS_READ(RTC_REG_B);

   /*32kHz Time Base, 8kHz interrupt freq.*/
   CMOS_WRITE(0x23, RTC_REG_A);
   ctemp = CMOS_READ(RTC_REG_B);
   ctemp &= 0x8f; /* Clear */
   /* Periodic interrupt enable */
   ctemp |= 0x40;
   CMOS_WRITE(ctemp, RTC_REG_B);

   rtl_hard_enable_irq (8);
   (void) CMOS_READ(RTC_REG_C);
   return 0;
}

void cleanup_module(void)
{
   rtf_destroy(FIFO_NO);
   /* restore the original mode */
   outb_p(0xb6, 0x43);
   CMOS_WRITE(save_cmos_A, RTC_REG_A);
   CMOS_WRITE(save_cmos_B, RTC_REG_B);
   rtl_free_irq(8);
}
```

5.3 Playing with the Parallel Port

These programs requires that you have your MiniRTL box connected to a second Linux PC via standard PLIP cable.

5.3.1 rt_irq_gen.o

rt_irq_gen.o waits for an input on the parallel port in polling mode. This polling-task is in a busy-wait loop until something occurred at the parallel-port, it will "freeze" your rtlinux box until something comes in or it is timed out. if you remove the timeout in rt_irq_gen.c then your rtlinux box will only stay "alive" as long as you run the lpt_irq program on your second linux box connecting it via a plip-cable.

5.3.2 rt_irq_gen.c

```
#include <rtl.h>
#include <rtl_time.h>
#include <rtl_fifo.h>
#include <asm/rt_irq.h>
#include <asm/rt_time.h>
#include <rtl_sched.h>
#include <asm/io.h>
#include <linux/cons.h>
#include <pthread.h>
```

```
#include <time.h>
#include <rtl_sched.h>
#include <rtl_sync.h>
#include <unistd.h>
#include <rtl_debug.h>
#include <errno.h>
#define LPT 0x378
#define LPT_IRQ 7
#define RTC_IRQ 8

struct sample {
   hrtime_t min;
   hrtime_t max;
};

/* all time values in nano-seconds */
#define TIMEOUT 10000000
#define SAMPLES 10

pthread_t thread;
hrtime_t min_response;
hrtime_t max_response;
struct sample samp;
int samples;

void * irq_gen(void *arg) {

/* putting these in registers prevents
 * additional delays due to load and
 * store to memory during the timing loop
 */
register int i;
register int orig;

int old_interrupts;
hrtime_t before, after, response;
struct sched_param p;
p . sched_priority = 1;
pthread_setschedparam (pthread_self(),
   SCHED_FIFO, &p);

pthread_make_periodic_np (pthread_self(),
   gethrtime(), 100000000);

while (1) {
   min_response = 2000000000;
   max_response = 0;
   for (samples = 0;samples < SAMPLES;samples++){

     /* turn off interrupts so we really get the */
     /* round-trip time by the busy-wait loop */
     rtl_no_interrupts(old_interrupts);
     outb_p(0xf, LPT);  /* ACK is 1 for logic 0 */
     orig = inb_p(LPT + 1);

     outb(0x0, LPT);  /* trigger an interrupt */
     before = gethrtime();
     i = 0;

    /* the busy wait loop,poling the interrupt */
    /* pin of the parallel port .*/
     while ((inb(LPT + 1) == orig) && i++ < TIMEOUT);
        after = gethrtime();

       /* if this loop were not present the system */
       /* would wait for ever until an interrupt is */
       /* polled on ACK of the parport. */
        if (i >= TIMEOUT) {
           rtl_printf("timeout!!!\n");
           return 0;
           }

      response = after - before;
      if (response < min_response) {
         min_response = response;
         }
      if (response > max_response) {
         max_response = response;
         }

     /* restore interrupts before sending */
     /* the thread to sleep */
      rtl_restore_interrupts(old_interrupts);
      pthread_wait_np();
      }
   samp.min = min_response;
   samp.max = max_response;
   rtf_put(0, &samp, sizeof(samp));
   }
   return 0;
}

int init_module(void)
{
rtf_destroy(0);
rtf_create(0, 4000);
return pthread_create (&thread,NULL,irq_gen,0);
}

void cleanup_module(void)
{
rtl\_printf ("Removing rt_gen_irq on CPU %d
   and clearing ACK on %d\n",rtl_getcpuid(),LPT);
outb_p(0xf, LPT);      /* clear ACK on parport */
pthread_delete_np (thread);
rtf_destroy(0);
}
```

5.3.3 lpt_irq

this is a simple non-rt busy-wait loop waits for ACK on the parallel port ,then it will toggle pins D0-D7 to produce an ACK. Again this runs on the second linux box and is connected to the MiniRTL box via PLIP-cable. if you drive the load high on the computer running lpt_irq you can see that the MiniRTL system will time out sooner or later since the non-rt program lpt_irq will fail to deliver the ACK in the required timeout defined in rt_irq_gen.c.
To prevent this behavior you can use rectangle.o on

the second box, which is a rt-program that does basically the same as lpt_irq just it does it in real-time and thus driving the load high on the PC will not influence the behavior of the MiniRTL system. This deterministic behavior is what its all about and this simple example shows the difference most drastic.

5.3.4 lpt_irq.c

```
#include <stdio.h>
#include <asm/io.h>
#include <unistd.h>
#include <sys/io.h>
#define LPT 0x378

int main(void)
{
   int in;
   if (ioperm(LPT, 3 , 1) < 0) {
      fprintf(stderr,
         "ioperm: error accessing IO-ports");
   exit(-1);
   }

  /* wait for ACK to go low (logic 1) on
   * the parallel port and then produce a
   * HI/LOW on D0-D8 to trigger an interrupt
   * as response , see PLIP.txt for pin-out
   */
   while(1) {
      in=inb(LPT+1);
      in = in >> 3;
      in = in & 0x0f;
      if(in==0){
         printf("got interrupt on LPT\n");
         outb(0xff,LPT); /* logic 0 on LPT*/
          usleep(100);
         outb(0x0,LPT);  /* logic 1 on LPT*/
        }
     }
}
```

5.4 Real-time Multitasking: rectangle.o

As mentioned above rectangle.c will also toggle the parallel port, but it will do it load independent, so if rectangle.o is running on the second PC then you will not be able to timeout rt_irq_gen.o on the MiniRTL system. At the same time rectangle.c is an example of simple real time multitasking. Two threads are running, one that sets the pins D0-D7 "Hi" and one that set them "Lo".

5.4.1 rectangle.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <rtl_time.h>
#include <rtl_sched.h>
#include <asm/io.h>
#include <time.h>
#include "common.h"

int period[2]={20000000,1000000};
int periodic_mode=0;
int nibl=0xff;

pthread_t THREAD[2];

void *bit_toggle(void *t) {

   int index = (int)t;
   pthread_make_periodic_np(THREAD[index],
      gethrtime(), period[index]);

   while (1){
      outb(nibl,LPT);
      nibl = ~nibl;
       pthread_wait_np();
      }
}


int init_module(void)
{
   pthread_attr_t attr;
   struct sched_param sched_param;

   pthread_attr_init (&attr);
   sched_param.sched_priority = 1;
   pthread_attr_setschedparam (&attr,
      &sched_param);
   pthread_create (&THREAD[0],  &attr,
      bit_toggle, (void *)0);
   pthread_create (&THREAD[1],  &attr,
      bit_toggle, (void *)1);

   return 0;
}

void cleanup_module(void)
{
   pthread_delete_np (THREAD[0]);
   pthread_delete_np (THREAD[1]);
}
```

6 Acknowledgments

Thanks goes to the head of the Computational Material Science Group, Prof. Dr. J. Hafner, for encouraging this slightly off-topic work at his institute. Further thanks to Victor Yodaiken of FSMLabs for supporting this work and supplying the Internet resources required to successfully run such a project.

7 Availability

MiniRTL is available from the following sites on the Internet.
`http://www.rtlinux.org/rtlinux.new/minirtl.html`
`ftp://ftp.rtlinux.org/pub/rtlinux/minirtl`
`http://www.fsmlabs.com/`
`ftp://ftp.fsmlabs.com/pub/rtlinux/minirtl/`
`http://www.thinkingnerds.com/`
`ftp://ftp.thinkingnerds.com/pub/projects/`

Help and support can also be found on the rtlinux mailing list hosted at rtlinux.org, to subscribe to this mailing list send mail to `majordomo@rtlinux.org` with the line `subscribe rtl YOUR_EMAIL_ADDRESS` in the body of the mail.

References

[1] M. Baraban, New Mexico Institute of Mining and Technology
A Linux-based Real-Time Operating System, Thesis (1997).

[2] Real Time Linux Home-Page,
`http://www.rtlinux.org/`,ftp://ftp.rtlinux.org/

[3] The Linux router project
`http://www.linuxrouter.org/`

[4] initrd-archive kernel patch
`ftp://ftp.psychosis.com/linux/initrd-arch/`

[5] linuxrc-always kernel patch
`ftp://ftp.psychosis.com/linux/initrd-arch/`

[6] Dave Cinege `<dcinege@psychosis.com>`, Eric B. Andersen `<andersee@debian.org>`, `ftp://ftp.lineao.com/pub/busybox`

[7] David A. Russling, The Linux Kernel,
`at any LDP-site tlk-0.8-3.ps.gz`