RTLinux Kickstart Session

Der Herr Hofr.at
The Thinking Nerds
Aignergasse 10, A-2020 Hollabrunn, Austria
der.herr@hofr.at

Abstract

RTLinux Kickstart session at the 4th Real Time Linux Workshop. This kickstart session introduces RTLinux/GPL instalation and first steps on a SuSE 8.0 system running RTLinux-3.2-pre1 (Kernel 2.4.18-rtl3.2-pre1). The basic steps should work out on other but SuSE 8.0 systems as well as long as they are based on gcc-2.95.X and not gcc-3.X .

1 Introduction

This manual is not a in depth introduction to installing and running RTLinux/GPL but, as the name says, a kickstart only. It should guide you step by step to get you up and running on RTLinux/GPL quickly. Although this document describes the steps for RTLinux/GPL on a SuSE-8.0 system, steps will more or less be the same on a different distribution, and should still give you some guidance for installing RTAI on a SuSE based Linux-box. Feedback, especially on trying this for other platforms, and a clean RTAI version (or RTAI specific comments) would be appreciated.

2 System Install

- CD1 basic installation (click through default install)
- Boot-screen: 'Installation - Safe settings'
- Language Selection: 'English(us)' -> [Accept]
- In the pop-up window, select '[*] New-Install' -> [OK]
- In the main window click -> [Accept]
- Now you get a list of the instalation settings (Language, Keyboard etc.) is things are OK -> [Accept]

For details on installing Linux refer to the Manual of the distribution you are using - SuSE 8.0 comes with a fairly complete manual and we don't anticipate to reproduce this manual here.

- Next you will be prompted for a final confirmation that a new install should be done - THIS WILL DELETE ANY EXISTING PARTITIONS AND INSTALLATIONS
- In the pop-up window - "Confirm installation" -> [Yes install]

The installation now runs on its own. If you want to check errors or if the instalation seems stuck then change to the text consoles by holding down `<CNTRL>-<ALT>` and pressing `<F1>` for the first console `<F2>` for the second and so forth - the installer is using all 10 consoles available by default to report progress, settings and problems. Especially if the installation fails or seems extremely slow the information on these screens is vital for analysis - and in many cases will be enough for you to fix it on your own - before you bother calling any help-desk or support hot-line (which generally are not that hot...) note down infos from these screens as this increases the probability that you can get help dramatically.

- Next pop-up - to inform you that lilo has been written to disk -> [OK]
- You are then prompted for the root password - enter it twice and (move from the first to the second field with the mouse or by pressing <TAB>), WRITE IT DOWN BEFORE clicking -> [Next]

  Now you can enter a user account - this is not really necessary for now, especially you should note that at this point low-security passwords for users are accepted where as these are not accepted later in the system (i.e. password == user name).
- The graphics desktop environment screen can be left at the defaults, adjustment can be done once the system is up and running, for this session we will use the text-consoles any way -> [Next]

The SuSE installer needs to run a number of scripts now to finalize the configuration of the system. After these scripts have run you could configure your peripherals. For this kickstart session we will skip this, the only thing to check is if your hardware is detected properly in the list. Configuration can be done later by launching yast (which will present you the same list).
The instalation terminates by going into run-level 5 and presenting a graphics install menu - if you only set up a root-user then no users are presented but you can login as root.
Switch to the first text-mode console by holding down <CNTRL>-<ALT> and pressing <F1>

```
linux login: root
Have a lot of fun...
linux:~ # reboot
```

Don't forget to remove the install CD now...

3 Preparation for RTLinux

(If the system boots...) login as root on tty 1 (<CNTRL>-<ALT>-<F1> if you have the graphics login in front of you)
The system is now a base instalation, to compile and install a new kernel we need a few extra packages. These could be installed with Yast, the SuSE instalation tool, or with some other install tool on other distributions, but behind all these tools are RPM packages, and in many cases you need to install RPM packages on your development system that are not provided by the Distribution provider, so lets use rpm tools to install the packages we need.
First mount cdrom Nr 2

```
linux:~ # mount /cdrom
```

This is mapped via `/etc/fstab`, so if you want to know your real CDROM device number do

```
linux:~ # grep cdrom /etc/fstab
/dev/cdrom /media/cdrom auto ro,noauto,user,exec 0 0
```

or

```
linux:~ # df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda5              9919796   1604360   8315436  17% /
/dev/hda1                23302      3857     18242  18% /boot
shmfs                    63264         0     63264   0% /dev/shm
```

after mounting it with mount `/cdrom`.
To locate the packages you are interested in on a newly installed system you should not expect to have any rpm-database or the like available (yet) as the instalation may not include these tools in a base install setup. so lets scan the cdrom for gcc in a first step.

```
linux:~ # cd /cdrom
linux:/cdrom # find . | grep gcc
/cdrom/suse/d2/gcc-2.95.3-216.i386.rpm
```

So now that we located the gcc rpm file for the 2.95.3 gcc (note that 2.95.X is still the official kernel compiler not 2.96 not 3.X...) we can try to install it.

```
linux:/cdrom # cd suse/d2/
```

Now lets install gcc manually using the rpm tool, you just need to type the first part of the rpm name (i.e. gcc-2) and then hit the <TAB> key, if the package name is unique after the first few entered characters it will be completed and you can hit <ENTER> if it beeps at you if you press <TAB> hit <TAB> a second time to get a list of all files that would match, then type in a few more characters of the file you want and use <TAB> again to complete it.

```
linux:/cdrom/suse/d2 # rpm -i gcc-2.95.3-216.i386.rpm
error: failed dependencies:
   binutils is needed by gcc-2.95.3-216
   glibc-devel is needed by gcc-2.95.3-216
```

So this fails because gcc needs some other packages to work properly, but rpm told us which packages were missing so lets install these first.

```
linux:/cdrom/suse/d2 # rpm -i binutils <TAB>
linux:/cdrom/suse/d2 # rpm -i glibc-devel <TAB>
```

Now we can try gcc again.

```
linux:/cdrom/suse/d2 \# rpm -i gcc-2.95.3-216.i386.rpm
```

This should now work without any errors. We still need some other packages, make is needed to build anything on a Linux system, so lets install make

```
linux:/cdrom/suse/d2 # rpm -i make <TAB>
```

The linux installer needs ncurses (you would notice it when you try to start it - giving you an error message that libncurses is missing...)

```
linux:/cdrom/suse/d2 # rpm -i ncurses-devel <TAB>
```

This is all we will need to build rtlinux and a new kernel, most of the other tools that are used during the build process all ready came with the base instalation (sed, awk, patch, etc...).
Leave the `/cdrom/suse/d2` directory now so that we can unmount the cdrom.

```
linux:/cdrom/suse/d2 # rpm -i ncurses-devel <TAB>
linux:/cdrom/suse/d2 # cd
linux:~ #
linux:~ # umount /cdrom
linux:~ # eject
```

Remove the SuSE CD2 from the drive - we will not need it any more for this kickstart session - and if you want a complete system then you should get yourself a SuSE 8.0 distribution :)

4 Kernel install

mount the proceedings CD

```
linux:~ # mount /cdrom
```

unpack vanilla kernel linux-2.4.18 from Proceedings CD

```
linux:~ # cp /cdrom/kernels/linux-2.4.18.tar.bz2 /usr/src/
linux:~ # cd /usr/src/
linux:/usr/src # tar -xjf linux-2.4.18.tar.bz2
linux:/usr/src # mv linux linux-2.4.18-rtl3.2
linux:/usr/src # ln -s linux-2.4.18-rtl3.2 linux
```

copy rtlinux from the Proceedings CD and unpack it

```
linux:/usr/src # cp /crom/rtlinux-3.2-pre1.tar.bz2 ./
linux:/usr/src # tar -xjf rtlinux-3.2-pre1.tar.bz2
linux:/usr/src # ln -s rtlinux-3.2-pre1 rtlinux
```

4.1 patch kernel

Decompress the kernel patch

```
linux:/usr/src # cd rtlinux/patches
linux:/usr/src/rtlinux/patches # bunzip2 kernel_patch-2.4.18-rtl3.2-pre1.bz2
```

First test the kernel patch to see if it applies properly

```
linux:/usr/src/rtlinux/patchs # cd /usr/src/linux
linux:/usr/src/linux\#patch -p1 --dry-run \
    < /usr/src/rtlinux/patches/kernel_patch-2.4.18-rtl3.2-pre1
```

If all goes well (sure it does...) patch the kernel now

```
linux:/usr/src/linux\#patch -p1 \
    < /usr/src/rtlinux/patches/kernel_patch-2.4.18-rtl3.2-pre1
```

4.2 configure kernel

check with `lsmod` what essential kernel modules we need in the SuSE install (forget sound modules...) check network modules and peripherals that are essential. you can also get the config file SuSE used from the `/proc` directory (`/proc/config.gz`), but you need to check this config simply copying it may lead to problems (i.e. APM enabled...) .

```
linux:/usr/src/linux\#make menuconfig

Code Maturity Level Options
 [*] Prompt for development and/or Incomplete code/drivers

Loadable Module Support
 [ ] Set Version Information on all module symbols

Processor Type and feature
 Select EXACTLY your CPU or a generic low-end CPU (check "cat /proc/cpuinfo")

Filesystem
 [*] Reiserfs support
 [*] /dev file system support (EXPERIMENTAL)
```

save and exit - Note the reiserfs is needed because SuSE-8.0 installed the basic system on a reiserfs partition - other distributions prefer other filesystems, check in `/etc/fstab` what filesystems you will need on the system.

```
linux:/usr/src/linux # cp .config myconfig
```

this saves the config in a way that will not be deleted by `make mrproper`.

4.3 compile and install kernel

```
linux:/usr/src/linux # make dep
linux:/usr/src/linux # make modules
linux:/usr/src/linux # make modules\_install
linux:/usr/src/linux # make bzlilo
...
a half a million lines of confusing output later...
cp /usr/src/linux-2.4.18-rtl3.2/System.map /
if [ -x /sbin/lilo ]; then /sbin/lilo; else /etc/lilo/install; fi
Added linux *
Added failsafe
Added memtest86
make[1]: Leaving directory `/usr/src/linux-2.4.18-rtl3.2/arch/i386/boot'
linux:/usr/src/linux #
```

edit /etc/lilo conf add entry for rtlinux use failsafe settings to start

```
linux:/usr/src/linux # cd /etc
linux:/etc # vi lilo.conf
```

Add the following entry, basically this is a copy of the failsafe entry (so leave the root= parameter as you find it there not the way it is here !), the only changes are, remove the initrd entry and change the image location to `/vmlinuz` where `make bzlilo` put it, give the entry a unique name "rtlinux".

```
  image  = /vmlinuz
  label  = rtlinux
  root   = /dev/hda5
  vga    = 791
  append = "ide=nodma apm=off acpi=off"
  optional
```

save and exit - Note that the very carful setting of `ide=nodma` is not a requirement of RTLinux, where as `apm=off` and `acpi=off` is a requirement if you want to guarantee hard-realtime performance.
Next we need to install the new boot-loader configuration to the disk by running lilo.

```
linux:/etc # lilo
Added linux *
Added rtlinux
Added failsafe
Added memtest86
```

this should run without any errors and show you the rtlinux image.
Now we can tell the system to boot rtlinux on the next reboot - this will not permanently change the boot kernel - so by default non-rt Linux will be booted and only if we select rtlinux at the boot-prompt or by running `lilo -R` rtlinux will rtlinux boot.

```
linux:/etc # lilo -R rtlinux
linux:/etc # reboot
```

After the system comes up again - login as root. check what we have running

```
linux:~ # uname -a
Linux linux 2.4.18-rtl3.2-pre1 #5 Sun Dec 1 07:12:11 PST 2002 i686 unknown
```

5 RTLinux

RTLinux is installed from sources on the Proceedings CD, no rpm's for RTLinux around. The procedure here applies not only to the rtlinux-3.2-pre1 version but is more or less identical for other versions. If you ever run into a problem of a module or a system behaving very strange, then pleas verify the strange behavior on a clean instalation as described here, often strange behavior is due to accumulating changes and build procedures for custom modules not being clean... For question pertaining to the basic setup of RTLinux you can also contact the community via the rtlinux mailing list at `www2.fsmalbs.com/mailman/listinfo.cgi/rtl`.

5.1 configure/compile rtlinux

```
linux:/etc # cd /usr/src/rtlinux
linux:/usr/src/rtlinux # make menuconfig

  Support option --->

  [*] Posix standard I/O
  [ ] POSIX Priority protection
  [*] Dev mem support
  [*] Enable debugging
  [*] rtl_printf uses printk
  [ ] Nolinux support
  [ ] RTLinux tracer support (experimental)
  [ ] Userspace Real Time
  [*] Floating Point Support
  [*] RTLinux V1 API support
  [*] RTLinux Debugger
  [ ] Synchronized clock support
```

lets leave it all defaults (shown above) for now, also the driver section can be left as it is - Save and exit menuconfig.

```
linux:/usr/src/rtlinux # make dep
linux:/usr/src/rtlinux # make 2>&1 | tee build.log
linux:/usr/src/rtlinux # make devices
```

The command `make 2$>$\&1` tee build.log— records the entire compiler output into the file build.log, so if anything goes wrong this can help you, and also help when you report errors to the mailing list.
The make devices is only necessary for the first instalation - this creates the rtlinux specific device files in `/dev/`.

5.2 Check your instalation

The regression test performs a number of sanity checks - it will not tell you if the setup is suited for hard realtime applications, it will basically tell you that the instalation worked and that rtlinux will not crash your box ;)

```
linux:/usr/src/rtlinux # ./scrips/regression.sh
```

the regressions script should ONLY return [OK], if you get anything else pleas let the community know . After the script terminated all rtlinux modules are unloaded.
now launch rtlinux

```
linux:/usr/src/rtlinux # ./scripts/inrtl
linux:/usr/src/rtlinux # lsmod
```

Check if the modules are loaded - it should return something like:

```
linux:/usr/src/rtlinux # dmesg -c
```

Clear the kernel message ring buffer so that we can see what messages popped up after we launched rtlinux examples.
Lets start with a very simple example - "hello World" in hard-realtime.

5.3 hello.o

```
linux:/usr/src/rtlinux # cd examples/hello
linux:/usr/src/rtlinux/examples/hello # sync ; insmod hello.o
```

Why do we do `sync ; insmod hello.o`. If you load a module that you are playing with and you made a mistake your system can crash fairly easily, as the filesystem may be in a inconsistent state at this point it could loose data or even be damaged (depending on how wildly you crash your system) so it is a good habit to sync your disk before loading a kernel module as this reduces the probability of loosing data considerably.

```
linux:/usr/src/rtlinux/examples/hello # dmesg
```

check the messages that the `hello.o` module is generating with `dmesg`. To stop our realtime "hello World" we remove the `hello.o` module and clear the kernel message buffer.

```
linux:/usr/src/rtlinux/examples/hello # rmmod hello
linux:/usr/src/rtlinux/examples/hello # dmesg -c
```

5.4 rt_process.o

Now to a more usable example - `rt_process.o`. This kernel module measure the scheduling jitter of the hardware. It sets up a thread to run periodically for ntests times in a loop and report the minimum and maximum deviation of the time it actually ran to the time it should have run by writing the data to a realtime fifo (rtf). The data can be retrieved from the fifo with the monitor program. The monitor will, by default, retrieve 10000 data samples and the terminate, by passing it the -s#para meter you can tell it to grab exactly # samples.

```
linux:/usr/src/rtlinux/examples/hello  # cd ../measurements
linux:/usr/src/rtlinux/examples/measurements  #
linux:/usr/src/rtlinux/examples/measurements  # sync ; insmod rt_process.o bperiod=0
```

load the measurement module with a sync again, if you don't want to do it, then simply `insmod rt_process.o` and find out the hard way why to sync your disk before loading a module ;) The `bperiod=0` module parameter instructs `rt_process.o` to launch only one thread and not launch a background thread that would compete for the CPU.

```
linux:/usr/src/rtlinux/examples/measurements  # ./monitor -s 1000 | tee data
```

We only collect 1000 samples and put them in the file data, while at the same time displaying them on the screen.

```
linux:/usr/src/rtlinux/examples/measurements  # ./gist data | tee data.out
```

Next we make a "histogram" of this data - Note that this is not a real histogram as we don't have access to the samples but only the min/max values of every ntests-large sample (default 500) so you can't interprete this data statistically. It does give you a fairly good overview of the systems rt-performance though - especially if you produce a high-load and high interrupt situation while running this test. If you launch `rt_process` without the `bperiod=0` parameter then you run two rt-threads and you can see what influence two threads completing at the same priority will have on the worst case scheduling-jitter of this specific system-hardware. So now lets clean up - removing the module and clearing the kernel message buffer again.

```
linux:/usr/src/rtlinux/examples/measurements  # rmmod rt_process
linux:/usr/src/rtlinux/examples/measurements  # dmesg -c
```

5.5 shut down rtlinux

```
linux:/usr/src/rtlinux/examples/measurements  # cd /usr/src/rtlinux
linux:/usr/src/linux  # ./scripts/rmrtl
linux:/usr/src/linux  # dmesg -c
```

just to check if there were any problems unloading the rtlinux modules !
Thats it - you now are rtlinux experts.... almost.

6 Debugging

To use gdb for debugging we need to reconfigure rtlinux and recompile it. To do this we first clean up and then launch menuconfig again.

```
linux:~ # cd /usr/src/rtlinux
linux:~ # make distclean
linux:~ # make menuconfig

  Support option --->

  [*] Posix standard I/O
  [*] POSIX Priority protection
  [*] Dev mem support
  [*] Enable debugging
  [*] rtl_printf uses printk
  [ ] Nolinux support
  [*] RTLinux tracer support (experimental)
  [*] Userspace Real Time
  [*] Floating Point Support
  [ ] RTLinux V1 API support
  [ ] RTLinux Debugger
  [*] Synchronized clock support
  [ ] RTLinux V1 API
  [*] RTLinux Debugger
```

save and exit.

```
linux:/usr/src/rtlinux # make dep
linux:/usr/src/rtlinux # make
```

RTLinux is now rebuilt with debugging flags, and with the RTLinux debugger module in the debugger subdirectory (`rtl_debug.o`)

6.1 RTLinux Debugger

Before we can launch the debugger we need to reload the modified rtlinux modules, we can use the scripts in the top-level rtlinux directory again.

```
linux:/usr/src/rtlinux # ./scripts/insrtl
linux:/usr/src/rtlinux #  cd debugger
linux:/usr/src/rtlinux/debugger # insmod rtl_debugger.o
```

RTLinux is not ready for debugging, before we insert the actual module to be debugged lets give it a look. If you look `hello.c` and compare it with `examples/hello/hello.c` you will find some debugging specific differences.

```
#include <rtl_debug.h>
...
void * start_routine(void *arg)
{
   ...
   if (((int) arg) == 1) {
       breakpoint();
   }
   ...
```

This `breakpoint();` instruction is the point where gdb will halt and you can continue from there on, if your module has not breakpoint and no bug that causes an exception then gdb can't connect, so either code a segfault or use the `breakpoint(); function ;)`

```
linux:/usr/src/rtlinux/debugger # gdb hello.o
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-suse-linux"...
(gdb) target remote /dev/rtf10
Remote debugging using /dev/rtf10
[New Thread -1012596736]
[Switching to Thread -1012596736]
start_routine (arg=0x1) at hello.c:37
37 for (i = 0; i < 20; i ++) {
warning: shared library handler failed to enable breakpoint
(gdb) l
32
33 if (((int) arg) == 1) {
34 breakpoint();
35 }
36
37 for (i = 0; i < 20; i ++) {
38 pthread_wait_np ();
39 rtl_printf("I'm here; my arg is %x\n", (unsigned) arg);
40 }
41 return 0;
(gdb) break rtl_printf
Function "rtl_printf" not defined.
(gdb) modaddsym ../modules/rtl.o
add symbol table from file "../modules/rtl.o" at
.text_addr = 0xc88da060
(gdb) modaddsym ../modules/rtl_time.o
add symbol table from file "../modules/rtl_time.o" at
.text_addr = 0xc88e0060
(gdb) modaddsym ../modules/rtl_sched.o
add symbol table from file "../modules/rtl_sched.o" at
.text_addr = 0xc88ea060
(gdb) break rtl_printf
Breakpoint 1 at 0xc88db189: file rtl_printf.c, line 39.
(gdb) c
Continuing.
```

```
Breakpoint 1, rtl_printf (fmt=0xc8905140 "I'm here; my arg is %x\n")
    at rtl_printf.c:39
39 {
(gdb) l
34 static char initial_printkbuf [MAX_PRINTKBUF];
/* need to protect in_printkbuf from overflowing */

35 static char in_printkbuf[MAX_PRINTKBUF]; /* please don't put this on my stack*/
36 static char *printkptr = &in_printkbuf[0];
37 static spinlock_t rtl_cprintf_lock = SPIN_LOCK_UNLOCKED;
38 int rtl_printf(const char * fmt, ...)
39 {
40 rtl_irqstate_t flags;
41 int i;
42 va_list args;
43
(gdb) quit
The program is running.  Exit anyway? (y or n)
```

What we did was connect to gdb via `/dev/rtf10` (that the '`target remote /dev/rtf10`' line), then gdb stopped at the breakpoint instruction, and we tried to set a breakpoint at `rtl_printf`, but that was not know because the symbols were not loaded, so next we use a convenient modaddsym macro found in the `.gdbini` file of the debugger directory, to load the symbol information form the rtlinux core modules. Now setting of the breakpoint works fine, and we can continue (command 'c' in gdb), stopping at the first `rtl_printf`. Next we list (command 'l' in gdb) the `rtl_printf` function. There really is not much to do in this example so we quite (command 'q' in gdb).
As we were debugging the module, it was off course not running in realtime, and if you now type dmesg, you can see that the execution of the individual threads (`hello.o` spawns two threads!) gets confused.

```
RTLinux Extensions Loaded (http://www.fsmlabs.com/)
RTLinux Debugger Loaded (http://www.fsmlabs.com/)
rtl_debug: exception 0x3 in hello (EIP=0xc89050a8),
    thread id 0xc3a50000; (re)start GDB to debug
I'm here; my arg is 1
I'm here; my arg is 1
I'm here; my arg is 1
I'm here; my arg is 2
I'm here; my arg is 1
I'm here; my arg is 2
I'm here; my arg is 1
I'm here; my arg is 2
```

So gdb is good to find a segfault or some other coding error, but it will not allow to debug race-conditions or to locate temporal misbehavior of your realtime module, and in fact the behavior of multi-threaded apps can be quite strange in gdb, even for those that work fine when run free !
We are done with our debugging intro so lets clean up.

```
linux:/usr/src/rtlinux/debugger # dmesg -c
linux:/usr/src/rtlinux/debugger # rmmod hello
linux:/usr/src/rtlinux/debugger # rmmod rtl_debug
```

6.2 RTLinux Tracer

When we reconfigured RTLinux, with menuconfig, above, we enabled the RTLinux Tracer. The problem with GDB was that it did not allow temporal debugging, and this is because gdb is taking control of the rt-thread, so this is a rt-thread under control of a non-rt executable. To allow temporal debugging the RTLinux tracer was designed that we will briefly introduce here.

```
linux:/usr/src/rtlinux/debugger # cd ../
linux:/usr/src/rtlinux # insmod modules/mbuff.o
```

The mbuff module, contributed by thomas motylewsky, is a shared memory module, allowing to share memory between rt-threads and user-space applications. The RTLinux Tracer records important events with timestamps (system events are hard-coded, like entering and exiting the scheduler routine) and user-defined events can be included in your application that trigger writes of the buffered data into shared memory when ever a user-specified condition is met. This way you can back-trace what events lead to the event that you are watching.

```
linux:/usr/src/rtlinux # cd tracer
```

If we look into `rt_process.c` in the tracer directory and compare it with `rt_process` in examples / measurements then we can find the following difference. The recording of the maximum in the inner loop changes from:

```
if(diff > max_diff){
   max_diff = diff;
}
```

to include the `RTL_TRACE_USER` event recording the new absolute jitter maximum, and the trace buffer is flushed to shared memory where the user-space application can read it. (there are a few other minor changes like `#include <rtl_trace.h>` and the variable `abs_max_diff` not shown here - but those should be quite self explaining).

```
if(diff > max_diff){
   max_diff=diff;
   if(max_diff > abs_max_diff){
      abs_max_diff=max_diff;
      rtl_trace2(RTL_TRACE_USER,(long) abs_max_diff);
      rtl_trace2(RTL_TRACE_FINALIZE,0);
   }
}
```

Now every time a new maximum jitter value is encountered the buffer will be flushed, this way we can trace the path of events that leeds to a jitter maximum and thus (hopefully) locate the hot-spot in the code.
Now lets load the tracer module and the modified `rt_process.o` to watch it.

```
linux:/usr/src/rtlinux/tracer # insmod rtl_tracer.o
linux:/usr/src/rtlinux/tracer # insmod rt_process.o ; ./tracer | tee trace.log
P0 131828416 rtl_restore_interrupts        0x46 <c88e09cd>
P0    576 scheduler out           0xc88ecd64 <c88ea95d>
P0    480 rtl_restore_interrupts       0x96 <c88ea96b>
P0    512 hard sti                        0 <c88e0865>
P0    576 rtl_no_interrupts            0x286 <c88e0707>
P0    480 rtl_spin_unlock         0xc88e15f8 <c88e0212>
P0    416 rtl_restore_interrupts      0x203 <c88e021d>
P0  41184 rtl_no_interrupts            0x203 <c88e01c7>
P0    448 rtl_spin_lock           0xc88e15f8 <c88e01d6>
...
---snip---
...
P0    448 rtl_restore_interrupts       0x46 <c88e09cd>
P0    640 rtl_switch_to           0xc3ab8000 <c88ea7a0>
P0   1952 scheduler out           0xc3ab8000 <c88ea95d>
P0    640 rtl_restore_interrupts       0x92 <c88ea96b>
P0    992 rtl_restore_interrupts      0x297 <c88eaea4>
```

```
P0   3488 user                        0x1cc0 <c894f325>
That was trace # 1
```

The commands for insmod and starting the tracer are concatenated to a command sequence to make sure we launch the tracer fast enough as it happen quite frequently that the maximum is reached right at the beginning and then we don't see anything. Furthermore we redirect the tracer output to a log file (`trace.log`), to terminate the tracer type `<CNTRL>-<C>` . The tracer output will scroll by on the screen as the new absolute maximum is frequently encountered at the beginning and then output will stall, if you want to produce a new worst-case maximum, then witch to a different console (`<CNTRL>-<ALT>-<F2>`) login as root again and start something like.

```
linux:~ # ls -lR /
```

Note that the RTLinux Tracer does introduce a slight distortion on the systems realtime behavior so you will not be able to find everything this way, but its about as close to temporal debugging that you can get. After we terminated the tracer we do the usual cleanup.

```
linux:/usr/src/rtlinux/tracer #  rmmod rt_process
linux:/usr/src/rtlinux/tracer #  rmmod rtl_tracer
linux:/usr/src/rtlinux/tracer # rmmod mbuff
```

Thats it - have fun with real time Linux !