Networked and Mobile Systems
A blog on software-defined networking, mobile technologies, cloud computing … and more.

Raspberry Pi Going Realtime with RT Preempt

Posted on **October 25, 2015**

[UPDATE 2016-05-13: Added pre-compiled kernel version 4.4.9-rt17 for all Raspberry Pi models (Raspberry Pi Model A(+), Model B(+), Zero, Raspberry Pi 2, Raspberry Pi 3). Added build instructions for Raspberry Pi 2/3.

A real-time operating system gives you deterministic bounds on delay and delay variation (jitter). Such a real-time operating system is an essential prerequisite for implementing so-called Cyber Physical Systems, where a computer controls a physical process. Prominent examples are the control of machines and robots in production environments (Industry 4.0), drones, etc.

RT Preempt is a popular patch for the Linux kernel to transform Linux into such a realtime operating system. Moreover, the Raspberry Pi has many nice features to interface with sensors and actuators like SPI, I2C, and GPIO so it seems to be a good platform for hosting a controller in a cyber-physical system. Consequently, it is very attractive to install Linux with the RT Preempt patch on the Raspberry Pi.

Exactly this is what I do here: I provide detailed instructions on how to install a Linux kernel with RT Preempt patch on a Raspberry Pi. Basically, I wrote this document to document the process for myself, and it is more or less a collection of information you will find on the web. But anyway, I hope I can save some people some time.

And to save you even more time, here is the pre-compiled kernel (including kernel modules, firmware, and device tree) for the Raspberry Pi Model A(+),B(+), Raspberry Pi Zero, Raspberry Pi 2 Model B, Raspberry Pi 3 Model B:

- kernel-4.4.9-rt17.tgz

To install this pre-compiled kernel, login to your Raspberry Pi running Raspbian (if you have not installed Raspbian already, you can find an image here: https://www.raspberrypi.org/downloads/raspbian/), and execute the following commands (I recommend to do a backup of your old image since this procedure will overwrite the old kernel):

```
pi@raspberry ~$ sudo rm -r /boot/overlays/
pi@raspberry ~$ sudo rm -r /lib/firmware/
pi@raspberry ~$ cd /tmp
pi@raspberry ~$ wget http://download.frank-durr.de/kernel-4.4.9-rt
pi@raspberry ~$ tar xzf kernel-4.4.9-rt17.tgz
pi@raspberry ~$ cd boot
pi@raspberry ~$ sudo cp -rd * /boot/
pi@raspberry ~$ cd ../lib
pi@raspberry ~$ sudo cp -dr * /lib/
pi@raspberry ~$ sudo /sbin/reboot
```

With this patched kernel, I could achieve bounded latency well below 200 microseconds on a fully loaded 700 MHz Raspberry Pi Model B (see results below). This should be safe for tasks with a cycle time of 1 ms.

Since compiling the kernel on the Pi is very slow, I will cross compile the kernel on a more powerful host. You can distinguish the commands executed on the host and the Pi by looking at the prompt of the shell in the following commands.

Install Vanilla Raspbian on your Raspberry Pi

Download Raspbian from https://www.raspberrypi.org/downloads/raspbian/ and install it on your SD card.

Download Raspberry Pi Kernel Sources

On your host (where you want to cross-compile the kernel), download the latest kernel sources from Github:

```
user@host ~$ git clone https://github.com/raspberrypi/linux.git
user@host ~$ cd linux
```

If you like, you can switch to an older kernel version like 4.1:

```
user@host ~/linux$ git checkout rpi-4.1.y
```

Patch Kernel with RT Preempt Patch

Next, patch the kernel with the RT Preempt patch. Choose the patch matching your kernel version. To this end, have a look at the Makefile. VERSION, PATCHLEVEL, and SUBLEVEL define the kernel version. At the time of writing this tutorial, the latest kernel was version 4.4.9. Patches for older kernels can be found in folder “older”.

```
user@host ~/linux$ wget https://www.kernel.org/pub/linux/kernel/pr
user@host ~/linux$ zcat patch-4.4.9-rt17.patch.gz | patch -p1
```

Install and Configure Tool Chain

For cross-compiling the kernel, you need the tool chain for ARM on your machine:

```
1. user@host ~$ git clone https://github.com/raspberrypi/tools.git
2. user@host ~$ export ARCH=arm
3. user@host ~$ export CROSS_COMPILE=/home/user/tools/arm-bcm2708/gcc
4. user@host ~$ export INSTALL_MOD_PATH=/home/user/rtkernel
```

Later, when you install the modules, they will go into the directory specified by INSTALL_MOD_PATH.

Configure the kernel

Next, we need to configure the kernel for using RT Preempt.

For Raspberry Pi Model A(+), B(+), Zero, execute the following commands:

```
1. user@host ~$ export KERNEL=kernel
2. user@host ~$ make bcmrpi_defconfig
```

For Raspberry Pi 2/3 Model B, execute these commands:

```
1. user@host ~$ export KERNEL=kernel7
2. user@host ~$ make bcm2709_defconfig
```

An alternative way is to export the configuration from a running Raspberry Pi:

```
1. pi@raspberry$ sudo modprobe configs
```

```
1. user@host ~/linux$ scp pi@raspberry:/proc/config.gz ./
2. user@host ~/linux$ zcat config.gz > .config
```

Then, you can start to configure the kernel:

```
1. user@host ~/linux$ make menuconfig
```

In the kernel configuration, enable the following settings:

- CONFIG_PREEMPT_RT_FULL: Kernel Features → Preemption Model (Fully Preemptible Kernel (RT)) → Fully Preemptible Kernel (RT)
- Enable HIGH_RES_TIMERS: General setup → Timers subsystem → High Resolution Timer Support (Actually, this should already be enabled in the standard configuration.)

Build the Kernel

Now, it's time to cross-compile and build the kernel and its modules:

```
1. user@host ~/linux$ make zImage
2. user@host ~/linux$ make modules
3. user@host ~/linux$ make dtbs
4. user@host ~/linux$ make modules_install
```

The last command installs the kernel modules in the directory specified by INSTALL_MOD_PATH above.

Transfer Kernel Image, Modules, and Device Tree Overlay to their Places on Raspberry Pi

We are now ready to transfer everything to the Pi. To this end, you could mount the SD card on your PC. I prefer to transfer everything over the network using a tar archive:

```
1. user@host ~/linux$ mkdir $INSTALL_MOD_PATH/boot
2. user@host ~/linux$ ./scripts/mkknlimg ./arch/arm/boot/zImage $INST
3. user@host ~/linux$ cp ./arch/arm/boot/dts/*.dtb $INSTALL_MOD_PATH/
4. user@host ~/linux$ cp -r ./arch/arm/boot/dts/overlays $INSTALL_MOD
5. user@host ~/linux$ cd $INSTALL_MOD_PATH
6. user@host ~/linux$ tar czf /tmp/kernel.tgz *
7. user@host ~/linux$ scp /tmp/kernel.tgz pi@raspberry:/tmp
```

Then on the Pi, install the real-time kernel (this will overwrite the old kernel image!):

```
1. pi@raspberry ~$ cd /tmp
2. pi@raspberry ~$ tar xzf kernel.tgz
3. pi@raspberry ~$ sudo rm -r /lib/firmware/
4. pi@raspberry ~$ sudo rm -r /boot/overlays/
5. pi@raspberry ~$ cd boot
6. pi@raspberry ~$ sudo cp -rd * /boot/
7. pi@raspberry ~$ cd ../lib
8. pi@raspberry ~$ sudo cp -dr * /lib/
```

Most people also disable the Low Latency Mode (llm) for the SD card:

```
1. pi@raspberry /boot$ sudo nano cmdline.txt
```

Add the following option:

```
1. sdhci_bcm2708.enable_llm=0
```

Reboot

```
1. pi@raspberry ~$ sudo /sbin/reboot
```

Latency Evaluation

For sure, you want to know the latency bounds achieved with the RT Preempt patch. To this end, you can use the tool cyclictest with the following test case:

- clock_nanosleep(TIMER_ABSTIME)
- Cycle interval 500 micro-seconds
- 100,000 loops
- 100 % load generated by running the following commands in parallel:
    - On the Pi:

```
pi@raspberry ~$ cat /dev/zero > /dev/null
```

- From another host:

```
user@host ~$ sudo ping -i 0.01 raspberrypi
```

- 1 thread (I used a Raspberry Pi model B with only one core)
- Locked memory
- Process priority 80

```
1. pi@raspberry ~$ git clone git://git.kernel.org/pub/scm/linux/kerne
2. pi@raspberry ~$ cd rt-tests/
3. pi@raspberry ~/rt-test$ make all
4. pi@raspberry ~/rt-test$ sudo ./cyclictest -m -t1 -p 80 -n -i 500 -
```

On a Raspberry Pi model B at 700 MHz, I got the following results:

```
1. T: 0 ( 976) P:80 I:500 C: 100000 Min: 23 Act: 40 Avg: 37 Max: 95
```

With some more tests, the worst case latency sometimes reached about 166 microseconds. Adding a safety margin, this should be safe for cycletimes of 1 ms.

I also observed that using other timers than clock_nanosleep(TIMER_ABSTIME)—e.g., system timers (sys_nanosleep and sys_setitimer)—, the latency was much higher with maximum values above 1 ms. Thus, for low latencies, I would only rely on clock_nanosleep(TIMER_ABSTIME).

This entry was posted in **Linux**, **Raspberry Pi**, **Realtime**, **Uncategorized** and tagged **linux**, **raspberry pi**, **realtime**, **rt preempt** by **Frank Dürr**. Bookmark the **permalink [http://www.frank-durr.de/?p=203]** .