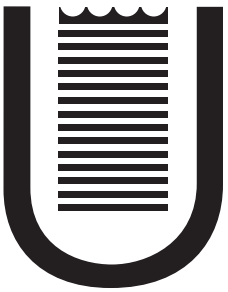

Sistemi Embedded e Real Time

Project Overview

4 novembre 2014

Università di Roma Tor Vergata

Contents

Chapter 1

Introduction

Real-time systems are computing systems that must react within precise time constraints to events in the environment. Embedded and Real Time Operative Systems (RTOS) play a crucial role in our society since an increasing number of applications rely in strict timing constraints. The main characteristic that distinguishes a real-time task from other types of computation is time. The correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced. In a critical application, a result produced after the deadline is considered to be a fatal fault, a late result produced by the job after the deadline may have disastrous consequences.

Real-time tasks are usually distinguished in two classes, hard and soft. The difference between them is based on the functional criticality of jobs and the usefulness of late results

- A real-time task is said to be soft if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment.
- A real-time task is said to be hard if missing its deadline may cause catastrophic consequences on the environment under control.

In a non simulated environment, a real time application should be designed to handle periodic, soft and hard tasks using different strategies. The operating system should guarantee to respect the individual timing constraints of both the hard real time and periodic task while minimizing the average response time of the soft activities.

- In the first section we will describe the basic steps required for the implementation of an Embedded System, based on kernel 3.8.13, using the commercial Raspberry Pi model B+ board. The development process will be handled by *Buildroot* which simplifies and automates the process of building a complete and bootable Linux environment for embedded systems.

- In order to pursue predictability, our work will be based on the dual kernel approach using *Xenomai* which is a real-time subsystem that can be tightly integrated with the Linux kernel. We will prepare, in the second section, the linux tree in order to include the Xenomai support provided by the interrupt pipeline.

- In the last section, we will measure latencies with a benchmark tool provided with Xenomai test suite, create a module to handle interrupts on a GPIO port and create a pwm signal with variable period.

All the above parts will be available online and updated constantly at `http://www.userk.co.uk/tutorials`

Chapter 2

Building a minimal Linux System

The widespread interest and enthusiasm generated by Linu's successful use in a number of embedded applications has led to the creation of a several articles, tools, companies, and documents all pertaining to embedded Linux. Usually building a minimal distribution from scratch involves different steps starting from gathering sources, configuration, compilation and finally installation. In this section we will prepare the micro sd card and build the system using the well known Buildroot.

2.1 Prepare the sd card

In order to use our operative system we need to format a micro sd card and create two partitions.

- The **Boot** Partition is a primary partition that contains all the files responsible for booting the operative system (OS). The bootloader initiates the bootstrap process by loading the OS into memory.
- The **Root** or System partition is a disk partition containing the operative system folder mounted at the root directory in Linux called with the slash symbol /.

Insert the SD in your card reader and run the fdisk command with root privileges to identify the name of SD card, the device name is preceded by the /dev folder and its name looks like "/dev/DeviceName".

```
$ sudo fdisk -l

Disk /dev/sda: 500.1 GB, 500107862016 bytes  <-- My Hard Disk
255 heads, 63 sectors/track, 60801 cylinders, total 976773168 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1               1   976773167   488386583+  ee  GPT

Disk /dev/sdb: 3941 MB, 3941597184 bytes    <-- The Sd Card
53 heads, 21 sectors/track, 6916 cylinders, total 7698432 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0002c262

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1            2048     7698431     3848192    b  W95 FAT32
```

It's worth noting that the hard disk appears in the first block and is denoted by sda. Note that all future operations will affect the other device, sdb. As we can see from the output, the sdb device has one partition in /dev/sdb1 formatted in FAT32. We need to unmount the partition, delete it and prepare the memory card to host the Operative System by creating the Boot and the Root partitions. We can create and modify partitions on Sd card using the fdisk tool which is a command-line utility that provides disk partitioning functions.

Let's run the utility specifying the disk to work on.

```
$ sudo fdisk /dev/sdb
```

Delete the partition

Delete the partition *sdb1* pressing **d**. The partition 1 will be automatically selected.

```
Command (m for help): d
Partition number (1-4): 1
```

The **p** command will show an empty partition table. Or if your card contained more than one partition, the command will show the others.

```
Command (m for help): p

Disk /dev/sdb: 7969 MB, 7969177600 bytes
246 heads, 62 sectors/track, 1020 cylinders, total 15564800 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00007519

   Device Boot      Start         End      Blocks   Id  Syste
```

If you have another partition, delete it with the same command.

2.1.1 Create the Boot partition

Create a new primary partition using the n command and setting the partition type as Primary (p). The utility will ask you the number of the partition you want to modify, which is in our case the default value (1). Then you will be asked for the sector, press Enter. Now you need to set the size, we will specify to write 10 MiB. Press +10M.

```
Command (m for help): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-7698431, default 2048): [Enter]
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-7698431, default 7698431): +10M
```

Toggle Bootable flag and specify partition type

Specify the partition's system id by pressing **t**, set the system type to FAT32 (c) and toggle the bootable flag by pressing **a**.

```
1  Command (m for help): t
2  Selected partition 1
3  Hex code (type L to list codes): c
4  Changed system type of partition 1 to e (W95 FAT32 (LBA))
5
6  Command (m for help): a  <-- Toggles the bootable flag
7  Partition number (1-4): 1
```

2.1.2 Create the Root partition

Create a second primary partition for the file system with the entire remaining space as dimension.

```
1  Command (m for help): n
2  Partition type:
3     p   primary (1 primary, 0 extended, 3 free)
4     e   extended
5  Select (default p): p
6  Partition number (1-4, default 2): 2
7  First sector (264192-7698431, default 264192): [Enter]
8  Using default value 264192
9  Last sector, +sectors or +size{K,M,G} (264192-7698431, default 7698431):[Enter]
10 Using default value 7698431
```

Display the partition table with command **p** and check if you have a 10MiB partition, formatted in Fat32 with the bootable flag and another one primary.

```
1  Command (m for help): p
2
3  Disk /dev/sdb: 7969 MB, 7969177600 bytes
4  246 heads, 62 sectors/track, 1020 cylinders, total 15564800 sectors
5  Units = sectors of 1 * 512 = 512 bytes
6  Sector size (logical/physical): 512 bytes / 512 bytes
7  I/O size (minimum/optimal): 512 bytes / 512 bytes
8  Disk identifier: 0x00007519
```

```
9
10     Device Boot      Start         End      Blocks   Id  System
11 /dev/sdb1   *        2048       22527       10240    c  W95 FAT32 (LBA)
12 /dev/sdb2           22528    15564799     7771136   83  Linux
```

Write the table to disk and exit by pressing **w**. If you get the above warning, umount the device from nautilus or with the following command and try again.

```
1 $ sudo umount /dev/sdb1 /dev/sdb2
2 $ sudo fdisk /dev/sdb
3 Command (m for help): w
4 The partition table has been altered!
5
6 Calling ioctl() to re-read partition table.
7 Syncing disks.
8 userk@dopamine:¬$
```

2.1.3 Add Label to partitions

The last step is to associate the names Boot and Root to respectively **/dev/sb1** and **/dev/sdb2**.

```
1 userk@dopamine:¬$ sudo /sbin/mkfs.vfat -n Boot /dev/sdb1
2 userk@dopamine:¬$ sudo /sbin/mkfs.ext4 -L Root /dev/sdb2
```

Remove and insert the Sd card and check if nautilus sees the partitions. In the next subsection we will set up the Cross-Compiler and build the minimal linux distribution.

2.2 Buildroot configuration

Buildroot simplifies the process of building a complete Linux system for an embedded system, using cross-compilation. It is able to generate a cross-compilation toolchain, a root filesystem, a Linux kernel image and a bootloader for the target, the Raspberry Pi.

We will first create our workspace for the project, in the home folder, then down-

load the latest version of Buildroot from the official website and untar the compressed file.

```
¬$ cd && mkdir RaspberryPi && cd RaspberryPi
¬/RaspberryPi$ wget http://buildroot.uclibc.org/downloads/buildroot-2014.11.tar.bz2
¬/RaspberryPi$ tar xjf buildroot-2014.11.tar.bz2 && cd buildroot-2014.11
¬/RaspberryPi/buildroot-2014.11$
```

Once downloaded we can configure the tool and compile it.

Buildroot has a graphical configuration tool. Please note that there is no need to run the command as root to configure and use Buildroot. Since the Buildroot team has prepared a configuration file for the Raspberry Pi, we will use it as a starting point. Please check the steps written in "buildroot-2014.11/board/raspberrypi/readme.txt".

```
¬/RaspberryPi/buildroot-2014.11$ sudo make ARCH=arm help | grep raspberrypi
  raspberrypi_defconfig                - Build for raspberrypi
¬/RaspberryPi/buildroot-2014.11$ sudo make raspberrypi_defconfig
[...]
#
# configuration written to .config
#
```

The next step is to run the configuration assistant.

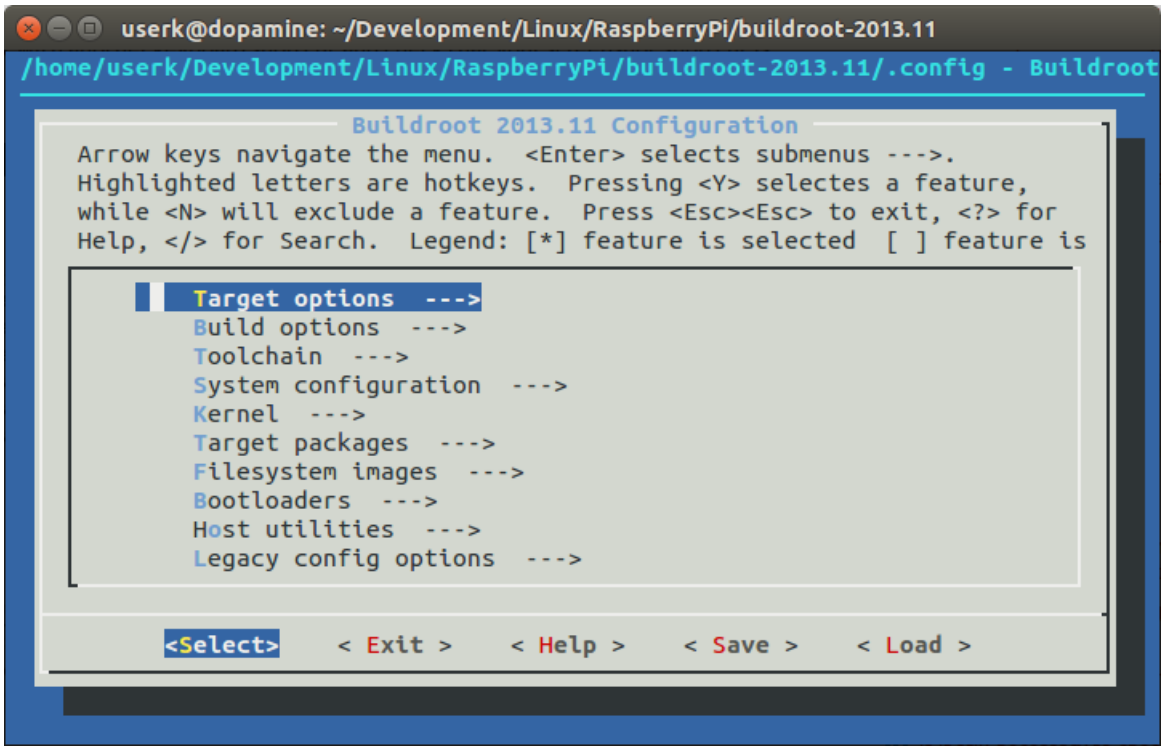

Figure 2.1: The configuration assistant

```
1 ¬/RaspberryPi/buildroot-2014.11$ sudo make menuconfig
```

We need to modify a few fields in this configuration tool. Starting from the header files we wish to use in order to patch the kernel with the Adeos and Xenomai patch and few utilities. The main settings are reported below.

- Build Options
  - Host dir: /usr/local/cross-rpi/
- Toolchain
  - Toolchain type: Buildroot toolchain

  - Kernel Headers: (Manually specified Linux version) (3.8.13) linux version
  - Custom kernel headers series (3.8.x)

- System Configuration
  - (x3n0B0mb) System hostname
  - (Welcome to x3n0B0mb) System banner
  - Root password: pitos

- Kernel
  - Kernel version: (Custom Git repository) (git://github.com/raspberrypi/linux.git) URL of custom repository
  - (d996a1b) Custom repository version
  - Kernel configuration (Using a defconfig) (bcmrpi_quick) Defconfig name
  - Kernel binary format: zImage

- Target Packages
  - (y) Show packages that are also provided by busybox
  - Debugging, profiling and benchmark
  - Development tools
  - Hardware handling
  - Interpreter languages and scripting: python3
  - Miscellaneous: mcrypt
  - Package managers: opkg

  - Networking applications
  - Real-Time
  - Shell and utilities
  - Text editors and viewers

The compilation requires an internet connection to download different packages specified in the configuration step. The make command will generally perform the following actions:

- download source files (as required);
- configure, build and install the cross-compilation toolchain, or simply import an external toolchain;
- configure, build and install selected target packages;
- build a kernel image, if selected;
- build a bootloader image, if selected;
- create a root filesystem in selected formats.

2.2.1 Prepare the Root partition

Extract the filesystem image just created in the Root partition located at /media/$USER/Root

```
1 ¬/RaspberryPi/buildroot-2014.11$ ls output/images/
2 rootfs.ext2  rootfs.ext4  rootfs.tar
3 ¬/RaspberryPi/buildroot-2014.11$ cd /media/$USER/Root/ && ls
4 lost+found
5 /media/userk/Root$ sudo tar -xvf /home/$USER/
6 RaspberryPi/buildroot-2014.11/output/images/rootfs.tar
7 userk@dopamine:/media/userk/Root$ ls
```

```
8 bin   etc   lib    linuxrc     media   opt   root   sbin   tmp   var
9 dev   home  lib32  lost+found  mnt     proc  run    sys    usr
```

2.2.2 Prepare the Boot partition

The RaspberryPi must find the following files in the Boot partition

- bootcode.bin
- config.txt
- fixup.dat
- start.elf
- zImage

Usually the kernel image name is kernel.img. It has been defined as zImage in the config.txt file.

```
1  /media/userk/Root$ cd ¬/RaspberryPi/buildroot-2014.11/output/images
2 ¬/buildroot-2014.11/output/images$ sudo cp rpi-firmware/* /media/$USER/Boot/
3 ¬/buildroot-2014.11/output/images$ sudo cp zImage /media/$USER/Boot/zImage
```

Since the filesystem, the bootloader and the kernel image have been created, we can run our minimal on the Raspberry Pi. Insert the micro SD in the model B+ board and login as **root** and insert **pitos** as password.

Chapter 3

Linux Kernel and Xenomai patch

3.1 Xenomai overview

Xenomai supports the running of real-time programs in user space. These tasks are exclusively controlled by the co-kernel during the course of their time-critical operations so that very low latencies are achieved for their code running inside a standard Linux kernel. To allow porting traditional RTOS APIs to Linux based real time frameworks, the Xenomai core provides generic building blocks for implementing real time APIs, also known as skins. The official Xenomai project website, offering source code, documentation, technical articles, and other resources, is `http://www.xenomai.org`.

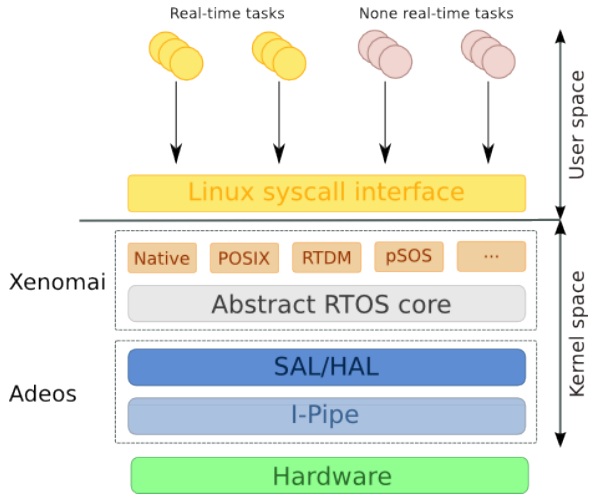

Figure 3.1: The Xenomai architecture

3.1.1 The interrupt pipeline

In order to keep latency predictable for real-time tasks, the system must ensure that the regular Linux kernel never defers external interrupts. The interrupts should be delivered as quickly as possible to Xenomai. Therefore, the interrupt pipeline or I-pipe acts as an additional software between the hardware, Linux, and Xenomai. The I-pipe organizes the system as a set of domains connected through a software pipeline. Within an I-pipe-featured kernel, Xenomai is the highest priority domain, ahead of the Linux kernel. The I-pipe dispatches events such as interrupts, system calls, processor faults, and exceptions to domains according to each domain's static priority.

The I-pipe implementation is available as patches against a number of Linux versions. In this section we will use Linux kernel 3.8.13 since it is compatible with the i-pipe arm Adeos patch and others required to fulfill the objective. We will then compile the kernel and substitute the image we previously created.

3.2 Download the cross compiler and Xenomai

In order to cross compile the kernel we could use the one provided by Buildroot but we will use the one included in the Raspberry Pi tools archive from github. Both cross compilers work.

```
$ cd && cd RaspberryPi
RaspberryPi$ mkdir Xenomai-RPI && cd Xenomai-RPI
RaspberryPi/Xenomai-RPI$ wget https://github.com/raspberrypi/tools/archive/
master.tar.gz
RaspberryPi/Xenomai-RPI$ tar xzf master.tar.gz
RaspberryPi/Xenomai-RPI$ wget -q -O - http://download.gna.org/xenomai/stab
le/xenomai-2.6.3.tar.bz2 | tar -xjf -
```

3.3 The Linux Kernel

Let's create a new directory and download the linux kernel version 3.8.13 from the Linux Kernel Archives.

```
RaspberryPi/Xenomai-RPI$ git clone -b rpi-3.8.y git://github.com/raspberrypi/
linux.git linux-3.8.13
RaspberryPi/Xenomai-RPI$ cd linux-3.8.13
RaspberryPi/Xenomai-RPI/linux-3.8.13$
```

The latest patches in the xenomai-2.6.3/ksrc/arch/arm/patches/raspberry folder.

```
RaspberryPi/Xenomai-RPI/linux-3.8.13$ patch -Np1 < ../xenomai-2.6.3/ksrc/arch/
arm/patches/raspberry/ipipe-core-3.8.13-raspberry-pre-2.patch
```

We can now apply the ipipe patch by running the prepare-kernel.sh script located in the scripts folder. We just need to specify the path of the target kernel source tree, the Adeos patch to apply against the tree and the target architecture.

```
RaspberryPi/Xenomai-RPI/linux-3.8.13$ ../xenomai-2.6.3/scripts/./prepare-kernel.sh
```

```
2   --arch=arm --linux=./ --adeos=../xenomai-2.6.3/ksrc/arch/arm/patches/
3   ipipe-core-3.8.13-arm-3.patch
```

The above command prepares the Linux tree located at ./linux-3.8.13 in order to include the Xenomai support. Once the target kernel has been prepared, the kernel should be configured following its usual configuration procedure. All Xenomai configuration options are available from the "Real-time subsystem" toplevel menu.

We can finally apply the last patch

```
1  RaspberryPi/Xenomai-RPI/linux-3.8.13$ patch -Np1 < ../xenomai-2.6.3/ksrc/arch/
2  arm/patches/raspberry/ipipe-core-3.8.13-raspberry-post-2.patch
```

3.3.1 Kernel compilation

Before running the cross compilation, we need to load a basic configuration optimized for the board's hardware and add a few components to include I2C, SPI support and remove frequency scaling and CPU Idle power management support.

I2C and SPI support

Since several sensors provide SPI and I2C transmission protocol, we need to enable the kernel support from the configuration assistant.

```
1  RaspberryPi/Xenomai-RPI/linux-3.8.13$ make menuconfig
```

Go to the Device Driver menu and check the I2C support pressing Y.

- Enter the I2C menu, go to I2C device interface and Press M for module support.
- Enter the I2C Hardware Bus support menu and press M to enable the module support for the BCM2708 BSC

Check the SPI support pressing Y and enter the sub menu. Press M near the BCM2708 SPI controller driver.

3.3.2 Optimization for Real Time systems

The Xenomai Learning system provides a few guidelines in order to set up a kernel for Xenomai in a dual kernel configuration. The CPU frequency scaling feature, option CONFIG_CPU_FREQ, allows you to change the clock speed of the CPU on the fly. This is a nice method to save power but in real time applications this could lead to unpredictable behavior and high latencies.

Disable the CPU Frequency scaling option in CPU Power Management. Press 'n' to disable.

Xenomai developers suggest to disable CONFIG_CPU_IDLE because it allows the CPU to enter deep sleep states, increasing the time it takes to get out of these sleep states, hence the latency of an idle system. Also, on some CPU, entering these deep sleep states causes the timers used by Xenomai to stop functioning.

Disable CPU idle PM support in the CPU Power Management menu. Save the configuration and cross-compile.

```
1 RaspberryPi/Xenomai-RPI/linux-3.8.13$ make ARCH=arm
2 CROSS_COMPILE=../tools-master/arm-bcm2708/
3 gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-
```

The kernel image is now available to be transferred in the Raspberry Pi Boot partition. The image called 'zImage' is located in the arch/arm/boot/ folder. You can copy the old kernel image in a secure place before replacing it.

```
1 RaspberryPi/Xenomai-RPI/linux-3.8.13$ sudo cp /media/$USER/Boot/zImage
2 RaspberryPi/Xenomai-RPI/linux-3.8.13$ sudo cp arch/arm/boot/zImage
```

```
3   /media/$USER/Boot/zImage
```

Kernel modules are installed into the /lib/modules/x.y.z directory on the target system. Copy them into the SD card in /media/$USER/Root.

```
1  RaspberryPi/Xenomai-RPI/linux-3.8.13$ sudo make ARCH=arm INSTALL_MOD_PATH=
2  /media/$USER/Root modules_install
```

3.4 Compile Xenomai user space

Let's compile Xenomai user space by first defining the absolute path of the cross-compiler and then running the configure command.

```
1  RaspberryPi/Xenomai-RPI/linux-3.8.13$ cd ../xenomai-2.6.3
2  RaspberryPi/Xenomai-RPI/xenomai-2.6.3$ PATH=$PATH:/home/$USER/RaspberryPi/
3  Xenomai-RPI/tools-master/ arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin
4  RaspberryPi/Xenomai-RPI/xenomai-2.6.3$ ./configure --host=arm-linux-gnueabihf
5  CFLAGS='-march=armv6' LDFLAGS='-march=armv6'
6  RaspberryPi/Xenomai-RPI/xenomai-2.6.3$ make
7  RaspberryPi/Xenomai-RPI/xenomai-2.6.3$ make DESTDIR=$(pwd)/RPI install
```

Compress the sbin, bin and lib folder and copy them to the Root partition

```
1  RaspberryPi/Xenomai-RPI/xenomai-2.6.3$ tar cjf xenomai-rpi.tar.bz2
2  usr/xenomai/bin/ usr/xenomai/sbin/ usr/xenomai/lib/ usr/xenomai/include/
3  RaspberryPi/Xenomai-RPI/xenomai-2.6.3$ sudo cp xenomai-rpi.tar.bz2
4  /media/$USER/Root/
5  RaspberryPi/Xenomai-RPI/xenomai-2.6.3$ cd /media/%USER/Root/
6  /media/%USER/Root/$ sudo tar xjf xenomai-rpi.tar.bz2 && sudo rm
7  xenomai-rpi.tar.bz2 && cd ..
8  /media/%USER/$ sudo umount /dev/sdb1 /dev/sdb2
```

Finally, we have our real time embedded system with Xenomai!

Chapter 4

Testing the system

It is common in the literature to report real-time test measurements made by the real-time system being tested. In this chapter we will present three experiments to measure the latencies of the system. We will first use the latency benchmark tool[1] provided by Xenomai and then implement two kernel modules to generate periodic signals with the GPIO pins of the Raspberry pi and register interrupts with the Real Time Xenomai API v2.6.4.

The Resources section of the Project website shows several benchmarks of the dual kernel over different versions of Linux. Tests compare the latencies of Xenomai 2.6.1 and Xenomai 3 (Cobalt) on x86 and the ARM architectures.

[1]For further information read the xeno-test manual page: http://www.xenomai.org/documentation/trunk/html/xeno-test/

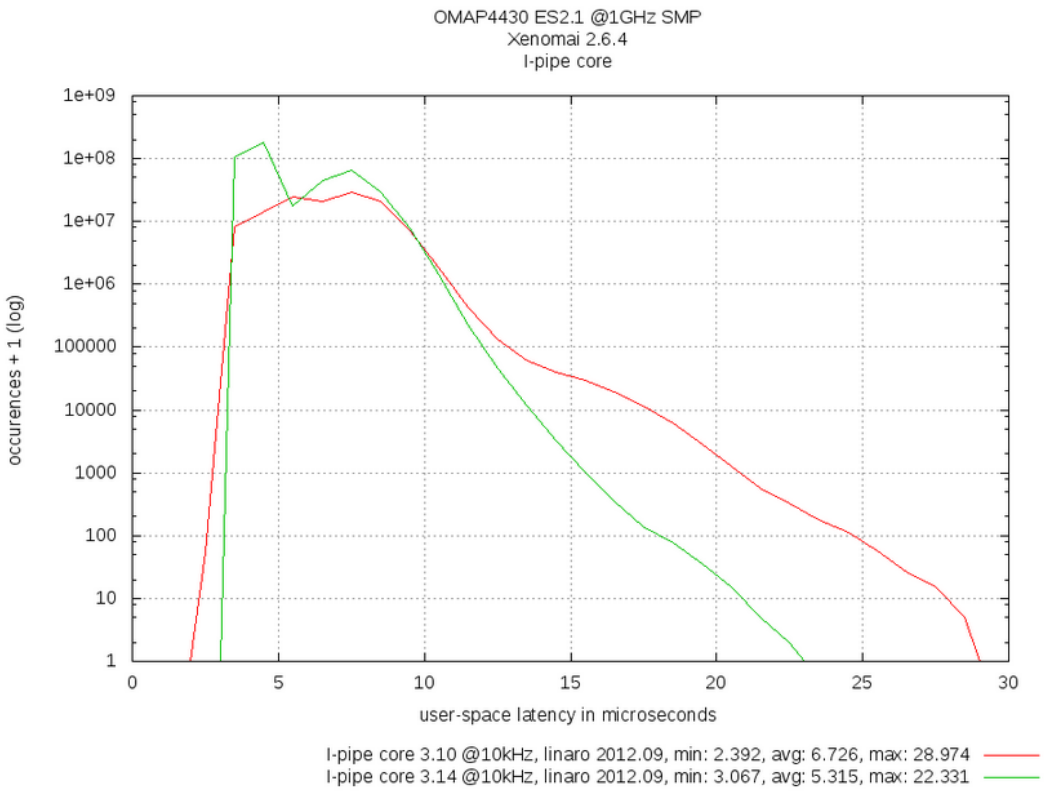

Figure 4.1: Texas Instrument Panda board, running a TI OMAP4430 processor at 1 GHz

As shown in Figure 4, the worst case latency with the Panda board is about 28 μs.

4.1 Test 1: Latency benchmark tool

Xenomai provides several benchmark tools to test real time features. They are available in the /usr/xenomai/bin folder and require the system to run a suitable Xenomai enabled kernel with the xeno_timerbench and xeno_native modules loaded. We will run the timer latency benchmark program written by Philippe Gerum and provided by the Xenomai test suite. The latency test displays a message every second with minimal, maximal and average latency values. We first run measurements when the

system is unloaded.

```
# echo 0 > /proc/xenomai/latency
# latency -p 100
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      2.000|      3.000|     17.000|        0|     0|      2.000|     17.000
RTD|      3.000|      4.000|     23.000|        0|     0|      2.000|     23.000
RTD|      3.000|      4.000|     21.000|        0|     0|      2.000|     23.000
RTD|      3.000|      4.000|     22.000|        0|     0|      2.000|     23.000
RTD|      3.000|      4.000|     20.000|        0|     0|      2.000|     23.000
RTD|      3.000|      4.000|     25.000|        0|     0|      2.000|     25.000
RTD|      2.000|      3.000|     24.000|        0|     0|      2.000|     25.000
RTD|      2.000|      3.000|     21.000|        0|     0|      2.000|     25.000
RTD|      3.000|      4.000|     25.000|        0|     0|      2.000|     25.000
RTD|      3.000|      4.000|     23.000|        0|     0|      2.000|     25.000
RTD|      3.000|      4.000|     24.000|        0|     0|      2.000|     25.000
RTD|      3.000|      4.000|     22.000|        0|     0|      2.000|     25.000
```

If we increase the workload and run the latency test again we obtain the following results.

```
# echo 0 > /proc/xenomai/latency
# latency -p 100
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  02:32:49  (periodic user-mode task, 100 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      5.000|      7.000|     24.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     23.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     19.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     22.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     31.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     21.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     22.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     20.000|        0|     0|      3.000|     38.000
RTD|      5.000|      8.000|     23.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     21.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     20.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     20.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     24.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     20.000|        0|     0|      3.000|     38.000
RTD|      5.000|      7.000|     26.000|        0|     0|      3.000|     38.000
```

It's worth pointing out that the latency average and minimal values increase while the maximal latency values remain the same (26 μs). These results underline the

deterministic behavior we are looking for.

4.2 Test 2: Interrupt response time

The most important metric for real time embedded systems is the time it takes for them to respond to outside events. Failure to handle such events efficiently can cause catastrophic results.

In this sectio, we will apply a technique for measuring the system's response time to an induced interrupt. The interrupts are triggered by an outside source, a frequency generator, by connecting it to an interrupt generating input pin on the target.

The aim of this experiment is to create a kernel module that implement an interrupt handler to toggle the state of one of the Raspberry's output pins and measure the exact time it takes for the system to respond to interrupts. The system's response will be measured and the square wave generated by the frequency generator will be plotted thanks to an oscilloscope connected to the board.

4.2.1 GPIO Interfaces

A "General Purpose Input/Output" is a flexible software-controlled digital signal. These interfaces are provided from many kinds of chip. Each GPIO represents a bit connected to a particular pin. To help catch system configuration errors, two calls are defined. The **gpio_request(unsigned gpio, const char *label)** requests GPIO, returning 0 or negative errno. The **gpio_free(unsigned gpio)** releases a previously claimed pin. If we want to use the pin as an input or an output, we can mark the direction using **gpio_direction_output(unsigned gpio, int value)** or **gpio_direction_input(unsigned gpio)**. The Raspberry pi B+ provides more than 29 pins with the expansion header. GPIO numbers are unsigned integers, so are IRQ

numbers. These make up two logically distinct namespaces. The following function maps pin numbers to IRQ numbers: **gpio_to_irq(unsigned gpio)**.

4.2.2 Interrupts

As discussed in 3.1.1, in order to keep latency predictable, the real-time system must ensure that the regular Linux kernel never defers external interrupts. The interrupts are masked at the hardware level and are delivered as quickly as possible to Xenomai. The interrupt pipeline relies on a modification of the Linux kernel sources for virtualizing the interrupt mask. The Xenomai co-kernel registers a set of handlers for various I-pipe events. These handlers notify Xenomai of any event before Linux can handle it.

The Interrupt management services of the Xenomai API contains a number of functions to deal with interrupts, such as create, enable, disable, and delete operations. In kernel space the **rtdm_irq_request** function takes a rtdm_irq_handler_t as argument and registers it with an IRQ line. We will use this function to associate an ISR to the external interrupt.

```
1 int rtdm_irq_request(
2         rtdm_irq_t *    irq_handle,
3         unsigned int    irq_no,
4         rtdm_irq_handler_t      handler,
5         unsigned long   flags,
6         const char *    device_name,
7         void *      arg)
```

The interrupt handler defines the operations that will be performed when the interrupt is triggered. In our case, we want to toggle the value of the GPIO pin every time the handler is invoked.

```
1 static int handler_interrupt(rtdm_irq_t * irq)
```

```
2  {
3      static int value = 0;
4      gpio_set_value(GPIO_OUT, value);
5      value = 1 - value;
6      return RTDM_IRQ_HANDLED;
7  }
```

4.2.3 The kernel module

In **irq-rtdm.c** module we will register an interrupt and associate an ISR to it in order to toggle the value of a GPIO pin. We will then measure the system's response time. The __init function requests two GPIO pins, configures one as input and the other as output and registers the interrupt handler. The __exit function releases the GPIO pins and the interrupt handler.

```
1  #include <linux/gpio.h>
2  #include <linux/interrupt.h>
3  #include <linux/module.h>
4
5  #include <rtdm/rtdm_driver.h>
6
7  // GPIO Input 22
8  #define GPIO_IN  22
9  // GPIO Output 22
10 #define GPIO_OUT 25
11
12 static rtdm_irq_t irq_rtdm;
13
14 static int handler_interrupt(rtdm_irq_t * irq)
15 {
16     static int value = 0;
17     gpio_set_value(GPIO_OUT, value);
18     value = 1 - value;
19     return RTDM_IRQ_HANDLED;
20 }
21
22 static int __init exemple_init (void)
23 {
24     unsigned int numero_interrupt = gpio_to_irq(GPIO_IN);
25     int err;
26
27     printk("Module loaded\n");
28     if ((err = gpio_request(GPIO_IN,"input")) != 0)
29     {
30         printk("error %d: could not request gpio: %d\n"
31         ,err, GPIO_IN);
32         return err;
33     }
34     else if (err == 0)
35     {
36         printk("info: GPIO %d requested.\n",GPIO_IN);
```

```
37      }
38
39      if ((err = gpio_direction_input(GPIO_IN)) != 0) {
40          gpio_free(GPIO_IN);
41          printk("error %d: could not request diretction
42          to gpio: %d\n",err, GPIO_IN);
43          return err;
44      }
45      if ((err = gpio_request(GPIO_OUT,"output")) != 0) {
46          printk("error %d: could not request gpio: %d\n"
47          ,err, GPIO_OUT);
48          gpio_free(GPIO_IN);
49          return err;
50      }
51      else if (err == 0)
52      {
53          printk("info: GPIO %d requested.\n",GPIO_OUT);
54      }
55
56      if ((err = gpio_direction_output(GPIO_OUT,1)) != 0) {
57          printk("error %d: could not request direction
58          to gpio: %d\n",err, GPIO_OUT);
59          gpio_free(GPIO_OUT);
60          gpio_free(GPIO_IN);
61          return err;
62      }
63
64      //irq_set_irq_type(numero_interrupt,  IRQF_TRIGGER_RISING);
65
66      if ((err = rtdm_irq_request(& irq_rtdm,
67                      numero_interrupt, handler_interrupt,
68                      RTDM_IRQTYPE_EDGE,
69                      THIS_MODULE->name, NULL)) != 0) {
70          printk("error %d: could not request gpio: %d\n",
71  err, GPIO_IN);
72          gpio_free(GPIO_OUT);
73          gpio_free(GPIO_IN);
74          return err;
75      }
76      return 0;
77  }
78
79  static void __exit exemple_exit (void)
80  {
81      printk("Module removed\n");
82      rtdm_irq_free(& irq_rtdm);
83      gpio_free(GPIO_OUT);
84      gpio_free(GPIO_IN);
85  }
86
87  module_init(exemple_init);
88  module_exit(exemple_exit);
89  MODULE_LICENSE("GPL");
```

To compile the module we will use the cross compiler provided by Buildroot. During the configuration step in Chapter two, we specified /usr/local/cross-rpi as host directory in the build options menu.

```
EXTRA_CFLAGS := -I /home/userk/Development/Linux/RaspberryPi/
    Xenomai-RPI/xenomai-2.6.3/RPI/usr/xenomai/include/

ifneq (${KERNELRELEASE},)
        obj-m += irq-rtdm.o
else
    ARCH ?= arm
    CROSS_COMPILE ?= /usr/local/cross-rpi/usr/bin/arm-linux-
    KERNEL_DIR = /home/userk/Development/Linux/RaspberryPi/
       Xenomai-RPI/linux-3.8.yOK/
    MODULE_DIR := $(shell pwd)
    CFLAGS := -Wall -g

.PHONY: all
all:: modules

.PHONY: modules
modules:
        ${MAKE} ARCH=${ARCH} CROSS_COMPILE=${CROSS_COMPILE}
           -C ${KERNEL_DIR} SUBDIRS=${MODULE_DIR}  modules

XENOCONFIG=/home/userk/Development/Linux/RaspberryPi/Xenomai-RPI/
   xenomai-2.6.3/RPI/usr/xenomai/bin/xeno-config

.PHONY: clean
clean::
   rm -f  *.o  .*.o  .*.o.* *.ko  .*.ko  *.mod.* .*.mod.* .*.cmd *¬
   rm -f Module.symvers Module.markers modules.order
   rm -rf .tmp_versions
endif
```

Once copied in the micro-sd card and moved in the current working directory on the target device, we can load the module with **insmod irq-rtdm.ko** and check the log with **dmesg | tail -3** command. The following messages should appear.

```
module loaded
info: GPIO 22 requested.
info: GPIO 25 requested.
```

The system's response time with no load is shown in Figure 4.1. As we can see, the latency is around 3 μs.

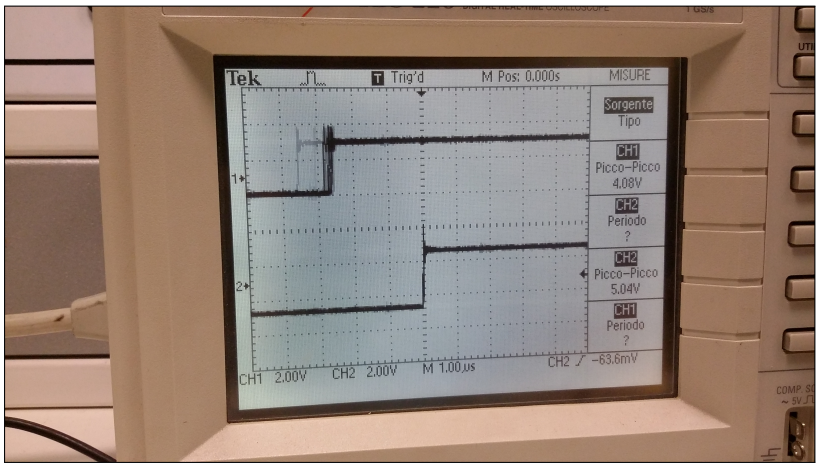

Figure 4.2: Response time with unloaded system

In order to increase the cpu usage to 100% we asked the system to perform the following operation[2]:

```
1 dd if=/dev/zero of=/dev/null
```

The test was conducted with the maximum system load for two hours. Figure 4.3 and Figure 4.4 show that the maximum latency of the system never exceeds 20μs.

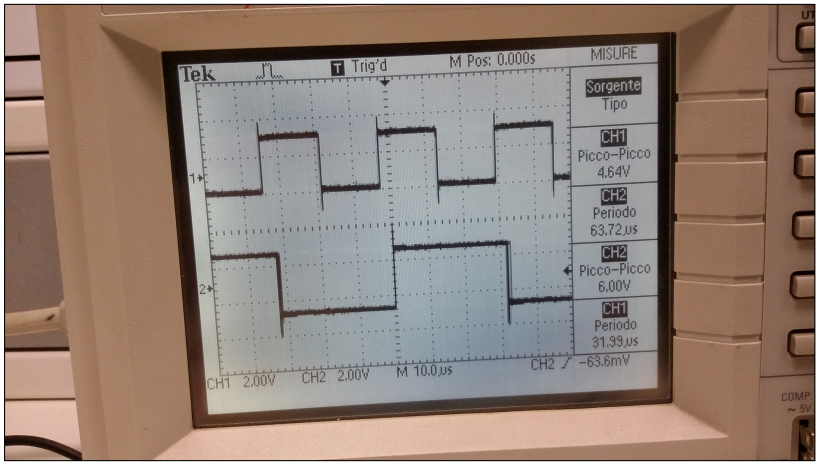

Figure 4.3: Response time with 100% cpu usage

[2]http://unix.stackexchange.com/questions/185559/which-commands-or-operations-can-be-used-to-put-the-cpu-under-intense-load

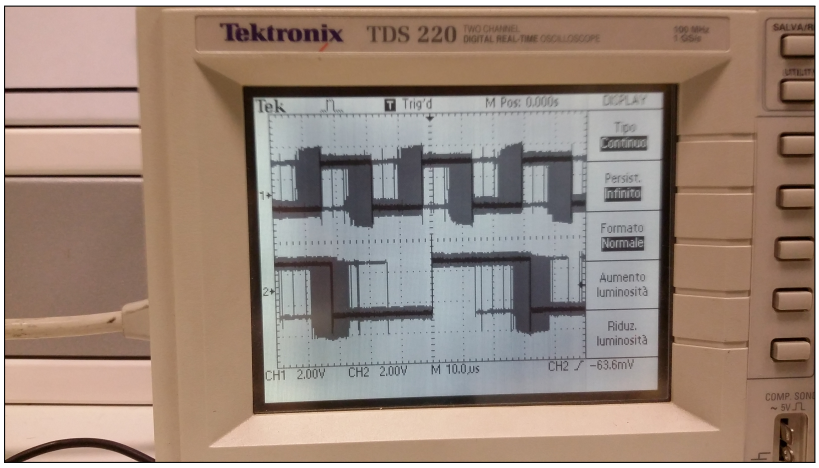

Figure 4.4: Response time with 100% cpu usage, 120 min.

4.3 Test 3: Periodic signal

The aim of this second test is to determine whether the Raspberry Pi is able to generate a reliable Pwm signal in order to control a brushless motor. Pulse With Modulation is a technique used to encode a message into a pulsing signal. Pulses of various lengths (the information itself) are sent at regular intervals. For simplicity's sake, we have written a basic kernel module simulating a 50% duty cycle thrust input as shown in Figure 4.5.

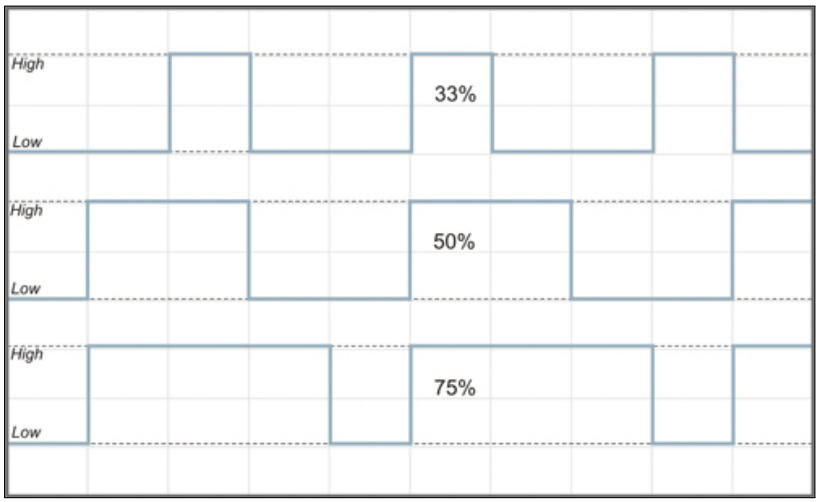

Figure 4.5: Pwm duty cycle

4.3.1 RealTime Driver Model and Timer services

In order to take advantage of the real time features of the system we will use the Timer services of the RealTime Driver Model (RTDM). The RTDM acts as a mediator between the application requesting a service from a certain device and the device driver offering it. Its API are addressable both from kernel and user space. The RTDM skin makes use of Xenomai's ability user space real time threads automatically between Hard Real Time and Linux operation mode.

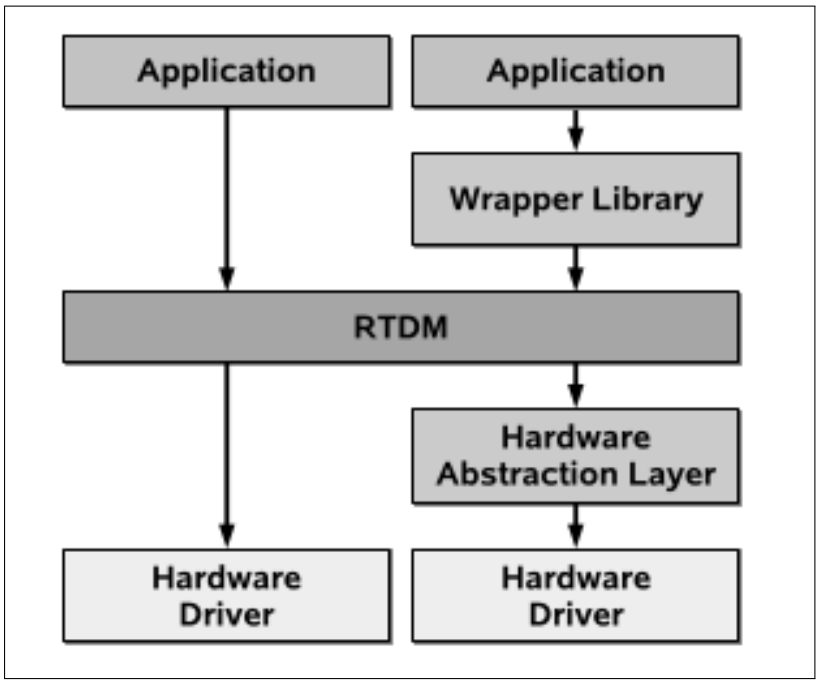

Figure 4.6: RTDM and related layers

The system we have created in the previous chapter provides a simple API for the construction and management of timers. It consists of functions for timer creation, cancellation, and management. In order to use timers the user must initialize it with **rtdm_timer_init** and start it with the **rtdm_timer_start** function specifying the firing time, the interval and the mode which in our case will be RTDM_TIMERMODE_REALTIME.

```
1 rtdm_timer_init(& rtimer, timer_osc, "Oscillator")
2 rtdm_timer_start(& rtimer, period_us*1000, period_us*1000,
3 RTDM_TIMERMODE_REALTIME)
```

With the initialized timer, the user need to specify the timer handler and the operations to perform. In our case we will just toggle the value of the GPIO pin.

```
1 static void timer_osc(rtdm_timer_t * unused)
```

```
2 {
3     static int value = 0;
4     gpio_set_value(GPIO_OSC, value);
5     value = 1 - value;
6 }
```

In the next subsection we will discuss the GPIO handling in kernel mode.

4.3.2 The kernel module

In this module we will implement a basic periodic square wave on pin 22. We will pass the period as a parameter to the module in order to assign the parameter value at load time by calling **insmod pwm period_us=x**. In the __init function we will first claim pin 22, configure it as an output, initiate and start the rtdm timer. Next, we define the timer handler as described in 4.3.1. And in the __exit function we release the GPIO pin, and stop and destroy the timer.

```
1  #include <linux/gpio.h>
2  #include <linux/module.h>
3  #include <linux/sched.h>
4  #include <linux/version.h>
5  #include <rtdm/rtdm_driver.h>
6
7  static int period_us = 1000;
8  module_param(period_us, int, 0644);
9
10 static void timer_osc(rtdm_timer_t *);
11 static rtdm_timer_t rtimer;
12
13 // pin 15 of the P1 header of the Raspberry Pi : GPIO 22
14 #define GPIO_OSC 22
15
16 static int __init init_osc(void)
17 {
18     int err;
19     printk(KERN_NOTICE "Pwm Kernel module loaded.
20     [pin,period]=[22,%d]\n", period_us);
21     if ((err = gpio_request(GPIO_OSC, THIS_MODULE->name))!= 0)
22     {
23         return err;
24     }
25     if ((err = gpio_direction_output(GPIO_OSC, 1))!= 0)
26     {
27         gpio_free(GPIO_OSCILLATEUR);
28         return err;
29     }
30
31     if ((err = rtdm_timer_init(& rtimer, timer_osc,
32          "oscillator")) != 0)
```

```
33      {
34          gpio_free(GPIO_OSC);
35          return err;
36      }
37
38      if ((err = rtdm_timer_start(& rtimer, period_us*1000,
39           period_us*1000, RTDM_TIMERMODE_REALTIME)) != 0)
40      {
41          rtdm_timer_destroy(& rtimer);
42          gpio_free(GPIO_OSC);
43          return err;
44      }
45      return 0;
46  }
47
48  static void __exit exit_osc (void)
49  {
50      rtdm_timer_stop(& rtimer);
51      rtdm_timer_destroy(& rtimer);
52      gpio_free(GPIO_OSC);
53  }
54
55  static void timer_osc(rtdm_timer_t * unused)
56  {
57      static int value = 0;
58      gpio_set_value(GPIO_OSC, value);
59      value = 1 - value;
60  }
61
62  module_init(init_osc);
63  module_exit(exit_osc);
```

We can cross-compile the module as we did for the second experiment using the following **Makefile**.

```
1   EXTRA_CFLAGS := -I /home/userk/Development/Linux/RaspberryPi/
2        Xenomai-RPI/xenomai-2.6.3/RPI/usr/xenomai/include/
3   ifneq (${KERNELRELEASE},)
4           obj-m += pwm.o
5   else
6       ARCH ?= arm
7       CROSS_COMPILE ?= /usr/local/cross-rpi/usr/bin/arm-linux-
8       KERNEL_DIR = /home/userk/RaspberryPi/Xenomai-RPI/linux-3.8.13/
9       MODULE_DIR := $(shell pwd)
10      CFLAGS := -Wall -g
11
12  .PHONY: all
13  all:: modules
14
15  .PHONY: modules
16  modules:
17          ${MAKE} ARCH=${ARCH} CROSS_COMPILE=${CROSS_COMPILE}
18           -C ${KERNEL_DIR}  SUBDIRS=${MODULE_DIR}  modules
19
20  XENO_DESTDIR:=/home/userk/RaspberryPi/Xenomai-RPI/xenomai-2.6.3/RPI/
21  XENO_CONFIG:=$(XENO_DESTDIR)/usr/xenomai/bin/xeno-config
22  XENO_POSI_CFLAGS:=$(shell DESTDIR=$(XENO_DESTDIR)
23       %(XENO_CONFIG) --skin=posix --cflags)
```

```
24 XENO_POSIX_LIBS:$(shell DESTDIR=$(XENO_DESTDIR)
25     $(XENO_CONFIG) --skin=posix --ldflags)
26
27 .PHONY: clean
28 clean::
29     rm -f  *.o  .*.o  .*.o.* *.ko  .*.ko  *.mod.* .*.mod.* .*.cmd *¬
30     rm -f Module.symvers Module.markers modules.order
31     rm -rf .tmp_versions
32 endif
```

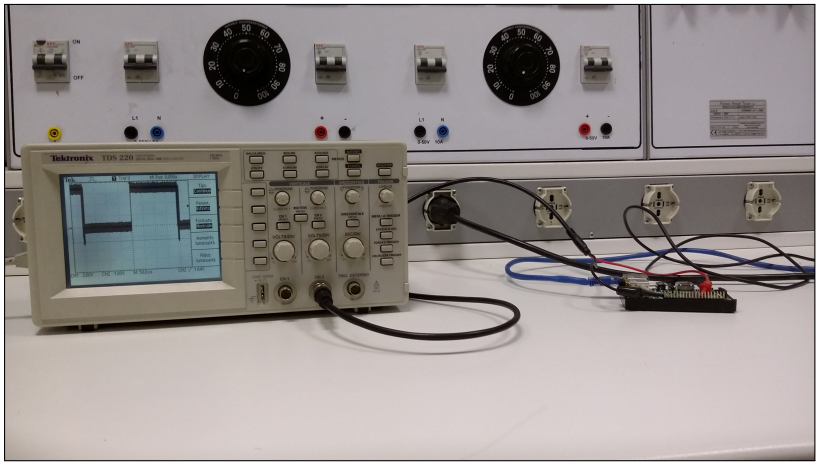

Figure 4.7: The Raspberry Pi B+ connected to an oscilloscope

4.3.3 Results and conclusions

The experiment was conducted with the 100% of cpu usage for two hours. The square wave has been displayed thanks to an oscilloscope connected to pin 22 as shown in Figure 4.3.3 and Figure 4.3.2.

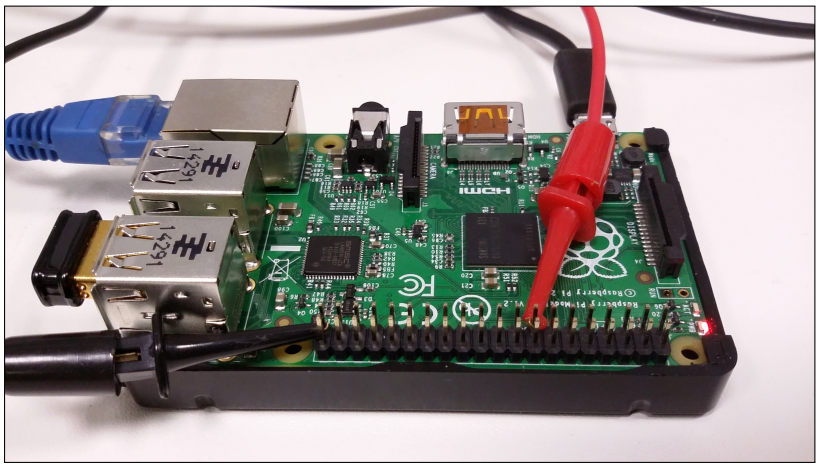

Figure 4.8: Raspberry Pi B+ setup

The following pictures show the maximal latency of the system under intense load. The experiments were conducted with 50μs and 200μs periods. As we can see, the maximal latency never exceeds **20 μ-seconds**, as expected from Xenomai benchmarks discussed in 4.1.

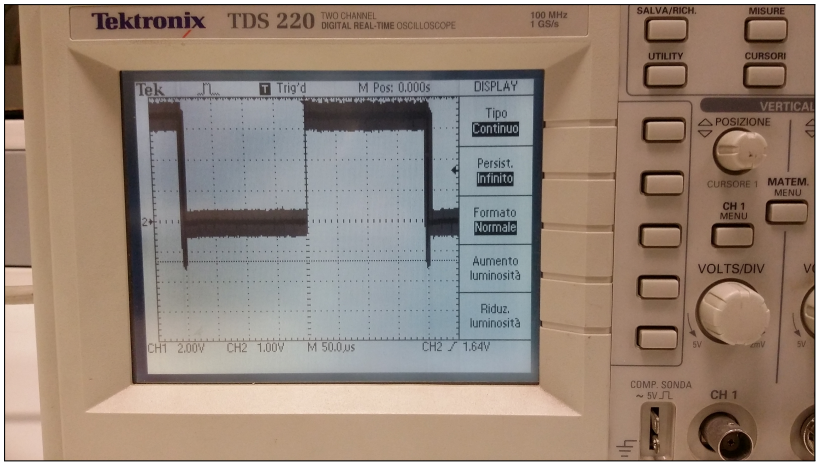

Figure 4.9: Square wave with 200μs period

If we now consider the objective of the application, the crucial point is to ensure that the pulse width remains exactly the same as long as we want the brushless motor to rotate at a constant rate. The Raspberry Pi controls the inputs of a quadrotor

system in which the roll τ_θ and pitch τ_φ torques are given by a similar expression:

$$\tau_\theta = Lk(\omega_1^2 - \omega_2^2) \tag{4.3.1}$$

in which k is a constant ω_i is the angular rate of the i-motor and L is the distance between the motors and the center of gravity of the object frame. From the results obtained we can conclude that if a square wave of x μs is required, the Raspberry Pi will produce a signal with a period of $(x \pm \varepsilon)\mu$s with $\varepsilon > 5\mu$s. If we consider the resulting pwm signal, this small uncertainty will end up in small change in angular velocity of the motor which will generate an undesired torque acting on the body frame. Although we have minimized the uncertainty in the response time, the real time system cannot be used to handle pwm signal controlling the attitude of the quadcopter. A dedicated micro-controller, such as Arduino, will be included in the system as a reliable PWM signal generator.

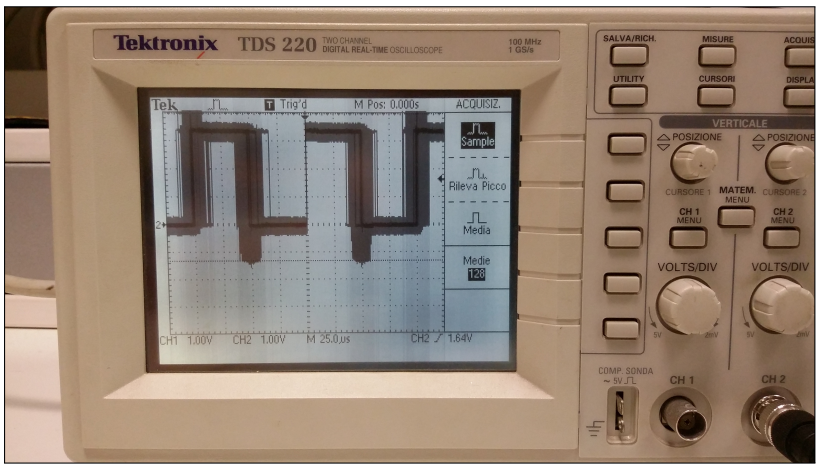

Figure 4.10: Square wave with 50μs period

Bibliography

[1] Giorgio C. Buttazzo, , *"Hard Real-Time Computing Systems."*, Second edition. Springer, 2005.

[2] Jane W. S. Liu, *"Real-Time Systems"*, Prentice Hall, 2000.

[3] Marco Cesati, Daniel P. Bovet *"Understanding the Linux Kernel"*, 2008.

[4] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, Philippe Gerum, *"Building Embedded Linux Systems"*, 2008.

[5] Sistemi Embedded e Real Time, *"http://sert14.sprg.uniroma2.it/"*, Marco Cesati, 2013.

[6] J. Kiszka, University of Hannover Appelstrasse 9A, 30167 Hannover, Germany, *"Real Time Driver Model and first applications"*.

[7] Xenomai official project page, *"http://xenomai.org"*.

[8] The Linux Kernel Archives , *"https://www.kernel.org/"*.

[9] Linux from scratch, *"http://www.linuxfromscratch.org/"*.

[10] Buildroot, Making Embedded Linux Easy, *"http://buildroot.uclibc.org/"*.