REAL TIME OPERATING SYSTEM (RTOS)

FOR RASPBERRY PI 3

Let's find an OS for Raspi!

Antoine BONNIN - abonnin@etud.insa-toulouse.fr

Clément CHADUC - chaduc@etud.insa-toulouse.fr

Kévin DUCONGÉ - duconge@etud.insa-toulouse.fr

Lucas REBOUÇAS - reboucas@etud.insa-toulouse.fr

Nabil SELLAM - sellam@etud.insa-toulouse.fr

GuangXue ZHANG - gxzhang@etud.insa-toulouse.fr

Toulouse – 02/11/2016

Abstract

The purpose of this document is to bring some elements about the Operating System (OS) of the Raspberry. The matters discussed in this document are:

- ➔ Choice of the technology
- ➔ Solution to implement the technology
- ➔ Test protocols
- ➔ Test results
- ➔ Way to make a back-up of OS
- ➔ How to code and build RT programs

Whenever possible, text is replaced by figures and tables to make an easy to read document.

Table of Contents

1 Elements of justification for OS choice

1.1 Why an RTOS?

Client requirements ask for a time guaranty system. Meaning whatever the amount of processes running, system must be able to deliver an answer at the same date. However, due to car speed (under 10 km/h), it is no need to offer an absolute real time system or a hard real time system: soft real time systems are easier and cheaper to develop.

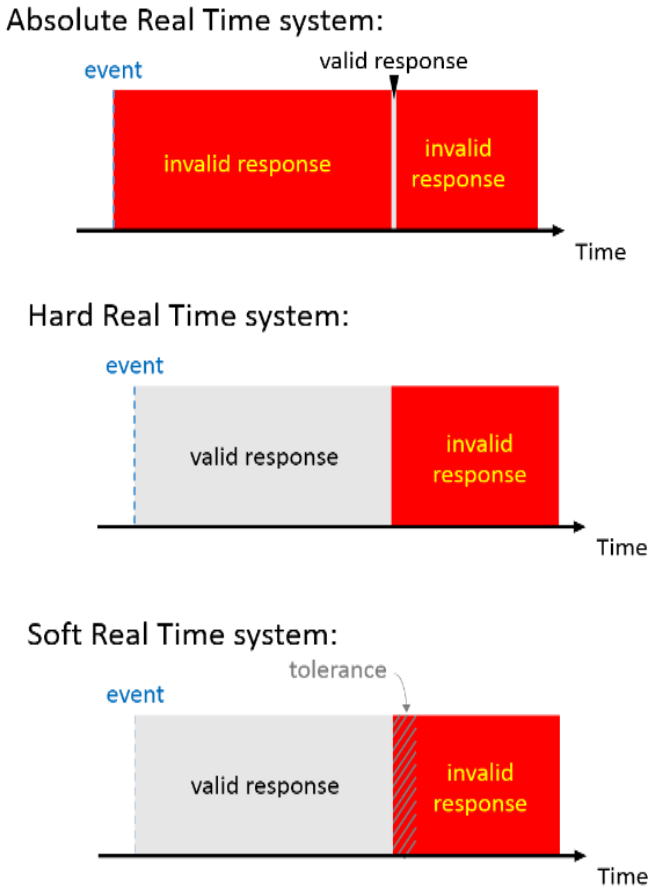

Figure 1: Comparison between Hard and Soft Real Time System

So, a soft RTOS is used. Meaning, answer time is independent from tasks whose priority is lower of the considerate task. Part 1.2 develops existing solutions.

1.2 Solution offering soft RTOS

Various solutions, with various performances, limitations and costs exist to deploy an RTOS. The first solution is to install directly an RTOS, the second consists in adding a co-kernel to a non RTOS.

Full RTOS against non-RTOS and co-kernel main advantage is a smaller jitter[1] (for Raspberry Pi 2, sources report less than 3% difference). Contrariwise, the way to install a full RTOS is much less documented and seems harder. To finish, using a full RTOS means use only RT drivers.

For Raspberry Pi, full RTOS solution consists in installing Xenomai and co-kernel solution consists in patching a non RTOS.

In view of the foregoing, we decided to install Raspbian (a non RTOS) on the Raspberry and to patch it with RT-PREMPT.

This scheme shows how a system with a co-kernel works:

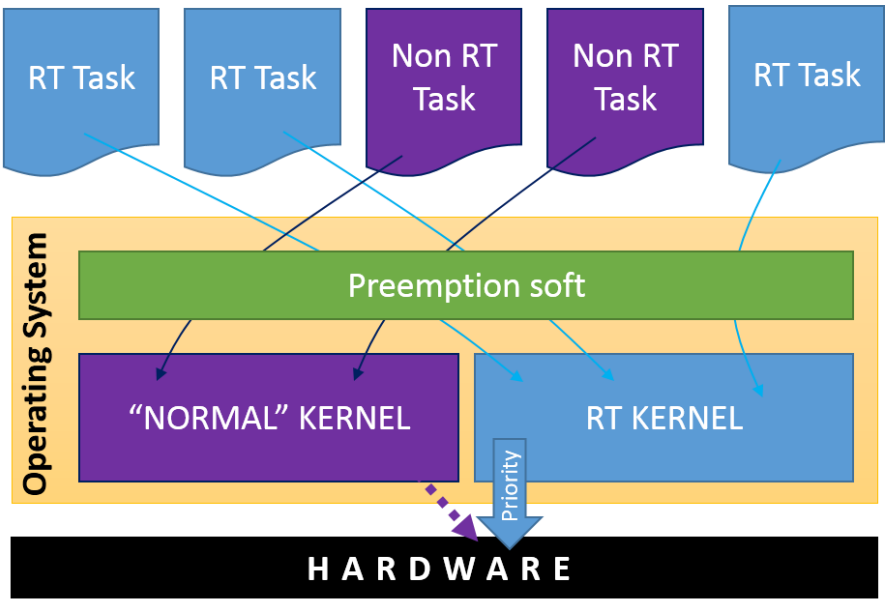

Figure 2: Tasks are distributed by preemption soft between the two kernels

[1] jitter = delay variation

2 Brief abstract on the way to implement co-kernel

2.1 From a non-compiled kernel

This solution needs first to install a Raspbian (lite or graphical). Then, it is recommended to cross compile the kernel on a machine with a higher computing power than a Raspberry Pi (a basic computer is enough).

A lot of tutorials are available on internet, but none of them were successful. Encountered problems were: errors at downloading (with time sources have moved or are not available anymore) or at compiling (it exists several Raspberry Pi versions, unfortunately cross compiler and sources are not adapted for the last ones).

2.2 From a precompiled kernel

This solution needs first to install a Raspbian (lite or graphical). It is also recommended to put on an USB flash drive the precompiled kernel.

Then, all the manipulations must be done on the Raspberry.

0.1	Check the information of installed system	`pi@raspberrypi:~ $ uname –a`

Normally answer is:

```
Linux raspberrypi 4.4.26-v7+ #915 SMP Thu Oct 20 17:08:44 BST 2016 armv7l GNU/Linux
```

It means system has a “classical” operating system. So, let’s install co-kernel.

1.1	Remove booting system	`pi@raspberrypi:~ $ sudo rm -r /boot/overlays/`
1.2	Remove firmware	`pi@raspberrypi:~ $ sudo rm -r /lib/firmware/`
1.3	Move to /tmp	`pi@raspberrypi:~ $ cd /tmp`
1.4	Copy pre-compiled kernel	`pi@raspberrypi:/tmp $ cp /media/pi/1605-BBEC/kernel-4.4.9-rt17.tgz ./`
1.5	Decompress pre-compiled kernel	`pi@raspberrypi:/tmp $ tar xzf kernel-4.4.9-rt17.tgz`
1.6	Move to new kernel boot directory	`pi@raspberrypi:/tmp $ cd boot/`
1.7	Implement new booting on system	`pi@raspberrypi:/tmp/boot $ sudo cp -rd * /boot/`
1.8	Move to new kernel lib directory	`pi@raspberrypi:/tmp/boot $ cd ../lib`
1.9	Implement new firmware on system	`pi@raspberrypi:/tmp/lib $ sudo cp -rd * /lib/`
1.10	Reboot system	`pi@raspberrypi:/tmp/lib $ sudo reboot`

Once system has rebooted, kernel patching is over:

2.1	Check the information of patched system	`pi@raspberrypi:~ $ uname –a`

Normally answer is:

```
Linux raspberrypi 4.4.9-rt17-v7+ #1 SMP PREEMPT RT Wed May 11 22:46:14 CEST 2016 armv7l GNU/Linux
```

To test the system, please see parts 3 and 4.

2.3 From an image disk

All images disks are delivered to client. So the easiest way for him is to directly burn on a SD-card one of it. Please read parts 5.1.2 and 5.2.2 for further information. By choosing this option, it is no need to install anything else: Raspbian is already installed and patched.

3 Test protocols for RTOS

3.1 Test philosophy

These tests are made to prove a system is real time. So the aim is to verify if a task scheduled is processed in time. Tests on a non-charged system are not sufficient: all systems are real time as long as they have only one task to process.

To do these tests, two softs are used: *cyclictest* and *hackbench*. The first one schedules a task every 1 millisecond and measure the treatment delay. The second one creates some tasks whose purpose is to charge the system (in fact these tasks are supposed to send messages with a given length the one to the others). Use such software guarantees to have the same conditions for all the tests.

3.2 Test protocol

3.2.1 Normal test procedure

"Normal" test procedure means tests are processed directly on Raspberry Pi with a graphical OS.

These operations must be done only one time:

0.1	Create directory test-RT at /home/pi	`pi@raspberry mkdir /home/pi/tests-RT`
0.2	Copy test folder in /home/pi/	`pi@raspberry cp cheminSource /home/pi/tests-RT/`

These operations must be done for each test:

1.1	Go in rt-tests directory	`pi@raspberry cd /home/pi/tests-RT/rt-tests`
1.2	Launch *cyclictest* and let it run	`pi@raspberry sudo cyclictest___` 3.3.1

In a new terminal:

2.1	Go in rt-tests directory	`pi@raspberry cd /home/pi/tests-RT/rt-tests`
2.2	Launch *hackench* and let it run	`pi@raspberry sudo hackbench_____` 3.3.2

When *hackbench* is over, you can go in first terminal and kill (Ctr+C) *cyclictest*.

Results are printed on shell. Please see 0 for interpretation.

3.2.2 Distant test procedure

This test can be launched threw SSH[2]. In this case, the easiest way is to have two different SSH. In this case test procedure is the one introduced at part 3.2.1. Otherwise, follow procedure of part 3.2.3.

3.2.3 Other test procedure

If it is not possible to launch two different terminals, you have to follow this procedure.

These operations must be done only one time:

0.1	Create directory test-RT at /home/pi	`pi@raspberry mkdir /home/pi/tests-RT`
0.2	Copy test folder in /home/pi/	`pi@raspberry cp cheminSource /home/pi/tests-RT/`
0.3	Create directory output	`pi@raspberry mkdir /home/pi/tests-RT/output`

These operations must be done for each test:

1.1	Go in tests-RT directory	`pi@raspberry cd /home/pi/tests-RT`

1.2	Launch *cyclictest* in background
	Note pid number
		`pi@raspberry`
		`sudo rt-tests/cyclictest___`[3.3.1] `> output/test1`
		`[1] XXX`

1.3	Launch *hackench* and let it run	`pi@raspberry sudo rt-tests/hackbench_____`[3.3.2]

Once *hackbench* is over :

2.1	Kill *cyclictest*	`pi@raspberry sudo kill XXX`

In this case, test result is not printed on the shell, but it is in the specified file (meaning /home/pi/tests-RT/ output/test1). Please note, that interesting values are at the end of the document. See 0 to interpret data.

[2] SSH = Secure Shell: a session on the Raspberry is open from a distant machine. For Windows machine, this can be done via *Putty*. For a Linux machine, you have to use a terminal and run a command like `ssh user@X.X.X.X`

3.3 Expected results and interpretation

3.3.1 *cyclictest* options

cyclictest can be used with or without any options. All options can be found by typing `/home/pi/tests-RT/rt-tests/cyclictest --help` in a terminal.

Recommended call for *cyclictest* is: `sudo cyclictest -p 99 -t 5 -n` where `-p X` set thread priority and `-t X` set number of thread, `-n` option is for use clock_nanosleep. Interval between two scheduled tasks can be set with `-i X` (in µs, default is 1000 µs).

When *cyclictest* exit is redirected in a file, you should use `-q` option to only keep a summary of test.

3.3.2 *hackbench* options

hackbench can be used with or without any options. All options can be found by typing `/home/pi/tests-RT/rt-tests/hackbench --help` in a terminal.

Recommended call for *hackbench* is: `sudo hackbench -p -g 20 -l 1000`

Following options must be considered:

`-p` allow to use or not pipes

`-g X` set the number of groups of tasks

`-f X` number tasks in each group

`-l` number of messages emitted by each task

`-s X` length (in bytes) of each message

Finally, *hackbench* creates *g* groups of *f* tasks which exchange *l* messages of *s* bytes in each direction.

3.3.3 Interpretation and validity

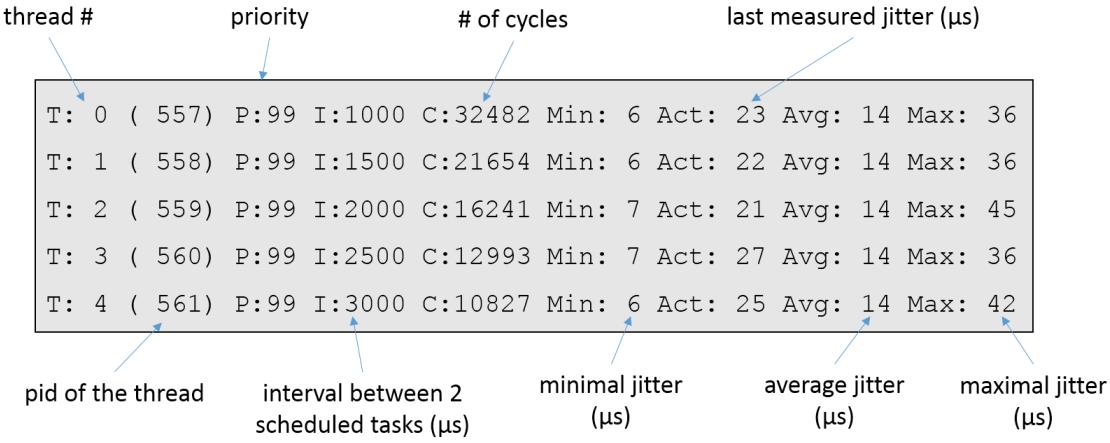

Figure 3: cyclictest results signification

Whatever the chosen options, a comparison between max jitter and interval between two scheduled tasks must be done.

4 Test results for RT-PREMPT on Raspberry

Test were performed on a Raspberry Pi 3 with a graphical interface. Four different system loads were chosen. However, whatever applied load, Raspbian gauge indicates that system load is maximal (100%).

- Small load: `sudo ./hackbench -p -g 20 -l 1000`
- Medium load: `sudo ./hackbench -p -g 50 -l 1000`
- Large load: `sudo ./hackbench -p -g 100 -l 10000`
- Extreme load: `sudo ./hackbench -p -g 200 -l 1000`

All tests are performed with the same parameters for *cyclictest*: `sudo cyclictest -p 99 -t 5 -n`

4.1 Non-RTOS results

	Maximum Jitter		Average Jitter		*Hackbench* execution time
Empty	36 µs	3,6 %	13 µs	1,3 %	----
Small	1494 µs	> 100 %	13 µs	1,3 %	10, 434 s
Medium	4970 µs	> 100 %	14 µs	1,4 %	
Large	Not Testable : Raspberry crash before the end of test				
Extreme					

Table 1: Results for tests performed on classic Raspbian OS

4.2 RTOS results

	Maximum Jitter		Average Jitter		*Hackbench* execution time
Empty	35 µs	3,5 %	14 µs	1,4 %	----
Small	73 µs	6,9 %	17 µs	1,7 %	39,331 s
Medium	80 µs	8,0 %	18 µs	1,8 %	97,957 s
Large	88 µs	7,3 %	18 µs	1,8 %	193,581 s
Extreme	108 µs	10,8 %	19 µs	1,9 %	367,022 s

Table 2: Results for tests performed on RTOS

4.3 Analysis

Finally, PREEMPT-RT is acceptable for the project. If system load stays acceptable, jitter is independent from system load. Co-kernel has also an interesting behavior as it avoid system crash in case of overload. However, this solution is not perfect as it implies longer execution time for programs.

5 Backup solution

Important point, some problems appear (burning is impossible) if SD-card size is smaller than image disk size. It seems logic, but be careful when using SD-cards from different suppliers: sizes vary from a few bits, and can cause troubles.

5.1 With a Linux (Debian) computer

Used apps are very common and can be installed by: `sudo apt-get install MissingApp`

All the following manipulations must be done in a shell on computer.

5.1.1 Save image disk on computer

0.1	Identify SD-card name	`klem@computer sudo fdisk -l`

Generally your disk is /dev/sda or similar

1.1	Launch SD-card copy and zip	`klem@computer sudo dd if=disk_path | gzip -9 > backup_path`

5.1.2 Burn image disk on SD-card

0.1	Identify SD-card name	`klem@computer sudo fdisk -l`

Generally your disk is /dev/sda or similar

1.1	Launch SD-card burning	`klem@computer sudo dd bs=1M if=source_path of= disk_path`

5.2 With a Windows (Seven) computer

Solution is very simple and only need to download *Win32DiskImager*. Software interface is quite simple:

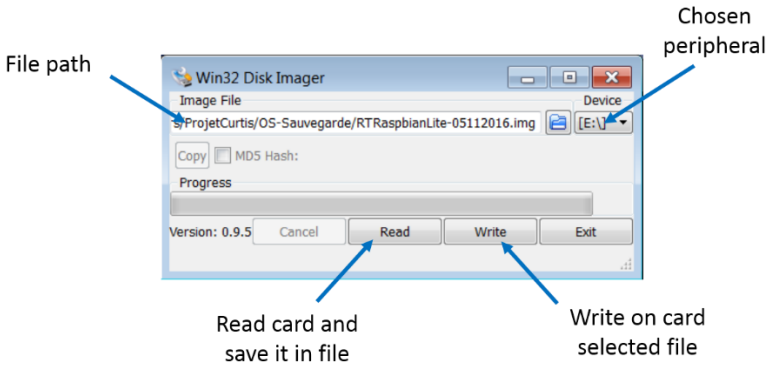

Figure 4: Win32DiskImager user interface

5.2.1 Save image disk on computer

To save an image disk, specify peripheral and output file path and click “read”.

5.2.2 Burn image disk on SD-card

To burn an image disk, specify peripheral and input and click “read”. Burning totally erase SD-card.

6 Write and compile RT code

6.1 Code environment

pthread library is used to build real time programs. This library is installed by default on all Unix systems (meaning on Raspbian too). You already used *pthread* library to run *cycletest* and *hackbench*.

6.2 Building tools

Compilation can directly be done on Raspberry.

Here is the command line to use: `gcc -o MYPROGRAM mysource.c -lpthread`

It is strongly recommended to put `-lpthread` at the end of the line (and not at the beginning) to be sure to be recognized by system.

6.3 RT code example

6.3.1 Basic RT code

This short example shows how to include *pthread* library in your code. It also indicates how to declare and create RT tasks.

6.3.2 Mutex using RT code

This short example shows how to use shared variables, especially by using mutex.

Bibliography TODO

Infos temps reel sur Raspi

http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMFHS-075/Raspberry-Pi-et-temps-reel

Procédures de test

http://2014.capitoledulibre.org/symposion_media/media/coverage/linux-temps-reel-sur-raspberry-pi/Linux_RPi_TR_Capitole2014.pdf

Installation noyau

http://www.frank-durr.de/?p203

Installation Win32DiskImager

https://sourceforge.net/projects/win32diskimager/

Développer en RT

http://gkemayo.developpez.com/tempsreel/programmation-multitache-posix/#LIII-B

List of Tables

List of Figures

Appendix

N/A

INSA Toulouse
135, avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

Université Fédérale Toulouse Midi-Pyrénées

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE