POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

Development of a real-time application based on Xenomai

Supervisor
Prof. Maurizio Rebaudengo

Candidate
Paolo Michelotti

Internship tutor
CREA Semiconductor Test Equipment
Dott. Ing. Marco Marcinnò

Academic Year 2018 – 2019

This page is intentionally left blank

Acknowledgments

I would like to thank first of all Eng. Marco Marcinnò for making this work possible and for following me throughout its development with valuable advice and useful suggestions. I also thank all the employees of the CREA company for their hospitality and for making me feel immediately part of a family.

I thank my supervisor, Prof. Maurizio Rebaudengo, for supporting me during the whole thesis, suggesting the path to follow.

I thank all my friends at Politecnico for sharing hours of classes, workshops and lunches. I thank all my old friends, for sharing with me all these years.

A special thank to my parents, for always supporting me in all my decisions.

Ringraziamenti

Vorrei ringraziare innanzitutto l'Ingegner Marco Marcinnò per aver reso possibile questo lavoro e per avermi seguito durante tutto il suo svolgimento con preziosi consigli e utili suggerimenti. Ringrazio inoltre tutti i dipendenti della ditta CREA per la loro accoglienza e per avermi fatto sentire fin da subito parte di una famiglia.

Ringrazio il mio relatore, professor Maurizio Rebaudengo, per avermi supportato durante tutto il periodo di tesi suggerendomi la strada da seguire.

Ringrazio tutti i miei amici del Poli, con cui ho condiviso ore di lezione, laboratori e pranzi. Ringrazio tutti i miei amici di sempre, per aver condiviso con me tutti questi anni.

Ringrazio i miei genitori, per avermi sempre supportato in ogni mia decisione.

Contents

List of Figures

Chapter 1

Introduction

The goal of this thesis work is the development of a real-time application which has to work inside an automatic test environment. The application will run on a System-on-Chip (SoC) as a server, receiving data and instructions from a user workstation and managing the underlaying test logic.

Starting from a prototype version of the system, the application will be improved and migrated to the real-time domain using Xenomai, a free and open project which brings POSIX and traditional RTOS(Real-Time Operating System) capabilities for porting time-critical applications to Linux-based platforms.

The application will be used to control a set of fifteen relays mounted on a Printed Circuit Board(PCB). This testbench will be used to understand limits and capabilities of the system. The basic idea behind this work is to develop an environment and a set of guidelines, rules and methodologies to create a generic real-time framework which will be used, in the next future, to control and manage all the equipments typically involved in the power semiconductor test scope, such as measuring units and generators.

This thesis work has been developed at CREA Semiconductor Test Equipment, an Italian company with headquarters located at Ciriè (TO). The company, established in 1992, is today a worldwide leader in power semiconductor testers, with distributors in Europe, USA and Asia.

After this brief introduction, chapter 2 will introduce the project giving a general overview of the whole system, from the hardware and software point of view. Instructions will be given to correctly set up and configure all the components to recreate the starting working environment. Chapter 3 will focus on the system calls used to implement a special communication protocol used in the application.

After the description of the system, chapter 4 will introduce the concept of real-time and the related requirements. Chapter 5 will present the Xenomai framework, used to bring real-time capabilities into the system. Chapter 6 will show the use of Xenomai and the rules and guidelines followed in the program development. A Xenomai based real-time driver module will be presented in chapter 7. In the last chapter the performance of the system will be analyzed.

Chapter 2

System overview

The main subject of this thesis work is the FiTRelay application. This program runs on a micro-controller and plays the role of a server; it receives data and commands from a user workstation, manages them and interact with the test logic, implemented on a FPGA (Field Programmable Gate Array).

The application, written in C/C++, is loaded on a board from Acme Systems, the FoxG20 board. This micro-controller runs a pre-configured Linux distribution based on the 2.6.38 kernel core, made available by the same manufacturer. The user workstation and the board communicate via Ethernet cable, exchanging data and commands in XML format.

The application plays the received commands and interact with the underlaying test logic, implemented on a FPGA from the same vendor, the FoxVHDL board. The two boards communicates with the Foxbone protocol, an ad-hoc communication scheme specifically developed by Acme System for the couple. The Foxbone protocol is managed with a set of system calls added to the kernel.

In this first chapter all the hardware components of the system will be introduced and briefly described, and instructions will be given to correctly set up and configure all the software; the characteristics of the boards, the communication protocol and the relative system calls, the operating system, the application and some specific library, etc.

2.1 Hardware components

The main hardware parts of the system are the two boards from Acme Systems, the micro-controller, used as a server, and the FPGA, implementing the test logic. A brief description is reported below.

2.1.1 FoxG20 board

The Acme Systems FoxG20 board is a single board computer built around the 400MHz ARM9 Atmel CPU AT91SAM9G20. This board has been developed for solid-state web application servers and embedded devices enhanced with Internet connectivity and Linux flexibility. The main features of the board are:

- built around the NetusG20 core, based on a 400MHz ARM9 Atmel CPU AT91SAM9G20;
- 256KB of FLASH memory for the bootloader;
- Up to 16GB storage on bootable microSD card;
- two USB 2.0 ports (12 Mbits);
- Ethernet 10/100 port;
- debug serial port (to be used with DPI module);
- 2 serial ports;
- 5V DC power supply input;
- Real Time Clock with on-board backup battery;
- GPIO lines (3.3V);
- 4 A/D converter lines, I2C bus, SPI bus;
- Average power consumption: 80 mA @ 5V (0.4 Watt) without microSD, ethernet link, USB devices or other peripherals;
- Operative temperature range: 0° + 70°.

The board can run a pre-configured 2.6.38 kernel based Linux operating system freely available at https://github.com/tanzilli/foxg20-linux-2.6.38.

In this project the FoxG20 board is used as a server; it receives commands from the user workstation and manages the FPGA board, sending and receiving data and control signals.

2.1.2 FoxVHDL board

The FoxVHDL is a daughterboard for the FoxG20 board which is useful to evaluate the huge potentiality of the FPGA used as a hardware co-processor on a Linux system. This board is equipped with a ACTEL (today Microsemi) ProAsic 3 FPGA with 250K gates that can be programmed at run-time directly from the FoxG20

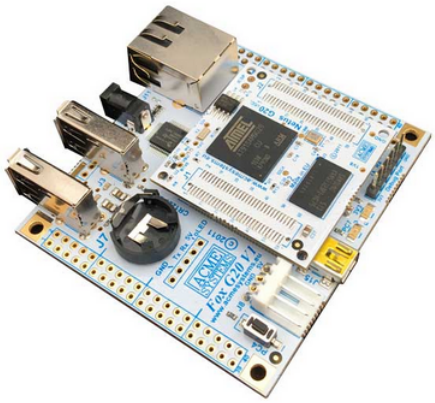

Figure 2.1. Acme Systems FoxG20 board

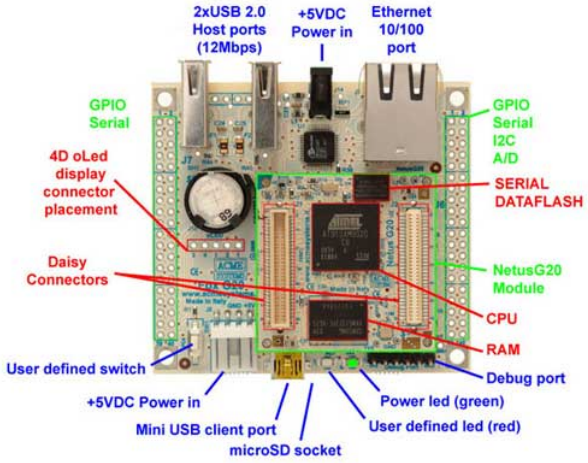

Figure 2.2. Acme Systems FoxG20 board detailed view

board without any additional hardware programming tool with user designed hardware circuits and peripherals.
The main features of this board are:

- based on the Actel A3P250 ProAsic3 FPGA with 250,000 gates in a FBGA package;
- two banks of 256K x 16 bits CMOS SRAM;
- D/A converter, can be connected to VGA;
- 40 pins header;

- same size of the FoxG20 board;
- easy to connect to FoxG20 board through JP1 and JP2 connector headers.

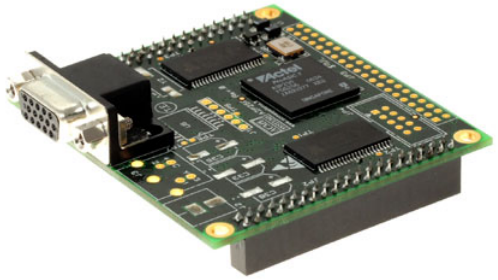

Figure 2.3. Acme Systems FoxVHDL board

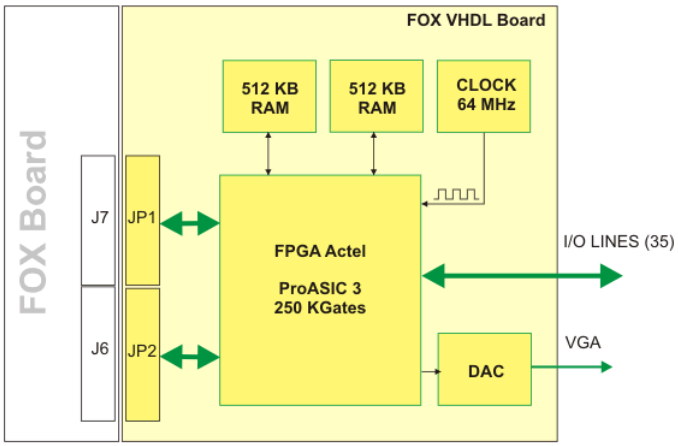

Figure 2.4. Acme Systems FoxVHDL board detailed view

In this project the FoxVHDL board is used to implement various peripherals and modules, such as memories, communication standard protocols (SPI, UART, etc), bus system, timers and general control logic. Every module has it's own control, data and status registers, which allows the FoxG20 board to manage and interact with it. The FPGA is then mounted onto a PCB on which reside all the target hardware components.

2.1.3 Relay PCB

The couple FoxG20 - FoxVHDL board can be used to drive every required target hardware. In this project they are used to drive a prototype board made of two

sets of fifteen relays. Two sets of LEDs are used to show the current state of each relay. Various control logic and power supply network complete the Printed Circuit Board. The whole system is shown in figure 2.5.

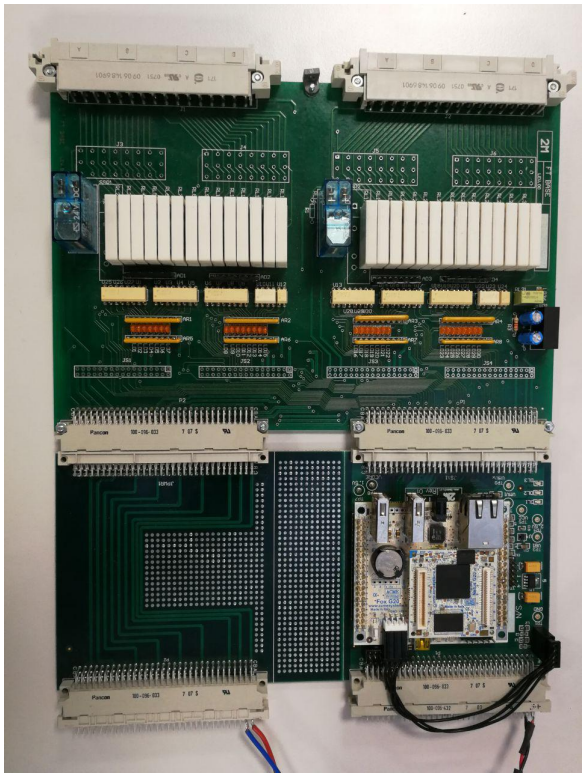

Figure 2.5. Relays prototype PCB.

2.2 SD-card

The Acme FoxG20 board boots from an SD-card. The board supports up to 16 GB SD-card. To install an operating system on it, we first need to create the partition table. The partition table lists all the sections in which the card has been divided. Each of these sections will host a different part of the system. Several softwares are available to perform this operation. In our case the free tool GParted has been used. The partition table has been configured as follows:

- 1 - 'kernel', Fat16 file-system, 32 MB are enough;
- 2 - 'rootfs', ext4 file-system, 800MB at least;
- 3 - 'data', ext4 file-system (optional);

- 4 - 'swap', linux-swap file-system, 128 MB are enough;

The 'kernel' partition will host the image of the kernel, 'rootfs' will host the file-system and 'swap' will be used for the memory swapping mechanism. The 'data' partition is optional an can be used for backup.

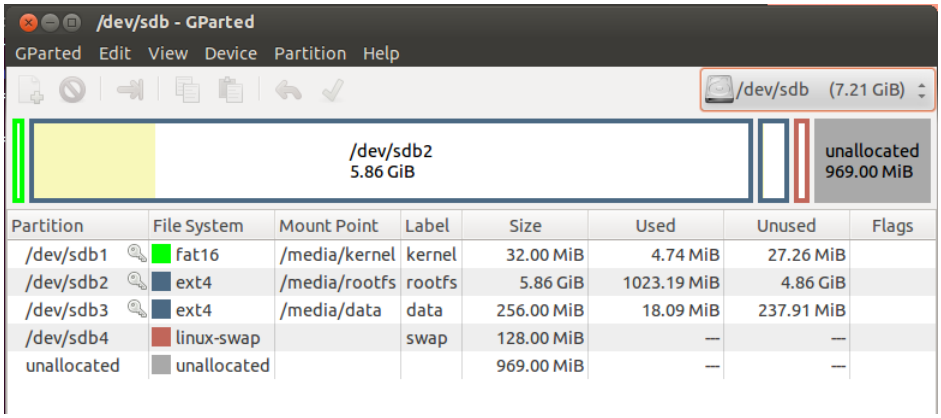

Figure 2.6. SD card partition table

2.3 Cross-compile toolchain

To compile kernel and applications for the FoxG20 board we need a suitable toolchain and a developing environment. A toolchain is the set of various tools (parser, compiler, linker, libraries, etc) needed to cross-compile an application for a specific architecture. The Linux based Ubuntu operating system has been chosen as a developing environment. In our case the target architecture is based on an ARM9 processor core; the GNU Arm toolchain is freely available and can be downloaded and installed directly from the Ubuntu repositories. In the following the list of installed packages.

```
#!/bin/bash
sudo apt-get install libc6-armel-cross libc6-dev-armel-cross
sudo apt-get install gcc-arm-linux-gnueabi g++-arm-linux-gnueabi
sudo apt-get install binutils-arm-linux-gnueabi
sudo apt-get install uboot-mkimage dpkg-cross
sudo apt-get install libncurses5-dev
```

2.4 Compiling and installing the kernel

Acme System makes available a Linux based Operating System (OS) pre-configured for the FoxG20 board. This OS is based on a Linux kernel 2.6.38, plus some added features to fully support the peripherals included on the board. It can be

downloaded from https://github.com/tanzilli/foxg20-linux-2.6.38.
It can be easily compiled running

```
> cd foxg20-linux-2.6.38
> make foxg20_defconfig
> make menuconfig
> make
```

The 'make foxg20-defconfig' will create the '.config' file, which lists all the configuration options and functionalities which have to be included into the kernel. If some default options have to be changed, the 'make menuconfig' command will show a sort of command line graphic interface which allows to configure the kernel in the desired way. The 'make' command will finally start the compilation. The generated kernel image will be in the 'uImage' file, which has to be copied into the kernel partition in the SD-card.

Acme System makes available also an archive, rootfs.tar.bz2, containing a basic file-system which can be downloaded from https://www.acmesystems.it/ and installed on the SD-card running

```
> sudo tar -xvjpSf rootfs.tar.bz2 -C /media/rootfs
```

The OS is now fully installed on the physical support and ready to work.

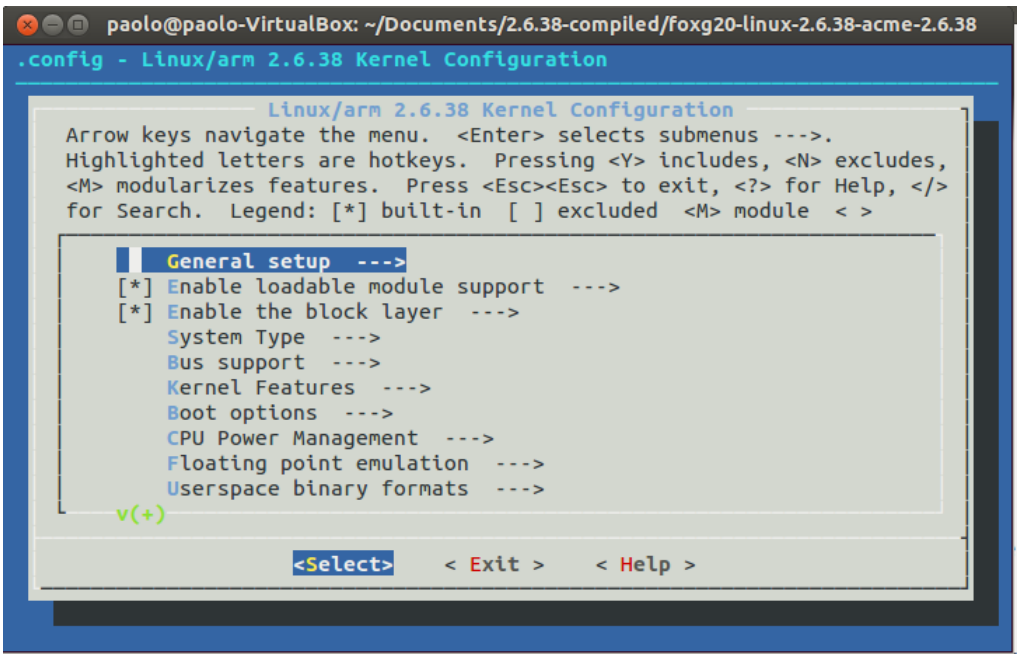

Figure 2.7. The command line interface which allows the configuration of the kernel functionalities.

2.5 Minicom

The Acme Systems made available a small DPI (Debug Port Interface) module to connect the serial output of the FoxG20 board to an USB input of the host computer for debugging purpose. To use this module we need a terminal emulator like Hyperterminal, PuTTY, Minicom, etc installed on the host PC. In our case the free Minicom has been used. It can be downloaded and installed from the Ubuntu repositories. Then it has to be configured to talk to the port on which the DPI module has been mapped onto the host computer (the mapping can be seen with the 'dmesg' command) running 'sudo minicom –setup'.

Figure 2.8. Debug Port Interface (DPI) module

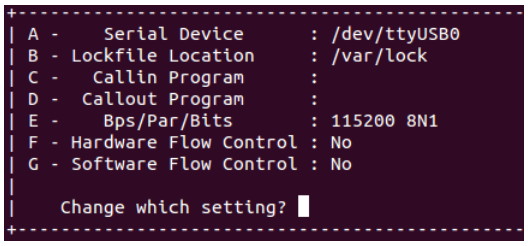

Figure 2.9. Minicom serial port configuration

2.6 LibExpat

LibExpat is a free library for the stream-oriented XML parsing written in C language. This project was originally developed by James Clark in 1997 and today is available both for Linux and Windows. In this project the XML language is used to exchange data and commands between the application and the user workstation through an Ethernet cable.

Expat has a lot of options and handlers which can be configured but, briefly speaking, it is based on four main functions:

- XML_ParserCreate : create a new parser object;
- XML_SetElementHandler : set handlers for start and end tags;
- XML_SetCharacterDataHandler : set handlers for text;
- XML_Parse : pass a buffer full of document to the parser.

An example of the usage of Expat is the 'outline' program, which simply prints on the screen the content of an XML document received as input.

outline.c

```
/************************************************************
 * outline.c
 *
 * Copyright 1999, Clark Cooper
 * All rights reserved.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the same terms as Perl.
 *
 * Read an XML document from standard input and print an element
 * outline on standard output.
 */
#include <stdio.h>
#include "xmlparse.h"
#define BUFFSIZE        8192
char Buff[BUFFSIZE];
int Depth;
void start(void *data, const char *el, const char **attr) {
  int i;
  for (i = 0; i < Depth; i++)
    printf("  ");
  printf("%s", el);
  for (i = 0; attr[i]; i += 2) {
    printf(" %s='%s'", attr[i], attr[i + 1]);
  }
  printf("\n");
  Depth++;
}  /* End of start handler */
void end(void *data, const char *el) {
  Depth--;
}  /* End of end handler */
void main(int argc, char **argv) {
  XML_Parser p = XML_ParserCreate(NULL);
  if (! p) {
    fprintf(stderr, "Couldn't allocate memory for parser\n");
    exit(-1);
  }
  XML_SetElementHandler(p, start, end);
  for (;;) {
    int done;
    int len;
    len = fread(Buff, 1, BUFFSIZE, stdin);
    if (ferror(stdin)) {
      fprintf(stderr, "Read error\n");
```

```
      exit(-1);
    }
    done = feof(stdin);
    if (! XML_Parse(p, Buff, len, done)) {
      fprintf(stderr, "Parse error at line %d:\n%s\n",
              XML_GetCurrentLineNumber(p),
              XML_ErrorString(XML_GetErrorCode(p)));
      exit(-1);
    }
    if (done)
      break;
  }
}  /* End of main */
```

LibExpat can be downloaded from https://github.com/libexpat/libexpat/releases and installed simply running

```
> tar -xvjf expat-2.2.6.tar.bz2
> cd expat-2.2.6/
> ./configure
> make
> make install
```

Now Expat is ready to be used. It can be included into a project as a standalone library with some simple flags added to the Makefile

```
CFLAGS += -I<expat install path>/include
LDFLAGS += -L<expat install path>/lib -lexpat
```

2.7 Test environment

The object of this thesis is the FiTRelay application. It works as a server, receiving communications from the user workstation and interacting with the underlaying FPGA board. The test sequence to be performed is specified by the user, which has the ability to configure every parameter of the test. This is possible thanks to a specifically developed test environment called FT Designer. This software is able to control and manage a fully configurable test sequence on a generic hardware module. The main interface is shown in figure 2.10.

FT Designer is an environment able to manage every phase of a generic test. It supports potentially every kind of target hardware component. Every target in FT Designer is implemented as a module. A module is, simply speaking, a logical entity described with a set of attributes; each attribute has a specific meaning and can be associated with a certain value. This structure, based on a set of configurable attributes, allows to describe every kind of peripheral to be tested.

The test steps can be fully configured in a similar way: every test step corresponds to a specific set of value for the attributes. Create a test sequence simply means specifying a value for each attribute of each resource for each step. Once

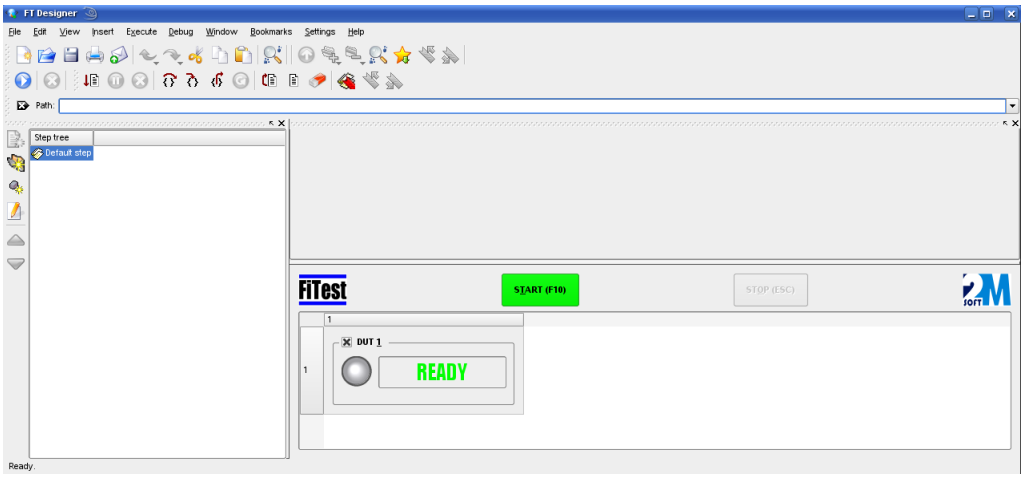

Figure 2.10. Graphic user interface of the FT Designer application.

defined, a step can be repeated many times and in parallel on many devices, in a fully automated way.

In this project a prototype board with two sets of 15 Relays has been used as a target. A possible example of test sequence is the following:

- turn on each Relay, one after the other, for 20ms;
- turn on all Relays, concurrently, for 20ms;
- turn on all Relays with different timing

Picture 2.11 shows the described example test sequence. On the left the test steps are shown in a folder tree like style. At the top each attribute of the resources (in this case Ton and Toff) can be configured modifying a sort of spreadsheet structure. Some details are shown in the next images.

2.8 Foxbone protocol

The Foxbone protocol was originally designed by Acme Systems to provide a flexible interface between the FoxG20 board and various custom embedded peripherals which can be implemented by the user in an FPGA, such as the FoxVHDL board. This protocol is based on a pretty simple architecture in which every data exchange between the board and a custom peripheral or module basically consists in a series of read/write operations from/to some addressable registers implemented on the FPGA side. In this way the board sees all the different modules as a simple group of data, control and status registers. This solution allows to have a very flexible and

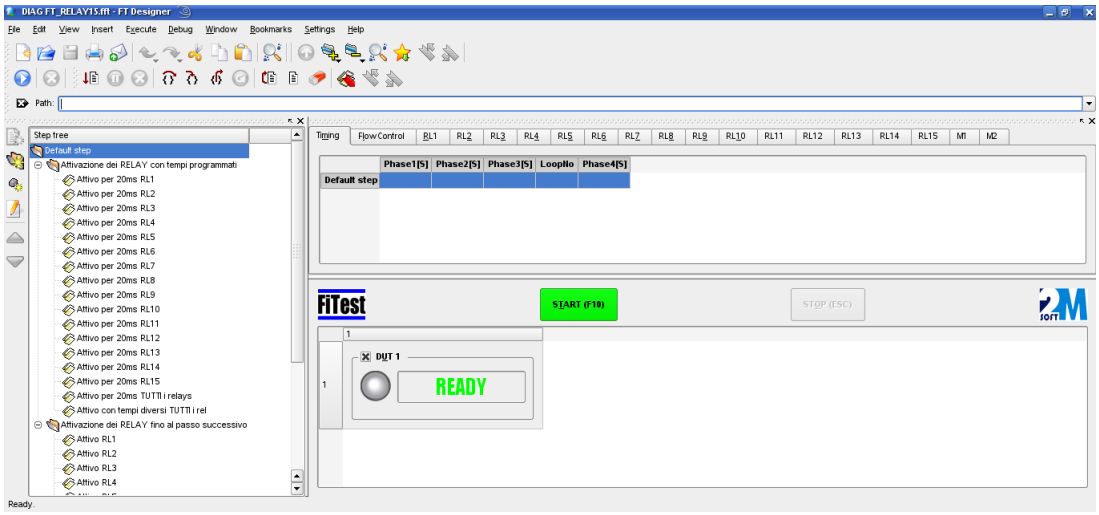

Figure 2.11. FT Designer running the example test sequence.

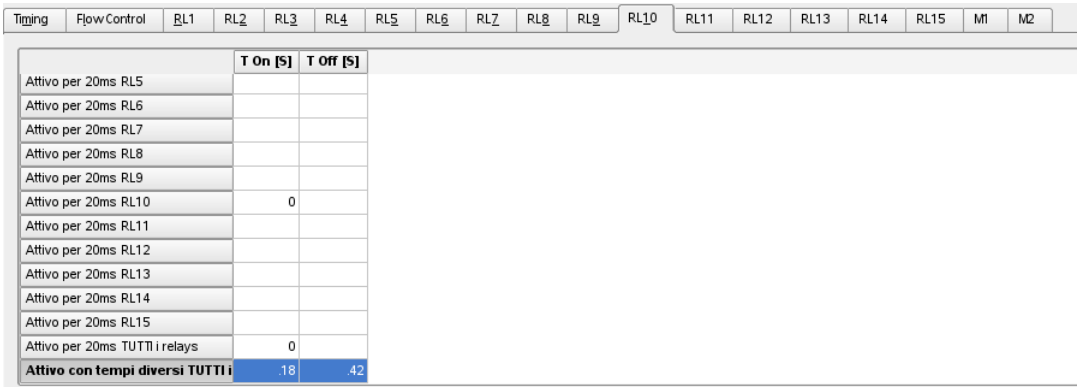

Figure 2.12. Detail of the spreadsheet structure of the module attributes.

general communication scheme, easy to manage and customize. An RTL (Register Transfer Level) description of a possible implementation of this protocol is depicted in figure 2.15. Data and addresses share a common 16 bits bus; when a read/write operation has to be performed, the address has to be provided first. When the address write signal goes active, the value present on the bus is stored into a dedicated address register. After, in case of a reading, the read control signal will go active and the read value will be provided on the bus. In case of a write, the data has to be sent on the bus, the write control signal will go active and the write operation will be performed at the desired location. In the following, a brief example of communication is reported (this example is extracted from the vendor website).

Figure 2.13. Detail of the folder-tree like test step sequence.

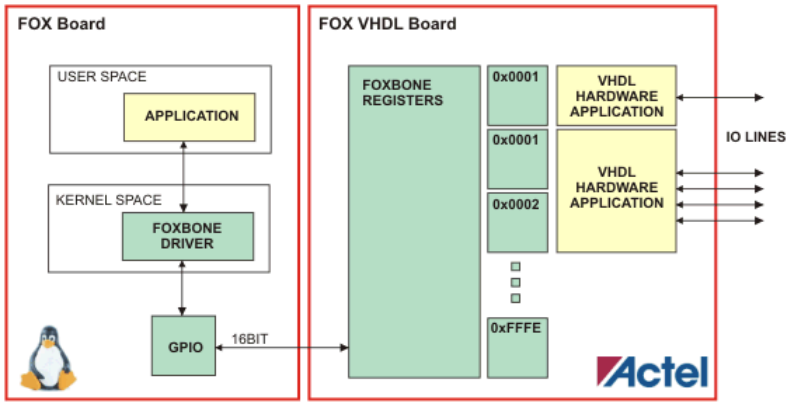

Figure 2.14. Communication with the Foxbone protocol

Writing a data word on the FPGA address 0x8000 and read it back:

- Set the address value 0x8000 on the 16 FOXBONE BUS address/data bits;
- Set high and then low the ADDRESS WRITE signal to store the address 0x8000 in the address register of the FPGA;
- Set the desired data to be stored in the 0x8000 register on the 16 FOXBONE BUS lines;

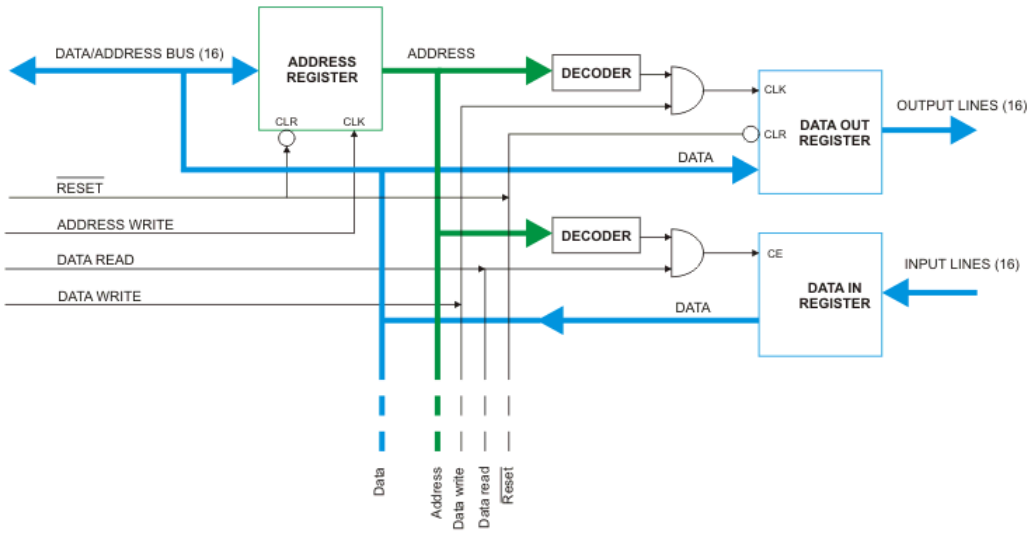

Figure 2.15. Hardware implementation of the Foxbone protocol

- Set high and then low the DATA WRITE signal to store the desired value in the FPGA 0x8000 register;
- Release the FOXBONE BUS to high impedance state (input state for the Fox) to prepare it for the reading phase from the FPGA;
- Rise the DATA READ signal to enable the FPGA output register selected to put the desired data on the FOXBONE BUS;
- Read the FOXBONE BUS containing now the FPGA 0x8000 register value;
- Lower the DATA READ signal to finish the reading phase and make the FPGA bus lines to return to a high impedance state.

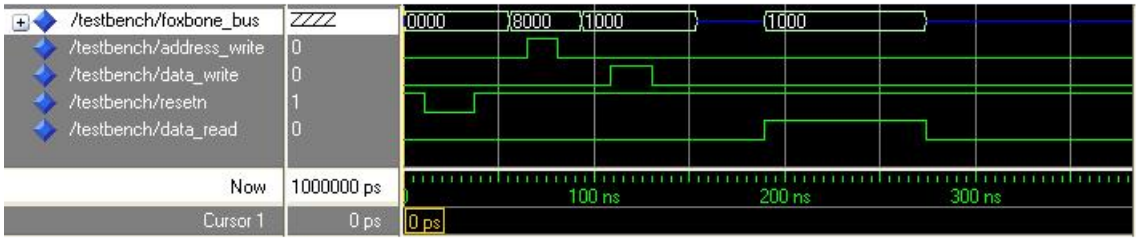

Figure 2.16. Example of communication with the Foxbone protocol

The Foxbone protocol in this project is managed through a set of system calls

added to the kernel. The meaning of system calls and the procedure to include them into the OS will be explained in the next chapter.

2.9 FiTRelay

The FiTRelay application is the core subject of this thesis work. It is written in C/C++ language and adopts a multi-thread approach. This program works as an interface between the user workstation(the FiTDesigner environment) and the test logic(the FPGA); it receives, through an Ethernet connection, the description of a generic test sequence in form of XML strings and has to decode it, checking for eventual errors and for the validity of the content of the strings. After the decoding phase, the application has to manage the test operations configuring, at each step, the underlaying circuitry and peripherals implemented into the FPGA to obtain the desired behavior.

More in detail, every time a new connection from the user-workstation occurs, a new thread is created to manage it; each of these threads will analyze the received XML strings thanks to the functionalities made available with the Expat libraries described before, performing at the same time an error check on the validity of the received strings. The next step depends of course on the received commands and data but, in general, it will result in the configuration of some parameters of a module. Every module implemented in the FPGA is accessed and controlled through a set of data, control and status register; these registers will be configured accordingly to the received commands. The communication with the FPGA is implemented through the Foxbone protocol described before. The configuration parameters are not sent singularly but in a single block; in other word all the parameters and data of all the modules are grouped in a structure, a sort of memory image, and sent all together to the FPGA in a single transaction.

The most important thread of the program is the FPGA handler; this thread is responsible for the communication with the programmable logic ad is in charge of managing at each step the memory image containing all the configuration parameters of the modules. Moreover, it is responsible for the timing of the application; reading the value of a timer from the FPGA it is able to check that all the operations are performed at the right time. The synchronization between all the threads is guaranteed by a set of mutexes and conditional variables.

This prototype program will be modified and ported to the real-time domain thanks to Xenomai; all the modifications be will described in chapter 6.

Chapter 3

System calls

Applications run generally in the so called user-space domain; at this level programs can access user-space libraries and services but don't have complete and direct access to the hardware or other services which require privileged permissions. This is made mostly for safety reason, to avoid operations which can damage the application itself or the whole system. Sometimes, however, a more accurate and precise control of the hardware is required, or the access to some critical service; in this case a switch to the so called kernel space is required. In this domain the user is guaranteed the highest permissions level and is allowed to perform direct operations on the hardware and all the operations which was previously denied.

A system call is a special mechanism which allows a user-space application to call some operating system kernel level service. Every time a system call is executed, a software interrupt (or trap instruction) occurs and the processor mode is switched from user to kernel mode. All the required operations can now be performed with privileged permissions. At the end of the execution, the processor operating mode is restored and the application keeps on his normal execution at the user level.

System calls are used for example for creating and managing processes(fork(), exec(), wait(), exit(), kill(), etc), for managing files and devices(open(), close(), read(), write(), etc), for communication between processes(message queues, pipe, shared memory, etc) and in general for getting various kind of information from the system(actual time, processes info, files and devices attributes, etc).

The full set of system calls available on a system is listed in the so called system call table; each row of this table represents a system call and contains the address of the routine to be executed when the corresponding system call is invoked. As an example, the standard set of system calls of a Unix-like operating system can be called thanks to the 'glibc' library.

In our application, custom system calls will be used to drive the generic I/O

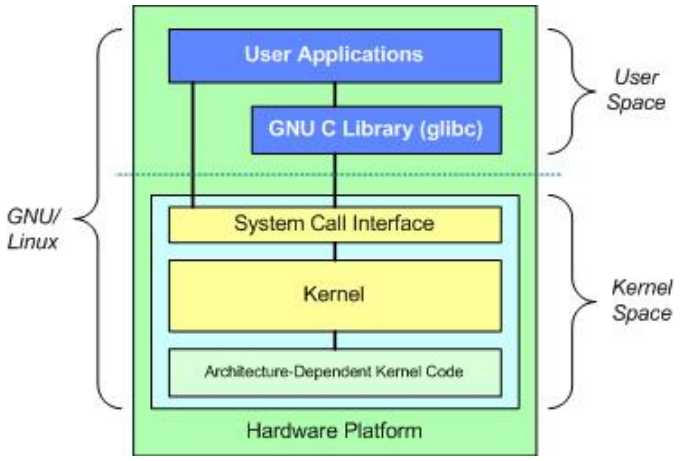

Figure 3.1. User-space applications can request kernel services through system calls interface.

pins implementing the physical layer of the Foxbone communication protocol.

Adding one or more system calls to a kernel requires to modify and recompile it; the new functions have to be added to the system calls table modifying some sources files. The procedure is pretty simple but there are some points that have to be followed carefully.

First of all, the C code implementing the required services has to be written. The prototype of a system call is pretty similar to a standard function prototype, as example 'asmlinkage long sys_foxboneread(unsigned long int reg);'. The 'asmlinkage' keyword is typically used in all Linux system calls: it means that the function will receive eventual input parameters only through the stack, not in registers.

In our case the sources were provided in three files, one for each system call: 'foxboneinit.c', 'foxboneread.c', 'foxbonewrite.c'. As an example the 'init' function code is listed below (the full code is reported in the appendix). This function initializes all the pins required for the communication.

foxbone_init.c

```
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/input.h>
#include <linux/gpio.h>
#include <mach/at91_pio.h>
#include <linux/slab.h>
#include<linux/linkage.h>
```

```
asmlinkage long sys_foxbone_init(void)
{
        void __iomem * iom;
        unsigned int mask;
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        __raw_writel(mask, iom + PIO_OWER);
        mask=0x3bc0c3f0;
        __raw_writel(mask, iom + PIO_OWDR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOA;
        mask=0x800002c0;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        mask=0x00000200;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        __raw_writel(mask, iom + PIO_SODR);
        mask=0x80000000;
        __raw_writel(mask, iom + PIO_CODR);
        mask=0x00000080;
        __raw_writel(mask, iom + PIO_CODR);
        mask=0x00000040;
        __raw_writel(mask, iom + PIO_CODR);
        return(0);
}
```

As it can be seen, this function initializes and sets all the pins and control status register required by the Foxbone protocol. All the pins and registers are accessed through an address, computed as a base address (AT91_VA_ BASE_SYS) plus a specific offset which identifies each different register (AT91_PIOA, AT91_PIOB, PIO_IDR,PIO_PDUR, PIO_OER, etc).

The sources files have been stored in the 'foxbone' folder, placed into the kernel source tree directory. The very simple rules to compile those files are listed in the following 'Makefile':

```
obj-y += foxbone_init.o
obj-y += foxbonewrite.o
obj-y += foxboneread.o
```

The 'foxbone' folder has to be included into the compilation path modifying the 'core-y' line in the top level Makefile: 'core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ foxbone/'.

The next step requires to modify the 'arch/arm/kernel/calls.S' file. This file contains the system calls table for our specific architecture. The following lines have to be added at the end of it:

```
CALL(sys_foxbone_init)
CALL(sys_foxbonewrite)
CALL(sys_foxboneread)
```

It is very important to take note of the number of the system calls, indicating their position inside the table. In this case the positions are 370, 371 and 372. These numbers will be used later.

The next file to modify is 'arch/arm/include/asm/unistd.h'. This header file contains the definition of all the standard POSIX API available on the system, such as fork(), open(), read(),exec(), etc. This file associates each system call with it's address in the system calls table. The following lines has to be added to the function list:

```
#define __NR_foxbone_init        (__NR_SYSCALL_BASE + 370)
#define __NR_foxbonewrite        (__NR_SYSCALL_BASE + 371)
#define __NR_foxboneread         (__NR_SYSCALL_BASE + 372)
```

Finally, the new functions have to be declared in the 'include/linux/syscalls.h' file to make them usable by the system:

```
asmlinkage long foxbone_init(void);
asmlinkage long foxbonewrite(unsigned long int reg,
unsigned long int value);
asmlinkage long foxboneread(unsigned long int reg);
```

Now it is possible to recompile the kernel running 'make' in the source tree directory.

As a final step the following header file have been introduced; it defines a more user friendly alias name for each function. This file of course is not mandatory, but is useful to ease the use of the system calls and improve the readability of the code.

foxbone_syscalls.h

```
#include<linux/unistd.h>
#define __NR_foxbone_init 370
#define __NR_foxbonewrite 371
#define __NR_foxboneread 372
long foxboneread(unsigned long int reg)
{
        return syscall(__NR_foxboneread,reg);
};
long foxbonewrite(unsigned long int reg, unsigned long int value)
{
        return syscall(__NR_foxbonewrite,reg,value);
```

```
};
long foxbone_init(void)
{
        return syscall(__NR_foxbone_init);
};
```

Chapter 4

Real-time systems

4.1 Introduction

A standard definition of a real-time system is the following: "A real-time system is one in which the correctness of the computation not only depends upon the logical correctness of the result but also upon the time at which the result is produced. If the timing constraints are not met, system failure is said to have occurred". In other words a real-time system is one in which the logical correctness of the results is not enough, the results also have to be provided at the right time.

A common mistake is to think of the word 'real-time' as a synonymous of 'fast'. Real-time does not means fast, otherwise a real-time system will simply require a better and more performing hardware instead of special design techniques. The concept of real-time is related to determinism and predictability; the system has to work in a timely controlled way, it's behavior has to be known in advance under every circumstances in the field of application. In other world a real-time system must guarantees that some deadlines, imposed by the environment in which it works, are respected. These concept can be better understood considering the following examples.

Let's consider for example the airbag control system of a car. The system continuously monitors some parameters of the car coming from various sensors such as speed, acceleration, pressure, inertial sensors, etc. When the system detects a collision, identified by a strong variation of a combination of those parameters, over a certain threshold, the airbag has to be opened as soon as possible. The control system in this case has not to perform a complex algorithm or expensive computations, as in the case of multimedia application for example. The crucial point here is the reaction time, defined as the time interval between the occurrence of an event or a condition and the system response to that event. As soon as a collision is detected, as soon as a response has to be produced, independently from the actual status of

the system. In this specific case, missing the deadline will result in a drastic failure.

Another possible example of a real-time system can be a monitoring system which acquires a set of samples from various sensors and send them to a central station with some communication protocol. These samples may have a certain periodicity. In this case the key point is not the reaction time but the determinism and the predictability. The system must guarantee a certain regularity; it has to be able to continuously schedule the sampling operation at regular intervals, also if other actions are being performed in parallel (the communication with the base station). If a deadline miss occurs, one or more samples will be lost. Depending on the application the result can be a drastic failure or simply a degradation of the performances.

As anticipated in the previous examples, a real-time system can be classified in three major categories depending on the consequences of a failure:

- Hard real-time: a system is said to be hard real-time if a failure will result in drastic, irreversible and catastrophic consequences, with possible damages to things and injuries for people. This is the case for example of the airbag control system mentioned above,a semaphore controller, medical and industrial robots, airplane and aerospace applications, power plant security systems, etc;
- Soft real-time: a system is said to be soft real-time if a failure will result in a degradation of the performances but not in a critical interruption of the functionality of the system. This is the case of the monitoring system mentioned above; missing some samples will simply degrade the performances and the result of the computation, and in many cases this can be considered as an acceptable error (for example in multimedia applications or in general in all that kind of applications in which missing samples can be replaced and restored with special technique such as interpolation). Of course all depends on the applications; if the monitoring system is responsible for the safety control of a nuclear power plant, this behavior will not be acceptable;
- Firm real-time: this is an intermediate category used for those hard real-time system which however can tolerate a certain percentage of deadline misses without becoming unsafe.

4.2 Is real-time really needed ?

As previously said real-time has not to be associated to the word faster; if more speed is required, a better and more performing hardware can solve the problem without the need of real-time capabilities, which will increase the complexity with no reason.

In the following a list of common situations in which real-time capabilities are really needed:

- There are deadlines to be respected. These deadlines are typically imposed by the physical environment in which the system operates. Missing one ore more deadlines will result in a failure; producing the correct result is not enough, it has to be delivered also at the right time. Depending on the consequences of a failure the system will be classified as hard, soft or firm real-time;
- One or more peripherals the system is connected to impose some kind of time boundary for data exchange and communication. In this case the deadlines are not imposed by the physical environment but by a part of the system itself;
- The application has to run with a higher priority than the system tasks. On a system, apart from user applications, there is commonly a set of running tasks needed to guarantee the functionalities of the system itself; if a user program requires a higher priority, a special mechanism is needed which allow the application to win the competition with the system tasks and gain the control of the CPU;
- The system requires to express or measure fine-grained delays. A system may work with delays in the range of millisecond for example, but in some cases a fine granularity is required; in the latter real-time capabilities are needed to ensure a correct behavior.

4.3 Latency

One of the most important keyword regarding real-time system is latency. Latency is defined, in general, as the time between the occurrence of an event/condition and the reaction of the system to that event. More in detail we can analyze two main kind of latency, called interrupt and dispatch latency.

The interrupt latency is defined as the time between the occurrence of an interrupt and the call to the corresponding ISR (Interrupt Service Routine). The interrupt management may be delayed for many reason: cache misses, bus contention, DMA (Direct Memory Access), interrupt masking, etc. If the response time to some interrupts is crucial for the application, all these factors have to be taken into account.

The dispatch or schedule latency is the time required by the system to allocate a new task on the CPU. When switching context between two process or when returning from an interrupt, the system may take some time to complete some operations before allocating a new process to the CPU.

In both cases, when speaking about latency, commonly the worst-case latency is considered, defined as the latency value corresponding to a combination of the worst case of all the factors which can affect it. In a real-time system, in which the response time to an event is a crucial point, the latency must be guaranteed to be predictable in every working condition inside the application field.

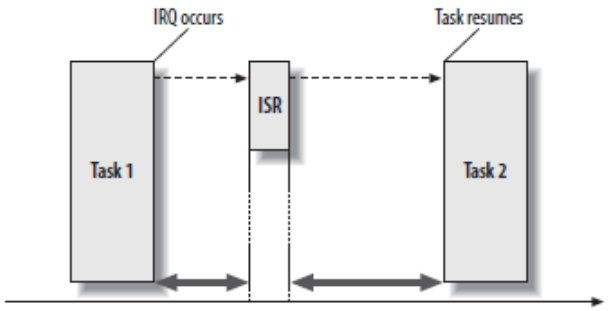

Figure 4.1. Graphical representation of interrupt latency and dispatch latency.

4.4 Kernel requirements

After introducing the concept of real time we can now analyze which are the characteristics and requirements which an operating system must satisfy to implement a real-time environment:

- As we have seen on a system may reside both real-time and non real-time processes. When an event occurs, or when a real-time task with higher priority becomes schedulable, it has to be served as soon as possible, switching out the actual running process. This procedure is called preemption and the time required to perform it is said preemption granularity. A real-time system must be able to always guarantee that a higher priority task could preempt a lower priority one, and this has to be done as quickly as possible. The system has to guarantee a fine preemption granularity;

- The dispatch latency must be bounded and predictable . There are cases in which the schedule of a higher priority task is delayed indefinitely due to a problem called priority inversion. Suppose that two task, A (high priority) and B(low priority), share some resource. If this resource is actually hold by the low priority task B, A, which is waiting for it, cannot preempt B until B releases the resource. In this way A is forced to wait for the completion of a lower priority task. In a real-time system this situation should be avoided;

- Interrupt latency must be bounded and predictable. Reaction time to external event is fundamental in a real-time system and so the interrupt latency must be known and bounded, within all the possible working load condition. This is important not only for external events but also for internal ones, such as timed interrupt used to implement periodic tasks.

Two main paths have been followed to introduces real-time functionalities into the Linux kernel. The first one is based on the so called co-kernel approach. Since we can not rely on the standard kernel for real-time purpose, the idea is to introduce a second kernel, running in parallel with the standard one, which is responsible for all the real-time related task and stuff. This is the strategy adopted by Xenomai, which is used in this project and which will be explained in detail in the next chapter.

A second approach consist in modifying the Linux kernel and transform it into a real-time one, using ad hoc real-time scheduling policies. The Linux kernel is pretty large, but not all of it has to be modified, only those part which can affect the scheduling and the timing aspect of the system. This is the approach followed by the RT_PREEMPT solution.

Before moving to the description of the Xenomai framework, we can look at two very simple example of real-time scheduling policies, just for reference.

4.5 Real-time scheduling policies

As already explained real time system requires special scheduling properties. A necessary (but not sufficient) condition is to have a preemptive and priority based policy; every task is associated with a integer value depending on it's importance(priority based) and a higher priority task can gain the control of the CPU over a lower priority task (preemptive).

In the following examples we will consider a periodic task, i.e. a task which requires the control of the CPU at regular time interval. First we have to define some parameter which are useful to model the problem:

- p : the periodicity of the task;
- t : processing time, the time required by the task to complete;
- d : deadline, the time limit before which the task has to complete;
- r : rate, defined as $1/p$;
- u : utilization, defined as t/p.

Of course it has to be ensured that

$$0 \leq t \leq d \leq p \tag{4.1}$$

Figure 4.2 shows an example of a periodic task, described with the introduced parameters.

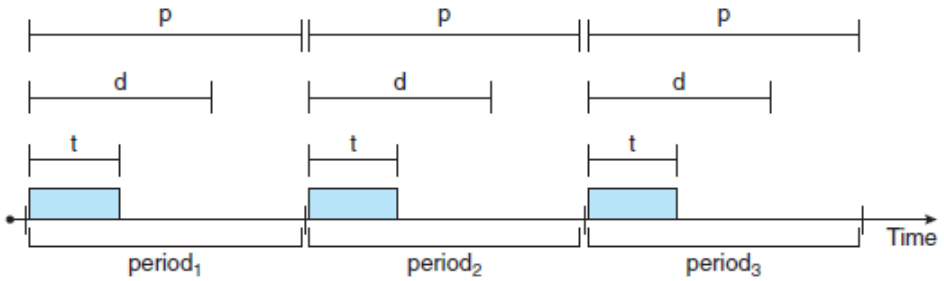

Figure 4.2. Mathematical model of a periodic process.

In general all real-time scheduling policy can be classified in two main categories:

- Static priority: the priorities of all the tasks are assigned statically, before the execution of the tasks. This can lead to the best performance but of course requires to know in advance all the parameters and characteristics of the processes;
- Dynamic priority: priorities of all the tasks are assigned at runtime, during the execution, depending on the system status and on the characteristics of incoming processes. This cannot lead to the best performances but allows more flexibility.

It must be taken into account that not always a schedule is feasible. There are cases in which a set of tasks is simply not schedulable under any policy; sometime is mathematically impossible for a set of tasks to met all their deadlines. This property goes under the name of schedulability, and has to be checked case by case depending on the chosen scheduling algorithm.

4.5.1 Rate-monotonic (RM) scheduling policy

The rate-monotonic (RM) algorithm schedules periodic tasks using a static priority policy with preemption. When entering the system, each task is assigned a priority inversely based on it's period. The shorter the period, the higher the priority; the longer the period, the lower the priority. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority

process. The idea behind this algorithm is to assign a higher priority to those processor which, having shorter period, will request the control of the CPU more frequently. For simplicity it is assumed that each task has a constant processing time, i.e. it requires the same time to complete for every CPU burst.

Suppose to have two tasks, P_1 and P_2, with the following characteristics: $p_1 = 50$, $p_2 = 100$, $t_1 = 20$, $t_2 = 35$. The deadline for each process requires that it complete its CPU burst by the start of its next period.

According to the Rate-monotonic algorithm, P_1 will be assigned a higher priority having a smaller period. The evolution in time is shown in figure 4.3. P_1 starts first and complete it's execution at time 20, thus respecting it's deadline. Then P_2 starts and runs until time 50, when it is preempted by P_1, which has a higher priority and requires the CPU due to it's periodicity. P_1 runs until time 70 and respects again it's deadline. Finally P_2 complete it's execution for the remaining 5 time unit, respecting it's deadline. Then this process iterates starting again at time unit 100.

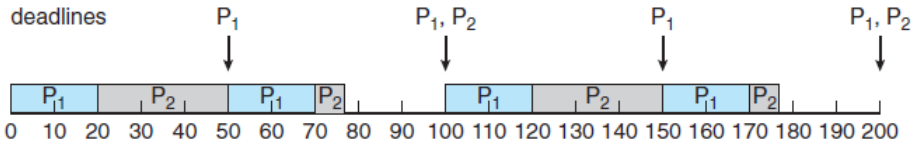

Figure 4.3. Scheduling of P_1 and P_2 according to the Rate-monotonic algorithm.

Rate-monotonic represents the optimal static priority policy, guaranteeing the best performance. It can be shown that if a set of process is not schedulable under the RM, it is not schedulable under any other static priority algorithm. It can be also demonstrated that a set of N tasks is schedulable under the RM if

$$\sum_{i=1}^{N} \frac{t_i}{p_i} \leq N(2^{\frac{1}{N}} - 1) \tag{4.2}$$

The main drawbacks of this algorithm is the CPU bound; this policy in fact is not able to maximize the CPU usage, which is always limited to

$$N(2^{\frac{1}{N}} - 1) \tag{4.3}$$

When N goes to infinite, the CPU usage is bounded to around 69%.

4.5.2 Earliest Deadline First (EDF) scheduling policy

The Earliest Deadline First (EDF) is a dynamic priority scheduling algorithm. It assigns priorities dynamically according to deadlines. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. When a new process becomes ready to run, it must inform the system about it's deadline requirements; according to EDF, the scheduler will reassign all the priorities according to the characteristics of the new set of runnable tasks.

Suppose to have two tasks, P_1 and P_2, with the following characteristics: $p_1 = 50$, $p_2 = 80$, $t_1 = 25$, $t_2 = 35$. The deadline for each process requires that it complete its CPU burst by the start of its next period.

According to the Rate-monotonic algorithm, P_1 will be assigned a higher priority having an earliest deadline. The evolution in time is shown in figure 4.4. P_1 starts first and complete it's execution at time 25. Then P_2 runs until 60, respecting again it's deadline. At time 50 P_2 is not preempted by P_1 cause now it's deadline is early, and so it has a higher priority then P_1. At 60 P_1 runs again until 85, meeting it's deadline. P_2 then runs until 100, when it is preempted by P_1, which has now an earlier deadline (150 against 160). P_2 will complete it's second burst at time 145.

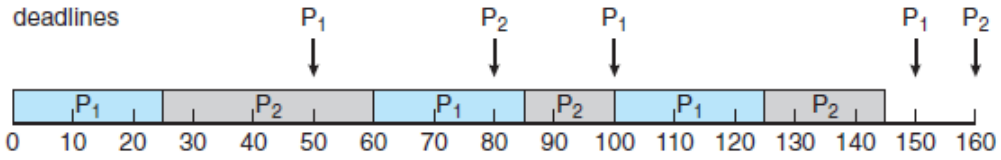

Figure 4.4. Scheduling of P_1 and P_2 according to the Earliest Deadline First algorithm.

Earliest Deadline First represents the optimal dynamic priority policy, guaranteeing the best performance. EDF does not require tasks to be periodic to work and, in theory, it is not CPU-usage bounded. It can be demonstrated that a set of N tasks is schedulable under the EDF if

$$\sum_{i=1}^{N} \frac{t_i}{p_i} \leq 1 \tag{4.4}$$

4.5.3 POSIX Real-time scheduling policies

POSIX standard defines and makes available two real-time scheduling policies, SCHED_$FIFO$ and SCHED_RR. SCHED_$FIFO$ schedules tasks according to

a first-come, first-served policy using a FIFO queue. However, there is no time slicing among threads of equal priority. This means that the highest-priority thread in front of the FIFO queue will be granted the CPU until it terminates or blocks. SCHED_RR uses a Round-Robin policy; it is similar to SCHED_$FIFO$ except that it provides time slicing among tasks of equal priority.

These two policies however are not able to guarantee the respect of the deadlines, they simply try to do it, without any predictability or certainty. In other world, these two policies are suitable only for soft real-time.

Chapter 5

Xenomai

Xenomai is a development framework that brings POSIX ad traditional RTOS API to Linux-based platform. If the Linux kernel is not able to meet the response time requirements of an application, Xenomai supports it with a real-time infrastructure that reschedule all the time critical activities independently from the main Linux scheduler.

Xenomai is a subsystem, strictly integrated with the kernel, which is able to guarantee predictable response time to user applications. It is implemented as a co-kernel, a second kernel core which run in parallel with the first, on the same hardware. Real-time tasks are managed exclusively by the Xenomai core, which is the only responsible for it.

Xenomai was first introduced in 2001 to ease the process of migration from traditional RTOS (Real-Time Operating System) to Linux based platform. Linux is not natively hard real-time and is not able to guarantee predictable response time; on the other hand, traditional small RTOS in some cases may lack some of the functionalities which Linux and the POSIX standard can offer. Xenomai try to solve both problems, supporting the standard Linux with a real-time framework running syde-by-syde to it. Xenomai provides different building blocks, called *skin*, each one able to mimic traditional RTOS functionalities in a Linux environment. Xenomai is offered under the GPL license.

5.1 Xenomai architecture

Xenomai provides real-time functionalities with a set of API emulators, which mimic exactly the behavior of the most typical real-time system calls introduced by RTOS vendor, guaranteeing in such a way compatibility with them. As previously said Xenomai is made of various blocks, called skins, each one porting some

functionalities for a specific target.

Skins act like an interface between user-space applications and the Xenomai core, called Nucleus. Every time a user-space applications requires a real-time service, it makes a call to a Xenomai skin; this call is managed by the skin, which behaves like a system call interface, and is correctly transmitted to the lower layers of the system. Xenomai manages only real-time related stuff. Non real-time services are managed with usual Linux system call interface; in this way the Xenomai architecture is transparent to traditional tasks.

A general overview of the Xenomai framework is depicted in figure 5.1. The picture shows the various skins placed between the user-space level and the Xenomai core. The graph also shows another important component of Xenomai, the I-pipe. This block, also called Adeos pipeline, is the fundamental module on which all the framework is built and will described in detail in the next section.

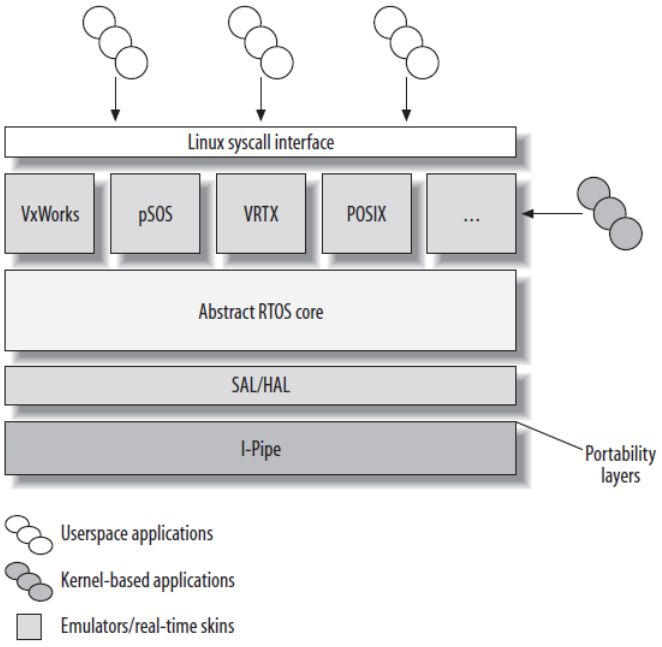

Figure 5.1. General overview of the Xenomai architecture.

5.2 Interrupt pipeline (I-pipe)

The interrupt pipeline I-pipe is the fundamental building block of the Xenomai framework. As explained Xenomai is a co-kernel running in parallel with the standard Linux kernel. Only real-time tasks will be managed by the Nucleus core while non real-time ones will be managed by the standard kernel. We need a way to separate this two environments.

Xenomai is organized in the so called *domains*. A domain can be seen as an independent module which typically contains a whole kernel. On a system more than one domain can exist. Each domain has a priority with respect to all the others, a sort of order of importance. In a Xenomai based system there are at least two domains. The first is the one containing the co-kernel introduced by Xenomai and has the higher priority; this is also called primary domain. The second is the one containing the standard Linux kernel, and has a lower priority; this is usually called the secondary domain or root domain. The two domains share the same address space, so they can access the same portion of memory and work on the same data; the two are in fact separated entities but have to cooperate. The primary domain is aware of the secondary, but the secondary is usually unaware of the primary; in other word the Linux kernel does not see the Xenomai co-kernel. A graphical representation of such architecture is depicted in figure 5.2.

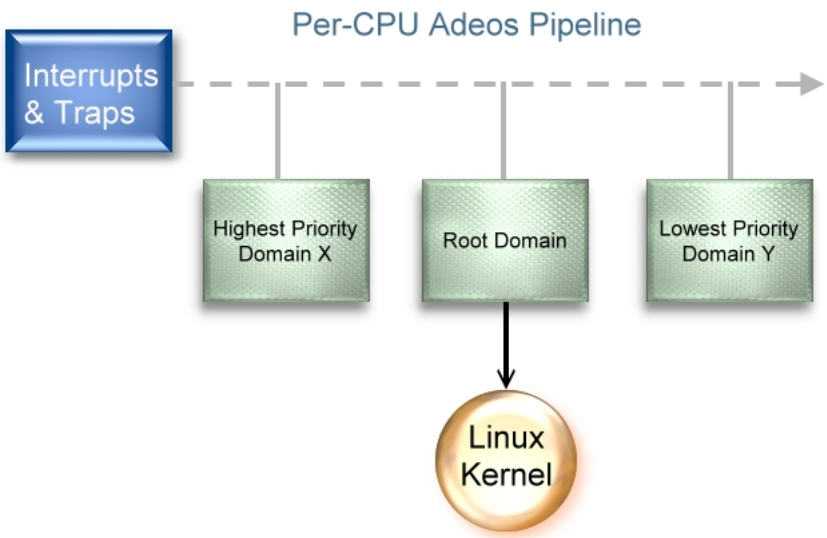

Figure 5.2. Domains view of a Xenomai based environment.

The two (or more) domains are connected through the Interrupt Pipeline. The

I-pipe, also known as Adeos pipeline, was first introduced in 2001 by Karim Yaghmour. It is essentially a resource virtualization layer, which enables the coexistence of multiple domains on the same system. As said before, domains, which are typically full kernel modules, does not necessarily see themselves, but all see the pipeline. The problem of having more kernels on the same machine is that they may compete for different resources and events control such as

- Incoming external events/interrupts, or auto-generated ones(es timed interrupts);
- System calls invoked by user applications;
- Every other kind of event triggered by a kernel, such as signals notification, memory managements problems, process/thread creation/exiting, task switching, etc.

The Adeos pipeline acts as a sort of arbiter, collecting all this kind of events, and reassigning them to the correct domain taking into account the priority of each domain. In other words it has the power to choose which domain has to manage each events. In this way it is able to deliver fastly every events to the target domains in an ordered manner, guaranteeing predictability of the delivery.

Every domain has a priority which is, simply speaking, an integer number statically assigned. The domains are ordered along the pipeline depending on their priority, from the highest to the lowest. When an events arrives, it is moved along the pipeline from the head to the tail until the target domains is reached, and the events is delivered to it to be correctly managed.

A potential risk for this structure is the possibility of a stall. A stall may happen if a domain decide to temporarily disable the reception of interrupts. There are many reason for which a kernel may decide to mask the interrupts, the main one is that it is executing a critical section. If a domain is in a stall condition, the whole pipeline will be stalled, preventing all others domain from receiving events/interrupt. In a real-time environment this has to be absolutely avoided. To solve this problem Xenomai adopt a strategy called optimistic interrupt protection, a technique which was first introduced in a paper by Stodolsky, Chen and Bershad. It is based on the introduction, for each domain, of a sort of interrupt logfile; this file is able to collect all the incoming events. If a domain decide to mask the interrupts, it's logfile will accumulate all the incoming events targeted to that domain. In this way incoming events does not get lost. Events targeted to other domains however are still allowed to move along the pipeline without stalling. In this way even if a domain is stalled, the rest of the pipeline will keep on working correctly. If a real-time event arrives, it will be delivered to the Xenomai co-kernel also if the

secondary domain is stalled. Picture 5.3 shows this architecture; each domain is associated with a logfile (actually one for each CPU) able to accumulate events in case of a stall.

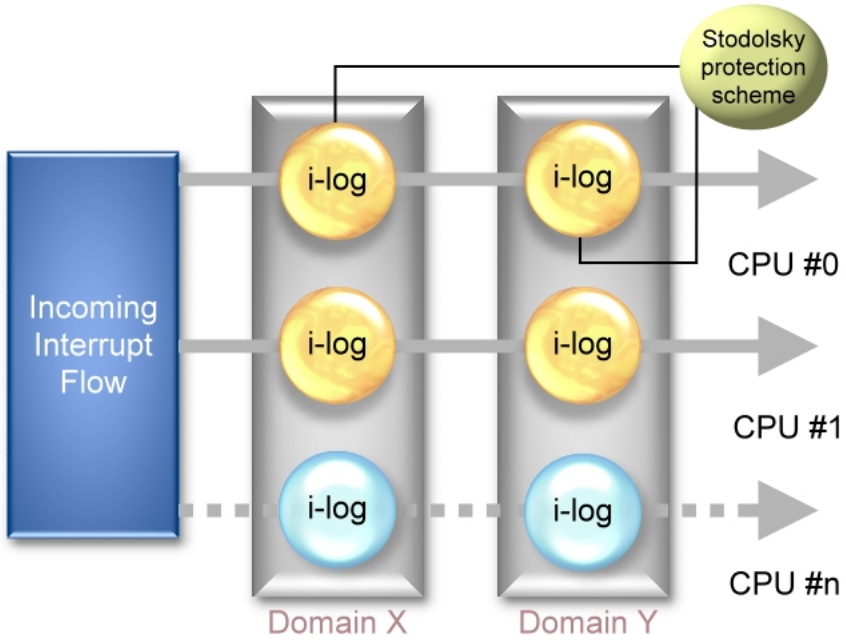

Figure 5.3. Optimistic interrupt protection scheme applied to the Xenomai architecture.

Based on this structure, Xenomai is able to catch and manage every events before the main kernel and independently from it's will, guaranteeing small and predictable interrupts latency, in the order of the microsecond.

The I-pipe is strongly hardware dependent. It is placed below the Hardware Abstraction Layer (HAL), directly in contact with the physical layer. The major number of request to the I-pipe will typically come from the HAL. Changing architecture means of course modifying the HAL, but the skin modules, which are built upon it, will suffer smaller modifications.

5.3 Primary and secondary domain

As previously explained the organization in domains allows to separate real-time from non real-time tasks. RT process will be managed exclusively by the Xenomai co-kernel, while non RT will be left to the standard Linux kernel. All RT thread are known by the RT scheduler. RT tasks however can be moved to the secondary domain while remaining managed by Xenomai. It may happen that a real-time

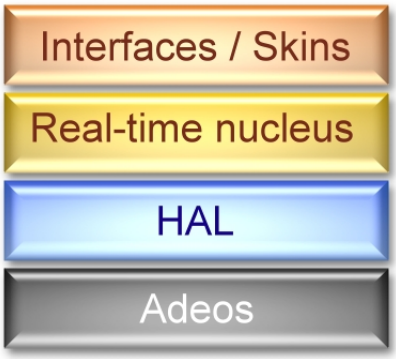

Figure 5.4. Layers view of the Xenomai architecture.

tasks requires a non real-time service; in this case it is switched to the secondary domain ad it will stay there as long as it uses non RT services. In this context it will suffer higher latencies such as regular Linux processes. As soon as it calls Xenomai API, it will be moved back to RT space. This switching capability allows more flexibility in the management of the processes. A task originally created as real-time however will always be known by the co-kernel, independently from it's actual location.

To support this switching capability Xenomai needs to fulfill the following requirements:

- Common priority scheme: RT and non RT tasks must have a coherent priority policy. When an RT task is switched to secondary domain we will have a sort of mixed situation, with 2 kind of processes residing in the same space; they need to have shared scheduling policies to correctly coexist. In other words an RT task must have his priority correctly enforced in every domain. Xenomai solve this problem adopting the so called root thread's mutable priority technique, by which the Linux kernel automatically inherits the priority of the Xenomai thread controlled by the real-time nucleus which happens to enter the secondary domain. This means that Xenomai threads currently running in the primary domain won't necessarily preempt those running in the secondary one, unless their effective priority is actually higher. A regular Linux task will always be preempted by a Xenomai thread running in the primary domain, but they will compete priority-wise for the CPU usage when running both in the secondary domain;
- Predictability of program execution times: when a Xenomai thread runs over the secondary domain, either executing kernel or application code, its timing should not be affected by non real-time Linux interrupt activities, or in general, by any low priority, asynchronous activity occurring at kernel level. A

simple method to avoid this situation is to starve Linux interrupts when a real-time task is running in the secondary domain. This can be done inserting a layer, called interrupt shield, between Xenomai and Linux domain. This shield prevent the delivery of non real-time events to the secondary domain when a RT task is running into it. This shield is deactivated automatically when no RT task are running in the Linux space. This shield can be activated on a per-thread or system-wide basis during the kernel build. The interrupt shield is shown in figure 5.5;

- Fine-grained Linux kernel: in order to get the best from the execution in the secondary space, we need the Linux kernel to exhibit the shortest possible non-preemptible section, so that rescheduling opportunities are taken as soon as possible after a Xenomai thread running in the secondary domain becomes ready-to-run. Additionally, this ensures that Xenomai threads can migrate from the primary to the secondary domain within a short and time bounded period of time, since this operation involves reaching a kernel rescheduling point. For this reason, Xenomai benefits from the continuous trend of improvements regarding the overall pre-emptibility of the Linux kernel, including Ingo Molnar's PREEMPT_RT extension. Of course, Xenomai threads which only run in the primary domain are not affected by the level of granularity of the Linux kernel, and always benefit from very low and bounded latencies, since they do not need to synchonize in any way with the Linux operations, which they actually always preempt unconditionally;

- Priority inversion management: both the real-time nucleus and the Linux kernel should handle the case where a high priority thread is kept from running because a low priority one holds a contended resource for a possibly unbounded amount of time. Xenomai provides support to avoid this situation at least in the primary domain, while Linux kernel does not do it natively.

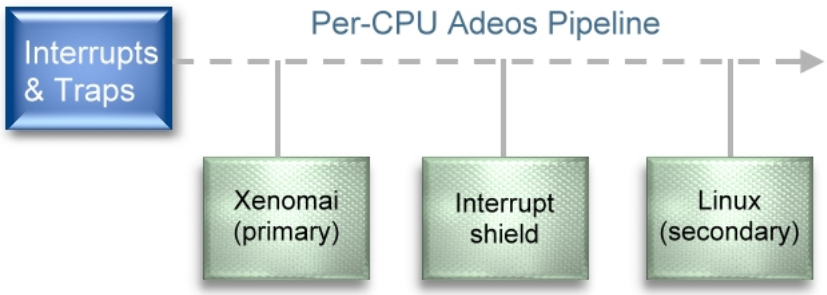

Figure 5.5. Xenomai interrupt shield.

In conclusion, for a correct real-time management, every time Xenomai's core is loaded, at least three domains are created in the Adeos pipeline:

- Primary domain, where the Xenomai co-kernel and Nucleus core reside;
- Secondary domain, where the standard Linux kernel reside;
- Interrupt shield, to prevent non real-time events delivery to Linux domain when RT task are running into it.

5.4 System calls

We have seen how the Adeos pipeline works and how it enables the Xenamai framework. We have also seen the organization in domains, which allows to separate RT and non RT space. Xenomai services, implemented as independent modules called skins, can be called from user and kernel space through ad-hoc system calls. However there are also the standard Linux system calls; as for interrupts, Xenomai needs a strategy to correctly manage all the incoming system calls. The Adeos pipeline allows to register a dedicated event handler which is able to:

- dispatch the real-time services requests from applications to the proper system call handlers, which are implemented by the various APIs running over the real-time nucleus;
- ensure that every system call is performed under the control of the proper domain, either Xenomai or Linux, by migrating the caller to the target domain as required. For instance, a Linux system call issued from a Xenomai thread running in the Xenomai domain will cause the automatic switching of the caller to the Linux domain, before the request is sent to the regular Linux system call handler. The other way around, a Xenomai thread which invokes a possibly blocking Xenomai system call will be moved to the Xenomai domain before the service is eventually performed, so that the caller may sleep under the control of the real-time nucleus.

5.5 Nucleus core

Xenomai framework is made of different skin modules, implementing a specific set of traditional RTOS API, built upon the I-pipe. Between these modules, one is particularly important and is mandatory in the framework, the Nucleus core. This is the central building block which allows the delivery of the real-time services offered by Xenomai. The Nucleus core provides the following components:

- A real-time thread object directly controlled by the Xenomai scheduler. The scheduler is based on a preemptive and fixed-priority policy, supporting a number of thread priority levels. Within a priority level,FIFO ordering applies. This scheduler also enables round-robin scheduling on a per-thread basis. Basic scheduling operations such as priority management, preemption control, and suspension/resumption are available thanks to the Nucleus interface;

- A memory allocator with predictable latencies that skins module can specialize to support the dynamic allocation of variable-size memory blocks;

- A generic interrupt object which connects the skin modules to any number of IRQ lines provided by the underlying hardware. Xenomai support also nested and shared interrupts;

- A synchronization object. This is one of the most important features of the Xenomai core. This object implements thread blocking on any kind of resource for all the real-time services. Xenomai supports timeouts, priority inheritance, and priority-based or FIFO queuing order when multiple threads have to block on a single resource. For instance, all kinds of semaphores, mutexes, condition variables, message queues, and mailboxes defined by the RT API are based on this object;

- Timer management, allowing any time-related service to create any number of software timers. Xenomai also implements a sort of time base, by which software timers which belong to different RT skins can be clocked separately and concurrently, with different and distinct frequencies.

5.6 Xenomai skins

In the following a brief list of the skins available on Xenomai. Each skin represents a set of API intended for a specific scope. Each skin is implemented as an independent kernel module which can be loaded and used upon the Nucleus core module.

The available modules are:

- Nucleus core: the central building blocks, already discussed;

- POSIX skin: a set of API which mimic and replace the standard POSIX services and the glibc libraries. Using this module it is possible to reuse, with very little modifications, portions of code written for standard Linux. This makes the porting of a Linux application into the Xenomai framework pretty easy;

- Native skin: a set of API born and developed specifically for Xenomai. It allows to create and manage real-time processes and other structures similarly to POSIX;
- RTDM skin: the RTDM (Real-Time Driver Model) is an interface to implement real-time drivers to be integrated with the kernel. This skin takes into account the presence of multiple domains: RTDM provides two version of each method, depending on the context(RT or non RT) who is managing the driver;
- Traditional RTOS services: Xenomai provides various sets of APIs compatible with the most common traditional RTOS and real-time framework available on the market, such as VxWorks, pSOS+, VRTX, uITRON, RTAI3.

After this theoretical introduction to real-time and Xenomai, we can move to a more practical part; the next chapter shows the procedure to install and use Xenomai and the changes introduced to the program to make it works, the main part of this thesis work.

Chapter 6

Work with Xenomai

The procedure to install Xenomai on a Unix-like system requires to patch the kernel. This means that the kernel sources will be modified and all the needed features (mainly the I-pipe structure and the Nucleus core, plus the desired skins) will be added to it. At the end the kernel will need to be recompiled.

The Xenomai and I-pipe source codes can be downloaded respectively from https://xenomai.org/downloads/xenomai/stable/ and https://xenomai.org/downloads/ipipe/. The user should pay attention to choose a patch version compatible with the target kernel and architecture. In our case the target is the Linux 2.6.38 kernel running on an ARM9 architecture and the downloaded files are xenomai-2.6.0-rc5.tar.bz2 and adeos-ipipe-2.6.38-arm. The procedure to apply the patch is the following:

```
> tar -xvjf xenomai-2.6.0-rc5.tar.bz2
> cd xenomai-2.6.0-rc5.tar.bz2
> scripts/prepare-kernel.sh --linux=<path to linux kernel>
  --adeos=<path to ipipe file>/adeos-ipipe-2.6.38-arm --arch=arm
```

The 'prepare-kernel.sh' script will apply the patch and install the chosen I-pipe version on the selected kernel.

Now the patch is applied and it is possible to recompile the kernel running the 'make' command in the kernel source directory. At the beginning of the compilation procedure we will be asked to choose which Xenomai features have to be added and how to configure them. In general, all the Nucleus, Native, POSIX and RTDM skins have been selected with default parameters values when needed. The details of the configuration of the kernel can be found in the '.config' file in the appendix (since this file is pretty large, only the Xenomai related settings are shown). At the end of the compilation the new kernel image can be copied into the kernel partition on the SD-card.

To have a working Xenomai framework we still need to install the user-space API that will allow us to call all the real-time functionalities that has been added at the kernel-level. The easiest way to perform this operation is to copy the Xenomai sources files (the xenomai-2.6.0-rc5 directory) into the file-system partition and then run

```
> cd xenomai-2.6.0-rc5
> ./configure
> make
> make install
```

Now Xenomai should be fully installed and working. It is possible to check it's functionalities with some demo programs included in the sources files, such as the 'trivialperiodic' program. This demo code, based on the Native skin API, simply creates a periodic task with a period of 1 second. This task implements an infinite loop in which, at each iteration, the actual value of a timer is read and it is subtracted the value of the previous iteration. In this way it is possible to check if the obtained period is equal to the expected one and have an idea of the time resolution Xenomai can reach. It has to be taken into account that the call to the printf() function used to print on the screen the elapsed time values will degrade the performances but here it is used only for demonstrative purpose. In a real program with timing constraint the use of this and many other functions should be avoided, as will be explained later.

The image 6.1 shows the execution of this demo code. As it can be seen the program shows a good deterministic behavior with an average time error less than $20\mu s$.

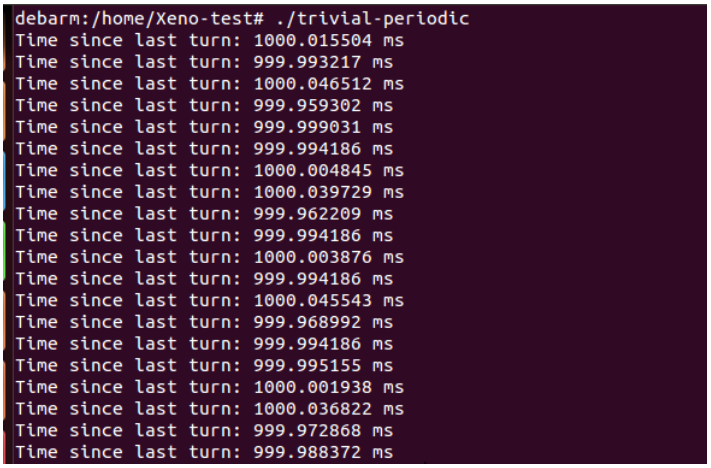

Figure 6.1. Trivial periodic demo application

trivial-periodic.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>
#include <native/task.h>
#include <native/timer.h>
RT_TASK demo_task;
/* NOTE: error handling omitted. */
void demo(void *arg)
{
        RTIME now, previous;
        /*
         * Arguments: &task (NULL=self),
         *            start time,
         *            period (here: 1 s)
         */
        rt_task_set_periodic(NULL, TM_NOW, 1000000000);
        previous = rt_timer_read();
        while (1) {
                rt_task_wait_period(NULL);
                now = rt_timer_read();
                /*
                 * NOTE: printf may have unexpected impact on the timing of
                 *       your program. It is used here in the critical loop
                 *       only for demonstration purposes.
                 */
                printf("Time since last turn: %ld.%06ld ms\n",
                       (long)(now - previous) / 1000000,
                       (long)(now - previous) % 1000000);
                       previous = now;
        }
}
void catch_signal(int sig)
{
}
int main(int argc, char* argv[])
{
        signal(SIGTERM, catch_signal);
        signal(SIGINT, catch_signal);
        /* Avoids memory swapping for this program */
        mlockall(MCL_CURRENT|MCL_FUTURE);
        /*
         * Arguments: &task,
         *            name,
         *            stack size (0=default),
         *            priority,
         *            mode (FPU, start suspended, ...)
         */
        rt_task_create(&demo_task, "trivial", 0, 99, 0);
        /*
         * Arguments: &task,
         *            task function,
         *            function argument
         */
        rt_task_start(&demo_task, &demo, NULL);
        pause();
        rt_task_delete(&demo_task);
}
```

The Makefile used to compile this program is shown in the following and can be used as a reference template for compiling a general application based on Xenomai.

The XENODIR variable points to the Xenomai installation directory, XENOCONFIG points to the 'xeno-config' script. This script is able to automatically return the requested compilation flags and parameters depending on some options:

- –cc will return the correct compiler to use;
- –skin=native –cflags will return the correct C flags to use the desired API (Native,POSIX,RTDM,etc);
- –sin=native –ldflags will return the correct LOAD flags to use the desired API (Native,POSIX,RTDM,etc);
- –libdir will return the Xenomai library path, to be used during the linking phase.

Makefile

```
#
# MAKEFILE template for Xenomai-native  application
#
#Compile for standard Linux
CC = gcc
XENOLDFLAGS = -lpthread -lrt
#Xenomai configuration -comment for standard linux
#Xenomai installation directory(default /usr/xenomai)
XENODIR = /usr/xenomai
#Set Xeno-config and wrap-link location
XENOCONFIG = $(shell PATH=$(XENODIR):$(XENODIR)/bin:$(PATH)
which xeno-config 2>/dev/null)
#Xenomai compiler,flags and library
CC = $(shell $(XENOCONFIG) --cc)
XENOCFLAGS = $(shell $(XENOCONFIG) --skin=native --cflags)
XENOLDFLAGS = $(shell $(XENOCONFIG) --skin=native --ldflags)
XENOLDFLAGS+=-Xlinker -rpath -Xlinker $(shell $(XENOCONFIG) --libdir)
#Application to be built
APPLICATION = trivial_periodic
OBJ = trivial-periodic.o
all:    $(APPLICATION)
clean:
        @echo "Cleaning up directory"
        rm -f core $(OBJ) *~ $(APPLICATION)
trivial-periodic:       $(OBJ)
                        $(CC) -o $@ $(OBJ) $(XENOLDFLAGS)
trivial-periodic.o :    trivial-periodic.c
                        $(CC) -c $< -o $@ $(XENOCFLAGS)
```

Another example of the Xenomai functionalities is the demo-periodic-thread-posix program. This code, realized by Marc Le Douarain and available on the Xenomai site, uses the POSIX skin to create a periodic thread with a period of 10ms. Every time the thread is entered, a counter variable is increased. At the end of the execution the counter value is compared with the theoretical expected value. The execution of the program is shown in picture 6.2.

demo-periodic-thread-posix.c

```
/*
   Demo of a simple periodic thread using Posix
   under 'standard' Linux
   and under Xenomai versions 2 and 3 using posix skin
   Marc Le Douarain, august 2015
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <errno.h>
```

```
debarm:/home/Xenomai-posix-test# ./demo_periodic_thread_posix
Simple periodic thread using Posix.
Version compiled for Xenomai 2 or 3.
Created thread 'DemoPosix' period=10000usecs ok.
Wait 10 seconds before ending...
Increment value during 10 secs = 900 (should be 900)
debarm:/home/Xenomai-posix-test#
```

Figure 6.2. Demo periodic thread posix application

```
#include <pthread.h>
#include <sys/mman.h>
#include <sys/timerfd.h>
#define PERIOD_MICROSECS 10000 //10millisecs
#define START_DELAY_SECS 1 //1sec
pthread_t MyPosixThread;
int TimerFdForThread = -1;
char ThreadRunning = 0;
int ResultIncValue = 0;
int CreatePosixTask( char * TaskName, int Priority, int StackSizeInKo,
 unsigned int PeriodMicroSecs, void * (*pTaskFunction)(void *) )
{
        pthread_attr_t ThreadAttributes;
        int err = pthread_attr_init(&ThreadAttributes);
        if ( err )
        {
                printf("pthread attr_init() failed for thread '%s'
                with err=%d\n", TaskName, err );
                return -10;
        }
#ifdef __COBALT__
        err = pthread_attr_setinheritsched( &ThreadAttributes,
        PTHREAD_EXPLICIT_SCHED );
        if ( err )
        {
                printf("pthread set explicit sched failed for thread '%s'
                with err=%d\n", TaskName, err );
                return -11;
        }
#endif
        err = pthread_attr_setdetachstate(&ThreadAttributes,
        PTHREAD_CREATE_DETACHED /*PTHREAD_CREATE_JOINABLE*/);
        if ( err )
        {
                printf("pthread set detach state failed for thread '%s'
                with err=%d\n", TaskName, err );
                return -12;
        }
#if defined(__XENO__) || defined(__COBALT__)
        err = pthread_attr_setschedpolicy(&ThreadAttributes, SCHED_FIFO);
        if ( err )
        {
                printf("pthread set scheduling policy failed for thread
                '%s' with err=%d\n", TaskName, err );
                return -13;
        }
        struct sched_param paramA = { .sched_priority = Priority };
        err = pthread_attr_setschedparam(&ThreadAttributes, &paramA);
        if ( err )
        {
                printf("pthread set priority failed for thread '%s'
                with err=%d\n", TaskName, err );
                return -14;
        }
#endif
        if ( StackSizeInKo>0 )
        {
```

```
                err = pthread_attr_setstacksize(&ThreadAttributes,
                StackSizeInKo*1024);
                if ( err )
                {
                        printf("pthread set stack size failed for thread
                        '%s' with err=%d\n", TaskName, err );
                        return -15;
                }
        }
        // calc start time of the periodic thread
        struct timespec start_time;
#ifdef __XENO__
        if ( clock_gettime( CLOCK_REALTIME, &start_time ) )
#else
        if ( clock_gettime( CLOCK_MONOTONIC, &start_time ) )
#endif
        {
                printf( "Failed to call clock_gettime\n" );
                return -20;
        }
        /* Start one seconde later from now. */
        start_time.tv_sec += START_DELAY_SECS ;
        // if a timerfd is used to make thread periodic (Linux / Xenomai 3),
        // initialize it before launching thread (timer is read in the loop)
#ifndef __XENO__
        struct itimerspec period_timer_conf;
        TimerFdForThread = timerfd_create(CLOCK_MONOTONIC, 0);
        if ( TimerFdForThread==-1 )
        {
                printf( "Failed to create timerfd for thread '%s'\n", TaskName);
                return -21;
        }
        period_timer_conf.it_value = start_time;
        period_timer_conf.it_interval.tv_sec = 0;
        period_timer_conf.it_interval.tv_nsec = PeriodMicroSecs*1000;
        if ( timerfd_settime(TimerFdForThread, TFD_TIMER_ABSTIME,
        &period_timer_conf, NULL) )
        {
                printf( "Failed to set periodic tor thread '%s'
                with errno=%d\n", TaskName, errno);
                return -22;
        }
#endif
        ThreadRunning = 1;
        err = pthread_create( &MyPosixThread, &ThreadAttributes,
        (void *(*)(void *))pTaskFunction, (void *)NULL );
        if ( err )
        {
                printf( "Failed to create thread '%s' with err=%d!\n",
                TaskName, err );
                return -1;
        }
        else
        {
                // make thread periodic for Xenomai 2 with
                pthread_make_periodic_np() function.
#ifdef __XENO__
                struct timespec period_timespec;
                period_timespec.tv_sec = 0;
                period_timespec.tv_nsec = PeriodMicroSecs*1000;
                if ( pthread_make_periodic_np(MyPosixThread, &start_time,
                &period_timespec)!=0 )
                {
                        printf("Xenomai make_periodic failed for thread
                        '%s' with err=%d\n", TaskName, err);
                        return -30;
                }
#endif
                pthread_attr_destroy(&ThreadAttributes);
#ifdef __XENO__
                err = pthread_set_name_np( MyPosixThread, TaskName );
#else
```

```
                err = pthread_setname_np( MyPosixThread, TaskName );
#endif
                if ( err )
                {
                        printf("pthread set name failed for thread '%s',
                        err=%d\n", TaskName, err );
                        return -40;
                }
                printf( "Created thread '%s' period=%dusecs ok.\n",
                TaskName, PeriodMicroSecs );
                return 0;
        }
}
void WaitPeriodicPosixTask( )
{
        int err = 0;
#ifdef __XENO__
        unsigned long overruns;
        err = pthread_wait_np(&overruns);
        if (err || overruns)
        {
                printf( "Xenomai wait_period failed for thread: err=%d,
                overruns=%lu\n", err, overruns );
        }
#else
        uint64_t ticks;
        err = read(TimerFdForThread, &ticks, sizeof(ticks));
        if ( err<0 )
        {
                printf( "TimerFd wait period failed for thread with
                errno=%d\n", errno );
        }
        if ( ticks>1 )
        {
                printf( "TimerFd wait period missed for thread:
                overruns=%lu\n", (long unsigned int)ticks );
        }
#endif
}
// our really simple thread just incrementing a variable in loop !
void * MySimpleTask( void * dummy )
{
        while( ThreadRunning )
        {
                WaitPeriodicPosixTask( );
                ResultIncValue++;
        }
        return 0;
}
int main( int argc, char * argv[] )
{
        int err;
        printf("Simple periodic thread using Posix.\n");
#if defined( __XENO__ ) || defined( __COBALT__ )
        printf("Version compiled for Xenomai 2 or 3.\n");
        mlockall(MCL_CURRENT | MCL_FUTURE);
#endif
        err = CreatePosixTask( "DemoPosix", 1/*Priority*/, 16/*StackSizeInKo*/,
        PERIOD_MICROSECS/*PeriodMicroSecs*/, MySimpleTask );
        if ( err!=0 )
        {
                printf( "Init task error (%d)!\n",err );
        }
        else
        {
                printf( "Wait 10 seconds before ending...\n" );
                sleep( 10 );
                ThreadRunning = 0;
                printf( "Increment value during 10 secs = %d (should be %d)\n",
                ResultIncValue, ((10-START_DELAY_SECS)*1000*1000)/PERIOD_MICROSECS );
        }
        return 0;
```

```
}
```

The Makefile for this program is very similar to the previous one. Here the –skin=posix flag is passed to the 'xeno-config' script. Another script is used in this case: the 'xeno-wrap.sh' script manages the final linking phase, in which the Xenomai libraries are used instead of the standard POSIX ones.

Makefile

```
#
# MAKEFILE template for Xenomai-posix application
#
#Compile for standard Linux
CC = gcc
XENOLDFLAGS = -lpthread -lrt
#Xenomai configuration -comment for standard linux
#Xenomai installation directory(default /usr/xenomai)
XENODIR = /usr/xenomai
#Set Xeno-config and wrap-link location
XENOCONFIG = $(shell PATH=$(XENODIR):$(XENODIR)/bin:$(PATH)
which xeno-config 2>/dev/null)
XENOWRAP = $(XENODIR)/bin/wrap-link.sh -v
#Xenomai compiler,flags and library
CC = $(shell $(XENOCONFIG) --cc)
XENOCFLAGS = $(shell $(XENOCONFIG) --skin=posix --cflags)
XENOLDFLAGS = $(shell $(XENOCONFIG) --skin=posix --ldflags)
XENOLDFLAGS+=-Xlinker -rpath -Xlinker $(shell $(XENOCONFIG) --libdir)
#Application to be built
APPLICATION = demo_periodic_thread_posix
OBJ = demo_periodic_thread_posix.o
all:    $(APPLICATION)
clean:
        @echo "Cleaning up directory"
        rm -f core $(OBJ) *~ $(APPLICATION)
demo_periodic_thread_posix:     $(OBJ)
                                $(XENOWRAP) $(CC) -o $@ $(OBJ) $(XENOLDFLAGS)
demo_periodic_thread_posix.o :  demo_periodic_thread_posix.c
                                $(CC) -c $< -o $@ $(XENOCFLAGS)
```

6.1 Use of the Xenomai POSIX skin

Xenomai POSIX skin allows the porting of a standard POSIX application to the real-time Xenomai infrastructure in a "soft" way. The POSIX skin in fact are able to replace many of the usual POSIX services with the correspondents real-time version. This means that, in theory, very small modifications should be done on the code, cause the majority of the Xenomai services are fully compliant with the prototype of the standard POSIX functions.

As already explained, a Xenomai application may run in two modes: either the primary domain, where it is scheduled by the Xenomai co-kernel, and benefits from hard real-time scheduling latencies, or the secondary mode, where it is an ordinary Linux thread, and as such may call any Linux services. A thread can change mode dynamically; when this thread calls a Xenomai real-time service while running in secondary domain, it switches to primary domain, when it calls any non real-time Xenomai service or any Linux service (including exceptions such as page faults)

while running in primary mode, it switches to secondary mode.

However, there are some rules that have to be respected to make Xenomai work properly, to exploit all of it's functionality and to obtain a robust real-time system. In the following some important points will be discussed.

The first key point to be considered is the memory locking. Linux typically uses an on demand paging scheme; when some data are required, the corresponding page is loaded from disk and stored in the main memory, available to all processes requiring it. It may happen that the main memory becomes full; in this case, according to some policy, the paging mechanism allows to move some part of the main memory back onto disk (memory swapping), creating free space for other pages and increasing the maximum amount of main memory on the machine. This is a very simple description of the so called virtual memory technique. In a real-time environment this can be a huge problem; reading data from disk is slower than reading from the main memory and will introduce a very large and not predictable latency. For this reason, a hard real-time application should always be locked in memory. Locking a process means that all the pages containing data related to the process will be loaded statically and blocked in the main memory; they will reside permanently in the memory (at least until the end of the execution) and will never be swapped out. Locking a process in memory can be done very simply with the 'mlockall(mcl_current | mcl_future)' instruction. This will lock in memory all the pages mapped to the address space of the calling process when the call is performed(mcl_current) and also the future ones(mcl_future),in case of dynamic allocation. Calling this function is mandatory in Xenomai before asking for any real-time service. It has to be considered that calling mlockall() will lock all the pages mapped to a process, also for example loaded libraries. For this reason the number of locked pages can be pretty high, and this can be a problem in case of systems with a very limited amount of memory. In general, only the real-time tasks should be locked to avoid a waste of memory.

Also the creation of new threads can be a problem when using mlockall(). Each thread in fact comes with a predefined maximum stack size which, depending on the system, may also reach some MB. When the application is locked, the full stack size of each thread will be allocated in memory, also if not really used. The occupied memory can grow very fast. To prevent this situation it is possible, also at runtime, to specify the maximum stack size of every new created thread. Knowing the requirements of the application, it is possible to estimate the real usage of the stack and set it's upper-bound to a more suitable (and possibly smaller) value. The desired maximum stack size can be set before creating the thread with the function 'pthread_attr_setstacksize()' specifying a pthread attribute descriptor and the desired stack size value in Bytes.

Another aspect to consider is how Xenomai can identify a real-time process. An application is typically made of several different tasks; among them, maybe only a few has to be real-time, the other being standard Linux ones. A method has to be defined to instruct Xenomai on which tasks has to be considered time critical and managed accordingly. By default, those tasks which have been defined with scheduling policy sched_fifo are considered hard real-time and managed by the co-kernel. sched_fifo is one of the two real-time policy made available by the POSIX standard. These policies in reality are soft real-time algorithm, cause they are not able to guarantee the deadlines. A sched_fifo tasks will be treated as a hard real-time one and will be given higher priority. The policy of the task can be set with ' pthread_attr_setschedpolicy()'. Attention has to be paid on the meaning of sched_fifo; such a process will be given high priority and will run until it suspends waiting on a resource or another process arrives with a higher priority. If none of these events happens, the process may potentially monopolize the control of the CPU and never release it in case, for example, of an infinite loop. Consider for example the following simple piece of code

```
pthread_mutex_lock(&mutex);
/* (...) */
while (!cond)
pthread_cond_wait(&cond, &mutex);
/* (...) */
pthread_mutex_unlock(&mutex);
```

if mutex or cond are not correctly initialized or if one of them it's destroyed, and there are no other processes with higher priority, this process will loop forever, never releasing the CPU and creating a stall condition.

Mutexes and condition variables have to be managed correctly. In particular Xenomai requires a system call to initialize these object; the usual pthread_cond_initializer and pthread_mutex_initializer static initializer will not work on Xenomai. It is mandatory to call 'pthread_ mutex_init()' or 'pthread_cond_init()' to initialize each mutex or condition variable. This operation has to be performed before entering the time critical section.

The POSIX skins are able to overrides the standard POSIX services with their corresponding real-time version. However in some cases it would be useful to keep the standard functions; this can be done using the following syntax

```
//This function will be overwritten
fd = socket(PF_INET, SOCK_DGRAM, 0);
//This function will not be overwritten
fd = __real_socket(PF_INET, SOCK_DGRAM, 0);
```

putting '___real_' in front of a function will prevent Xenomai from overriding it with it's real-time implementation.

Xenomai services are provided and managed on a per-process basis. This means that it is not possible to use in a process objects(mutex, semaphore, etc) which have been created and initialized in another one. This introduces a big difference with respect to the standard Linux; when using fork(), it is not possible to share data between father and children. In other words, a child cannot use objects which have been initialized by the father. If the two needs to share data, the initialization has to be done after the fork(), singularly for each process. In case the two needs to share a POSIX object (semaphore, mutex, etc), pthread_mutexattr_setpshared(), pthread_condattr_setpshared(), etc have to be used.

Xenomai threads can run mainly in two ways, continuously or periodically. A continuous thread will run until it is preempted by another one with higher priority. A periodic thread will be scheduled with a predefined periodicity. Depending on the application, one choice will be preferred to the other. Every new thread will be created as continuous; it can be made periodic with the 'pthread_make_periodic_np()' function, specifying it's starting time and periodicity.

Following all these rules should lead to a working Xenomai application. However it is still possible that performances are not the best. As said before a real-time thread which requires a non real-time service will be switched to the secondary domain, where it will suffer larger latencies. Switching to secondary domain should be avoided. In the following a list of common situation which can generate a domain switch and some possible solution.

- Every call to a Linux service which happens while running in the primary domain will generate a switch to the secondary domain. This kind of switch is pretty easy to spot and the solution seems trivial; do not use Linux system call in a real-time critical section. This solution, although very simple in theory, can require some modification of the code; if some operations were performed using a system call, they need to be replaced and performed in some other way. More difficult to spot some functions which may rise a system call only some times, such as a malloc() in case of error;
- Access to driver through the usual functions such as open(), close(), write(), read(), etc will generate a domain switch. Here the solution is offered by Xenomai itself with the RTDM skin, which allows to implement a real-time driver;
- Writing/reading from files, or in general all input/output operations are generally managed with system calls and so they will generate a domain switch. A solution can be to use buffers and other data structures to store all the needed information, and make access to files and memory in general only before or after the time critical section;

- Dynamic allocation, such as malloc(), may generate system calls and exceptions and so should be avoided. All the memory and resources required by the task should be allocated statically before entering the time critical section.

Another possible cause of domain switch is the situation depicted in figure 6.3. This is a sort of priority inversion which may generate not really a switch but a condition with similar effects. When thread 1 lock the mutex, it is switched to primary domain. When thread 2 will try to lock the same mutex, it will be suspended until thread 1 releases it. Suppose now that thread 1 perform some operation which cause it to switch to secondary domain; thread 2, despite being still in primary domain, will suffer higher latencies because it is suspended on a tasks which is in a non real-time context. Different cause, but same effect. The only way to avoid this condition is to organize very well the architecture of the program and the interaction between threads.

Thread 1	Thread 2
`pthread_mutex_lock(&mutex);`	(...)
(...)	`pthread_mutex_lock(&mutex);`
`write(fd, buffer, sizeof(buffer));`	(...)

Figure 6.3. A possible example of suspension on a non real-time task, which will cause higher latency.

Since this situation can be difficult to find in debug, Xenomai offers a kernel option, config_xeno_opt_debug_synch_relax, which will perform a check every time a mutex is contended. This option can be activated during the compilation of the kernel.

As it can be seen, there are a lot of different possible causes for a domain switch, and finding them all can be tricky; to help the developers, Xenomai provides a tool to check for domain switch at run-time, on a per thread basis, the pthread_warnsw bit. If this bit is set, a sigxcpu signal will be generated every time the thread switch to the non real-time domain. This signal can be used for debugging purpose; in the following, a very simple example in which it is used to count the number of context switch.

```
Real_time_thread()
{
        //Check domain switching
        signal(SIGXCPU,sigxcpu_handler);
        pthread_set_mode_np(0,PTHREAD_WARNSW);
        //...Main code...
}
//Count number of domain switching
void sigxcpu_handler(int sig)
{
        if(sig == SIGXCPU)
        {
                signal(SIGXCPU,sigxcpu_handler);
                domain_switch_numb++;
        }
}
```

These are the main guidelines which have to be followed when working with the POSIX skin. The FiTRelay application has been completely rewritten according to these rules. The program has been locked in memory, default thread stack size has been reduced, the process tree has been modified such as the organization and managing of mutexes and condition variables, etc. Different possible causes of domain switch have been identified and removed.

Particular attention has to be paid to the Foxbone protocol, used for the communication between the micro-controller and the FPGA. As previously explained, this communication scheme was originally managed through a set of system calls, which of course would have generated a large number of domain switch. For this reason the system calls have been replaced with a real-time driver module based on the RTDM skin. The next chapter, after a general introduction to module and drivers, will describe the developed driver.

Chapter 7

Real-time device driver module

7.1 Introduction to drivers and modules

A device driver is a piece of software intended to manage a specific hardware component. The driver act as an interface between the operating system and the hardware part: it allows the kernel to correctly configure and manage the hardware in a safe and robust way, and allows in general the data exchange between the two. Every system, from a small and simple embedded micro-controller to a large and complex control system, interacts with a large number of hardware parts; for each of them, a specific device driver is needed.

In some way, the kernel itself with it's Hardware Abstraction Layer (HAL) can be seen as a driver, which makes available to the user space programs all the physical resources present on the machine. Some drivers are essential in a machine, like as the CPU controller and the memory manager, others are optional and depends on the configurations of the system, like as drivers for disks, printers, keyboards, etc.

Drivers interacts directly with the hardware. For this reason, they always run in the kernel space, with the highest permission levels. They have to be developed very carefully to guarantee stability, robustness, performances, safety of the operations.

As previously said, a driver is developed for a specific hardware component. The developer has to know the internal structure of the module very accurately cause he will address the specific registers and configuration bits of that physical device. For this reason it will not be compatible with a different one, in general. Furthermore, the driver needs to interact with the operating system; being an interface between

the kernel and the device, it has to respect some standard rules of communication to correctly exchange data and information with it and ask for it's functionality. This means that writing a driver require to keep in mind two targets, the physical device and the running operating system. Drivers are typically written in a low level language, such as C or in some cases even Assembly.

Today there is a very large variety of different devices which can be connected to a system, so it is difficult to make a sort of classification. Driver however are typically grouped in three main families:

- Character driver: physical device which can be seen and managed as a standard file. It uses regular operations like open(),close(),read() and write(). Data can be typically read and written sequentially. The keyboard and the console are two example of this device class;
- Block driver: physical device which is typically read and written in block of fixed size. This class is used for memories and disks in general. This driver class allows to mount a file-system onto the device;
- Interface driver: physical device implementing some kind of communication/net protocol. Data are typically transmitted in packages made up of fixed fields, such as target identifiers, data type, data length, control and configuration fields, etc.

Independently from the class, all physical devices connected to a Linux system are seen as a file, and so are accessed through the usual file operations, among which:

- open() : open and eventually initialize the device;
- close() : safely close the device;
- read() : transfer data from the device to the system;
- write() : transfer data from the system to the device.

Depending on the the device, the meaning of these functions will change; the driver in fact will implement each of these calls in the proper way for that specific peripheral. In other word a driver is simply a collection of function which overrides the usual meaning of these calls in the desired way.

There are mainly two ways of inserting a driver in an operating system:

- Statically : all the required drivers of a system are compiled with the kernel and strictly integrated with it. This solution can lead to high stability and optimization, robustness and performances, but on the other hand a small

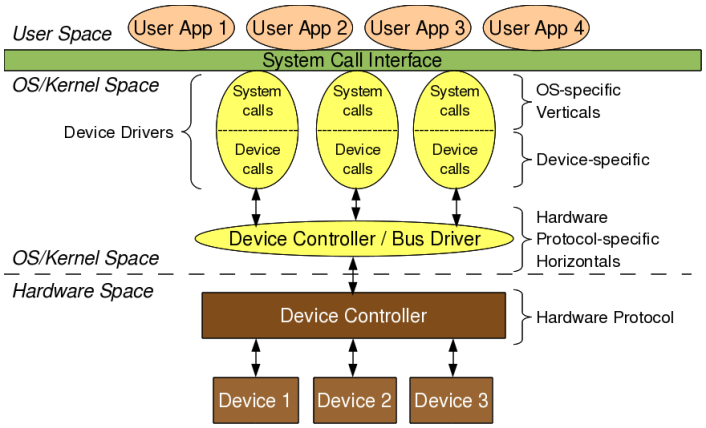

Figure 7.1. Each device connected to the system is accessed through a specific driver which implements and makes available to the system all the methods required to correctly interact with it.

bug in a piece of the code can affect all the system and so the driver has to be developed very carefully. This solution is feasible when a small number of driver is needed and known a priori; on a general purpose machine, which may potentially interact with an enormous number of different peripheral, this will require to install on the machine all the possible drivers available on the market, that is of course impossible;

- Dynamically : only a small subset of the most common drivers comes preinstalled with the kernel. When a peripheral is connected to the system it's driver, which is typically present into the device, is dynamically loaded and made available to the system. This method is not able to guarantee such a level of optimization and performances as the static one, but is much more flexible and practical to use. This is the most common strategy for general purpose system.

From a more technical point of view, in the majority of the cases drivers are implemented as kernel modules. A module is a particular kind of object which is supported by the operating system. Simply speaking, it can be seen as an independent box, a block of code which contains functions, methods, data type, structure, etc, which run in the kernel space with privileged permission. A module can be used in theory for every kind of application, also a user program but, due to it's nature, it is commonly used for standalone piece of code or independent blocks such as drivers.
Modules are part of the kernel and have to interact with it; for this reason they have to be compliant with some rules that have to be taken into account when writing the module. Every driver will be different but, summarizing, the general operations performed by a module are the following:

- Init function : this is a special function which is executed automatically every time the module is loaded into the kernel. This function is mandatory for every module and is typically used to register the device (the meaning of this operation will be explained later);
- Open function : prepares the device performing an eventual initialization and configuration of it. Typically this function is also used to allocate all the resources needed by the device, such as dynamic data structures, interrupt lines, files, I/O pins and other hardware;
- Read : transfers data from the device to the kernel. The actual implementation depends of course on the kind of device;
- Write : transfers data from the kernel to the device. The actual implementation depends of course on the kind of device;
- Close : deallocates all the resources assigned to the device, performing in general the opposite operations of the open function;
- Exit function : this is a special function which is executed automatically every time the module is unloaded from the kernel. This function is mandatory for every module and is typically used to unregister the device (the meaning of this operation will be explained later);
- other device specific functions.

This list is simply to be intended as a guideline, every driver will perform different operations in different functions depending on the case.
As previously explained, every peripheral is accessed through the standard file operations such as open(), read(), write(), close(), etc. Every device moreover is identified by two numbers, called minor and major number. The major number is an integer which is associated to the driver controlling the device. When a driver module is registered, it is automatically given a major number identifying it. It may happen then two or more devices are associated with the same driver, and so will share the same major number. The minor number is used by the driver module to identify a specific device; it may happen that two or more entities of the same device are connected to the system and so will be identified by different minor numbers. Minor numbers are used only inside the driver.

7.2 Real-time driver

After this brief introduction to drivers and modules, we can look more in detail to the developed module. Xenomai makes available the RTDM skin, which stands for Real Time Driver Module. This is a collection of functions and methods which can

be used inside the Xenomai real-time environment to describe the behavior and functionalities of a kernel module. Services of an RTDM driver can be called from a real time process without introducing a domain switch.

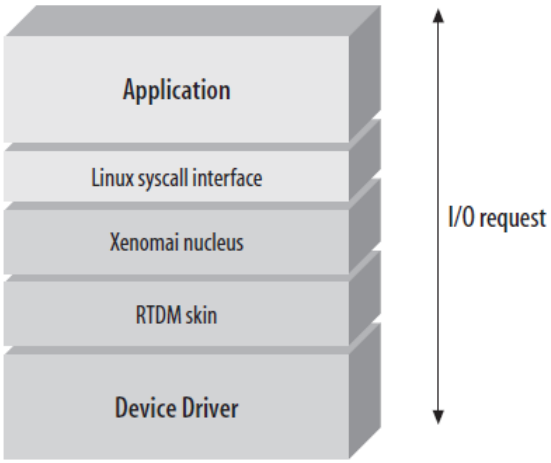

Figure 7.2. Xenomai Real-Time Driver Model layer.

A significant extract of the developed module is reported below.

foxbone-RTdriver-module.c

```
//-----------------------------------------------
// Xenomai based Real time module driver for
// the Foxbone protocol on the FOXG20 board
//-----------------------------------------------
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/input.h>
#include <mach/at91_pio.h>
#include <linux/linkage.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/time.h>
#include <linux/gpio.h>
#include <linux/slab.h>
#include <rtdm/rtdm_driver.h>
#include <rtdk.h>
#define MOD_LICENSE "GPL"
#define MOD_AUTHOR "Paolo Michelotti"
#define MOD_DESCRIPTION "Xenomai RT module driver for the Foxbone protocol
on FOXg20 board"
#define DEVICE_NAME "FoxboneG20"
#define MODULE_NAME "FoxboneRT-driver"
#define DEBUG 1
typedef struct {
        char data[4];
} buffer_t;
static ssize_t rtdm_read_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, char* buf, size_t nbyte);
static ssize_t rtdm_write_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, char* buf, size_t nbyte);
static int rtdm_open_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, int oflags);
static int rtdm_close_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info);
MODULE_LICENSE (MOD_LICENSE);
MODULE_AUTHOR (MOD_AUTHOR);
MODULE_DESCRIPTION(MOD_DESCRIPTION);
//RTDM device descriptor
static struct rtdm_device device = {
        .struct_version = RTDM_DEVICE_STRUCT_VER,
        .device_flags = RTDM_NAMED_DEVICE,
        .context_size = sizeof( buffer_t),
        .device_name = DEVICE_NAME,
        .open_nrt = rtdm_open_rt,
        .open_rt = rtdm_open_rt,
        .ops = {
                .close_nrt = rtdm_close_rt,
                .close_rt = rtdm_close_rt,
                .read_nrt = rtdm_read_rt,
                .read_rt = rtdm_read_rt,
                .write_nrt = rtdm_write_rt,
                .write_rt = rtdm_write_rt,
        },
        .driver_name = MODULE_NAME,
        .peripheral_name = DEVICE_NAME,
        .proc_name = device.device_name,
};
//Module init/exit functions
int __init rtdm_init(void);
void __exit rtdm_exit(void);
//
//      Init function
//
int __init rtdm_init(void){
        int ret;
```

```
        ret = rtdm_dev_register(&device);
        if(DEBUG) rtdm_printk(KERN_INFO "Module %s registered with
        error code %d\n",MODULE_NAME,ret);
        return ret;
}
//
//      Exit function
//
void __exit rtdm_exit(void){
        int ret;
        ret = rtdm_dev_unregister(&device,1000);
        if(DEBUG) rtdm_printk(KERN_INFO "Module %s unregistered with
        error code %d\n",MODULE_NAME,ret);
}
//
//      Open function
//
static int rtdm_open_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, int oflags){
        void __iomem * iom;
        unsigned int mask;
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        __raw_writel(mask, iom + PIO_OWER);
        mask=0x3bc0c3f0;
        __raw_writel(mask, iom + PIO_OWDR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOA;
        mask=0x800002c0;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        mask=0x00000200;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        __raw_writel(mask, iom + PIO_SODR);
        mask=0x80000000;
        __raw_writel(mask, iom + PIO_CODR);
        mask=0x00000080;
        __raw_writel(mask, iom + PIO_CODR);
        mask=0x00000040;
        __raw_writel(mask, iom + PIO_CODR);
        return 0;
}
//
//      Close function
//
static int rtdm_close_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info){
        //Nothing to be done
        return 0;
}
//
//      Write function
//
static ssize_t rtdm_write_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, char* buf, size_t nbyte){
```

```
unsigned long int reg;
unsigned long int value;
void __iomem * iom;
unsigned int mask;
unsigned int data;
unsigned int iovalue;
unsigned int foxreg_address;
unsigned int tmp;
//Extract address and data from buffer
reg = *(buf)&0x00ff;
reg = reg|((*(buf+1)<<8)&0xff00);
value = *(buf+2)&0x00ff;
value = value|((*(buf+3)<<8)&0xff00);
if(DEBUG) rtdm_printk(KERN_INFO "Writing data %d at address
%d\n",(int)value,(int)reg);
//Set bus to output
iom = (void __iomem *)AT91_VA_BASE_SYS;
iom+=AT91_PIOB;
mask=0x04000000;
__raw_writel(mask, iom + PIO_IDR);
__raw_writel(mask, iom + PIO_PUDR);
__raw_writel(mask, iom + PIO_OER);
__raw_writel(mask, iom + PIO_PER);
iom = (void __iomem *)AT91_VA_BASE_SYS;
iom += AT91_PIOB;
mask=0xc03f3c0f;
__raw_writel(mask, iom + PIO_IDR);
__raw_writel(mask, iom + PIO_PUDR);
__raw_writel(mask, iom + PIO_OWER);
__raw_writel(mask, iom + PIO_OER);
__raw_writel(mask, iom + PIO_PER);
//Address
tmp=0x0000;
foxreg_address=0x0000;
tmp=(reg&0x00f0)<<8;
tmp=tmp|((reg&0x000f)<<4);
tmp=tmp|((reg&0x0f00)>>8);
tmp=tmp|((reg&0xf000)>>4);
foxreg_address=tmp;
iovalue=0x000f&foxreg_address;
iovalue=iovalue|((0x00f0&foxreg_address)<<6);
iovalue=iovalue|((0x3f00&foxreg_address)<<8);
iovalue=iovalue|((0xc000&foxreg_address)<<16);
//Write address
iom = (void __iomem *)AT91_VA_BASE_SYS;
iom += AT91_PIOB;
mask=0x04000000;
__raw_writel(mask, iom + PIO_SODR);
iom = (void __iomem *)AT91_VA_BASE_SYS;
iom += AT91_PIOB;
mask=0xc03f3c0f;
__raw_writel(iovalue, iom + PIO_ODSR);
iom = (void __iomem *)AT91_VA_BASE_SYS;
iom += AT91_PIOA;
mask=0x00000080;
__raw_writel(mask, iom + PIO_SODR);
__raw_writel(mask, iom + PIO_CODR);
//Data
tmp=0x0000;
data=0x0000;
tmp=(value&0x00f0)<<8;
tmp=tmp|((value&0x000f)<<4);
tmp=tmp|((value&0x0f00)>>8);
tmp=tmp|((value&0xf000)>>4);
data=tmp;
iovalue=0x000f&data;
iovalue=iovalue|((0x00f0&data)<<6);
iovalue=iovalue|((0x3f00&data)<<8);
iovalue=iovalue|((0xc000&data)<<16);
```

```
        //Write data
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(iovalue, iom + PIO_ODSR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x80000000;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_CODR);
        return(0);
}
//
//      Read function
//
static ssize_t rtdm_read_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, char* buf, size_t nbyte){
        unsigned long int reg;
        void __iomem * iom;
        unsigned int mask;
        unsigned int iovalue;
        unsigned short int localvalue;
        unsigned long int foxreg_address;
        unsigned long int tmp;
        unsigned int value;
        //Extract address from buffer
        reg = *(buf)&0x00ff;
        reg = reg|((*(buf+1)<<8)&0xff00);
        if(DEBUG) rtdm_printk(KERN_INFO "Reading data from address
        %d\n",(int)reg);
        //Set bus to output
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OWER);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        mask=0x3bc0c3f0;
        __raw_writel(mask, iom + PIO_OWDR);
        //Address
        tmp=0x0000;
        foxreg_address=0x0000;
        tmp=(reg&0x00f0)<<8;
        tmp=tmp|((reg&0x000f)<<4);
        tmp=tmp|((reg&0x0f00)>>8);
        tmp=tmp|((reg&0xf000)>>4);
        foxreg_address=tmp;
        iovalue=0x000f&foxreg_address;
        iovalue=iovalue|((0x00f0&foxreg_address)<<6);
        iovalue=iovalue|((0x3f00&foxreg_address)<<8);
        iovalue=iovalue|((0xc000&foxreg_address)<<16);
        //Write address
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_SODR);
```

```
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(iovalue, iom + PIO_ODSR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000080;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        //Set bus to input
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_ODR);
        __raw_writel(mask, iom + PIO_PER);
        //Read data
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000040;
        __raw_writel(mask, iom + PIO_CODR);
        __raw_writel(mask, iom + PIO_SODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        iovalue=0x00000000;
        iovalue=__raw_readl(iom + PIO_PDSR);
        iovalue=__raw_readl(iom + PIO_PDSR);
        mask=0xc03f3c0f;
        iovalue=mask&iovalue;
        localvalue=0x0000000f&iovalue;
        localvalue=localvalue|((0x00003c00&iovalue)>>6);
        localvalue=localvalue|((0x003f0000&iovalue)>>8);
        localvalue=localvalue|((0xc0000000&iovalue)>>16);
        value=0x0000;
        value=(localvalue&0x00f0)>>4;
        value=value|((localvalue&0xf000)>>8);
        value=value|((localvalue&0x0f00)<<4);
        value=value|((localvalue&0x000f)<<8);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000040;
        __raw_writel(mask, iom + PIO_CODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_CODR);
        return(value);
}
///Module entry/exit functions
module_init(rtdm_init);
module_exit(rtdm_exit);
```

After the declaration of the header files to be included we find the prototypes of the four typical device functions, open, close, read and write

```
static ssize_t rtdm_read_rt(...);
static ssize_t rtdm_write_rt(...);
static int rtdm_open_rt(...);
static int rtdm_close_rt(...);
```

according to the standard required by RTDM skin.

Later we find three macros which are used to specify some general information about the module, such as the author, the license, the device name

```
MODULE_LICENSE (MOD_LICENSE);
MODULE_AUTHOR (MOD_AUTHOR);
MODULE_DESCRIPTION(MOD_DESCRIPTION);
```

These information are not really needed by the driver but, once the module is compiled, they can be shown running the 'modinfo' command.

```
debarm:/home/foxbone-RTdriver-module# modinfo foxbone-RTdriver-module.ko
filename:       foxbone-RTdriver-module.ko
description:    Xenomai RT module driver for the Foxbone protocol on FOXg20 board
author:         Paolo Michelotti
license:        GPL
depends:
vermagic:       2.6.38 mod_unload ARMv5
debarm:/home/foxbone-RTdriver-module#
```

Figure 7.3. Output of the modinfo command.

Then we find the 'rtdm-device' struct; this is the basic data type which contains all the parameters of the module and allows to manage it. More in detail we find:

- Struct version : Current revision of RTDM structure;
- Device flags : Device addressing method (via ID, text name,etc);
- Context size : Size (in Byte) of the module context, i.e the additional space allocated for data passed/returned to/from the module;
- Device name : the name of the device driver;
- Open-rt : open() routine when called from a real-time context;
- Open-nrt : open() routine when called from a non real-time context;
- Ops : set of all the functions and methods made available by the driver, read(), write(), close(), both real-time and non real-time version;
- Driver name : the name of the module;
- Peripheral name : name of the physical peripheral associated to the driver;
- Proc name : the name of the module when loaded into the kernel, useful for debugging purpose.

After we have the declaration of the Init/Exit functions, the two procedures which are invoked automatically when the module is registered/unregistered

```
int __init rtdm_init(void);
void __exit rtdm_exit(void);
```

followed by the correspondents function bodies. Here we can see the procedures used to register/unregister the driver

```
ret = rtdm_dev_register(&device);
ret = rtdm_dev_unregister(&device,wait-time);
```

Then we see the bodies of the main functions, open(), close(), read() and write(). They are almost identical to the system calls, except from some shift ad masking operations needed due to the different way the input parameters are passed. At the end we see two standard macros which are used to register the Init/Exit function of the module.

```
module_init(rtdm_init);
module_exit(rtdm_exit);
```

Since a module is a particular structure which can be loaded dynamically and has to interact strictly with the kernel, it cannot be compiled simply as a standard application, but we will need to slightly modify the Makefile.

Makefile

```
#
# MAKEFILE template for Xenomai-RTDM  application
#
#Sources of currently running kernel
KSRC = /home/paolo/Documents/2.6.38-Xenomai/foxg20-linux-2.6.38-acme-2.6.38
#Sources of Xenomai
EXTRA_CFLAGS += -I/home/paolo/Documents/work/xenomai-2.6.0-rc5/include
#Modules to be built
obj-m += foxbone-RTdriver-module.o
all:
        $(MAKE) -C $(KSRC) SUBDIRS=$(shell pwd) modules
clear:
        $(MAKE) -C $(KSRC) SUBDIRS=$(shell pwd) clean
```

Looking at the main line of the Makefile we can see the main differences. The '-C' flag tells the system to move into the kernel source directory, pointed by the variable ksrc. Subdirs points to the actual directory, containing the module sources. The meaning of the keyword 'modules' is straightforward, it instruct the system about the kind of program it has to compile. Eventual libraries can be included with the extra_cflags variable, that in our case points to the RTDM sources. The module can be compiled simply running 'make'. Finally, a simple header file can be used to ease the use of the module introducing some alias, like previously done also for the system calls.

foxbone-driver.h

```
#include <rtdm/rtdm.h>
#define DEVICE_NAME FoxboneG20
int foxbone_open(void)
{
        return rt_dev_open(DEVICE_NAME,0);
};
int foxbone_close(void)
{
        return rt_dev_close(DEVICE_NAME);
};
```

```
int foxbone_read(unsigned long int reg)
{
        char buf[2];
        buf[0] = (reg&0x00ff);
        buf[1] = (reg&0xff00)>>8;
        return rt_dev_read(DEVICE_NAME,(void*)&buf,2);
};
int foxbone_write(unsigned long int reg, unsigned long int value)
{
        char buf[4];
        buf[0] = (reg&0x00ff);
        buf[1] = (reg&0xff00)>>8;
        buf[2] = (value&0x00ff);
        buf[3] = (value&0xff00)>>8;
        return rt_dev_write(DEVICE_NAME,(void*)&buf,4);
};
```

Once compiled, the module object will be created with extension '.ko'. It can be 'launched', i.e loaded into the kernel, running 'insmod foxbone-RTdriver-module.ko'. The list of currently active modules can be seen with the 'lsmod' command. The driver in the running state is depicted in the picture 7.4.

```
debarm:/home/foxbone-RTdriver-module# lsmod
Module                  Size  Used by
foxbone_RTdriver_module     1864  0
ipv6                   221082  10
debarm:/home/foxbone-RTdriver-module#
```

Figure 7.4. Foxbone real time driver module in the active state.

A small test application has been developed to check the driver functionalities. This program simply initializes the device, reads and writes some random numbers at random addresses ad then close it.

foxbone-RTdriver-test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <sys/types.h>
#include <native/task.h>
#include <rtdm/rtdm.h>
#include <rtdk.h>
#include "foxbone_driver.h"
RT_TASK test_task;
void test(void *arg)
{
        int device;
        unsigned long int reg,value;
        rt_task_set_mode(0,T_CONFORMING,NULL);
        device = foxbone_open();
        if(device < 0){
            rt_printf("Error opening device %s\n",DEVICE_NAME);
            exit(1);
        }
        reg = 5;
        value = 6;
        rt_printf("Writing %d at address %d\n",value,reg);
        foxbone_write(reg,value);
        reg = 100;
        value = 200;
        rt_printf("Writing %d at address %d\n",value,reg);
        foxbone_write(reg,value);
        reg = 15000;
        value = 20000;
        rt_printf("Writing %d at address %d\n",value,reg);
        foxbone_write(reg,value);
        reg = 128;
        rt_printf("Reading from address %d\n",reg);
        value = foxbone_read(reg);
        reg = 256;
        rt_printf("Reading from address %d\n",reg);
        value = foxbone_read(reg);
        reg = 32568;
        rt_printf("Reading from address %d\n",reg);
        value = foxbone_read(reg);
        device = foxbone_close();
        exit(0);
}
int main(int argc, char* argv[])
{
        /* Avoids memory swapping for this program */
        mlockall(MCL_CURRENT|MCL_FUTURE);
        rt_print_auto_init(1);
        rt_task_create(&test_task, "test", 0, 99, 0);
        rt_task_start(&test_task, &test, NULL);
        rt_task_join(&test_task);
        rt_task_delete(&test_task);
        return 0;
}
```

The developed driver module has been loaded into the kernel in substitution of the Foxbone system calls, to eliminate this source of domain switches. Now the FitRelay application is fully compliant with the Xenomai rules ad guidelines and can be considered fully real-time. It's performance will be analyzed in the next chapter.

Chapter 8

Real-time performance

In the final part of this thesis work, the performance of the FiTRelay application have to be analyzed. As said at the beginning, the main goal of this work is the development of a generic real time environment which combines predictable latencies and all the functionalities of a Linux based operating system. This framework will be used inside an automatic test environments to manage generators, measuring units, etc. In this way, the relays PCB used in this project has not to be considered as a final target for the application but just as a testbench to evaluate it's performance. We need to understand both potentialities and limits of the system to identify a set of fields of application which can be acceptable or not depending on the specific case.

In the following three versions of the program will be tested in different working conditions. The first one is the starting point program, without real-time capabilities; then we will move to the RT domain analyzing other two implementations of FiTRelay. The first, in which the main process in charge of managing the test logic is implemented as a high priority continuous thread; in this way it is expected to always keep control of the CPU, unless a higher priority process arrives. The second, in which this process is implemented as a periodic thread, without monopolizing the control of the CPU.

In all cases, performances have been analyzed in different working conditions. The workload of the application is represented by the frequency of the test sequence. The timebase of the test is implemented as a programmable counter in the FPGA, which can be configured and read form our application. The program must be able to follow the speed of the timebase without missing any tick of the counter and to correctly configure the test logic before the next test step arrives.The frequency of the timebase has been set to [1-10]kHz, a suitable range for the intended use of the application.

Every version of the program has been tested running a test sequence with the timebase in the range [1-10]kHz, step of 1kHz, 50 iterations each. A dedicated function in the main process was used to continuously read and check the value of the timebase counter, extracting statistics on the number of missed deadlines and writing them into a logfile after the execution. This simple function can be found in the appendix.

8.1 FiTRelay

Let's start with the original version of the application, without Xenomai and so without real-time capabilities. The performance of this application have been tested running the following test sequence:

- activate each relay for 20ms, one after the other;
- activate all relays together for 20ms;
- activate all relays together for a random time between 0 and 1 second;

	FiTRelay	
Frequency [KHz]	**Avg Deadline Miss**	**Avg miss value**
1	0,27	0,5
2	1	2,06
3	1,29	4,16
4	1,66	4,24
5	1,68	4,66
6	2,07	4,96
7	2,32	4,98
8	2,68	5,04
9	2,73	6,18
10	4,16	7,72

Figure 8.1. FiTRelay performance

Table 8.1 shows the results of the testing phase. The left column shows the timebase frequency. Avg deadline miss is the average number of deadline misses; avg miss value is the average number of tick skipped for every missed deadline. As it can be seen, the application can't be considered real-time cause missed deadlines are present in every frequency range. The number of misses grows with frequency; coherently, also the number of tick misses increases with the frequency. This was

the expected behavior; if we assume the miss value around constant in absolute time, more ticks will be lost when their frequency increases.

8.2 FiTRelay-RT

In this RT version of the program, the main process managing the test logic has been implemented as a continuous thread with high priority; we expect it to keep control of the CPU unless a higher priority process arrives. The performance of this application have been tested running the following program:

- activate each relay for 20ms, one after the other;
- activate all relays together for 20ms;
- activate all relays together for a random time between 0 and 1 second;

	FiTRelay-RT	
Frequency [KHz]	Avg Deadline Miss	Avg miss value
1	0,27	0,4
2	0,63	1,11
3	0,64	2,34
4	0,81	3,28
5	1,03	3,77
6	1,04	4,15
7	1,15	4,31
8	1,48	4,65
9	2,08	4,89
10	2,24	5,34

Figure 8.2. FiTRelay RT performance

Table 8.2 shows the results of the testing phase. The columns represents the same data as before. In this case we can see an improvement of the performances; the number of deadline misses is lower in all the frequency ranges. This happens thanks to the Xenomai scheduler, which allows the real-time task to preempt all the others and introduce more predictability in the system. However, misses have not been totally eliminated; still the application can't be considered hard real-time. In theory, we should expect the main thread to monopolize the control of the CPU, being the task with the highest priority; however, this does not happen. A possible explanation is the presence of some kernel tasks, needed for keeping the

operating system itself alive and correctly running. These tasks cannot be delayed forever and, at some point, will temporarily gain the control of the CPU, stealing it from real-time operations and causing a deadline miss. A possible solution is to implement the real-time task as a periodic thread; in this way some time will be free for other processes at every cycle. This solution has been implemented in the third version of the application and it's performance are analyzed in the next section.

8.3 FiTRelay-RT-periodic

In this version of the program, the main process managing the test logic has been implemented as a periodic thread, with a periodicity coherent with the timebase frequency. The performance of this application have been tested running the following program:

- activate each relay for 20ms, one after the other;
- activate all relays together for 20ms;
- activate all relays together for a random time between 0 and 1 second;

FiTRelay-RT-periodic			
Frequency [KHz]	**Task Period [µs]**	**Avg Deadline Miss**	**Avg miss value**
1	1000	0	0
2	500	0	0
3	333	0	0
4	250	0	0
5	200	1406,28	1,31
6	166	2851,96	1,68
7	142	4263,77	1,84
8	125	4815,54	2,14
9	111	4914	2,27
10	100	4987	2,73

Figure 8.3. FiTRelay RT periodic performance

Table 8.3 shows the results of the testing phase. As it can be seen, the application can be considered hard real time until around 4kHz; in this frequency range there are no deadline misses. Over this threshold, the number of misses grows very fastly.

This phenomena can be explained thinking about the periodicity of the thread. The period of the main real-time task has been set every time accordingly to the timebase frequency; 1000μs for 1kHz, 500μs for 2kHz, 250 μs for 4 kHz etc. The scheduler will continuously try to schedule the next iteration of the thread after the required time interval. However, if the time required by the application to perform the operations of one cycle is greater than the periodicity, we will have a deadline miss at almost every cycle. For this reason the number of misses increases a lot. From this we can also deduce the execution time of the process, that is around 250μs for every cycle; forcing a shorter period will certainly result in a failure.

8.4 Conclusion

At the end of this thesis work, we can sum up the main results achieved. Starting from a standard Linux kernel we have developed a real-time environment based on Xenomai and on a co-kernel approach. We have defined a set of rules, guidelines and methodologies which can help in the design of real-time application, exploiting all the potentialities of the Xenomai framework. The FiTRelay application, used to control and manage a prototype board equipped with a set of relays, has been built upon this layer; this testbench has been used to evaluate the performance of the system. Three version of the FiTRelay application has been developed. The first one, without real-time capabilities, was not able to respect the required deadlines. The second, based on the Xenomai skin and on a RT scheduler, despite an improvement of the performance, still was unable to fulfill hard real-time requirements. This program, however, can be used depending on the case for soft real-time requirements, when some percentage of deadline misses can be tolerated. In the last version the main thread has been made periodic, avoiding to monopolize the CPU; in this case the program was compliant with the hard real-time specification, at least in a subset of the frequency range under exam.

In the future, the designed real-time framework will be used, coherently with it's limits and capabilities, to drive and manage the hardware logic needed inside an automatic test environment, such as generators and measuring units, for the development of a new semiconductor power testing machine.

.1 Appendix A : kernel configuration file

.config file

```
#
# Real-time sub-system
#
CONFIG_XENOMAI=y
CONFIG_XENO_GENERIC_STACKPOOL=y
CONFIG_XENO_FASTSYNCH_DEP=y
CONFIG_XENO_FASTSYNCH=y
CONFIG_XENO_OPT_NUCLEUS=y
CONFIG_XENO_OPT_PERVASIVE=y
CONFIG_XENO_OPT_PRIOCPL=y
CONFIG_XENO_OPT_PIPELINE_HEAD=y
# CONFIG_XENO_OPT_SCHED_CLASSES is not set
CONFIG_XENO_OPT_PIPE=y
CONFIG_XENO_OPT_VFILE=y
CONFIG_XENO_OPT_PIPE_NRDEV=32
CONFIG_XENO_OPT_REGISTRY_NRSLOTS=512
CONFIG_XENO_OPT_SYS_HEAPSZ=256
CONFIG_XENO_OPT_SYS_STACKPOOLSZ=128
CONFIG_XENO_OPT_SEM_HEAPSZ=12
CONFIG_XENO_OPT_GLOBAL_SEM_HEAPSZ=12
CONFIG_XENO_OPT_STATS=y
CONFIG_XENO_OPT_DEBUG=y
CONFIG_XENO_OPT_DEBUG_NUCLEUS=y
CONFIG_XENO_OPT_DEBUG_XNLOCK=y
CONFIG_XENO_OPT_DEBUG_QUEUES=y
CONFIG_XENO_OPT_DEBUG_REGISTRY=y
CONFIG_XENO_OPT_DEBUG_TIMERS=y
CONFIG_XENO_OPT_DEBUG_SYNCH_RELAX=y
CONFIG_XENO_OPT_WATCHDOG=y
CONFIG_XENO_OPT_WATCHDOG_TIMEOUT=4
# CONFIG_XENO_OPT_SHIRQ is not set
CONFIG_XENO_OPT_SELECT=y
CONFIG_XENO_OPT_HOSTRT=y
#
# Timing
#
# CONFIG_XENO_OPT_TIMING_PERIODIC is not set
CONFIG_XENO_OPT_TIMING_VIRTICK=1000
CONFIG_XENO_OPT_TIMING_SCHEDLAT=0
#
# Scalability
#
# CONFIG_XENO_OPT_SCALABLE_SCHED is not set
CONFIG_XENO_OPT_TIMER_LIST=y
# CONFIG_XENO_OPT_TIMER_HEAP is not set
# CONFIG_XENO_OPT_TIMER_WHEEL is not set
#
# Machine
#
CONFIG_IPIPE_WANT_PREEMPTIBLE_SWITCH=y
CONFIG_XENO_HW_FPU=y
CONFIG_XENO_HW_UNLOCKED_SWITCH=y
#
# Interfaces
#
CONFIG_XENO_SKIN_NATIVE=y
CONFIG_XENO_OPT_NATIVE_PERIOD=0
CONFIG_XENO_OPT_NATIVE_PIPE=y
CONFIG_XENO_OPT_NATIVE_PIPE_BUFSZ=1024
CONFIG_XENO_OPT_NATIVE_SEM=y
CONFIG_XENO_OPT_NATIVE_EVENT=y
CONFIG_XENO_OPT_NATIVE_MUTEX=y
CONFIG_XENO_OPT_NATIVE_COND=y
CONFIG_XENO_OPT_NATIVE_QUEUE=y
CONFIG_XENO_OPT_NATIVE_BUFFER=y
CONFIG_XENO_OPT_NATIVE_HEAP=y
CONFIG_XENO_OPT_NATIVE_ALARM=y
CONFIG_XENO_OPT_NATIVE_MPS=y
# CONFIG_XENO_OPT_NATIVE_INTR is not set
CONFIG_XENO_OPT_DEBUG_NATIVE=y
```

```
CONFIG_XENO_SKIN_POSIX=y
CONFIG_XENO_OPT_POSIX_PERIOD=0
CONFIG_XENO_OPT_POSIX_SHM=y
CONFIG_XENO_OPT_POSIX_INTR=y
CONFIG_XENO_OPT_POSIX_SELECT=y
CONFIG_XENO_OPT_DEBUG_POSIX=y
# CONFIG_XENO_SKIN_PSOS is not set
# CONFIG_XENO_SKIN_UITRON is not set
# CONFIG_XENO_SKIN_VRTX is not set
# CONFIG_XENO_SKIN_VXWORKS is not set
# CONFIG_XENO_OPT_NOWARN_DEPRECATED is not set
CONFIG_XENO_SKIN_RTDM=y
CONFIG_XENO_OPT_RTDM_PERIOD=0
CONFIG_XENO_OPT_RTDM_FILDES=128
CONFIG_XENO_OPT_RTDM_SELECT=y
CONFIG_XENO_OPT_DEBUG_RTDM=y
CONFIG_XENO_OPT_DEBUG_RTDM_APPL=y
#
# Drivers
#
#
# Serial drivers
#
# CONFIG_XENO_DRIVERS_16550A is not set
#
# Testing drivers
#
CONFIG_XENO_DRIVERS_TIMERBENCH=y
# CONFIG_XENO_DRIVERS_KLATENCY is not set
# CONFIG_XENO_DRIVERS_IRQBENCH is not set
CONFIG_XENO_DRIVERS_SWITCHTEST=y
# CONFIG_XENO_DRIVERS_RTDMTEST is not set
#
# CAN drivers
#
# CONFIG_XENO_DRIVERS_CAN is not set
#
# ANALOGY drivers
#
# CONFIG_XENO_DRIVERS_ANALOGY is not set
#
# Real-time IPC drivers
#
# CONFIG_XENO_DRIVERS_RTIPC is not set
CONFIG_FREEZER=y
```

.2 Appendix B : System calls

foxbone_init.c

```
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/input.h>
#include <linux/gpio.h>
#include <mach/at91_pio.h>
#include <linux/slab.h>
#include<linux/linkage.h>
asmlinkage long sys_foxbone_init(void)
{
        void __iomem * iom;
        unsigned int mask;
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        __raw_writel(mask, iom + PIO_OWER);
        mask=0x3bc0c3f0;
        __raw_writel(mask, iom + PIO_OWDR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOA;
        mask=0x800002c0;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        mask=0x00000200;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        __raw_writel(mask, iom + PIO_SODR);
        mask=0x80000000;
        __raw_writel(mask, iom + PIO_CODR);
        mask=0x00000080;
        __raw_writel(mask, iom + PIO_CODR);
        mask=0x00000040;
        __raw_writel(mask, iom + PIO_CODR);
        return(0);
}
```

foxbonewrite.c

```
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/input.h>
#include <linux/gpio.h>
#include <mach/at91_pio.h>
#include <linux/linkage.h>
#include <linux/slab.h>
asmlinkage long sys_foxbonewrite(unsigned long int reg, unsigned long int value)
{
        void __iomem * iom;
        unsigned int mask;
        unsigned int data;
        unsigned int iovalue;
        unsigned int foxreg_address;
        unsigned int tmp;
        //Set bus to output
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OWER);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        //Address
        tmp=0x0000;
        foxreg_address=0x0000;
        tmp=(reg&0x00f0)<<8;
        tmp=tmp|((reg&0x000f)<<4);
        tmp=tmp|((reg&0x0f00)>>8);
        tmp=tmp|((reg&0xf000)>>4);
        foxreg_address=tmp;
        iovalue=0x000f&foxreg_address;
        iovalue=iovalue|((0x00f0&foxreg_address)<<6);
        iovalue=iovalue|((0x3f00&foxreg_address)<<8);
        iovalue=iovalue|((0xc000&foxreg_address)<<16);
        //Write address
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_SODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(iovalue, iom + PIO_ODSR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000080;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        //Data
        tmp=0x0000;
        data=0x0000;
        tmp=(value&0x00f0)<<8;
        tmp=tmp|((value&0x000f)<<4);
        tmp=tmp|((value&0x0f00)>>8);
        tmp=tmp|((value&0xf000)>>4);
```

```
        data=tmp;
        iovalue=0x000f&data;
        iovalue=iovalue|((0x00f0&data)<<6);
        iovalue=iovalue|((0x3f00&data)<<8);
        iovalue=iovalue|((0xc000&data)<<16);
        //Write data
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(iovalue, iom + PIO_ODSR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x80000000;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_CODR);
        return(0);
}
```

foxboneread.c

```
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/input.h>
#include <linux/gpio.h>
#include <mach/at91_pio.h>
#include <linux/slab.h>
#include<linux/linkage.h>
asmlinkage long sys_foxboneread(unsigned long int reg)
{
        void __iomem * iom;
        unsigned int mask;
        unsigned int iovalue;
        unsigned short int localvalue;
        unsigned long int foxreg_address;
        unsigned long int tmp;
        unsigned int value;
        //Set bus to output
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OWER);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        mask=0x3bc0c3f0;
        __raw_writel(mask, iom + PIO_OWDR);
        //Address
        tmp=0x0000;
        foxreg_address=0x0000;
        tmp=(reg&0x00f0)<<8;
        tmp=tmp|((reg&0x000f)<<4);
        tmp=tmp|((reg&0x0f00)>>8);
        tmp=tmp|((reg&0xf000)>>4);
        foxreg_address=tmp;
        iovalue=0x000f&foxreg_address;
        iovalue=iovalue|((0x00f0&foxreg_address)<<6);
        iovalue=iovalue|((0x3f00&foxreg_address)<<8);
        iovalue=iovalue|((0xc000&foxreg_address)<<16);
        //Write address
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_SODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(iovalue, iom + PIO_ODSR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000080;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        //Set bus to input
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
```

```
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_ODR);
        __raw_writel(mask, iom + PIO_PER);
        //Read data
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000040;
        __raw_writel(mask, iom + PIO_CODR);
        __raw_writel(mask, iom + PIO_SODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        iovalue=0x00000000;
        iovalue=__raw_readl(iom + PIO_PDSR);
        iovalue=__raw_readl(iom + PIO_PDSR);
        mask=0xc03f3c0f;
        iovalue=mask&iovalue;
        localvalue=0x0000000f&iovalue;
        localvalue=localvalue|((0x00003c00&iovalue)>>6);
        localvalue=localvalue|((0x003f0000&iovalue)>>8);
        localvalue=localvalue|((0xc0000000&iovalue)>>16);
        value=0x0000;
        value=(localvalue&0x00f0)>>4;
        value=value|((localvalue&0xf000)>>8);
        value=value|((localvalue&0x0f00)<<4);
        value=value|((localvalue&0x000f)<<8);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000040;
        __raw_writel(mask, iom + PIO_CODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_CODR);
        return(value);
}
```

foxbone_syscalls.h

```
#include<linux/unistd.h>
#define __NR_foxbone_init 370
#define __NR_foxbonewrite 371
#define __NR_foxboneread 372
long foxboneread(unsigned long int reg)
{
        return syscall(__NR_foxboneread,reg);
};
long foxbonewrite(unsigned long int reg, unsigned long int value)
{
        return syscall(__NR_foxbonewrite,reg,value);
};
long foxbone_init(void)
{
        return syscall(__NR_foxbone_init);
};
```

.3 Appendix C : Real time device driver module

foxbone-RTdriver-module.c

```
//-----------------------------------------------
// Xenomai based Real time module driver for
// the Foxbone protocol on the FOXG20 board
//-----------------------------------------------
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/ioport.h>
#include <asm/io.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/input.h>
#include <mach/at91_pio.h>
#include <linux/linkage.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/time.h>
#include <linux/gpio.h>
#include <linux/slab.h>
#include <rtdm/rtdm_driver.h>
#include <rtdk.h>
#define MOD_LICENSE "GPL"
#define MOD_AUTHOR "Paolo Michelotti"
#define MOD_DESCRIPTION "Xenomai RT module driver for the Foxbone protocol
on FOXg20 board"
#define DEVICE_NAME "FoxboneG20"
#define MODULE_NAME "FoxboneRT-driver"
#define DEBUG 1
typedef struct {
        char data[4];
} buffer_t;
static ssize_t rtdm_read_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, char* buf, size_t nbyte);
static ssize_t rtdm_write_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, char* buf, size_t nbyte);
static int rtdm_open_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, int oflags);
static int rtdm_close_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info);
MODULE_LICENSE (MOD_LICENSE);
MODULE_AUTHOR (MOD_AUTHOR);
MODULE_DESCRIPTION(MOD_DESCRIPTION);
//RTDM device descriptor
static struct rtdm_device device = {
        .struct_version = RTDM_DEVICE_STRUCT_VER,
        .device_flags = RTDM_NAMED_DEVICE,
        .context_size = sizeof( buffer_t),
        .device_name = DEVICE_NAME,
        .open_nrt = rtdm_open_rt,
        .open_rt = rtdm_open_rt,
        .ops = {
                .close_nrt = rtdm_close_rt,
                .close_rt = rtdm_close_rt,
                .read_nrt = rtdm_read_rt,
                .read_rt = rtdm_read_rt,
                .write_nrt = rtdm_write_rt,
                .write_rt = rtdm_write_rt,
        },
        .driver_name = MODULE_NAME,
        .peripheral_name = DEVICE_NAME,
        .proc_name = device.device_name,
};
//Module init/exit functions
int __init rtdm_init(void);
void __exit rtdm_exit(void);
//
```

```
//      Init function
//
int __init rtdm_init(void){
        int ret;
        ret = rtdm_dev_register(&device);
        if(DEBUG) rtdm_printk(KERN_INFO "Module %s registered with
        error code %d\n",MODULE_NAME,ret);
        return ret;
}
//
//      Exit function
//
void __exit rtdm_exit(void){
        int ret;
        ret = rtdm_dev_unregister(&device,1000);
        if(DEBUG) rtdm_printk(KERN_INFO "Module %s unregistered with
        error code %d\n",MODULE_NAME,ret);
}
//
//      Open function
//
static int rtdm_open_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, int oflags){
        void __iomem * iom;
        unsigned int mask;
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        __raw_writel(mask, iom + PIO_OWER);
        mask=0x3bc0c3f0;
        __raw_writel(mask, iom + PIO_OWDR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOA;
        mask=0x800002c0;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        mask=0x00000200;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        __raw_writel(mask, iom + PIO_SODR);
        mask=0x80000000;
        __raw_writel(mask, iom + PIO_CODR);
        mask=0x00000080;
        __raw_writel(mask, iom + PIO_CODR);
        mask=0x00000040;
        __raw_writel(mask, iom + PIO_CODR);
        return 0;
}
//
//      Close function
//
static int rtdm_close_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info){
        //Nothing to be done
        return 0;
}
//
```

```
//      Write function
//
static ssize_t rtdm_write_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, char* buf, size_t nbyte){
        unsigned long int reg;
        unsigned long int value;
        void __iomem * iom;
        unsigned int mask;
        unsigned int data;
        unsigned int iovalue;
        unsigned int foxreg_address;
        unsigned int tmp;
        //Extract address and data from buffer
        reg = *(buf)&0x00ff;
        reg = reg|((*(buf+1)<<8)&0xff00);
        value = *(buf+2)&0x00ff;
        value = value|((*(buf+3)<<8)&0xff00);
        if(DEBUG) rtdm_printk(KERN_INFO "Writing data %d at address
        %d\n",(int)value,(int)reg);
        //Set bus to output
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OWER);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        //Address
        tmp=0x0000;
        foxreg_address=0x0000;
        tmp=(reg&0x00f0)<<8;
        tmp=tmp|((reg&0x000f)<<4);
        tmp=tmp|((reg&0x0f00)>>8);
        tmp=tmp|((reg&0xf000)>>4);
        foxreg_address=tmp;
        iovalue=0x000f&foxreg_address;
        iovalue=iovalue|((0x00f0&foxreg_address)<<6);
        iovalue=iovalue|((0x3f00&foxreg_address)<<8);
        iovalue=iovalue|((0xc000&foxreg_address)<<16);
        //Write address
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_SODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(iovalue, iom + PIO_ODSR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000080;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        //Data
        tmp=0x0000;
        data=0x0000;
        tmp=(value&0x00f0)<<8;
        tmp=tmp|((value&0x000f)<<4);
        tmp=tmp|((value&0x0f00)>>8);
        tmp=tmp|((value&0xf000)>>4);
        data=tmp;
```

```
        iovalue=0x000f&data;
        iovalue=iovalue|((0x00f0&data)<<6);
        iovalue=iovalue|((0x3f00&data)<<8);
        iovalue=iovalue|((0xc000&data)<<16);
        //Write data
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(iovalue, iom + PIO_ODSR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x80000000;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_CODR);
        return(0);
}
//
//      Read function
//
static ssize_t rtdm_read_rt(struct rtdm_dev_context* context,
rtdm_user_info_t* user_info, char* buf, size_t nbyte){
        unsigned long int reg;
        void __iomem * iom;
        unsigned int mask;
        unsigned int iovalue;
        unsigned short int localvalue;
        unsigned long int foxreg_address;
        unsigned long int tmp;
        unsigned int value;
        //Extract address from buffer
        reg = *(buf)&0x00ff;
        reg = reg|((*(buf+1)<<8)&0xff00);
        if(DEBUG) rtdm_printk(KERN_INFO "Reading data from address
        %d\n",(int)reg);
        //Set bus to output
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom+=AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_OWER);
        __raw_writel(mask, iom + PIO_OER);
        __raw_writel(mask, iom + PIO_PER);
        mask=0x3bc0c3f0;
        __raw_writel(mask, iom + PIO_OWDR);
        //Address
        tmp=0x0000;
        foxreg_address=0x0000;
        tmp=(reg&0x00f0)<<8;
        tmp=tmp|((reg&0x000f)<<4);
        tmp=tmp|((reg&0x0f00)>>8);
        tmp=tmp|((reg&0xf000)>>4);
        foxreg_address=tmp;
        iovalue=0x000f&foxreg_address;
        iovalue=iovalue|((0x00f0&foxreg_address)<<6);
        iovalue=iovalue|((0x3f00&foxreg_address)<<8);
        iovalue=iovalue|((0xc000&foxreg_address)<<16);
        //Write address
```

```
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_SODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(iovalue, iom + PIO_ODSR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000080;
        __raw_writel(mask, iom + PIO_SODR);
        __raw_writel(mask, iom + PIO_CODR);
        //Set bus to input
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0xc03f3c0f;
        __raw_writel(mask, iom + PIO_IDR);
        __raw_writel(mask, iom + PIO_PUDR);
        __raw_writel(mask, iom + PIO_ODR);
        __raw_writel(mask, iom + PIO_PER);
        //Read data
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000040;
        __raw_writel(mask, iom + PIO_CODR);
        __raw_writel(mask, iom + PIO_SODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        iovalue=0x00000000;
        iovalue=__raw_readl(iom + PIO_PDSR);
        iovalue=__raw_readl(iom + PIO_PDSR);
        mask=0xc03f3c0f;
        iovalue=mask&iovalue;
        localvalue=0x0000000f&iovalue;
        localvalue=localvalue|((0x00003c00&iovalue)>>6);
        localvalue=localvalue|((0x003f0000&iovalue)>>8);
        localvalue=localvalue|((0xc0000000&iovalue)>>16);
        value=0x0000;
        value=(localvalue&0x00f0)>>4;
        value=value|((localvalue&0xf000)>>8);
        value=value|((localvalue&0x0f00)<<4);
        value=value|((localvalue&0x000f)<<8);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOA;
        mask=0x00000040;
        __raw_writel(mask, iom + PIO_CODR);
        iom = (void __iomem *)AT91_VA_BASE_SYS;
        iom += AT91_PIOB;
        mask=0x04000000;
        __raw_writel(mask, iom + PIO_CODR);
        return(value);
}
///Module entry/exit functions
module_init(rtdm_init);
module_exit(rtdm_exit);
```

foxbone_driver.h

```
#include <rtdm/rtdm.h>
#define DEVICE_NAME FoxboneG20
int foxbone_open(void)
{
        return rt_dev_open(DEVICE_NAME,0);
};
int foxbone_close(void)
{
        return rt_dev_close(DEVICE_NAME);
};
int foxbone_read(unsigned long int reg)
{
        char buf[2];
        buf[0] = (reg&0x00ff);
        buf[1] = (reg&0xff00)>>8;
        return rt_dev_read(DEVICE_NAME,(void*)&buf,2);
};
int foxbone_write(unsigned long int reg, unsigned long int value)
{
        char buf[4];
        buf[0] = (reg&0x00ff);
        buf[1] = (reg&0xff00)>>8;
        buf[2] = (value&0x00ff);
        buf[3] = (value&0xff00)>>8;
        return rt_dev_write(DEVICE_NAME,(void*)&buf,4);
};
```

Makefile

```
#
# MAKEFILE template for Xenomai-RTDM  application
#
#Sources of currently running kernel
KSRC = /home/paolo/Documents/2.6.38-Xenomai/foxg20-linux-2.6.38-acme-2.6.38
#Sources of Xenomai
EXTRA_CFLAGS += -I/home/paolo/Documents/work/xenomai-2.6.0-rc5/include
#Modules to be built
obj-m += foxbone-RTdriver-module.o
all:
        $(MAKE) -C $(KSRC) SUBDIRS=$(shell pwd) modules
clear:
        $(MAKE) -C $(KSRC) SUBDIRS=$(shell pwd) clean
```

foxbone-RTdriver-test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <sys/types.h>
#include <native/task.h>
#include <rtdm/rtdm.h>
#include <rtdk.h>
#include "foxbone_driver.h"
RT_TASK test_task;
void test(void *arg)
{
        int device;
        unsigned long int reg,value;
        rt_task_set_mode(0,T_CONFORMING,NULL);
        device = foxbone_open();
        if(device < 0){
            rt_printf("Error opening device %s\n",DEVICE_NAME);
            exit(1);
        }
        reg = 5;
        value = 6;
        rt_printf("Writing %d at address %d\n",value,reg);
        foxbone_write(reg,value);
        reg = 100;
        value = 200;
        rt_printf("Writing %d at address %d\n",value,reg);
        foxbone_write(reg,value);
        reg = 15000;
        value = 20000;
        rt_printf("Writing %d at address %d\n",value,reg);
        foxbone_write(reg,value);
        reg = 128;
        rt_printf("Reading from address %d\n",reg);
        value = foxbone_read(reg);
        reg = 256;
        rt_printf("Reading from address %d\n",reg);
        value = foxbone_read(reg);
        reg = 32568;
        rt_printf("Reading from address %d\n",reg);
        value = foxbone_read(reg);
        device = foxbone_close();
        exit(0);
}
int main(int argc, char* argv[])
{
        /* Avoids memory swapping for this program */
        mlockall(MCL_CURRENT|MCL_FUTURE);
        rt_print_auto_init(1);
        rt_task_create(&test_task, "test", 0, 99, 0);
        rt_task_start(&test_task, &test, NULL);
        rt_task_join(&test_task);
        rt_task_delete(&test_task);
        return 0;
}
```

Makefile

```
#
# MAKEFILE for foxbone-RTdriver module test
#
#-----------------------------#
#----XENOMAI configuration----#
#-----------------------------#
#Xenomai installation directory(default /usr/xenomai)
XENODIR = /usr/xenomai
#Set Xeno-config and wrap-link location
XENOCONFIG = $(shell PATH=$(XENODIR):$(XENODIR)/bin:$(PATH)
which xeno-config 2>/dev/null)
XENOWRAP = $(XENODIR)/bin/wrap-link.sh -v
#Xenomai compiler,flags and library
CC = $(shell $(XENOCONFIG) --cc)
XENOCFLAGS = $(shell $(XENOCONFIG) --skin=native --cflags)
$(shell $(XENOCONFIG) --skin=rtdm --cflags)
XENOLDFLAGS = $(shell $(XENOCONFIG) --skin=native --ldflags)
$(shell $(XENOCONFIG) --skin=rtdm --ldflags)
# This includes the library path of given Xenomai into the
#binary to make live easier for beginners if Xenomai's libs
#are not in any default search path.
XENOLDFLAGS+=-Xlinker -rpath -Xlinker $(shell $(XENOCONFIG) --libdir)
#**************************************#
CFLAGS += $(XENOCFLAGS)
LDFLAGS += $(XENOLDFLAGS)
APPLICATION = foxbone-RTdriver-test
all:
        $(CC) foxbone-RTdriver-test.c foxbone_driver.h -o $(APPLICATION)
        $(CFLAGS) $(LDFLAGS)
clean:
        @echo "Cleaning up directory."
        rm -f $(APPLICATION) *.o core *~
```

.4 Appendix D : Performance analysis

logfile.h

```
#ifndef LOGFILE_H
#define LOGFILE_H
#define LOGFILE_PATH "/home/FiTRelay-RT-periodic/FiTRelay-RT-periodic.csv"
#define MSG_LENGTH_MAX 1000
#define DATE_SIZE 23
/*******************************/
/*Print a message in the log file*/
/*******************************/
int logfile_msg(char* msg,...);
#endif // LOGFILE_H
```

logfile.c

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <stdarg.h>
#include <stdlib.h>
#include "logfile.h"
/*********************************/
/*Print a message in the log file*/
/*********************************/
int logfile_msg(char* msg,...){
  FILE* logfile;
  time_t current_time;
  struct tm* local_time;
  char* date;
  char* mystring;
  va_list argptr;
  date = (char*)malloc(DATE_SIZE);
  mystring = (char*)malloc(MSG_LENGTH_MAX);
  //Variable arguments management
  va_start(argptr,msg);
  vsprintf(mystring,msg,argptr);
  //Get current date in format [dd/mm/yyyy - hh:mm:ss]
  current_time = time(NULL);
  local_time = localtime(&current_time);
  strftime(date,DATE_SIZE,"[%d/%m/%y - %H:%M:%S]",local_time);
  logfile = fopen(LOGFILE_PATH,"a");
  if(logfile == NULL){
    return -1;
  }
  if(strlen(msg) > MSG_LENGTH_MAX || msg == NULL){
    fprintf(logfile,"%s\tError in the received log message.\n",date);
          fclose(logfile);
    return -1;
  }
  fprintf(logfile,"%s\t%s",date,mystring);
  fclose(logfile);
  va_end(argptr);
  return 0;
}
```

checktiming.h

```
#define DIFF_BUF_SIZE 5000
#define TICK_BUF_SIZE 15000
/***************************/
/* Manage timing statistics */
/***************************/
typedef struct {
    int total_access_numb;
    int total_miss_numb;
    int diff[DIFF_BUF_SIZE];
    int diff_position[DIFF_BUF_SIZE];
    float avg_diff_tick;
    int max_diff_tick;
    int max_diff_tick_pos;
    unsigned int ticks[TICK_BUF_SIZE];
    int domain_switch_numb;
    struct timespec t[TICK_BUF_SIZE];
    struct timespec t1[TICK_BUF_SIZE];
    struct timespec t2[TICK_BUF_SIZE];
}check_timing_strc;
int reset_check_timing(check_timing_strc* timing);
int print_check_timing(check_timing_strc* timing);
int print_check_timing_table(check_timing_strc* timing);
```

checktiming.c

```
#include <stdio.h>
#include <time.h>
#include "logfile.h"
#include "checktiming.h"
int reset_check_timing(check_timing_strc* timing){
    int i;
    timing->total_access_numb = 0;
    timing->total_miss_numb = 0;
    timing->domain_switch_numb = 0;
    for(i = 0; i < DIFF_BUF_SIZE ; i++){
        timing->diff[i] = 0;
        timing->diff_position[i] = 0;
    }
    for(i = 0 ; i < TICK_BUF_SIZE ; i++){
        timing->ticks[i] = 0;
        timing->t1[i].tv_sec = 0;
        timing->t1[i].tv_nsec = 0;
        timing->t2[i].tv_sec = 0;
        timing->t2[i].tv_nsec = 0;
        timing->t[i].tv_sec = 0;
        timing->t[i].tv_nsec = 0;
    }
    timing->avg_diff_tick = 0;
    timing->max_diff_tick = 0;
    timing->max_diff_tick_pos = 0;
    return 0;
}
int print_check_timing(check_timing_strc* timing){
    int i;
    logfile_msg("Total access number: %d\n",
        timing->total_access_numb);
    logfile_msg("Total tick-miss number: %d\n",
        timing->total_miss_numb);
    if(timing->total_miss_numb == 0) return 0;
    for(i = 0; i < timing->total_miss_numb ; i++)
        timing->avg_diff_tick += (float)timing->diff[i];
    timing->avg_diff_tick=timing->avg_diff_tick/timing->total_miss_numb;
    timing->max_diff_tick = timing->diff[0];
    timing->max_diff_tick_pos = timing->diff_position[0];
    for(i = 1 ; i < timing->total_miss_numb ; i++){
        if(timing->diff[i] > timing->max_diff_tick){
                timing->max_diff_tick = timing->diff[i];
                timing->max_diff_tick_pos = timing->diff_position[i];
                }
    }
    logfile_msg("Average tick diff: %f\n",
        timing->avg_diff_tick);
    logfile_msg("Max tick diff: %d at position tick %d\n",
        timing->max_diff_tick,timing->max_diff_tick_pos);
    logfile_msg("Number of domain switching: %d\n",
        timing->domain_switch_numb);
   return 0;
}
int print_check_timing_table(check_timing_strc* timing){
    int i;
    //Print in table format-> total_access_numb;
        //total_miss_numb;avg_diff_tick;max_diff_tick;
        //domain_switch_numb
    if(timing->total_miss_numb == 0){
        logfile_msg(";%d;%d;0;0;%d\n",
        timing->total_access_numb,timing->total_miss_numb,
        timing->domain_switch_numb);
        return 0;
```

```
    }
    for(i = 0; i < timing->total_miss_numb ; i++)
        timing->avg_diff_tick += (float)timing->diff[i];
    timing->avg_diff_tick=timing->avg_diff_tick/timing->total_miss_numb;
    timing->max_diff_tick = timing->diff[0];
    for(i = 1 ; i < timing->total_miss_numb ; i++){
        if(timing->diff[i] > timing->max_diff_tick){
            timing->max_diff_tick = timing->diff[i];
        }
    }
    logfile_msg(";%d;%d;%f;%d;%d\n",timing->total_access_numb,
        timing->total_miss_numb,timing->avg_diff_tick,
        timing->max_diff_tick,timing->domain_switch_numb);
    return 0;
}
```

Bibliography

[1] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef and Philippe Gerum *Building embedded Linux systems*. O'Reilly Media, Sebastopol, California, 2008.

[2] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman *Linux Device Drivers*. O'Reilly Media, Sebastopol, California, 2005

[3] Robert Love *Linux kernel development*. Addison-Wesley

[4] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne *Operating system concepts*. Wiley

[5] M. D. Marieska and P. G. Hariyanto and M. F. Fauzan and A. I. Kistijantoro and A. Manaf *On performance of kernel based and embedded Real-Time Operating System: Benchmarking and analysis*. International Conference on Advanced Computer Science and Information Systems, Dec 2011

[6] Xenomai website,
`https://xenomai.org/`

[7] Xenomai Gitlab,
`https://gitlab.denx.de/Xenomai/xenomai`

[8] LibExpat website,
`https://libexpat.github.io/`

[9] Acme System website,
`https://www.acmesystems.it/`

[10] FoxG20 kernel source,
`https://github.com/tanzilli/foxg20-linux-2.6.38/`

[11] Linux kernel archive,
`https://www.kernel.org/`

[12] Microsemi website,
`https://www.microsemi.com/`