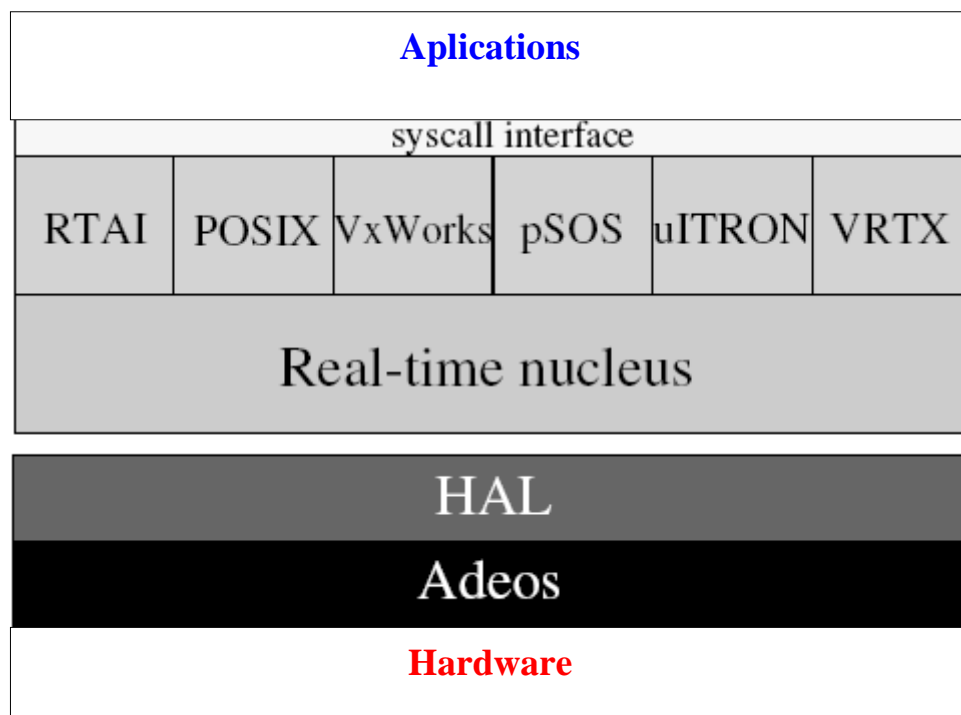


1 Introduction to real time applications development with Xenomai 2.5.x

Xenomai was developed from RTAI (Real Time Application Interface) fusion. It adds an abstraction layer to the Linux operating system, which supports launching and execution of real time tasks. To ensure that Linux does not interfere with real time tasks, all the Linux OS is launched as a low priority task by a real-time micro-kernel. Thus the real time tasks, launched by the micro-kernel, are never interrupted by Linux. In fact Linux only runs when no real-time task, with higher priority, is running.

Xenomai depends on Adeos for managing interrupts in real time (ipipe: interrupt pipeline). An application accesses the services provided by the real-time API calls through system calls, as shown in the diagram below:



O Xenomai supports, in addition to a native API (also called skin), others: rtai, rtdm, posix, psos, uitron, vrtx e vxworks.

1.1 Application implementation with Xenomai

1.1.1 Hello World

Unlike its predecessor, RTAI, real time tasks can be launched in user space. However it is also possible to launch them from a kernel module.

1.1.1.1 User space

The following user space application shows only the command line parameters:

```
#include <stdio.h>

int main(int argc, char** argv) {
    int i;

    for (i=0; i<argc; ++i) printf("%s\n", argv[i]);
    return 0;
}
```

To compile a real time application, a generic **makefile** can be used. This **makefile** runs script **xeno-config** to determine parameters required to compile and link a user space application with Xenomai support for the given API (skin). Note that it is only required to specify, in the first line, the C source file name without extension, ie. the name of the executable, and the skin in the second line:

```
target = hello
skin    = native

CFLAGS := $(shell xeno-config --skin=$(skin) --cflags)
LDFLAGS := $(shell xeno-config --skin=$(skin) --ldflags)

$(target): $(target).c
    $(CC) -o $@ $< $(CFLAGS) $(LDFLAGS)

clean:
    @rm $(target)
```

To run the application it is necessary to load some xenomai support, in the form of kernel modules, including the native API. This procedure is automated by the script **xeno-load** that search in **.runinfo** file the necessary information. This file has the following format:

```
<tag used in xeno-load>:<modules to load >:<actions ;;;>:<startup message>
```

For this example:

```
xenomaiHello:native:exec ./hello;popall:control_c
```

Which loads the native API and executes **./hello** (action: **exec ./hello**). After this action (the **hello** executable) end, or being interrupted with **ctrl-C**, removes all previously inserted modules (action: **popall**). The startup message is printed before the application is executed.

Normally to run an application:

```
xeno-load    <directory where the executable and .runinfo are>:<tag specified
              in .runinfo> {command line arguments}
```

from the directory where the executable and **.runinfo** file are:

```
xeno-load    ../xenomaiHello um dois tres
```

where the argument has the form **<executable directory>:<tag> { command line arguments }**

Command line arguments can also be passed in **.runinfo** file:

```
xenomaiHello:native:!../hello um dois três;popall:control_c
```

Note that **exec** can be substituted by **“!”** or even omitted in some cases.

1.1.1.2 Kernel space

In kernel space can also be launched real time tasks. However let start with a simpler module that just prints a message when loaded and another when unloaded. For a deeper introduction to modules drivers and hardware check the references at the end of this document.

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_ALERT */

int init_module(void) {
    printk("<1>Hello world 1.\n");
    return 0; // A non 0 return means init_module failed; module can't be
    loaded.
}

void cleanup_module(void) {
    printk(KERN_ALERT "Goodbye world 1.\n");
}
```

Function **int init_module(void)** is executed when module is inserted into the kernel and **void cleanup_module(void)** when unloaded.

A generic **makefile** can be used to compile this module:

```
obj-m += hello.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
```

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

To compile a module with Xenomai support another generic **makefile** is required:

```
obj-m      := hello.o
KDIR       := /lib/modules/$(shell uname -r)/build
PWD        := $(shell pwd)
EXTRA_CFLAGS := -I/usr/xenomai/include -I/usr/include/

all:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
```

The essential difference is marked **red**.

The **makefile** must be stored in the same directory as the source code. To compile just change to this directory and type **make** or **make all** at the command line. **make clean** deletes all files created when compiling.

To insert the module into the kernel (load it):	insmod hellomod.ko
To list all modules inserted in the kernel:	lsmod
To remove the module (unload it):	rmmod hellomod

To check messages added to kernel log with **printk**:

```
cat /var/log/kernel/warnings | tail -n3
or: dmesg | tail
```

.runinfo and **xeno-load** can also be used to insert and remove modules.

Let **.runinfo** be:

```
load:native:push hello.ko:Module loaded
remove::pop hello;popall:Module unloaded
```

To insert module:	xeno-load .:load
To list inserted modules:	lsmod
To remove module:	xeno-load .:remove

1.1.2 Launch real time task in user space

A real time task entry point is a function: **void f (void*)**

Tasks can be created with **rt_task_create (...)** and started with **rt_task_start (...)** or in just one step with **rt_task_spawn (...)**, as shown below:

```
#include <native/task.h>
#include <sys/mman.h>          //mloockall
```

```

#include <stdio.h>

#define TASK_PRIO  20 // 99 is Highest RT priority, 0 is Lowest
#define TASK_MODE  0      // No flags
#define TASK_STKSZ 0      // Stack size (use default)

RT_TASK task_desc;

void task_body (void *cookie) {
    printf ("hello world\n");
}

int main (int argc, char *argv[]) {
    int err;

    /* Ensures that process memory pages, present and future, will remain in
       main memory (will not be swaped), until munlockall() called.
    */
    mlockall(MCL_CURRENT|MCL_FUTURE);

    //create and start task
    err = rt_task_spawn(&task_desc, "simpleTask",
                       TASK_STKSZ, TASK_PRIO, TASK_MODE,
                       &task_body, NULL); //returns 0 if OK or <0 if error
    // ...
    if (!err) rt_task_delete(&task_desc);
    printf ("Good Bye\n");
    return 0;
}

```

The task **void task_body (void *cookie)** it is launched, which only prints a message.

Files **.runinfo** and **makefile**, for user space programs, can be similar to the ones presented already in **Error! Reference source not found..**

1.1.3 Periodic real time tasks in user space

Next example shows how to launch a task which is executed periodically. After the task is completed, it is removed from the ready tasks queue by the scheduler, until the next moment of execution. This task only prints the time elapsed each time it is executed. In **oneshot** timer mode, time is expressed in nanoseconds, while in periodic mode in **jiffies**.

A Jiffy is the smallest quantum of time for the kernel. To convert to seconds, simply multiply by the constant **HZ**. Thus in the second there are **HZ** jiffies. For example, if **HZ == 250** a Jiffy is 4 ms ($1/250 = 0,004$).

With Xenomai the duration of a Jiffy, in nanoseconds, can be programmed with the function:

```
rt_timer_set_mode(jiffy_duration_nseg);
```

If **jiffy_duration_nseg == 0**, ie **TM_ONESHOT** oneshot timer mode is selected.

The **oneshot** mode is more accurate at the expense of some overhead due to hardware reprogramming at each clock tick. Note that although time can be expressed in nanoseconds, this does not imply that the minimum time resolution is 1 nanosecond. So it does not mean that a task can be scheduled to run every nanosecond. The smallest amount of time depends on the system.

To select a task period, it should be used the following function defined in the Xenomai API:

```
rt_task_set_periodic(NULL, TM_NOW, rt_timer_ns2ticks(TASK_PERIOD));
```

In next example mode **oneshot** is selected and task period set to 0.5 seconds:

```
#include <stdio.h>
#include <stdlib.h>           //exit
#include <sys/mman.h>         //mlockall
#include <native/task.h>
#include <native/timer.h>

#define TASK_PRIO  20 // 99 is Highest RT priority, 0 is Lowest
#define TASK_MODE  0 // No flags
#define TASK_STKSZ  0 // default Stack size
#define TASK_PERIOD 500000000 // 0.5= 500000000 ns

RT_TASK tA;

void periodic_task (void *arg) {
    RTIME now, previous;

    previous= rt_timer_read();
    for (;;) {
        //task migrates to primary mode with xeno API call
        rt_task_wait_period(NULL); //deschedule until next period.
        now = rt_timer_read(); //current time

        //task migrates to secondary mode with syscall
        //so printf may have unexpected impact on the timing
        printf("Time elapsed: %ld.%04ld ms\n",
               (long)(now - previous) / 1000000,
               (long)(now - previous) % 1000000);
        previous = now;
    }
}

int main (int argc, char *argv[]) {
    int e1, e2, e3, e4;
    RT_TIMER_INFO info;

    mlockall(MCL_CURRENT|MCL_FUTURE);
    e1 = rt_timer_set_mode(TM_ONESHOT); // Set oneshot timer
    e2 = rt_task_create(&tA, "periodicTask", TASK_STKSZ, TASK_PRIO, TASK_MODE);

    //set period
    e3 = rt_task_set_periodic(&tA, TM_NOW, rt_timer_ns2ticks(TASK_PERIOD));
    e4 = rt_task_start(&tA, &periodic_task, NULL);
}
```

```

    if (e1 | e2 | e3 | e4) {
        fprintf(stderr, "Error launching periodic task....\n");
        rt_task_delete(&tA);
        exit(1);
    }

    printf("Press any key to end....\n");
    getchar();
    rt_task_delete(&tA);
    return 0;
}

```

Please note that first task is created with **rt_task_create()**, then its period selected with **rt_task_set_periodic()** and only then started with **rt_task_start()**.

If task created and launched in one step, and only then period set:

```

e2 = rt_task_spawn(&tA, "periodicTask",
                  TASK_STKSZ, TASK_PRIO, TASK_MODE,
                  &periodic_task, NULL);
e3 = rt_task_set_periodic(&tA, TM_NOW, rt_timer_ns2ticks(TASK_PERIOD));

```

during some time task would be executed without a defined period set.

However periodic tasks can be created and launched in one step with **rt_task_spawn()**, if period will be set at the start of the task itself:

```

void periodic_task (void *arg) {
    RTIME now, previous;

    rt_task_set_periodic(NULL, TM_NOW, rt_timer_ns2ticks(TASK_PERIOD));
    for (;;) {
        //...
    }
}

```

To deschedule task until next execution period, and thus freeing system resources:

```

rt_task_wait_period(NULL); //deschedule until next period

```

When a real time task is launched it runs in primary mode (Xenomai domain), taking precedence over tasks in the Linux domain (either in kernel space or user space). However if a call to the Linux API is required, ie. a system call such as **write** or **read**, the task automatically migrates to secondary mode (Linux domain). In this case the Linux kernel inherits the priority of the real time task, so that tasks in Linux domain may take priority over real-time tasks in secondary mode, depending on each task effective priority.

While a task in real time usually do not perform system calls, in this example time is printed on the console through a **printf**, so **write** a system call is issued, the task migrates to secondary domain and thus there is no guarantee that its period be fulfilled.

On the other hand a variable latency is always introduced. Its maximum or worst latency case can be determine, in each system, running the test:

```
/usr/xenomai/Bin/latency
```

Task automatically migrates back to the primary mode when a Xenomai service requiring the primary mode is called.

A task can be force to migrate to the primary mode with:

```
rt_task_set_mode(0, T_PRIMARY, NULL);
```

and to secondary mode with:

```
rt_task_set_mode(T_PRIMARY, 0, NULL);
```

It is worth noting than a real time task in kernel space will always be executed in Xenomay domain, since it is not possible to access directly the Linux API in the kernel.

1.1.4 Synchronized periodic tasks in user space

Next application launches two tasks: one writes a value in the I/O port 0x81 every 500ms, and the second reads the value from that port every 500 ms. If reader task period is increased to 1 second information is lost, since it is read only one of every two values written in the port:

```
#include <stdio.h>
#include <sys/mman.h>          //mlockall
#include <native/task.h>
#include <native/timer.h>
#include <sys/io.h>             //inb outb ioperm
#include <stdlib.h>             //exit

#define TASK_PRIO 20           // 99 is Highest RT priority, 0 is Lowest
#define TASK_MODE 0            // No flags
#define TASK_STKSZ 0           // default Stack size

#define MS 1000000              // 1 ms = 1000000 ns
#define TIMER_BASE 1000000     // TIMER_BAS = 1 ms
#define TASK_PERIOD 500*MS     // 500 MS = 0.5 s

#define PORT 0x81

RT_TASK tdw, tdr;
```



```

void write_port (void *arg) {
    int i=0;
    printf ("start writer\n");
    rt_task_set_periodic(NULL, TM_NOW, rt_timer_ns2ticks(TASK_PERIOD));
    for (;;) {
        rt_task_wait_period(NULL);
        outb(i++, PORT);
    }
}

void read_port (void *arg) {

    printf ("start reader\n");
    //2*TASK_PERIOD to loose one sample each time
    rt_task_set_periodic(NULL, TM_NOW, rt_timer_ns2ticks(TASK_PERIOD));
    for (;;) {
        rt_task_wait_period(NULL);
        printf ("%d\n", inb(PORT));
    }
}

int main (int argc, char *argv[]) {
    //acesso ao porto 0x81
    if(ioperm(PORT,1,1)) {
        perror("Port access denied\nMust be root\n");
        exit(1);
    }

    mlockall(MCL_CURRENT|MCL_FUTURE);
    rt_timer_set_mode(TM_ONESHOT); // Start timer base
    outb(128, PORT); //init PORT value out of sequence

    rt_task_spawn(&tdr, "readerTask",
        TASK_STKSZ, TASK_PRIO-1, TASK_MODE,
        &read_port, NULL); //retorna 0 se OK ou <0 se erro
    rt_task_spawn(&tdw, "writerTask",
        TASK_STKSZ, TASK_PRIO, TASK_MODE,
        &write_port, NULL); //retorna 0 se OK ou <0 se erro

    getchar();
    rt_task_delete(&tdw);
    rt_task_delete(&tdr);
    return 0;
}

```

Task period is set with:

```

rt_task_set_periodic(NULL, sttime,
    rt_timer_ns2ticks(TASK_PERIOD)); //set period 0.5 s

ou

rt_task_set_periodic(NULL, sttime,
    rt_timer_ns2ticks(2*TASK_PERIOD)); //set period 1 s (loose halve
//the samples)

```

Note that the reader task is started first, see the order in the **main()** function. So it will read a value from the I/O port before the writing task put the initial value of the counter: zero. A task can be started with a delay. The following changes make the reader task start be delayed by 100 ms. This may be useful to ensure that any work is performed by a producer task before the consumer is started

```
void read_port (void *arg) {
    RTIME sttime;

    printf ("reader started\n");
    // start time: 100 millisecond from now
    sttime = rt_timer_read()+ rt_timer_ns2ticks(100*MS);
    rt_task_set_periodic(NULL, sttime,
        rt_timer_ns2ticks(TASK_PERIOD)); //set period
    for (;;) {
        //... idêntico
    }
}
```

However it may not be possible to determine the time required for a task to accomplish its job. In these cases semaphores can be used to synchronize a producer and a consumer:

```
#include <native/sem.h>
RT_SEM semA;

void write_port (void *arg) {
    //...
    outb(i++, PORT);
    rt_sem_v(&semA); //increment semA counter (data written)
}

void read_port (void *arg) {
    printf ("start reader\n");
    for (;;) {
        rt_sem_p(&semA, TM_INFINITE); //decrement semA counter wait for writer
        printf ("%d\n", inb(PORT));
    }
}

int main (int argc, char *argv[]) {

    int e1 = rt_sem_create(&semA, "SemBinA", 0, S_FIFO);
    if (e1) {
        fprintf(stderr, "Error creating semaphore....\n");
        exit(1);
    }

    //...

    rt_sem_delete(&semA);
    return 0;
}
```

Yet another semaphore could be used, to ensure that the writer task only writes the next value of the counter after the reader task have read the previous. Or even after print it on the standard output. However this last case could generate some unpredictability as writing to the standard output is not accomplished in real time.

1.1.5 Periodic real time tasks in kernel space

In this example a periodic producer task running in kernel space writes the value of a counter in an I/O port, which is read and printed by a consumer task running in user space. As in the previous example a Semaphore is used to signal the consumer that can read data.

As stated in Xenomai roadmap, starting at future Xenomai version 3, *skins* will not export their interface to kernel modules anymore, being the exception the RTDM device driver API, which must be used from kernel space for writing real-time device drivers. So since version 2.5 deprecated uses of the Xenomai APIs from kernel space will cause warning messages to be issued at build time:

```
/usr/xenomai/include/native/task.h: In function "rt_task_spawn":
/usr/xenomai/include/native/task.h:317: warning: "rt_task_create" is deprecated
(declared at /usr/xenomai/include/native/task.h:250)
```

However those kernel interfaces will remain fully functional throughout the Xenomai v2.x series, until they are eventually removed in Xenomai 3. Stating this, the following module will not compile with version 3, since it uses the native API. Changes can however be made to its code to use the RTDM API instead of the native.

So, the module that implements the producer **writeportK.c**, for Xenomai version 2.x:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <native/task.h>
#include <native/sem.h>
#include <native/timer.h>

#define TASK_PRIO 20 // 99 is Highest RT priority, 0 is Lowest
#define TASK_MODE 0 // No flags
#define TASK_STKSZ 0 // default Stack size

#define MS 1000000 //1000000 ns = 1 ms
#define US 1000 //1000 ns = 1 us (micro sec)
#define TASK_PERIOD 250000*US // in micro seconds = 250 ms

#define PORT 0x81

RT_TASK td_pw; //Real time task pointer
RT_SEM semA; //semaphore descriptor
```

```

void portwrite (void *arg) {
    static unsigned char count=0;

    rt_task_set_periodic(NULL, TM_NOW,
        rt_timer_ns2ticks(TASK_PERIOD)); //set period
    for (;;) {
        rt_task_wait_period(NULL); //deschedule until next period
        outb (count++, PORT);        //outb takes aprox. 1 Micro sec.
        //printf("%d, ", count);    //check with dmesg
        if (count==100) count=0;
        rt_sem_v(&semA); //increment semA counter
    }
}

int init_module(void) {
    printk("\nwriteportK sucessfully loaded with\n");
    rt_timer_set_mode(TM_ONESHOT); // Start oneshot timer
    if (rt_sem_create(&semA, "semA", 0, S_FIFO)) {
        printk("Error creating semaphore....\n");
        return -ENOMEM;
    }
    return rt_task_spawn(&td_pw, "portwrite", TASK_STKSZ, TASK_PRIO,
        TASK_MODE, &portwrite, NULL);
}

//retorna 0 se OK ou <0 se erro
}

void cleanup_module(void) {
    rt_task_delete(&td_pw);
    rt_sem_delete(&semA);
    printk("\nportwriter unloaded\n");
}

```

The user space consumer **readport.c**:

```

#include <stdio.h>
#include <sys/mman.h>        //mlockall
#include <native/task.h>
#include <native/sem.h>
#include <sys/io.h>          //inb outb ioperm
#include <stdlib.h>          //exit
#include <signal.h>
#include <sys/mman.h>        //mlockall

#define PORT 0x81

RT_TASK tdr;
RT_SEM semA;

void end(int sig) { //ctrl-c handler
    rt_sem_unbind(&semA);
    printf ("Ended!\n");
    exit(0);
}

int main () {
    int err;

    signal (SIGINT, end); //ctrl-c handler

```

```

        if(ioperm(PORT,1,1)) {          //ioperm: obtain hardware i/o access permission
            puts("error");                //for io port DATAPORT
            exit(1);                      //No need in mode kernel
        }

mlockall(MCL_CURRENT|MCL_FUTURE);
rt_task_shadow (&tdr, "readport", 50, 0); //migrate to Xenomai domain

err=rt_sem_bind(&semA, "semA", TM_INFINITE);
if (err) {
    fprintf(stderr, "Error %d binding to semA\n", err);
    exit(1);
}

while(1) {                                //press ctrl-c to quit
    rt_sem_p(&semA, TM_INFINITE); //decrement semA counter wait for writer
    printf ("%03d\n", inb(PORT));
}
}

```

To generate the module a generic **Makefile** with Xenomai support, as already presented, can be used:

```

obj-m    := writeportK.o

KDIR     := /lib/modules/$(shell uname -r)/build
PWD      := $(shell pwd)
EXTRA_CFLAGS := -I/usr/xenomai/include -I/usr/include/

default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean

```

To generate the user space application a **MakefileUS** with Xenomai support, as also already presented, can be used:

```

target =  readport
skin    =  native

CFLAGS := $(shell xeno-config --skin=$(skin) --cflags)
LDFLAGS := $(shell xeno-config --skin=$(skin) --ldflags)

$(target): $(target).c
    $(CC) -o $@ $< $(CFLAGS) $(LDFLAGS)

clean:
    @rm $(target)
    @rm *~

```

The script **build.sh** can be used to call both Makefiles:

```
make
make -f MakefileUS
```

And the **.runinfo** file to insert **writeportK** module and execute **readport** user space application until <ctrl-c> pressed:

```
#<target tag supplied xeno-load>:<module to load>:<actions ; ; ;>:<start message>
port:native:push writeportK;!. /readport;popall:control_c
```

It is worth noting that **popall** removes not only native API module **native**, but also modules inserted with **push** (in this case **writeportK**).

Now calling script **run.sh**:

```
xeno-load .:port
```

all the application (user and kernel parts) are executed:

```
. run.sh
```

1.2 User space real time tasks programmed with C++

To compile a C++ application with support to Xenomai in user space, it is only required to indicate in the **makefile** that C++ compiler must be used: **g++**. This is not so simple for kernel modules. Assuming that a C++ implementation of the above **readport.c** is required:

```
#include <iostream>
using namespace std;

#include <sys/mman.h> //mlockall
#include <native/task.h>
#include <native/sem.h>
#include <sys/io.h> //inb outb ioperm
#include <stdlib.h> //exit
#include <signal.h> //
#include <sys/mman.h> //mlockall

#define PORT 0x81

RT_TASK tdr;
RT_SEM semA;

void end(int sig) {
    rt_sem_unbind(&semA);
    cout << "Ended!\n";
    exit(0);
}

int main () {
    int err;

    signal (SIGINT, end);
```

```

    if(ioperm(PORT,1,1)) {      //ioperm: obtain hardware i/o access permission
        cout << "error\n";      //for io port DATAPORT
        exit(1); }              //No need in mode kernel

    mlockall(MCL_CURRENT|MCL_FUTURE);
    rt_task_shadow (&tdr, "readport", 50, 0); //migrate to Xenomai domain

    err=rt_sem_bind(&semA, "semA", TM_NONBLOCK);
    if (err) {
        cerr << "Error binding to semA = " << err << "\n";
        exit(1);
    }

    while(1) {                  // press ctrl-c to quit
        rt_sem_p(&semA, TM_INFINITE); //decrement semA counter wait for writer
        int b = inb(PORT);
        cout << b << "\n";
        //if (err>3) break;
    }
}

```

Add one line to the start of **MakefileUS** to specify which compiler to use:

```

CC = g++
#ou CC := g++

```

Run again **build.sh** e **run.sh**.

1.3 Conclusion

After this short introduction to Xenomai it is recommended that the examples from STR lectures will also be implemented.

1.4 Bibliography

Xenomai home page:

http://www.xenomai.org/index.php/Main_Page

Xenomai Docs home:

<http://www.xenomai.org/documentation/>

<http://www.xenomai.org/documentation/xenomai-2.5/>

Xenomai API 2.5.x:

<http://www.xenomai.org/documentation/xenomai-2.5/html/api/index.html>

Xenomai API tour:

<http://www.xenomai.org/documentation/xenomai-2.5/pdf/Native-API-Tour-rev-C.pdf>

Xenomai roadmap: changes in version 3:

http://www.xenomai.org/index.php/Xenomai:Roadmap#What_Will_Change_With_Xenomai_3

Xenomai Local System Docs:

<Xenomai install dir>/share/doc/xenomai

Xenomai manpages:

man xeno-load, xeno-test, xeno-info, xeno-config, runinfo

Xenomai demos:

<Xenomai install dir>/share/xenomai/testsuite (executables)

<Xenomai source dir>/src/testsuite (source code)

where normally:

<Xenomai install dir> = /usr/xenomai

<Xenomai source dir> = /usr/src/xenomai-2.5.x

Xenomai install on Debian Linux:

<http://www.csg.is.titech.ac.jp/~lenglet/howtos/realtimelinuxhowto/>

Kernel modules and drivers:

<http://www.deei.fct.ualg.pt/PIn>

<http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/Modules.pdf>

<http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/Module-HOWTO.pdf>

<http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/IO-Port-Programming.pdf>

<http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf>