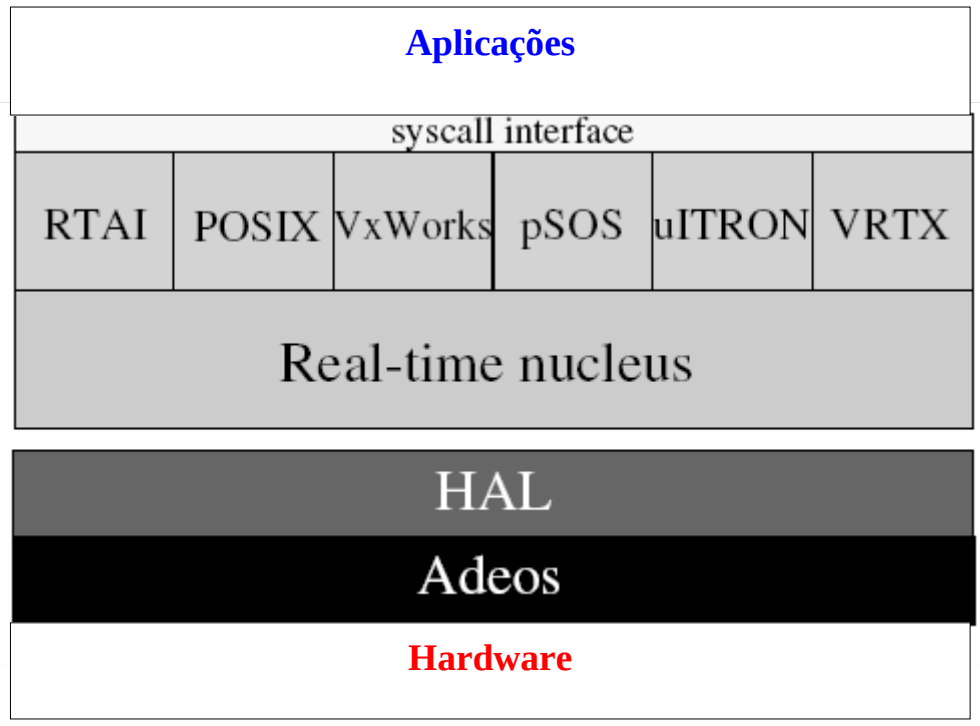
1 Introdução ao desenvolvimento de aplicações de tempo real no Xenomai 2.5.x

O Xenomai foi desenvolvido a partir da versão *fusion* do RTAI (Real Time Application Interface) e adiciona uma camada de abstracção (**Adeos**) ao sistema Linux, que disponibiliza comandos para lançamento e execução de tarefas em tempo real. De modo a garantir que o Linux não interfere com as tarefas de tempo real, todo o SO Linux é lançado como uma tarefa de baixa prioridade por um *micro-kernel* de tempo real. Deste modo as tarefas de tempo real lançadas pelo *micro-kernel* nunca são interrompidas pelo Linux. De facto o Linux só se executa quando nenhuma tarefa de tempo-real, com alta prioridade, se está a executar.

O Xenomai depende do **Adeos** para gerir interrupções em tempo real (**ipipe**: interrupt pipeline). Uma aplicação acede aos serviços da API de tempo real através de chamadas ao sistema, como mostra o diagrama seguinte:



O xenomai suporta além de uma API (ou *skin*) nativa (native), também outras APIs: rtai, rtdm, posix, psos, uitron, vrtx e vxworks.

1.1 Implementação de uma aplicação em Xenomai

1.1.1 Hello World

Ao contrário do seu antecessor RTAI, as tarefas de tempo real podem ser lançadas no espaço do utilizador. No entanto é também possível lança-las a partir de um módulo do kernel.

1.1.1.1 Espaço de utilizador

A aplicação seguinte, em espaço de utilizador, mostra apenas os parâmetros da linha de comandos:

```
#include <stdio.h>

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

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

Para compilar uma aplicação de tempo real, pode-se usar um **makefile** genérico, que interroga o script **xeno-config** pelos parâmetros necessários para compilação e *linkagem* de uma aplicação com suporte Xenomai para espaço de utilizador, para a API (*skin*) indicada. Repare-se que basta especificar na primeira linha o nome do ficheiro C fonte, sem extensão, isto é o nome do executável e na segunda a API (*skin*):

```
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)
```

Para executar a aplicação é necessário carregar algum suporte para o Xenomai, na forma de módulos do kernel, nomeadamente a API nativa. Este procedimento é automatizado pelo script **xeno-load** que busca no ficheiro **.runinfo** a informação necessária. Este ficheiro tem o seguinte formato**:**

```
<etiqueta usada em xeno-load>:<modulos a carregar>:<acções ;;;>:<mensagem inicial>
```

Para este caso o formato pode ser:

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

Que carrega a API nativa, executa **./hello** (acção: **exec ./hello)** e após a acção anterior ter terminado, ou ser interrompido com crtl-C, remove todos os módulos inseridos previamente (acção: **popall)**. A mensagem inicial indica que deve ser pressionado <control-c> para terminar a aplicação.
Na maior parte dos casos para executar:

```
xeno-load   <directório onde se encontra o executável e .runinfo>:<etiqueta
            indicada em .runinfo> {argumentos da linha de comando}
```

por exemplo a partir do directório do executável e que também contém .runinfo:

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

onde o argumento tem a forma <directório do executável>:<etiqueta> {argumentos da linha de comando da aplicação indicada na etiqueta xenomaiHello}
Também se podem passar argumentos da linha de comandos para a aplicação em .runinfo:

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

exec pode ser substituído por "!" e mesmo em alguns casos omitido.

1.1.1.2 Espaço do kernel

No espaço do Kernel, podem também ser lançadas tarefas de tempo real. Abaixo tem-se a implementação de um simples módulo do kernel que apenas imprime uma mensagem quando carregado e outra quando descarregado, não sendo lançada nenhuma tarefa de tempo real. Para uma melhor abordagem sobre módulos, drivers e hardware consultar as referências indicadas no final desta ficha.

```
#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");
}
```

A função **int init_module(void)** é executada quando o módulo é carregado no Kernel, e **void cleanup_module(void)** quando o módulo é removido.

Pode-se usar um **makefile** genérico para compilar módulos do kernel:

```
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
```

Ou embora não necessário para este exemplo, pretendendo-se incluir suporte para o Xenomai, o **makefile** poderá ser:

```
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
```

A diferença essencial está indicada a **vermelho**.

Bastando escrever **make** ou **make all** na linha de comando para compilar e **make clean** para limpar todos os ficheiro de compilação.

Para inserir o módulo no kernel pode-se usar: **insmod hellomod.ko**
Para listar módulos inseridos no kernel usar: **lsmod**
Para remover o módulo do kernel usar: **rmmod hellomod**

Para ver as mensagens enviadas com **printk** usar:

```
cat /var/log/kernel/warnings | tail -n3
```

ou:

```
dmesg | tail
```

Pode ser também usado **.runinfo** e **xeno-load** para inserir e remover módulos. Este script usa **insmod** e **rmmod** para esse efeito. Seja **.runinfo**:

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

Para inserir o módulo: **xeno-load .:load**

Para listar módulos: **lsmod**
Para remover o módulo: **xeno-load .:remove**

1.1.2 Lançamento de tarefa em espaço de utilizador

O ponto de entrada no código de uma tarefa deve ser definido numa função do tipo: **void f (void*)**
A tarefa pode ser criada com **rt_task_create (...)** e iniciada com **rt_task_start (...)** ou num só passo com **rt_task_spawn** (...), como mostra o código seguinte:

```
#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;

    /* Faz com que as páginas correntes e futuras do processo permaneçam
       na memória principal até que seja chamado munlockall()
    */
    mlockall(MCL_CURRENT|MCL_FUTURE);

    //cria tarefa e lança-a
    err = rt_task_spawn(&task_desc, "simpleTask",
                        TASK_STKSZ, TASK_PRIO, TASK_MODE,
                        &task_body, NULL); //retorna 0 se OK ou <0 se erro
    // ...
    if (!err) rt_task_delete(&task_desc);
    printf ("Good Bye\n");
    return 0;
}
```

É lançada a tarefa **void task_body (void *cookie)** que apenas imprime uma mensagem.
Os ficheiros **.runinfo** e **makefile**, para programas no espaço de utilizador, podem ser semelhantes aos apresentados no ponto 1.1.1.1.

1.1.3 Tarefas periódica em espaço de utilizador

No próximo exemplo é apresentado código para lançar uma tarefa periodicamente. Após a tarefa estar concluída é retirada da fila de tarefas prontas a executar, pelo *scheduler* até ao próximo instante de execução. Essa tarefa apenas imprime o tempo passado cada vez que é executada. No modo timer **oneshot**, a contagem de tempo é efectuada em nanosegundos, ao passo que no modo **periodic** em *jiffies***.**

Um **jiffy** em Linux representa o menor pacote de tempo do *kernel*. Para converter para segundos pode-se multiplicar pela constante **HZ**. Assim no segundo existem **HZ** *jiffies*. Por exemplo se HZ tomar o valor 250 um *jiffy* representa 4 ms (1/250=0.004).
Em Xenomai a duração de um *jiffy* em nanosegundos poderá ser programada com a função:

```
rt_timer_set_mode(duração_jiffy_nseg);
```

Se o valor do argumento for 0, isto é TM_ONESHOT o modo **oneshot** é seleccionado.
O modo **oneshot** é mais preciso à custa de alguma sobrecarga devido a programação de *hardware* em cada clock tick. Repare-se que embora os tempos possam ser expressos em nanosegundos, não implica que o mínimo de resolução temporal é de 1 nanosegundo. Ou de outra forma, não quer dizer que possamos programar uma tarefa para que se execute em cada nanosegundo. O menor *quantum* de tempo depende do sistema. Além disso terá de se contar com o tempo de execução no pior caso da própria tarefa.

É aconselhado o uso das funções da API para seleccionar o período da tarefa:

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

No exemplo é seleccionado o modo **oneshot**, e o período de execução da tarefa é seleccionado para 0.5 segundos.

```
#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= 50000000 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(); //cureent 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;
}
```

Repare-se que a tarefa é criada primeiro com **rt_task_create()**, depois seleccionado o seu período com **rt_task_set_periodic()** e só depois iniciada com **rt_task_start()**.

Se fosse usada uma função para criar a tarefa e lança-la de uma só vez e só depois seleccionado o seu período:

```
        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));
```

Durante algum tempo a tarefa seria executada sem o período seleccionado, até que **rt_task_set_periodic()** terminasse a sua execução. Se for pretendido usar **rt_task_spawn()**, o período deve ser seleccionado no inicio da função da tarefa, com o 1º argumento igual a NULL para que seja seleccionado o período da tarefa corrente:

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

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

Para que a tarefa de tempo real não consuma tempo de CPU enquanto espera pelo próximo período de execução, deve ser usada a função:

```
rt_task_wait_period(NULL); //deschedule until next period
```

Esta função retira a tarefa do *scheduler* até ao próximo período de execução.

Quando uma tarefa de tempo real é lançada executa-se em modo primário (domínio Xenomai), tendo prioridade sobre tarefas no domínio Linux (quer seja em espaço do *kernel* ou espaço do utilizador). No entanto se for necessário efectuar uma chamada à API do Linux, isto é uma chamada ao sistema ou *system call* como por exemplo um **write** ou **read,** a tarefa migra automaticamente para modo secundário (domínio Linux). Neste caso o kernel Linux herda a prioridade da tarefa de tempo real, e as tarefas em domínio Linux podem tomar prioridade sobre tarefas de tempo real em modo secundário dependendo da sua prioridade efectiva.
Assim embora uma tarefa em tempo real normalmente não efectue chamadas ao sistema, neste caso de demonstração é impresso o valor do tempo através de um **printf** de modo que é efectuada uma chamada **write**, não havendo garantia sobre o período de execução.
Por outro lado existe sempre uma latência variável, embora seja possível determinar o seu máximo, isto é o pior caso de latência, para cada sistema executando o teste:

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

A tarefa migra automaticamente para o modo primário assim que for chamado um serviço do Xenomai que requeira o modo primário. É possível fazer uma tarefa a migrar para o modo primário com:

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

e para o modo secundário com:

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

Deve ser referido que uma tarefa de tempo real em modo kernel executa-se sempre no domínio Xenomai, visto que no kernel não é possível aceder directamente à API do Linux.

1.1.4 Tarefas periódicas em espaço de utilizador sincronizadas

A aplicação seguinte lança duas tarefas: uma escreve um valor no porto de E/S 0x81 a cada 500ms, e a segunda lê o valor desse porto a cada 500 ms. Variando o período da tarefa de leitura para 1 segundo verifica-se que perde informação, pois lê apenas um de cada dois valores colocados no porto.

```
#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;
}
```

A periodicidade das tarefa é seleccionada com:

```
	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)
```

Repare-se que como a tarefa de leitura é iniciada primeiro, veja-se a ordem de lançamento na função **main()**, vai ler um valor do porto antes da tarefa de escrita colocar o valor inicial do contador: zero.

Uma tarefa pode ser iniciada com um atraso. As alterações seguintes fazem com que a tarefa de leitura seja atrasada 100 ms em relação à de escrita. Isto poderá ser útil para garantir que algum trabalho é efectuado por uma tarefa produtora antes de se iniciar a consumidora.

```
	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
		      }
	}
```

De qualquer forma este atraso não garante que o produtor termine o seu trabalho. Podem ser usados semáforos para este efeito:

```
#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;
}
```

Ainda poderia usar-se outro semáforo para garantir que a tarefa de escrita só escreveria o próximo valor do contador após a tarefa de leitura o ler e o escrever na saída padrão. No entanto isto poderia gerar imprevisibilidade pois a escrita na saída padrão não é feita em tempo real.

1.1.5 Tarefas periódicas em espaço do núcleo

Neste exemplo uma tarefa periódica produtora no espaço do núcleo escreve o valor de um contador num porto de E/S, que é lido e impresso por uma tarefa consumidora no espaço do utilizador. Tal como no exemplo anterior um semáforo indica ao consumidor que pode ler os dados.

Como indicado no *Xenomai roadmap*, a partir da versão 3, as APIs deixarão de ser exportadas para o espaço do *kernel*, com excepção da RTDM, que tem de ser usada no espaço do *kernel* para a escrita de *drivers* de tempo real. Assim a partir da versão 2.5, uso das APIs no *kernel* gerará alertas durante a compilação:

```
/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)
```

No entanto estas funcionalidades permanecerão activas no espaço do kernel nas versões 2.x, até serem eventualmente removidas na versão 3. Posto isto, o módulo seguinte não poderá ser compilado com a versão 3, no entanto o seu código poderá ser modificado para usar a RTDM API em vez da nativa.

Assim o módulo **writeportK.c** que implementa o produtor, para a versão 2.x do Xenomai, é:

```
#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.
      //printk("%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");
}
```

E o **readport.c** consumidor no espaço de utilizador:

```
#include <stdio.h>
#include <sys/mman.h>        //mloockall
#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 Xenomay 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));
      }

}
```

Para gerar o módulo pode-se usar o ficheiro **Makefile** semelhante ao já apresentado para módulos do kernel com suporte para Xenomai:

```
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
```

Para gerar a aplicação no espaço de utilizador o ficheiro **MakefileUS**, semelhante ao já apresentado para aplicações no espaço de utilizador com suporte para Xenomai:

```
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 *~
```

Pode-se automatizar a geração de ambos com o script **build.sh**:

```
make
make -f MakefileUS
```

Para executar a aplicação **readport** inserindo o módulo **writeportK** previamente e removendo-o quando for terminada a aplicação com <ctrl-c>, pode-se usar o ficheiro **.runinfo**:

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

Convém referir que **popall** remove não só o modulo da API nativa: **native**, mas também quaisquer módulos inseridos com **push** (neste caso apenas **writeportK**).
Assim correndo o script **run.sh:**

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

pode-se executar a aplicação:

```
. run.sh
```

1.2 Tarefas em espaço de utilizador em C++

Para compilar uma aplicação C++ com suporte para Xenomai no espaço de utilizador, basta indicar no makefile que se deve usar o compilador g++. Tal não é assim tão simples para módulos do kernel.
Supondo que se pretende usar C++ na aplicação do ponto anterior, **readport.c**, por exemplo para I/O com a consola:

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

#include <sys/mman.h>   //mloockall
#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 Xenomay 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;
      }

}
```

Adicionar uma linha ao ficheiro MakefileUS indicando o compilador a usar

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

E voltar a correr os scripts **build.sh** e **run.sh**.

1.3 Conclusão

Após esta breve introdução ao Xenomai, recomenda-se que se implementem também os exemplos dos capítulos teóricos de STR.

1.4 Bibliografia

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 roadmap: changes in version 3:

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

Xenomai API tour:

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

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