UNIVERSIDADE ESTADUAL DO RIO GRANDE DO SUL
UNIDADE DE GUAÍBA
CURSO DE ENGENHARIA DE COMPUTAÇÃO
GRUPO DE PESQUISA EM SISTEMAS EMBARCADOS E DE TEMPO REAL

Guia de programação NKE0.7d

Bolsistas desenvolvedores
Aline Schropfer Fracalossi
Alleff Dymytry Pereira de Deus
Bruna Quinhones de Melo
Cássio Nunes Brasil
Philippe Silveira
Gabrieli Michelon
Guilherme Debom
Leonardo da Luz Silva
Lucas Murliky
Tiago da Silva Duarte

Professor orientador
Dr. Celso Maciel da Costa
celsocostars@gmail.com

Guaíba – RS / Brasil
2016

Sumário

1. Como utilizar a placa ARM LPC23XX

A seguir serão descritos os passos a serem seguidos para executar um programa de aplicação escrito em C, na placa LPC23xx, sem a utilização do Nanokernel, apenas para fins de aprendizado e familiarização com a placa.

A placa utilizada é a LPC2378. Trata-se de um ARM7 com microprocessador Cortex-M3, possui 512Kb de memória Flash programável e 56Kb de RAM, figura 1, a seguir.

Figura 1 - Placa LPC2378.

1.1 Preparação do Ambiente

Para utilização desta placa ARM recomenda-se instalar em seu computador um sistema operacional Linux, 64 bits (x64) ou 32 bits (x86). Para este tutorial recomenda-se o sistema operacional Ubuntu. Antes de preparar o ambiente para programar é necessário verificar qual a arquitetura do computador. Para isso se utiliza o comando:

$ file /bin/bash | cut -d' ' -f3

Ao desenvolver aplicações para arquitetura ARM, precisa-se de um compilador C específico para esta arquitetura (***arm-elf-gcc***). Existem duas versões deste compilador, para sistemas de 32 bits e sistemas de 64 bits.

Para sistemas de 32 bits, a versão do compilador se encontra no arquivo compactado "***arm-elf-gcc-4.1.0-i586-5.tgz***". Para descompactá-lo utiliza-se o seguinte comandoe m um terminal no diretório do onde se encontra o arquivo:

$ sudo tar -xzf arm-elf-gcc-4.1.0-i586-5.tgz -C/

Caso o seu sistema seja de 64 bits, a versão do compilador se encontra no arquivo compactado "***arm-gcc-4.3.2.64.tar.gz***". Para descompactá-lo utiliza-se o seguinte comando em um terminal no diretório do onde se encontra o arquivo:

$ sudo tar -xzf arm-gcc.4.3.2-64.tar.gz -C/

Obs.: É recomendável evitar descompactar o compilador na Área de Trabalho do

sistema operacional.

1.2 Permitindo Acesso a Porta Serial

Para carregar um programa na placa através da porta USB é necessário conceder ao usuário do sistema a permissão para acessar a porta serial. Antes da permissão, é necessário reiniciar o equipamento. Pode-se reiniciar o computador através do comando:

$ kill -9 -1

Para permitir ao usuário o acesso à porta serial, usa-se o comando:

***$ sudo* usermod -a -G dialout <usuário>**
(Em "<usuário>" coloque o nome do usuário).

Além disso, é preciso alterar as permissões dos arquivos na pasta que contém o nanokernel (***"../NKEx.x"***), para isso usa-se o comando:

$chmod 777 –R ../NKEx.x

1.3 Execução de uma Aplicação Simples na LPC2378 (sem NKE)

O diretórioS*imple App* possui os arquivos necessário para programação na placa. No arquivo *main.c*, na pasta *Aplicação*, há um programa simples, escrito em C, que soma dois números e imprime o resultado.

O arquivo *Makefile (.../Aplicacao/Makefile)* contém os comandos para compilar, para deletar arquivos gerados pela compilação e para carregar o programa compilado na memória flash da placa. No *Makefile* são incluídos os arquivos *uart.c*, *uart.h*, *main.c* e *crt.s*. Os arquivos *uart.c* e *uart.h* são utilizados para operações de entrada e saída na porta serial (USB). O arquivo *crt.s* contém o carregador. O arquivo *main.c* é o programa que soma dois números e imprime o resultado. Segue abaixo o trecho do código do *Makefile* que indica os arquivos que serão compilados e executados:

```
...
CSOURCES = main.c uart.c
ASOURCES = .../Lpc2378/crt.s
...
HEADERS = uart.h
...
```

Em "*CSOURCES*" são incluídos todos os arquivos *".c"*, em "*ASOURCE*", todos os arquivos *".s"* e em "*HEADERS*", todos os arquivos *".h"*.

O arquivo *main.c* contém o seguinte código:

```
#include <arch/nxp/lpc23xx.h>
#include "uart.h"

int main(void)
{
  int x,y;
  U0init();
```

```
  x=2;
 y=5;
  x=x+y;
  nkprint("Resultado da soma: %d\n",x);
  return 0;
 }
```

Primeiramente foram importados a biblioteca*"lpc23xx.h"*, que foi instalada "juntamente com o compilador, e o arquivo "*uart.h"*, que está na pasta "simple_app\Aplicacao".

Para executar este programa siga os seguintes passos:

1.Conecte a placa em seu PC com o cabo USB;

2. Abra o terminal de comando;

3.Acesse o diretório onde está o arquivo Makefile (...\simple_app\Aplicacao);

4. Digite o comando "*make clean*" para limpar arquivos gerados durante compilações anteriores;

5. Digite o comando "make" para compilar;

6. Digite o comando "*make ispu*" para carregá-lo na placa;

7. Estando no mesmo diretório, digite "./terminalxx", sendo "xx" a versão do seu S.O. (32 ou 64 bits). Este programa monitora a comunicação serial entre a placa e o computador, através isso podemos visualizar os resultados.

8. Após visualizar os resultados no terminal, pressione "Ctrl+c" para sair ou "Ctrl+r" para reiniciar a execução na placa.

2. Programando com NKE

Neste capítulo serão descritos todos os passos para se executar uma aplicação na placa LPC23xx com o Nanokernel. Esses passos são iniciais, apenas para aprendizado e familiarização com o ambiente.

Para programar usando o NKE, localiza-se o arquivo "Makefile" em *".../NKEx.x/Placa"*, modifica-se a linha onde está explícito "*CSOURCES=*" incluindo o nome de sua aplicação. Por exemplo, para executar uma aplicação cujo nome é "*pipeline.c"*, que está na pasta *Aplicações*, deve-se incluir esse arquivo na linha indicada da seguinte forma: "*.../Aplicacoes/pipeline.c*". Os demais arquivos não podem ser excluídos, ou seja, apenas se altera o primeiro da linha.

2.1 Compilação

Utilizando o terminal, acesseo diretório ".../NKEx.x/Placa", onde está o arquivo "*Makefile*". Feito isto, siga os seguintes passos:

a) Delete os arquivos compilados previamente:

$ make clean

b) Compile os códigos fontes:

$ make

c) Envie para a placa:

$ make ispu

2.2 Execução de uma aplicação no NKE

O arquivo *Makefile (.../Aplicacao/Makefile)*é utilizado para compilação, para deletar arquivos gerados pela compilação e para carregar o código compilado do programana memória flash. Nele incluiremos também os arquivos que devem ser executados, inclusive a aplicação a ser executada. O arquivo contendo o código da aplicação deve estar na pasta *Aplicacoes* e precisa ser indicado no *Makefile*. Note que se pode compilar apenas um arquivo da pasta *Aplicacoes* por vez. O trecho do Makefile que deve ser alterado está destacado a seguir:

```
#Para executar outro programa mude o nome do primeiro arquivo.c
# (pipeline.c no exemplo abaixo)
CSOURCES = ../Aplicacoes/pipeline.c $(wildcard ../Kernel/*.c) $(wildcard ../Placa/*.c)
ASOURCES = $(wildcard ../Kernel/*.S) $(wildcard ../Placa/*.S)...
HEADERS = $(wildcard ../Kernel/*.h) $(wildcard ../Placa/*.h)
```

Em "***CSOURCES***" são incluídos todos os arquivos "*.c*", em "***ASOURCE***", todos os arquivos "*.s*" e em "***HEADERS***", todos os arquivos "*.h*".

O programa de aplicação que faremosserá formado por uma *task* que soma dois números e envia seu resultado pela serial. O resultado enviado será visualizado no computador através de um programa que monitora a comunicação USB. Primeiramente crie um arquivo chamado *minha_aplicacao.c* na pasta ***/Aplicacao***. Neste arquivo inclua o seguinte código:

```
#include "../Kernel/kernel.h"
```

```
void task(){
        int x;
        int y;
        int r;
        x=4;
        y=3;
        r=x+y;
        nkprint("Soma:\n",0);
        nkprint(" %d ",&x);
        nkprint(" +  %d",&y);
nkprint(" = %d\n",&r);
        taskexit();
}
int main (int argc, char *argv[]){
        int t0;
        taskcreate(&t0,task);
        start(RR);
        return 0;
}
```

Toda aplicação deve importar o arquivo "*.../Kernel/kernel.h*", esse arquivo é o cabeçalho principal do kernel.A chamada***nkprint*** possui apenas dois parâmetros: o primeiro recebe uma string, que pode conter uma variável, e o segundo parâmetro recebe o endereço desta variável ou zero quando a string não possuir variável. O ***main*** sempre termina a sua execução na chamada de sistema ***start***. Essa chamada faz com que o nanokernel bloqueie a ***main*** e passe a executar as tarefas que foram criadas com a primitiva ***taskcreate***. A análise de cada chamada implementada no Nanokernel é abordada nocapitulo 3 - Chamadas de sistema nativas do NKE.

No arquivo *Makefile*,na pasta*Placa**,*** inclua o nome do programa a ser executado como no exemplo a seguir:

```
#Para executar outro programa mude o nome do primeiro arquivo .c
CSOURCES = ../Aplicacoes/minha_aplicacao.c $(wildcard ../Kernel/*.c) $(wildcard ../Placa/*.c)
ASOURCES = $(wildcard ../Kernel/*.S) $(wildcard ../Placa/*.S)
HEADERS = $(wildcard ../Kernel/*.h) $(wildcard ../Placa/*.h)
```

Para executar um programa seguimos os seguintes passos:

1. Conecte a placa em seu PC.
2. Abra o terminal de comando.
3. Acesse o diretório “NKEX.X”.
4. Execute o comando chmod 777 -R NKEX.X (sendo X.X a versão do NKE).
5. Acesse o diretório onde está o arquivo *Makefile*.
6. Digite o comando “***makeclean***” para limpar arquivos gerados durante compilações anteriores.
7. Digite o comando “***make***” para compilar.
8. Digite o comando “***makeispu***” para carregá-lo na placa.
9. Retorne ao diretório “NKEX.X”.

10. Execute o programa que monitora a USBcom o comando “./terminalxx”, sendo “xx” a versão do seu SO (32 ou 64 bits)

Após visualizar os resultados pressione “Ctrl+c” para sair ou “Ctrl+r” para reiniciar a execuçãoda placa.

3. Chamadas de sistema nativas do NKE

Importante: Todas as chamadas de sistema possuem apenas letras minúsculas.

Gerência de tasks

taskcreate
start
taskexit
setmyname
getmynumber

Sincronização

semwait
sempost
seminit

Entrada e Saída

ligaled
writelcdn
writelcds
nkprint
nkread

Gerência de tempo

sleep
msleep
usleep

3.1 Primitivas de Gerência de *Tasks*

O corpo de uma *task* (o código que ela executa) é semelhante a uma função em C. Enquanto funções comuns podem receber parâmetros e retornar valores, uma *task* deve ser declarada como *void* e não deve receber nenhum parâmetro. Todo código da tarefa deve estar contido dentro da função da *task*. O NKE cria as tarefas dentro do sistema através da primitiva

*void taskcreate(int *ID,void (*funcao)())*

que é executada na função *main* da aplicação. A função recebe dois parâmetros, sendo o primeiro, *int *ID*, um ponteiro de inteiro que armazena o número de ID da *task* no sistema. O segundo parâmetro, "*void (*funcao)()"* recebe o endereço da função que contém o código da *task*.

Exemplo de um programa utilizando*taskcreate*:

```
#include "../Kernel/kernel.h"
int t0,t1;
void task0() {
while(1);
}
void task1() {
while(1);
}
```

```
int main (int argc, char *argv[ ]) {
  taskcreate(&t0,task0);
  taskcreate(&t1,task1);
  start(RR);
  return 0;
}
```

Esse exemplo mostra duas tarefas, *task0* e *task1*. Ambas executam um laço vazio infinito. Na *taskcreate* foi definida a criação da *task0*, sendo que a variável *t0* armazenará o ID da *task* no sistema durante toda execução. Na criação da *task1* a variável *t1* foi definida para armazenar o ID da mesma no sistema. Neste exemplo as tarefas nunca são terminadas.

A chamadas *taskcreate* cria as *tasks* no sistema, porém esses não passam a ser executadas imediatamente. A execução da função *main* deve ser interrompida para que as *tasks* sejam executadas. Além disso,enquanto as *tasks* são executadas a função *main* deve permanecer bloqueada para que a aplicação não se encerre. A primitiva responsável por fazer isso é a

void start(int scheduler).

O NKE possui uma lista de *tasks* prontas para execução: a *ReadyList*. A primitiva *taskcreate* cria as tarefas, mas não as insere nessa lista. Já a primitiva *start* bloqueia a execução da função *main* e insere todas as tarefas criadas até então na *ReadyList* para que sejam executadas. Além disso, é através da primitiva *start* que se define o algoritmo de escalonamento a ser utilizado na execução das *tasks*. O algoritmo de escalonamento é informado através do parâmetro *int scheduler*. Este parâmetro pode assumir os valores *RR*, *RM* e *EDF*, que correspondem aos algoritmos de escalonamento *Round-Robin*, *Rate-Monotonic* e *Earliest Deadline First*, conforme mostra a tabela a seguir:

Algoritmo	**Valor**
Round-Robin	RR
Rate-Monotonic	RM
Earliest Deadline First	EDF

Quando em execução, uma task executa apenas o código contido em sua função. Para que seja encerrada, uma task deve possuir em seu código a primitiva

void taskexit(void)

Essa chamada deve estar no corpo da *task* que se deseja encerrar. Ao executar essa chamada a tarefa será finalizada pelo Nanokernel e não será mais inserida na *ReadyList* do NKE.

Exemplo de um código utilizando *taskexit*:

```
#include "../Kernel/kernel.h"
void task(){
        for (int i=0; i<10; i++);
```

```
        taskexit();
}
int main (int argc, char *argv[]){
        int t0;
        taskcreate(&t0,task);
        start(RR);
        return 0;
}
```

Este exemplo de aplicação cria uma tarefa chamada *task*. A primitiva start insere esta tarefa na *ReadyList* e define *Round-Robin* como algoritmo de escalonamento a ser usado pelo sistema.Quando em execução, a task executa um laço e depois é encerrada pelo Nanokernel através da chamada *taskexit*.

O NKE possui um descritor de tarefas que armazena informações como nome, ID, prioridade e estados das tasks criadas. Durante a execução de uma task é possível alterar o nome desta task no descritor de tarefas através da primitiva

*void setmyname(const char *name)*

que recebe um ponteiro de char contendo o novo nome a ser atribuído à tarefa.

Exemplo de um código utilizando *setmyname*:

```
#include "../Kernel/kernel.h"
void task() {
        setmyname("Name");
        taskexit();
}
int main (int argc, char *argv[ ]) {
        int t0;
        taskcreate(&t0,task);
        start(RR);
        return 0;
}
```

Este exemplo de aplicação cria uma tarefa que define seu próprio nome como "Name".

Algumas informações do descritor de tarefas do NKE não podem ser alteradas durante a execução do sistema, como ID, por exemplo. Entretanto é possível que a task obtenha o número do próprio ID através da primitiva

*void getmynumber(int *number)*

que recebe um ponteiro de uma variável. Essa primitiva atribui à variável apontada o valor do ID da própria *task*.

Exemplo de código utilizando *getmynumber*:

```
#include "../Kernel/kernel.h"
void task() {
        int myID;
        getmynumber(&myID);
```

```
        //myID = task ID
        taskexit();
}
int main (int argc, char *argv[ ]) {
        int t0;
        taskcreate(&t0,task);
        start(RR);
        return 0;
}
```

Este exemplo de aplicação cria uma tarefa que possui uma variável *myID*. O endereço dessa variável é repassado na primitiva *getmynumber*. A variável *myID* passa a conter o valor do ID da *task* no sistema.

3.2 Primitivas de Sincronização

No NKE as tarefas podem ser sincronizadas através de semáforos. Semáforo é uma estrutura de dados capaz de fazer o bloqueio lógico de uma *task*em um determinado trecho do código. No NKE essa estrutura recebe o nome de *sem_t*. Todo semáforo precisa ser inicializado, isso é feito através da primitiva

void seminit(sem_t *semaforo, int ValorInicial)

que recebe dois parâmetros. O primeiro parâmetro é o ponteiro do semáforo que será inicializado enquanto o segundo é o valor inicialmente assumido pelo semáforo. Esse valor definirá se o semáforo será inicializado, sendo admitidos apenas os valores 1(aberto) ou 0(fechado).

A primitiva que solicita o acesso ao semáforo é a

*void semwait(sem_t *semaforo)*

que recebe como parâmetro apenas o ponteiro do semáforo que a task solicitante deseja acessar. Essa primitiva funciona como uma “entrada” para o semáforo, se este estiver aberto a tarefa solicitante estará livre para seguir adiante, caso esteja fechado, a tarefa permanecerá bloqueada até que o semáforo seja liberado. A primitiva que libera um semáforo bloqueado é a

void sempost(sem_t *semaforo)

que recebe apenas o ponteiro do semáforo que se deseja liberar. Essa primitiva seria a “saída” do semáforo. A tarefa solicitante já executou o código crítico e pode liberá-lo à outras tarefas. Quando o semáforo é liberado a task bloqueada à espera do mesmo é acordada para continuar sua execução.

Exemplo de um programa utilizando semáforos:

```
#include "../Kernel/kernel.h"
sem_t s0; //Semáforo declarado
int num=0;
void soma_20000(){
```

```
   for(int i=0;i<10000;i++){
semwait(&s0);
      num += 2;         //Região crítica 1
      sempost(&s0);
   }
   taskexit();
}
void soma_50000(){
   for(int i=0;i<10000;i++){
      semwait(&s0);
      num += 5;         //Região crítica 2
      sempost(&s0);
   }
   taskexit();
}
int main(int argc, char *argv[]){
int t2,t3;
  seminit(&s0, 1);
  taskcreate(&t2,soma_20000);
  taskcreate(&t2,soma_50000);
  start(RR);
  return 0;
}
```

Este exemplo de aplicação cria duas tarefas, *soma_20000* e *soma_50000*. Ambas as *tasks* incrementam uma variável compartilhada *num.* O acesso a essa variável é sincronizado através do semáforo *s0*. Se uma tarefa acessa e ocupa *s0,*mas, devido ao escalonamento, perde o processador antes de liberar *s0*, quando a outra tarefa for executada a mesma será bloqueada ao tentar acessar *s0*. Uma vez que esse semáforo for liberado a tarefa bloqueada será acordada e poderá, enfim, acessar sua região crítica.

3.3 Primitivas de Entrada e Saída

O NKE possui primitivas para periféricos de entrada e saída da LPC2378, sendo estes um conjunto de LEDs, um display LCD e uma porta serial USB. A primitiva que liga os LEDs da placa é a

void ligaled(int valor)

que recebe como parâmetro um valor inteiro correspondente aos LEDs que deverão ser acesos. A placa possui 8 LEDs enfileirados que podem ser entendidos como 8 bits, ou seja, se o valor inteiro for o binário “0000 0001”, apenas os primeiro LED será aceso. Se o valor for o binário “1111 1111”, todos os LEDs serão acesos. A tabela a seguir mostra os valores para se acender um único LED de cada vez:

Entrada Hexadecimal	**Saída**
0x01	LED 1
0x02	LED 2

0x04	LED 3
0x08	LED 4
0x10	LED 5
0x20	LED 6
0x40	LED 7
0x80	LED 8

Exemplo de um código utilizando ligaled:

```
#include "../Kernel/kernel.h"
void task() {
        ligaled(1); //Liga o primeiro LED
        ligaled(255); //Liga todos os LEDs
        taskexit();
}
int main (int argc, char *argv[ ]) {
        int t0;
        taskcreate(&t0,task);
        start(RR);
        return 0;
}
```

Este exemplo de aplicação cria uma tarefa que liga o primeiro LED e logo em seguida acende todos os LEDS.

Além do conjunto de LEDs a placa LPC2378 possui um display LCD com duas linhas sendo cada uma de 15 caracteres. A primitiva do NKE que imprime um número inteiro no display LCD é a

void writelcdn(int msg,int pos).

Esta primitiva recebe dois parâmetros, sendo o primeiro um valor inteiro que será exibido no display e o segundo um valor referente a posição na qual o número será exibido no display. Os valores das posições na primeira linha vão de 0x80 (primeira posição) até 0x8F (última posição), enquanto que na segunda linha os valores das posições vão de 0xC0 até 0xCF.

Exemplo de um código utilizando a primitiva writelcdn:

```
#include "../Kernel/kernel.h"
void task() {
        int n = 8;
        writelcdn( n, 0xC2);
        taskexit();
}
int main (int argc, char *argv[ ]) {
        int t0;
        taskcreate(&t0,task);
```

```
        start(RR);
        return 0;
}
```

Este exemplo de aplicação cria uma tarefa que imprime o número “8” na posição 0xC2 do display LCD.

Além de números, o NKE também imprime strings no display LCD. Isso é feito através da primitiva

void writelcds(char msg, int pos)*

que recebe dois parâmetros, sendo o primeiro a string a ser exibida no display e a segunda um valor referente a posição do primeiro caractere da string no display LCD. Os valores das posições na primeira linha vão de 0x80 (primeira posição) até 0x8F (última posição), enquanto que na segunda linha os valores das posições vão de 0xC0 até 0xCF.

Exemplo de um código utilizando writelcds:

```
#include "../Kernel/kernel.h"
void task() {
        char string[6] = {‘s’, ’t’, ’r’, ’i’, ’n’, ’g’};
        writelcds (string, 0xC2);
        taskexit();
}
int main (int argc, char *argv[ ]) {
        int t0;
        taskcreate(&t0,task);
        start(RR);
        return 0;
}
```

Este exemplo de aplicação cria uma tarefa que imprime a string *“string”* a partir da posição 0xC2 do display LCD.

Além desses periféricos, o NKE também utiliza a comunicação serial (porta USB) da LPC2378. A primitiva

*void nkprint(char *fmt,void *number)*

envia uma mensagem pela porta serial. O comportamento desta primitiva é semelhante ao da função *printf* padrão da biblioteca C. Porém, o *nkprint*recebe sempre dois parâmetros, mesmo que a mensagem não contenha nenhuma variável. O primeiro parâmetro é a mensagem a ser enviada, essa mensagem pode conter no máximo uma variável, *“Number is %d \n”*, por exemplo. O nkprint suporta diferentes tipos de variáveis, sendo possível utilizar os formatos: %c, %d, %s e %f.

O segundo parâmetro é a variável que contém o valor citado na mensagem. Quando a mensagem não possui nenhuma variável este parâmetro deve receber o valor “0”.

Exemplo de um código utilizando nkprint:

```
#include "../Kernel/kernel.h"
int main (int argc, char *argv[ ]) {
```

```
        int num = 2305;
        nkprint("Result = %d \n", num);
        nkprint("My proc: %s \n", "ARM7");
        start(RR);
        return 0;
}
```

Este exemplo de aplicação envia pela porta serial as mensagens “Result = 2305 \n” e “My proc: ARM7 \n”.

Além de escrever na serial, o NKE lê as informações durante a comunicação. A leitura da serial é feita através da primitiva

*void nkread (char *tipo, void *value)*

que recebe dois parâmetros. O comportamento desta chamada é similar ao da função *scanf* da biblioteca padrão C. O primeiro parâmetro recebe uma string contendo o tipo de dado esperado, *“%d”*, por exemplo. Os possíveis tipos são: *%c* para lerum caractere, *%s* para ler uma string, *%d* para ler um valor inteiroe por *%f* para ler um valor de ponto flutuante (estilo não científico). O segundo parâmetro recebe um ponteiro de uma variável que receberá o valor lido na serial.

A faixa de valores possíveis são:

Tipo	Valores
Inteiros (32 bits)	-32768 até 32767
Floats (32bits)	-32768,0 até 32767,0
Caracteres (8bits)	Aa-Zz, 'ç' não é permitido.
Strings	Até 25 caracteres (incluindo \n).Permite espaço, “ç”, números e todos os outros caracteres.

Exemplo:

nkread(“%.2f”, &var1)

Neste caso a variável *var1* receberá o primeiro valor do tipo float lido na serial. Este valor conterá apenas duas casas decimais. Se a precisão não for especificada o padrão é 6 casas decimais.

Exemplo de um código utilizando nkread:

```
#include "../Kernel/kernel.h"
int main (int argc, char *argv[ ]) {
        int num = 2305;
        nkread(“%d”, &num);
        start(RR);
        return 0;
}
```

Este exemplo de aplicação lê a serial e o armazena na variável *num* o primeiro valor inteiro encontrado.

3.4 Primitivas de Gerência de Tempo

O NKE possui primitivas de bloqueio em função do tempo. Essas primitivas bloqueiam uma determinada *task* durante um tempo predeterminado. A primitiva

void sleep(int time)

recebe um parâmetro do tipo inteiro referente ao tempo, em segundos, no qual uma tarefa permanecerá bloqueada. Após este tempo a task será acordada e reinserida na ReadyList.A mesma função é feita pela primitiva

void msleep(int time)

sendo a única diferença a contagem de tempo em milissegundos. Nessa primitiva o valor armazenado na variável *time* é referente ao tempo de bloqueio da tarefa em milissegundos.Utiliza-se a primitiva

void usleep(int time)

para bloquear uma tarefa durante um dado intervalo de tempo em microssegundos. Essa primitiva também possui a mesma função da primitiva *sleep*, tendo como única diferença a contagem de tempo em microssegundos.

Exemplo de um código utilizando sleep:

```
#include "../Kernel/kernel.h"
void task0(){
        sleep(2);
        msleep(2);
        usleep(2);
        taskexit();
}
int main (int argc, char *argv[ ]) {
        int t0,t1;
        taskcreate(&t0,task0);
        start(RR);
        return 0;
}
```

Este exemplo de aplicação cria uma *task* que é bloqueada durante 2 segundos, 2 milissegundos e 2 microssegundos. Após esse tempo a *task* é encerrada.

4. Inserindo uma nova chamada

As chamadas de usuário no NKE são implementadas com o uso da instrução SWI. Um programa de usuário roda no modo usuário do processador. Quando o programa do usuário necessita de um serviço do NKE, chama uma função que recebe os parâmetros e os armazena na estrutura de dados *Parameters* juntamente com o número da chamada *CallNumber.*Após isso a instrução SWI passa o processador para o modo *system*,passando a executar o *kernel* do sistema.A seguir é apresentada a estrutura de dados *Parameters*:

CallNumber	p0	p1	p2	p3

A estrutura *Parameters* possui 5 campos, sendo estes *CallNumber*, *p0*, *p1*, *p2* e *p3*. Sendo *CallNumbe*ro número da chamada de sistema a ser executada. *p0* é o endereço do primeiro argumento da chamada de sistema, *p1*,do segundo argumento da chamada de sistema, *p2*, do terceiro argumento da chamada de sistema e *p3*, do quarto argumento da chamada de sistema.

Chamadas de sistema com mais de quatro argumentos podem ser implementadas alterando-se a estrutura *Parameters*, que está definida em *Kernel/syscall.h*. Para criar uma nova chamada segue-se os seguintes passos:

1. A nova chamada de usuário deve ser declarada no arquivo *usercall.h*, na pasta K*ernel*. Exemplo: "*void newcall(int var);*"
2. A nova chamada de usuário deve ser declarada no arquivo *usercall.c*, na pasta K*ernel*. Este arquivo conterá a implementação da chamada, onde cada parâmetro deve ser repassado a estrutura *Parameters*. Segue a seguir um exemplo:

```
void newcall(int var){
  Parameters arg;
  arg.CallNumber=NEWCALL;
  arg.p0=(int *)var;
  CallSWI(0,&arg);
}
```

   A função *CallSWI* gera uma interrupção de softwareno sistema para que este mude para o modo *kernel*.
3. A nova chamada deve ser inserida na lista do ***enum*** do arquivo *syscall.h*, da pasta kernel, conforme mostra o exemplo:

```
enum sys_temCall{
  TASKCREATE,
  SEM_WAIT,
  ...
  NKPRINT,
NEWCALL,
};
```

4. A nova chamada deve ser inserida no ***switch*** da função *DoSystemCall*, no arquivo *syscall.c* da pasta Kernel, conforme é mostrado no exemplo abaixo. A função *DoSystemCall*repassa a execução à chamada de sistema correspondente ao *CallNumber*. As chamadas de sistema devem possuir o mesmo nome da chamada de usuário, adicionando "sys_" ao início.

```
switch(arg->CallNumber) {
        case WAITPERIOD:
                sys_waitperiod();
                break;
```

```
        case SEM_WAIT:
                sys_semwait((sem_t *)arg->p0);
                break;
        ...
        ...
        case NEWCALL:
                //Chamada de sistema ainda não criada;
                break;
        case default:
                break;
}
```

5. A nova chamada de sistema deve ser declarada no arquivo *syscall.h*, na pasta K*ernel*. Essa chamada deve possuir o mesmo nome de sua chamada de usuário equivalente, acrescentando-se “sys_” no início.
   Exemplo: “*void sys_newcall(int var);*”
6. A nova chamada de usuário deve ser declarada no arquivo *syscall.c*, na pasta K*ernel*, conforme o exemplo abaixo. Este arquivo conterá a implementação da chamada que manipulará os dados armazenados na estrutura *Parameters* pela chamada de usuário.

```
void sys_newcall(int var){
        //Insira aqui o código da nova chamada.
}
```

7. Uma vez que a chamada e sistema está implementada, deve se atualizar o ***switch*** da função *DoSystemCall*, no mesmo arquivo (*syscall.c*). Para isso é necessário levar em consideração como os argumentos da estrutura *Parameters* foram carregados na chamada de sistema. Em nosso exemplo apenas o parâmetro p0 recebeu um argumento, sendo assim o ***switch*** ficará assim:

```
switch(arg->CallNumber) {
        case WAITPERIOD:
                sys_waitperiod();
                break;
        case SEM_WAIT:
                sys_semwait((sem_t *)arg->p0);
                break;
        ...
        ...
        case NEWCALL:
                sys_newcall((int*)arg->p0);
                break;
        case default:
                break;
}
```

Para implementar chamadas com mais de 4 parâmetros é necessário alterar também a estrutura Parameters no arquivo kernel.h, na pasta Kernel.

5. Bibliografia

Costa, Celso Maciel da; Fragoso, João Leonardo; Murliky, Lucas; Silva, Leonardo da Luz; Fracalossi, Aline; Duarte, Tiago; Melo, Bruna; Brasil, Cássio; Debom, Guilherme; Matias Jr., Rivalino; “NKE – Um Nanokernel Educacional para Microprocessadores ARM”. IV Simpósio Brasileiro de Engenharia de Sistemas Computacionais Manaus/AM, 3 a 7 de novembro de 2014.