MONTAGEM DE PROTÓTIPO PARA TRANSPORTE DE CARGA UTILIZANDO O KIT LEGO MINDSTORMS NXT E TÉCNICAS DE PROGRAMAÇÃO EM TEMPO REAL

Cleber Albert Moreira Marques[1]
Euclério Barbosa Ornellas Filho[2]
Rafael G. Bezerra de Araújo[3]

Resumo
O Veículo de Carga Interativo em Tempo Real (Vecitr) é um protótipo autônomo transportador de cargas que faz parte de um sistema de manufatura integrado (SIM). A função do Vecitr no SIM é receber uma carga na caçamba por um protótipo, transportá-la para outro local e descarregá-la sobre o protótipo competente por continuar o processo. O objetivo ao desenvolver o Vecitr foi proporcionar aos projetistas a experiência de desenvolver um protótipo capaz de move-se, reagir a cores e comunicar com outros protótipos utilizando técnicas de programação para Tempo Real. O Vecitr foi desenvolvido utilizando kits de *smart brick* da linha LEGO Mindstorms NXT composto por *smart bricks*, sensores, controlador lógico programável, *software* e miscelâneas e foi programado utilizando a linguagem de programação NXC com técnicas de gerenciamento de recurso compartilhado, comunicação interprocessual, *multithreading* e boas práticas de programação; temas estudados na disciplina de Sistemas de Tempo Real da Escola de Engenharia e TI (EETI) da Universidade Salvador (Unifacs).

Palavras-chave: Sistemas de Tempo Real, *Multithreading*, Exclusão Mútua, Comunicação Interprocessual.

Abstract
The Real-Time Cargo Interactive Vehicle (RTVCI) is a cargo carrier autonomous prototype, part of an Integrated Manufacturing System (IMS). The RTVCI function in the IMS is to receive the cargo in the hopper from a prototype, carry it to another location and discharge it from the hopper on another prototype responsible to continue the process. The objective to develop the RTVCI was to provide the designers the experience to develop a prototype able to move, react to colors and communicate with others prototypes using real-time programming techniques. The RTVCI was developed using the LEGO Mindstorms NXT smart brick kits, composed by smart bricks, sensors, programmable logic controller, software and miscellaneous, and was programmed using NXC programming language with techniques of shared resource management, interprocess communication, multithreading and good programming practices; topics studied in the Real-Time System subject of the Escola de Engenharia e TI (EETI) at the Universidade Salvador (Unifacs).

Keywords*:* Real-Time System, Multithreading, Mutual Exclusion, Interprocess Communication.

INTRODUÇÃO

O Veículo de Carga Interativo em Tempo Real (Vecitr) com LEGO Mindstorms NXT consiste em um protótipo integrado a um Sistema Integrado de Manufatura (SIM) que transporta cargas de um protótipo a outro, atendendo exigências de prazos de entrega e com capacidade de parear e interagir com outros protótipos do SIM. A finalidade do Vecitr é

[1] Graduando em Engenharia Elétrica da Escola de Engenharia e Tecnologia da Informação - Universidade Salvador (Unifacs). cleber.amm@gmail.com
[2] Professor da Escola de Engenharia e Tecnologia da Informação (EETI) - Universidade Salvador (Unifacs). euclerio.ornellas@unifacs.br
[3] Professor da Escola de Engenharia e Tecnologia da Informação (EETI) - Universidade Salvador (Unifacs). rafael.araujo@unifacs.br

receber a carga de protótipo com dimensões físicas e comunicação compatíveis, caminhar sobre uma rota específica e não retilínea e entregá-la a outro protótipo.

A trajetória que a carga percorre é dividida em 6 trechos e cada trecho é percorrido por um protótipo com métodos de manobra e transporte de carga distintos, como ilustrado na figura 1. Para fazer o sistema funcionar, os protótipos que interagem têm total compatibilidade. Cada protótipo foi desenvolvido por uma equipe de alunos e no total 6 equipes desenvolveram 6 protótipos.

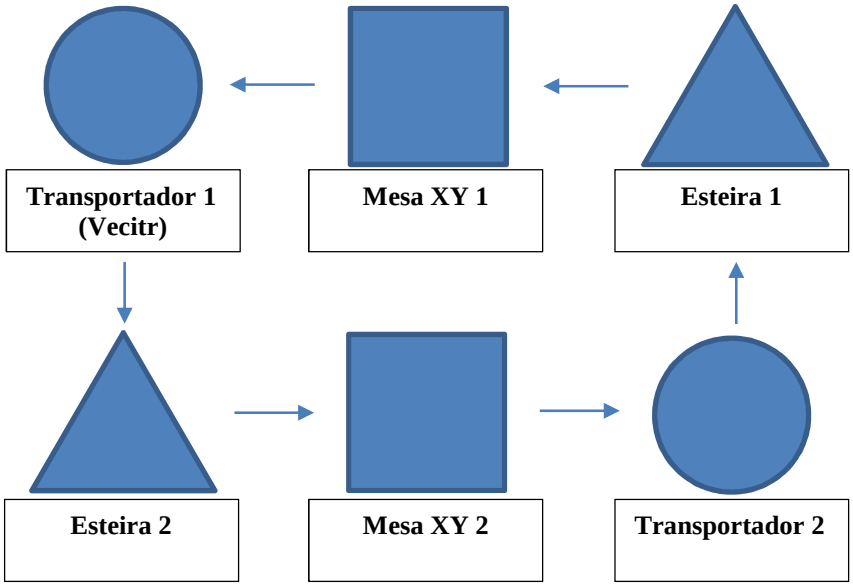

Figura 1: Diagrama do sistema. Fonte: Elaboração própria, 2014.

Os projetistas de cada protótipo, projetam as dimensões físicas e definem as características da comunicação em conjunto, durante a fase de projeto, para garantir que os sistemas que interagem durante o transporte sejam compatíveis. Portanto, a estrutura mecânica do protótipo foi contruida em LEGO e o recurso de memória compartilhada é realizada para prover a comunicação interprocessual entre os programas dos protótipos.

O kit de *smart brick* da linha LEGO Mindstorms NXT usa vários eletrônicos avançados para proporcionar um sistema com ampla funcionalidade e de fácil montagem, através dos blocos de encaixe. O kit é composto pelos blocos, por sensores ativos, sensores passivos, atuadores, controlador lógico programável (CLP), fonte de alimentação e miscelâneas. No Vecitr foram utilizados blocos de encaixe, um CLP, dois sensores de luz, 1 módulo Bluetooth, três servo-motores, uma fonte de alimentação e miscelâneas.

O objetivo no desenvolvimento do protótipo foi de propiciar aos projetistas a experiência prática sobre o conhecimento adquirido na disciplina de Sistemas de Tempo Real

da Escola de Engenharia e TI (EETI) da Universidade Salvador (Unifacs), por isto o enfoque deste artigo é voltado para a abordagem das técnicas de Tempo Real utilizadas no Vecitr.

DESENVILVIMENTO

Para a elaboração do protótipo Vecitr foi implementado um sistema de tempo real multitarefa, utilizando o kit LEGO NXT da Mindstorms composto por processador que inclui sistema operacional de tempo real (*RTOS*) TOPPERS, atuadores (servo-motores), sensores (sensores de luz), módulo Bluetooth e blocos de encaixe para montagem mecânica, além da plataforma de desenvolvimento para tempo real em código aberto nxtOSEK para satisfazer as restrições explícitas de tempo de resposta do sistema. (LEJOS-OSEK, 2009; TOPPERS, 2003)

O atendimento às restrições explícitas de tempo de resposta é feito pelo *RTOS*, organizando as tarefas a serem executadas de maneira que não falte tempo para executá-las, evitando uma degradação do sistema por perdas de *deadline*. Para isto, o TOPPERS utiliza o algoritmo de escalonamento não preemptivo *FCFS* (*First Come First Served*), em que os processos são processados de acordo a ordem de entrada na tabela de processamento e independe de prioridade. O sistema não é determinístico para tempo e classificado como *Soft*, ou seja, o *deadline* das tarefas para todos os possíveis comportamentos do sistema não é conhecido e as perdas de alguns *deadlines* podem degradar a qualidade do serviço, mas não compromete o funcionamento do sistema. (TRON ASSOCIATION, 2002)

O programa computacional gravado no CLP do Vecitr foi desenvolvido utilizando a linguagem de programação NXC (*Not Exactly C*), com fluxo procedural e variáveis globais, *threads*, poucas funções e estruturas dinâmicas e sem *polling* ou dados do tipo *struct* ou *double*. Estas são boas práticas de programação para otimizar o tempo de execução das tarefas. Apesar de funções serem estruturas dinâmicas, por utilizarem apontadores para outras regiões da memória onde as instruções estão armazenadas, funções são necessárias para implementação de *thread*, recurso essencial para a implementação de um programa multitarefa como o do Vecitr. A figura 2 ilustra o fluxograma do programa.

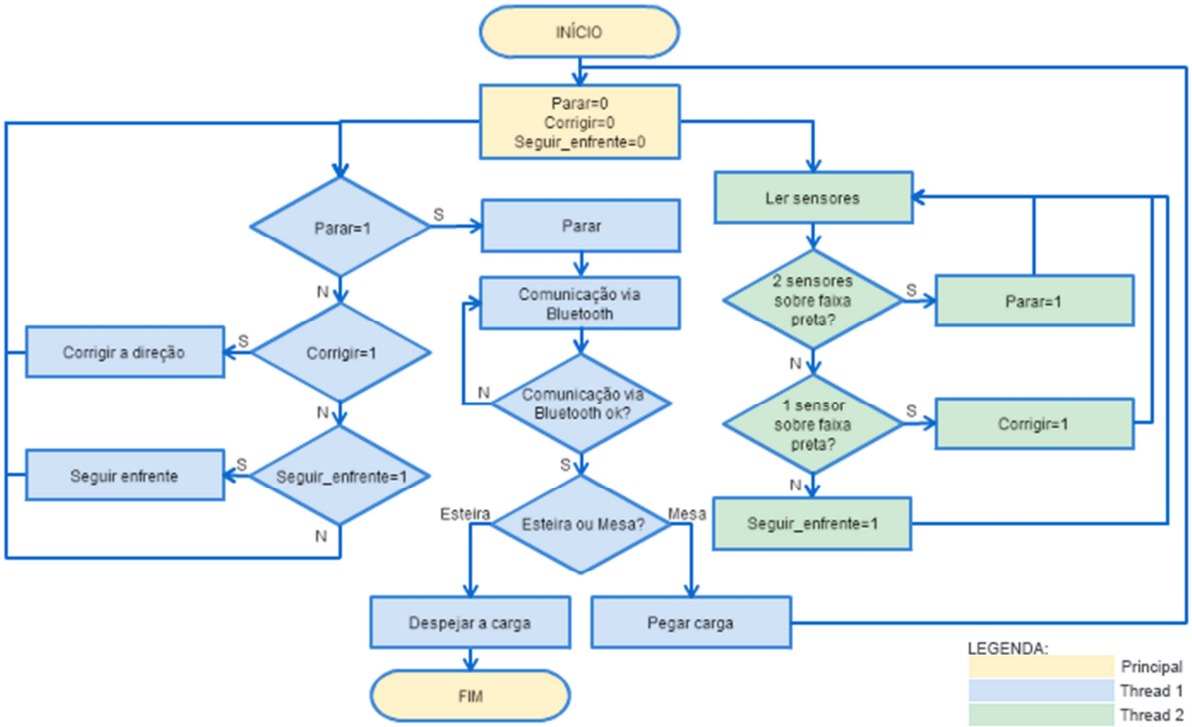

Figura 2: Fluxograma do programa do Vecitr. Fonte: Elaboração própria, 2014.

O programa é multitarefa, ou seja, mais de uma tarefa pode ser executada simultaneamente pelo processador. No Vecitr, o acionamento dos atuadores, comunicação Bluetooth e a leitura dos sensores são duas funções distintas que são executadas ao mesmo tempo. Enquanto o veículo se desloca, os sensores leem a trajetória que está sendo percorrida no mesmo instante. A implementação de um programa multitarefa é possível através do uso de t*hreads*, como já foi dito no parágrafo anterior.

Threads permitem que, além do fluxo principal do programa, haja outra (s) rotina (s) sendo executada (s) simultaneamente, simulando processamento paralelo. No caso de processadores *multicore* (multinúcleo), os processamentos de *threads* podem realmente serem paralelos entre si e ao fluxo principal. Um *thread* é uma unidade básica de utilização do processador, composta por um *ID* de *thread*, conjunto de registradores, inclusive o registrador contador de programa e uma pilha. Um *thread* é um processo "de baixo peso" e analogamente, um processo sem *thread* é chamado de processo tradicional ou ainda, de processo "pesado" por ter apenas um fluxo de controle. No entanto, processos com múltiplos *threads* (*multithreading*) permite que, através de escalonamento, mais um *thread* seja executado dentro de um único processo como ilustra a figura 3, compartilhando os mesmos recursos e com múltiplas tarefas com uma única entrada na tabela de processos. *Thread* pode ser suportado pela camada do usuário (*User-Level Thread* ou *ULT*) ou pela camada do Kernel

(*Kernel-Level Thread* ou *KLT*) do sistema operacional (SO) e muitos sistemas podem oferecer suporte a *thread* pelas duas camadas, resultando em combinações e diversos modelos de *multithreading* (SILBERSCHATZ, 2000; SHAW, 2003).

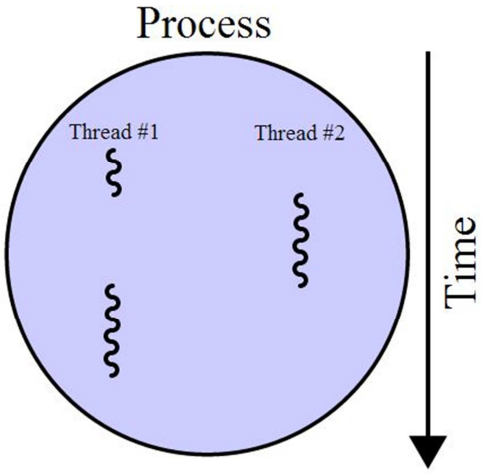

Figura 3: Um processo com dois threads. Fonte: Wikipédia, 2014.

Threads implementados na camada Kernel do SO são criados, escalonados e gerenciados pelo Kernel. Como o gerenciamento é feito pelo Kernel, quando uma tarefa de um *thread* tenta realizar um acesso que é bloqueado pelo SO, apenas o *thread* é bloqueado e não todo o processo. E em ambiente multiprocessado, o Kernel tem a capacidade de escalonar os *threads* em diferentes processadores, fazendo com que o processamento seja realmente paralelo. Diferente de KLT, *Threads* implementados pela camada Usuário do SO são subsidiados pela linguagem de programação, portanto, são escalonados e gerenciados por uma biblioteca de t*hread* implementado diretamente pelo programador. Como o gerenciamento é feito na camada usuário, o *multithreading* é invisível ao Kernel, portanto, quando uma tarefa de um *thread* tenta realizar um acesso que é bloqueado pelo SO, todo o processo é bloqueado porque o Kernel enxerga todo o processo como um único *thread*. A figura 4 ilustra a comparação entre os dois métodos de implementação de *thread.* (ARAÚJO, 2013; The University of Texas, 2011; FARINES, 2000)

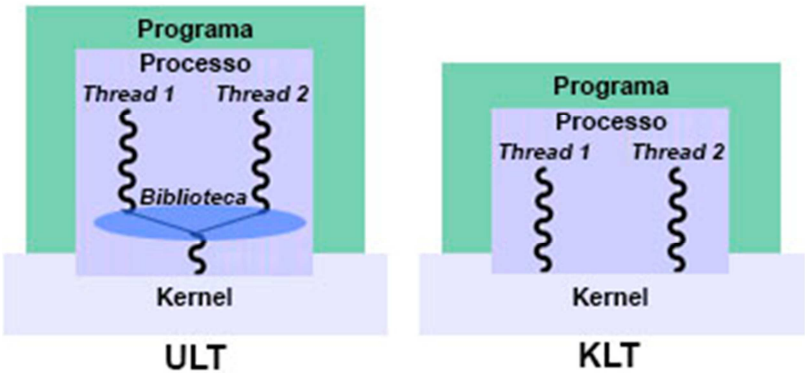

Figura 4: User-Level Thread x Kernel-Level Thread. Fonte: Elaboração própria, 2014.

Em NXC, o *multithreading* é programado através da função *Precedes* que executa, simultaneamente, as tarefas passadas por parâmetro de forma paralela ao *main*. Normalmente, em outras linguagens de programação um *thread* pode ser iniciado a qualquer momento do processo que o invoca (chamado também por processo pai), porém em NXC, utilizando a função *Precedes*, a instrução deve aparecer na última linha de código do processo pai, porque os *threads* serão inicializados apenas após o processo pai ser executado até a última linha de comando. Apesar da função *Precedes* aparecer no final do código, o processo pai não é completamente finalizado porque ele continua realizando o controle dos *threads*, portanto, será encerrado apenas quando o (s) processo (s) filho (*thread*) for (em) encerrado (s). Segue abaixo um esboço do código fonte da implementação de *multithreading* no NXC usando a função *Precedes*.

```
     task acting(){//Thread de controle dos atuadores e comunicação
Bluetooth
          […]
     }
     task reading(){//Thread de sensoriamento da trajetória
          […]
     }
     task main(){// Fluxo principal
          […]
          Precedes(acting, reading); /* Chama a thread; as tarefas
para execução simultânea */
     }
```

Para o protótipo Vecitr existem dois *threads*: O objetivo de um dos *threads* é o controle de movimento, fazendo com que o carro não saia da faixa do percurso. Outra é no controle de carga e descarga dependendo da posição em que ele está localizado - se na Mesa XY ou na Esteira. Os *threads* funcionam simultaneamente e é muito importante lembrar que quando tarefas iniciadas estão em execução ao mesmo tempo, pode ocorrer resultados inesperados, quando mais de uma tarefa tenta acessar um mesmo recurso compartilhado. Para evitar esses problemas, foi utilizado uma técnica da programação concorrente chamada de semáforo (exclusão mútua).

O semáforo é um método de exclusão mutua para sincronização de processos ou *threads* para evitar que haja condição de disputa, que é uma falha por consequência de mais de um evento tentando utilizar o mesmo recurso compartilhado simultaneamente. A técnica do semáforo é implementada declarando uma variável do tipo *mutex* (*mutual exclusion;* exclusão mutua) que permite que a região crítica de cada evento (trechos do código onde as instruções não podem ser intercaladas com as de outros processos ou *threads*, pois utiliza algum recurso compartilhado), seja executada individualmente. A variável do tipo *mutex* é utilizada em conjunto com os comandos *Acquire* e *Release*. O comando *Acquire* deve ser utilizado imediatamente antes da primeira linha de código da região crítica e o comando *Release* deve ser utilizado imediatamente depois da última linha de código da mesma região. Segue abaixo um esboço da programação sobre uma implementação de semáforo. (ARAÚJO, 2013; LAPLANTE; OVASKA, 2004; BURNS; WELLINGS, 2009; BENEDETTELLI, 2007)

```
     mutex moveMutex //Declaração da variável do tipo mutex
     task reading(){//Thread de sensoriamento da trajetória
          […]
          Acquire(moveMutex);
          [Região crítica. Aqui utiliza-se pelo menos um recurso
compartilhado]
          Release(moveMutex);
          […]
     }
```

A comunicação entre o Vecitr e os outros protótipos é implementada através de comunicação interprocessual (*IPC – Inter-process Communication*) assíncrona utilizando o método de fila de mensagens com o recurso de mensagem do NXT (protocolo de

comunicação Bluetooth). A configuração da conexão dos elementos que participam da comunicação (mestre e escravos) e as mensagens a serem trocadas entre os CLPs NXT são realizados através de linhas de código na fase de projeto. Na comunicação, o Vecitr é o escravo e fica aguardando alguma mensagem de algum dos outros protótipos envolvidos no SIM, a fim de sinalizar que a carga está disponível para ser entregue da Esteira para o do Vecitr ou que a carga está disponível para ser recebida pela Mesa.

Os sensores de luz e o Bluetooth geram sinais que são interpretados para informar a posição do Vecitr em relação a rota e *status* do processo de carga e descarga, respectivamente. Os dados interpretados pelo programa geram interrupções internas que mudam o fluxo do programa e são tratadas em outro trecho do programa. Ao ocorrer uma interrupção interna, o sistema operacional nunca a ignora e a rotina de interrupção adequada é enviada pelo SO para a fila de processamento. As rotinas de interrupção controlam o acionamento dos servos motores responsáveis pelas trações nas rodas (responsáveis pelo deslocamento do Vecitr) e na caçamba (responsáveis pelo despejo da carga).

Há dois sensores de luz, um ao lado de cada roda tracionada que detectam o posicionamento do Vecitr em relação a faixa preta fixada sobre o piso para ilustrar a rota a ser percorrida pelo protótipo. A faixa preta deve ficar posicionada entre as rodas do Vecitr e quando este desvia da direção correta, um dos sensores cruza a faixa pela esquerda ou pela direita e o sensor de luz do respectivo lado detecta a presença da faixa e corrige a direção através de uma compensação na rotação da roda do lado oposto. Quando os dois sensores detectam a faixa preta, esta interrupção significa que o Vecitr completou o percurso. Ao finalizar a rota é iniciada a manobra de despejo da carga sobre o receptor, através do acionamento do motor da caçamba que a inclinará a um ângulo obtuso a fim de provocar o deslocamento da carga por conta da lei da gravidade.

CONSIDERAÇÕES FINAIS

Para a elaboração do protótipo Vecitr, foram levadas em consideração as dimensões físicas e a compatibilidade física e de comunicação com os demais protótipos envolvidos no sistema integrado de manufatura. No decorrer do projeto foram feitas diversas adaptações na estrutura, a fim de obter um protótipo com medidas compatíveis com os outros projetos e cada vez mais firme, compacto e leve, de forma a precisar de tão pouca velocidade angular nas

rodas tracionadas quanto fosse possível e consecutivamente, promover um deslocamento mais seguro e preciso do veículo e da carga.

As técnicas de tempo real empregadas garantiram o funcionamento esperado, estável e sem falhas do Vecitr. Ao final do projeto, o Vecitr atendeu as exigências de prazos de entrega e deslocamento seguro, fazendo com que o transporte de cargas acontecesse no tempo esperado e com a capacidade de interação com os outros protótipos que também compõe o SIM.

A utilização do kit de *smart brick* da linha LEGO Mindstorms NXT foi determinante para a praticidade do desenvolvimento do projeto, devido a sua facilidade e diversidade de componentes para montagem do protótipo. Desenvolver um protótipo para transporte de carga utilizando o KIT LEGO Mindstorms NXT e técnicas de programação em tempo real foi uma experiência rica e importante para a abordagem prática dos conteúdos estudados na disciplina de Sistemas de Tempo Real da Escola de Engenharia e TI da Universidade Salvador (Unifacs).

Substituir os seis CLPs por apenas um é uma proposta relevante para otimizar do SIM abordado neste trabalho. Desta forma, os protótipos deixariam de ser autônomos para serem totalmente dependentes de um processamento externo, o que seria uma desvantagem, entretanto por se tratar de um SIM que requer poucos recursos e processamento, esta proposta é relevante no ponto de vista de redução de custo, visando um sistema integrado de manufatura simular em um chão de fábrica.

REFERÊNCIAS

ARAÚJO, Rafael. Aulas de Sistemas de Tempo, 2013. Disponível em: <http://www.harpia.eng.br/str.html>. Acesso em: 15 novembro 2013.

LEJOS-OSEK. What is nxtOSEK?, 2009. Disponível em: <http://lejos-osek.sourceforge.net/whatislejososek.htm>. Acesso em: 15 novembro 2013.

TOPPERS. Toyohashi OPen Platform for Embedded Real-Time Systems, 2003. Disponível em: <http://www.toppers.jp/en/jsp-kernel-e.html>. Acesso em: 15 de novembro 2013.

The University of Texas. A User-Level Thread Package, 2011. Disponível em: <http://www.cs.utexas.edu/users/dahlin/Classes/UGOS/labs/labULT/proj-ULT.html>. Acesso em: 8 abril 2013.

SHAW, Alan C. Sistemas e software de tempo real. Porto Alegre: Bookman, 2003. 240 p.

TRON ASSOCIATION. µLTRON4.0: Specification, 2002. Disponível em: <http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf>. Acesso em: 04 de janeiro 2014.

LAPLANTE, Phillip A; OVASKA, Seppo. Real-time systems design and analysis. 3rd ed. Estados Unidos da América: Instutute of Electrical and Electronics Engineers, 2004. 505 p.

BURNS, Alan; WELLINGS, Andrew. Real-time systems and programming languages: Ada, Real-time Java and C/Real-time POSIX. 4rd ed. Inglaterra: Pearson Education Limited, 2009. 602 p.

BENEDETTELLI, Daniele. Programming LEGO NXT Robots using NXC, 2007. Disponível em: <http://www.bricxcc.sourceforge.net/nbc/nxcdoc/NXC_tutorial.pdf>. Acesso em: 22 de fevereiro 2014.

FARINES, Jean-Marie. Sistemas de Tempo Real, 2000. Disponível em: <http://www.das.ufsc.br/~romulo/livro-tr.pdf>. Acesso em: 11 de junho 2014.

SILBERSCHATZ, Abraham. Sistemas Operacionais. Rio de Janeiro: Campus Ltda, 2000. 585 p.