UMA METODOLOGIA DE PORTE DE SISTEMA OPERACIONAL DE TEMPO REAL PARA PLATAFORMAS DE HARDWARE

Tiago N. Ukei*, Denis S. Loubach*

*Laboratório de Computação Avançada, Controle e Sistemas Embarcados - FEM
Universidade Estadual de Campinas - UNICAMP
Campinas, SP, Brasil

Emails: tiago.ukei@gmail.com, dloubach@fem.unicamp.br

Abstract— Real-time embedded systems have become increasingly more complex, requiring the development of more reliable and more efficient code. Although the testing stage of these systems has been improved in the past few years, it is not able to simulate all the possible conditions to which the system will be exposed during its lifecycle, making it necessary to develop techniques and methodologies in order to increase quality and reliability of such systems. Despite the research performed on this subject, a general and unified method for porting a real-time operating system on a hardware platform is hardly found in the literature. This article presents, therefore, a methodology with this purpose, which was then tested and validated through FreeRTOS porting on an ARM platform. We suggest the use of our methodology in future works for further testing and possible improvements.

Keywords— Embedded systems, Porting, Methodology, RTOS.

Resumo— Sistemas embarcados de tempo real têm se tornado cada vez mais complexos, o que exige o desenvolvimento de códigos mais confiáveis e eficientes. Embora a fase de testes desses sistemas venha sendo aprimorada nos últimos anos, ela não é capaz de reproduzir todas as condições a que o sistema estará exposto durante seu funcionamento, tornando-se necessário o desenvolvimento de técnicas e metodologias para aumentar a qualidade e confiabilidade de tais sistemas. Apesar das pesquisas nessa área, ainda não se encontra na literatura um método único e generalista para realizar o porte de um sistema operacional de tempo real em uma plataforma de hardware. Este artigo apresenta uma metodologia com essa finalidade, a qual foi testada e validada por meio de um porte realizado do FreeRTOS em uma plataforma ARM. Sugere-se que a metodologia apresentada seja utilizada em trabalhos futuros para testes e eventuais aprimoramentos.

Palavras-chave— Sistemas embarcados, Porte, Metodologia, RTOS.

1 Introdução

Sistemas embarcados de tempo real estão presentes em praticamente todos os lugares: automóveis, aeronaves, eletrodomésticos, televisões, celulares, MP3, porteiros eletrônicos, entre muitos outros. Com as aplicações tornando-se cada vez mais complexas, surge a necessidade de um gerenciamento confiável e eficiente das tarefas executadas por um dispositivo. Portanto, os desenvolvedores de software têm se apoiado cada mais em sistemas operacionais para desempenhar suas funções.

Embora uma série de testes seja realizada antes da liberação do produto para o mercado, não é possível replicar todas as condições a que o sistema estará exposto quando colocado em funcionamento, ou seja, o teste de software não representa, por si só, uma solução para alcançar a previsibilidade nos sistemas de tempo real. Para se obter uma performance mais robusta sob várias condições de operação, é necessário utilizar metodologias de projeto avançadas, além de uma análise aprofundada do código fonte e mecanismos específicos do sistema operacional, projetados para suportar aplicações com restrições temporais (Buttazzo, 2011).

Apesar do número de pesquisas e estudos relacionados aos sistemas computacionais de tempo real na atualidade, não se encontra na literatura um método único e generalista que possa ser utilizado para realizar o porte de um sistema operacional de tempo real (*real-time operating system* - RTOS) em uma plataforma de hardware qualquer. A literatura apresenta, entretanto, alguns métodos tratando parte desta questão. Devido à complexidade envolvida nesse processo, é importante que haja uma sequência bem definida das atividades necessárias para que o porte do sistema seja efetuado com êxito.

Dado esse contexto, o objetivo deste artigo é apresentar uma metodologia geral de porte de um RTOS em uma plataforma de hardware, bem como validá-la por meio da implementação do porte do FreeRTOS em uma plataforma de desenvolvimento FRDM-KL25Z, seguindo os passos da metodologia apresentada.

O restante deste artigo é organizado nas seguintes seções. A Seção 2 traz uma revisão de literatura com as principais definições e trabalhos relacionados. Na Seção 3 é apresentada nossa proposta de metodologia de porte de um RTOS em forma de um fluxograma de processo, o qual é utilizado para a implementação de um porte específico, descrita na Seção 4. A Seção 5 apresenta os resultados obtidos do experimento e, por fim, na Seção 6, são apresentadas as conclusões e principais contribuições deste artigo.

2 Revisão de literatura

Sistemas embarcados com hardware relativamente simples (*e.g.*, relógio digital), onde não se justifica a adoção de um sistema operacional devido ao *overhead* introduzido na aplicação, são geralmente implementados com base em uma arquitetura *bare metal*. Em contrapartida, aplicações mais complexas, como um sistema aviônico, requerem um gerenciamento mais preciso do escalonamento das tarefas, o que justifica a utilização de um RTOS.

Segundo Barabanov (1997, p. 1), "um sistema operacional de tempo real é um sistema operacional capaz de garantir os requisitos de tempo dos processos sob seu controle". Ele deve escalonar a execução dos programas de maneira temporizada, gerenciar os recursos do sistema e prover suporte básico para sincronização, comunicação, temporização precisa, gerenciamento de dispositivos de entrada/saída (E/S) e uma fundação consistente para o desenvolvimento do código de aplicativo (Li and Yao, 2003; Stankovic and Rajkumar, 2004).

Para que o RTOS se comunique de forma adequada com o hardware, é necessário um conjunto de *drivers* específicos para os componentes de hardware do sistema, conhecido como *board support package* (BSP) (Li and Yao, 2003). Segundo Kumar et al. (2013), o BSP fornece o suporte mínimo para carregar o sistema operacional e contém os *drivers* para todos os dispositivos no sistema *target*, disponibilizando uma abstração do hardware para as camadas mais altas do software. De acordo com Hedlund and Aronson (2002), a estrutura de um BSP inclui a configuração da CPU, da memória, das interrupções de hardware, do *timer* do escalonador do sistema e os *drivers* de dispositivos.

O desenvolvimento de um BSP não é uma tarefa trivial, dependendo do nível de complexidade do hardware. Para se desenvolver um BSP é necessário conhecer a arquitetura de hardware do *target*, o fluxo de dados, o mapa de memória, o mapa de interrupções, além da linguagem de montagem específica para o microprocessador. Boa parte dos desenvolvedores, no entanto, utiliza o BSP de um porte anteriormente realizado para uma plataforma de hardware similar e faz as devidas adaptações no código para a aplicação específica.

A metodologia empregada por Young (2015) para realizar o porte do FreeRTOS em uma plataforma de desenvolvimento FRDM-KL25Z resume-se nos seguintes passos:

1. Pesquisar sobre a arquitetura do processador/esquema de gerenciamento de memória. Definir as constantes e macros apropriadas nos arquivos de cabeçalho, necessárias para a configuração do *clock*, tamanho do *heap* e da pilha, níveis de prioridade, etc;
2. Utilizar o código de um porte anterior para plataforma semelhante, realizar as modificações necessárias no arquivo `port.c`, implementando inicialmente a função de inicialização da pilha;
3. Determinar quais funções são dependentes de inicialização da pilha e implementá-las de forma incremental até que seja possível compilar o FreeRTOS; e
4. Implementar um BSP mínimo para suportar a utilização do LED RGB (testes de temporização).

No estudo apresentado por Schäfer (2007), em que é feito o porte do RTEMS em uma plataforma *Blackfin*, também é adotada uma abordagem incremental, iniciando-se por um porte simples e, posteriormente, adicionando-se funcionalidades mais complexas ao sistema, como rotinas de tratamento de interrupções. As tarefas essenciais apontadas pelo autor para que o porte fosse realizado com sucesso envolvem a implementação das funções de gerenciamento da troca de contexto e das rotinas de tratamento de interrupções, além dos *drivers* de dispositivos e configuração do *clock*. É necessário ainda configurar adequadamente algumas macros para a correta inicialização e utilização da pilha, tais como as que definem a direção de crescimento, o tamanho mínimo e o alinhamento.

No trabalho apresentado por Souza (2007) é desenvolvida um metodologia iterativa para realizar o porte do Linux em sistemas embarcados. A metodologia proposta consiste, basicamente, na detecção das partes dos códigos-fonte que devem ser modificadas para que seja realizado o porte do sistema operacional para a nova plataforma, respeitando-se os passos (Souza, 2007, p. 52): (i) avaliar os requisitos iniciais do projeto usando um modelo de tomada de decisão; (ii) identificar quais elementos de hardware não tem correspondente suporte de software; (iii) identificar quais códigos de software devem ser modificados; (iv) identificar as dependências entre os componentes de software; (v) identificar a precedência de realização das modificações.

Dentro deste contexto, nossa proposta de metodologia visa ser mais generalista e abstrata em comparação com as apresentadas até aqui, entretanto, ela ainda é realizada de forma manual.

Além disso, também se busca definir claramente, por meio de um fluxograma de processo, a sequência das atividades necessárias para realizar o porte de um RTOS qualquer em uma plataforma de hardware, conforme introduzido na seção a seguir.

3 Metodologia proposta para porte de um RTOS em uma plataforma de hardware

A nossa metodologia baseou-se num estudo previamente realizado de um BSP existente e funcional fornecido pelo KSDK (*Kinetis Software Development Kit*) para o porte do FreeRTOS para a plataforma FRDM-KL25Z, utilizando-se o *Processor Expert*, que é uma ferramenta integrada ao ambiente de desenvolvimento para auxiliar na geração automática de código.

Após a análise da estrutura e a implementação do código, foi desenvolvida uma metodologia que visa fornecer os passos, em aspectos gerais, para a realização do porte de um RTOS qualquer em uma plataforma de hardware, sem considerar suas peculiaridades. A Figura 1 mostra o fluxograma do processo de porte proposto. Os pontos mais importantes desta metodologia serão discorridos adiante.

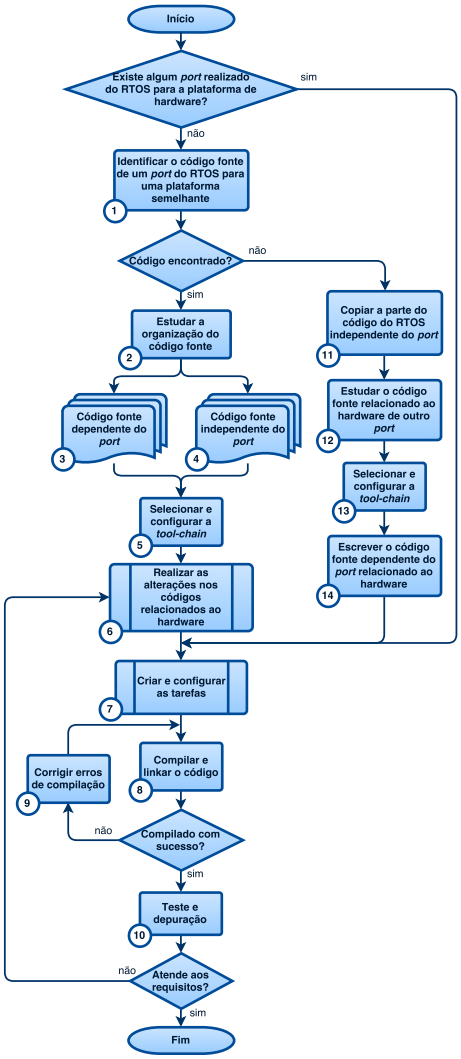

Figura 1: Fluxograma do processo de porte segundo a metodologia proposta.

Recomenda-se que os códigos fonte do *kernel* e do BSP sejam divididos em dois diretórios principais: um, contendo os arquivos dependentes do porte e outro, os independentes. Os arquivos específicos ao *target* (ID 3) são responsáveis pela configuração do *clock*, do *timer* gerador de *ticks* (interrupções periódicas para o escalonador), do controlador de interrupções, da memória e pela troca de contexto. A parte do *kernel* independente do porte (ID 4) está relacionada essencialmente à manipulação de tarefas, filas de mensagens, *mutexes*, listas e corrotinas.

O processo ID 6, detalhado na Figura 2, é o passo central desta metodologia, o qual consiste em realizar as alterações nos códigos relacionados ao hardware. Esse passo pode ser dividido em três blocos:

- **Inicialização do hardware:** compreende os códigos que são essenciais para a inicialização do sistema, porém são independentes do RTOS sendo portado, tais como o vetor de inicialização e as funções de configuração do *clock* e PLL (*Phase Locked Loop*).
- ***Port* do RTOS:** são os arquivos de configuração que adaptam o RTOS para ser executado na plataforma de hardware específica.
- ***Drivers* de dispositivos:** é a parte do BSP adicional, necessária para a utilização dos periféricos pelo aplicativo.

Muitos microprocessadores possuem o código em *assembly* do vetor de inicialização disponível para download, fornecido pelos próprios fabricantes. No entanto, caso seja necessário implementá-lo, deve-se recorrer à documentação do microprocessador, que possui todas as informações necessárias para sua configuração, tais como os modos de operação do processador, a configuração da pilha, o mapa de registradores, de memória e do vetor de inicialização, o gerenciamento de energia, o conjunto de instruções e os periféricos do processador.

Outro documento essencial para o porte é o *datasheet* ou manual de referência do microcontrolador, o qual contém todas as informações necessárias para configurar o hardware, incluindo o *clock*, o controlador de interrupção, os *timers* e demais periféricos. Entre essas informações, destacam-se o endereço dos registradores, os bits de configuração e os modos de operação de cada dispositivo. Um arquivo de cabeçalho contendo o mapeamento dos registradores dos periféricos é geralmente fornecido pelo fabricante do microcontrolador ou desenvolvedores *third party*.

Para realizar o porte do sistema operacional, é necessário ainda consultar o manual de porte e a documentação fornecida pelos desenvolvedores do RTOS, que servem de guia para a implementação da interface completa para que o sistema operacional se comunique adequadamente com o hardware.

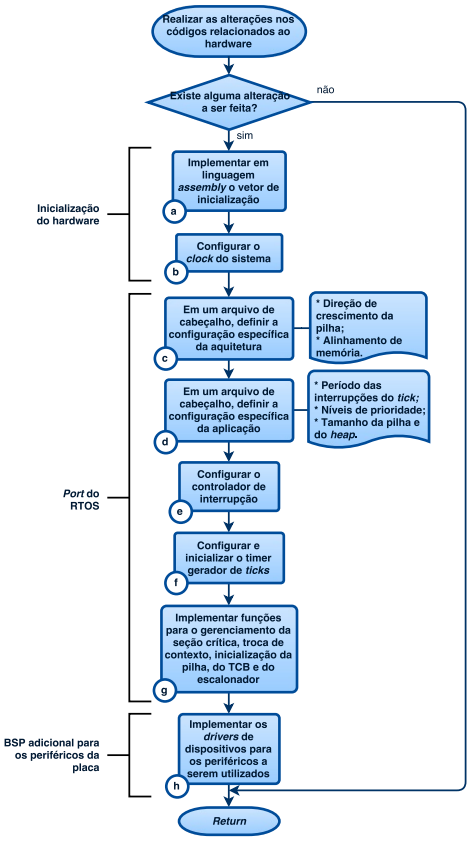

Figura 2: Desdobramento do processo ID 6 da metodologia apresentada.

Em geral, essa interface é composta por uma lista de constantes que devem ser definidas e um conjunto de funções que devem ser implementadas.

Os passos subsequentes da metodologia são necessários para verificar e validar o porte realizado, apesar de não fazerem parte do processo de porte propriamente dito. O passo ID 7, expandido na Figura 3, consiste em implementar um código mínimo de aplicativo e a função `main()` para verificar se o porte foi realizado com sucesso.

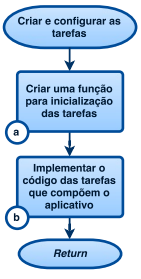

Figura 3: Desdobramento do processo ID 7 da metodologia apresentada.

O passo ID 8 equivale à geração da imagem executável e carregamento no *target*. Em caso de erro de compilação, deve-se realizar as correções necessárias (ID 9) e recompilar o código, em um processo iterativo, até que se obtenha a imagem executável com êxito.

Por fim, deve-se testar e depurar o programa (ID 10) para verificar se o sistema atende aos requisitos funcionais e, principalmente, temporais, validando o processo de porte do RTOS. Recomenda-se fortemente que esse processo seja iterativo e implementado gradativamente, isto é, deve-se compilar e testar o programa inicialmente com uma estrutura básica para seu funcionamento e então adicionar, de forma gradual, as funcionalidades do RTOS no programa de usuário.

4 Teste e validação da metodologia

Para validação da metodologia proposta, foi realizado o porte do FreeRTOS em uma plataforma FRDM-KL25Z e de um programa aplicativo básico capaz de testar algumas funcionalidades críticas do RTOS, tais como temporização, escalonamento de tarefas e bloqueio de recursos. O diagrama de requisitos do sistema é ilustrado na Figura 4.

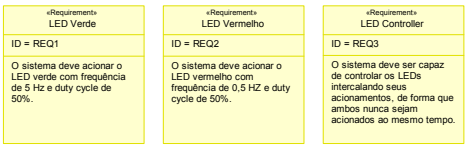

Figura 4: Diagrama de requisitos do sistema.

Primeiramente foi feito um estudo do código do BSP obtido com o auxílio do *Processor Expert*, bem como de sua organização. Para organizar o código fonte do RTOS conforme a metodologia, foi consultado o guia de porte do FreeRTOS, o qual informa a estrutura de diretórios e a organização do código fonte do sistema operacional. Em seguida, realizou-se o download dos arquivos fonte do FreeRTOS a partir de seu site oficial (`http://www.freertos.org`) e, então, organizou-se o código da maneira apresentada na Figura 5, a qual mostra o diagrama de pacotes utilizado na modelagem do programa implementado no experimento.

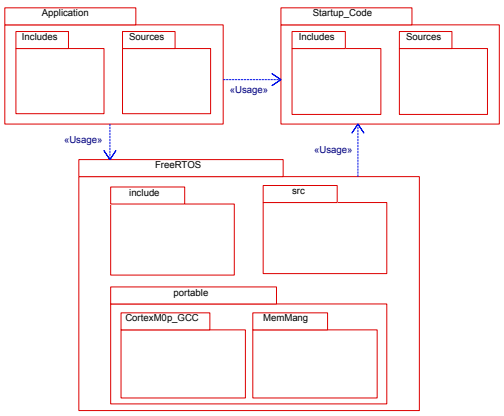

Figura 5: Diagrama de pacotes implementados.

Conforme os passos ID 3 e 4 da nossa metodologia, o código fonte do FreeRTOS foi dividido em dois diretórios separados. No diretório `portable` foram colocados os arquivos que dependem do porte, os quais ainda foram subdivididos nas pastas CortexM0p_GCC e MemMang. Na primeira pasta foram colocados os arquivos `port.c` e `portmacro.h`, que dependem do processador e do compilador, e na segunda, foi copiado o arquivo que gerencia a memória *heap*. No outro diretório (`src`) foram colocados os arquivos do *kernel* que não são específicos ao porte realizado, tais como `tasks.c`, `list.c` e `queue.c`.

Foi configurado então o ambiente de desenvolvimento KDS (*Kinetis Design Studio*) e criado um novo projeto do *Processor Expert*, chamado `tcc_blink`, porém o KSDK não foi utilizado para a geração automática de código. Dessa forma, o projeto importou apenas o `makefile`, os arquivos de configuração do *linker* e do *debugger* e os arquivos específicos ao microcontrolador, fornecidos pela própria NXP, fabricante da plataforma.

A *tool-chain* utilizada para geração da imagem executável foi a *GNU ARM Embedded Toolchain* `arm-none-eabi-`.

Os arquivos responsáveis pela inicialização do hardware, relacionados aos passos 6.a e 6.b da Figura 2, são definidos no padrão CMSIS (*Cortex Microcontroller Software Interface Standard*) e fornecidos pela NXP. O CMSIS-Core define os arquivos `startup_MKL25Z.S` e `system_MKL25Z4.c`, contidos na pasta `Startup_Code`.

No arquivo `system_MKL25Z4.h` foi selecionada a configuração desejada para o *clock* do sistema - modo PEE (*PLL Engaged External*), que usa um cristal externo de 4 MHz e um PLL para alcançar a frequência de 48 MHz de alta precisão.

As definições do passo 6.c foram realizadas no arquivo `portmacro.h`. O arquivo `FreeRTOSConfig.h` contém uma lista de definições para configurar o RTOS conforme as necessidades do software aplicativo (ID 6.d). Os passos 6.e a 6.g foram implementados no arquivo `port.c`, o qual contém as funções necessárias para que o RTOS execute na plataforma de hardware utilizada.

Os arquivos contidos no diretório `Application` foram então implementados para executar as tarefas do aplicativo. Os arquivos `led_driver.c` e `led_driver.h` implementam um *driver* mínimo para utilização do LED RGB contido na plataforma de desenvolvimento (ID 6.h).

O passo ID 7 consiste na criação e configuração das tarefas `Task1` e `Task2` do software aplicativo, de forma a satisfazer os requisitos do sistema.

Por fim, o programa foi compilado e testado de forma iterativa, até que os requisitos do sistema fossem satisfeitos, conforme sugerem os passos ID 8 a 10 da metodologia.

O diagrama de definição de blocos dos principais componentes do sistema são apresentados na Figura 6. O código completo pode ser encontrado em <`https://github.com/tiagoukei/tcc_blink`>.

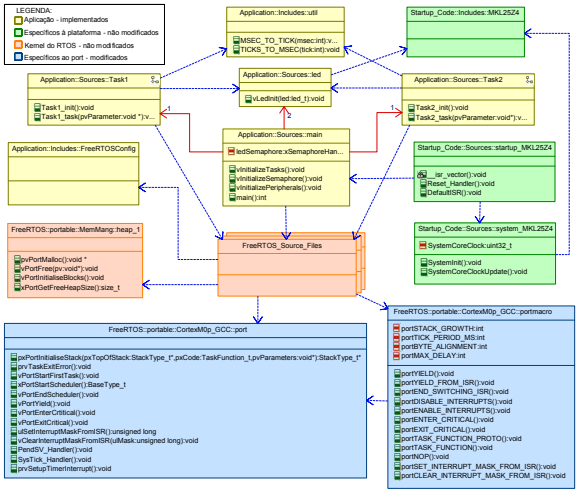

Figura 6: Diagrama de definição de blocos.

Os blocos destacados em amarelo na Figura 6 constituem o software aplicativo, implementado pelo usuário, e o arquivo `FreeRTOSConfig.h`, que faz parte do código baixado do site do FreeRTOS, onde foram definidas as constantes para customizar o *kernel* do sistema operacional para a aplicação.

Os blocos verdes são específicos ao hardware e fornecidos pela NXP, enquanto os laranjas são os arquivos fonte do RTOS, os quais não sofreram modificação.

Já os blocos em azul são os arquivos centrais do porte, que permitem a comunicação do software com o hardware. Ambos os arquivos (`port.c` e `portmacro.h`) foram reaproveitados de um porte anterior realizado em um processador Cortex-M0, considerando as modificações necessárias para adaptá-los para o Cortex-M0+.

5 Resultados

O sistema utilizado para validação da nossa metodologia, apesar de ser simples, possui várias características dos sistemas embarcados de tempo real, incluindo restrições temporais bem definidas, tratamento de interrupções, troca de contexto e gerenciamento de recursos compartilhados, bem como acesso a periféricos da plataforma de desenvolvimento.

A Figura 7 ilustra as tensões nos pinos de saída do microcontrolador obtidas com a utilização de um osciloscópio de dois canais. Observa-se que os requisitos característicos de sistemas de tempo real mencionados anteriormente são satisfeitos com precisão ($\varepsilon < 1$ms). Esta figura também ilustra a parte de troca de contexto das tarefas.

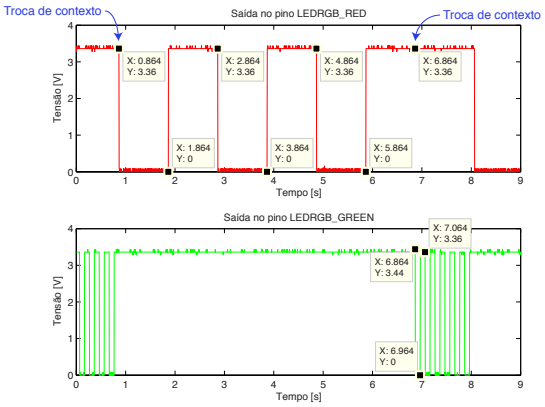

Figura 7: Tensão nos pinos que acionam os LEDs.

É importante ressaltar também que a quantidade de memória obtida no porte seguindo a nossa metodologia (18.556 bytes) foi reduzida em relação a obtida segundo o *Processor Expert* (20.872 bytes), o que equivale a uma redução de aproximadamente 11%, decorrente do fato de o programa ser customizado para a aplicação, sendo removidas todas as partes não utilizadas.

6 Conclusões

Este artigo apresentou uma metodologia geral de porte de um RTOS em uma plataforma de hardware, bem como sua validação por meio da implementação do porte do FreeRTOS na plataforma FRDM-KL25.

A metodologia apresentada mostrou-se adequada para o porte realizado, uma vez que satisfez os requisitos de tempo real do sistema. O fluxograma proposto fornece uma sequência de passos bem definidos, evitando que o desenvolvedor escreva código desnecessário, ao mesmo tempo em que garante que as atividades essenciais para o porte sejam realizadas, minimizando os erros decorrentes do processo.

Nossa metodologia difere-se das estudadas na literatura no sentido em que define claramente, por meio de um fluxograma de processo, a sequência das atividades necessárias para realizar o porte de um RTOS e também fornece uma abstração que procura viabilizar a aplicação desses conceitos para qualquer combinação de sistema operacional e plataforma de hardware, embora a implementação do BSP seja bastante específica.

Pretende-se, em trabalhos futuros, realizar o porte de diferentes RTOSs em diversas plataformas utilizando-se desta metodologia, visando expandir a validação de sua aplicabilidade e, eventualmente, aprimorá-la.

No contexto da Internet das Coisas, bastante difundido atualmente, intentamos realizar o porte dos RTOSs Zephyr (*Zephyr Project*, 2016) e Xenomai (Barbalace et al., 2008) em diferentes plataformas de hardware, como por exemplo *Beaglebone Black, Edison e Raspberry*. Sugere-se ainda, estudar a automatização em alguns aspectos do processo de porte, com o intuito de minimizar eventuais erros de codificação manual introduzidos durante sua geração.

Agradecimentos

Agradecemos à UNICAMP e a FEM por fornecerem os meios necessários para o desenvolvimento deste trabalho que foi parcialmente suportado pelo processo no 2014/24855-8, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)[1].

Referências

Barabanov, M. (1997). A Linux-based real-time operating system.

Barbalace, A., Luchetta, A., Manduchi, G., Moro, M., Soppelsa, A. and Taliercio, C. (2008). Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application, *IEEE Transactions on Nuclear Science* **55**(1): 435–439.

Buttazzo, G. C. (2011). *Hard Real-Time Computing Systems*, 3 edn, Springer US, New York.

Hedlund, M. and Aronson, F. (2002). Evaluation of real-time operating systems for safety-critical systems.

Kumar, K. E., Kamaraju, M. and Yadav, A. K. (2013). Porting and BSP customization of Linux on ARM platform, *International Journal of Computer Applications* **80**(15): 36–42.

Li, Q. and Yao, C. (2003). *Real-Time Concepts for Embedded Systems*, CMP Books, Berkeley.

Schäfer, P. A. (2007). Dynamic loading and linking native code on a real-time operating system.

Souza, O. (2007). Metodologia para porte do sistema operacional Linux para sistemas embarcados.

Stankovic, J. A. and Rajkumar, R. (2004). Real-time operating systems, *Kluwer Academic Publishers* **28**: 237–253.

Young, R. (2015). Porting a real-time operating system and developing a board support package.

Zephyr Project (2016). `https://www.zephyrproject.org`.

[1] As opiniões, hipóteses e conclusões ou recomendações expressas neste material são de responsabilidade dos autores e não necessariamente refletem a visão da FAPESP