UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

TRABALHO DE CONCLUSÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Implementação de um Sistema Operacional Simplificado de Tempo Real em Simulink com Geração Automática de Código Embarcado

Autor: Joel De Conto

Orientador: Prof. Dr. Marcelo Götz

Porto Alegre, dezembro de 2017

Sumário

Agradecimentos

Gostaria de agradecer a minha família, que me apoiou e incentivou desde o início de meus estudos, ajudando assim no meu desenvolvimento pessoal e profissional.

Agradeço a minha namorada Jennifer, pelo amor, carinho e paciência, mesmos nos momentos de dificuldade nestes últimos anos, além das convenientes instruções para aprimorar a escrita deste trabalho.

Agradeço ao professor Marcelo Götz pelo tempo dedicado e os ensinamentos transmitidos na orientação deste trabalho.

Agradeço também aos demais professores, colegas, amigos e funcionários da UFRGS que de alguma forma colaboraram para minha formação.

Resumo

Considerando o crescente avanço de tecnologias utilizadas em hardwares computacionais, a capacidade e poder de processamento de sistemas embarcados também é aprimorado, demandando cada vez mais tempo e conhecimento para serem programados. Algumas formas de facilitar e agilizar este processo são a implementação de estruturas que gerenciem os recursos disponíveis e a utilização de linguagens de programação de alto nível. Desta forma, este trabalho objetiva desenvolver um sistema operacional simplificado no ambiente de diagrama de blocos do software Simulink, para que possa ser utilizado na programação de sistemas embarcados. Para isso, será analisado um sistema operacional escrito na linguagem C++ para a plataforma Arduino, servindo como base para o desenvolvimento da lógica implementada no Simulink. Pelas verificações realizadas, observa-se que o sistema operacional apresenta um comportamento previsível e satisfatório, tanto na simulação quanto na implementação em hardware, o que permite concluir que os objetivos deste trabalho foram alcançados, disponibilizando uma estrutura de gerenciamento de tarefas que facilita o ciclo de programação.

Palavras chaves

Sistema embarcado; Sistema operacional; Tempo Real; Simulink; Arduino.

Abstract

Considering the increasing advance of technologies used in computer hardware, the capacity and processing power of embedded systems is also developed, demanding more time and knowledge to be programmed. Some ways to facilitate and speed up this process are to implement structures that manage the available resources, and the use of high-level programming languages. Therefore, this work aims to develop a simplified operating system in the block diagram environment of the Simulink software, in order that it can be used in the programming of embedded systems. For this purpose, an operating system written in the C ++ language for the Arduino platform will be analyzed, as a basis for the development of the logic implemented in Simulink. Through the verification performed, it is observed that the operating system presents a predictable and satisfactory behavior, both in simulation and hardware implementation, which allows concluding that the objectives of this work were achieved, providing a task management structure that facilitates the programming.

Keywords

Embedded system; Operational system; Real time; Simulink; Arduino.

Lista de Figuras

Lista de Tabelas

1 Introdução

Sistemas embarcados fazem parte do cotidiano. Mesmo passando despercebidos na maioria das vezes, realizam funções importantes na vida moderna, atuando em uma vasta gama de equipamentos, desde dispositivos simples, de uso pessoal, até complexos sistemas industriais. Estima-se que mais de 98% dos processadores utilizados no mundo estejam operando em sistemas embarcados e, embora possam não ser considerados computadores por alguns, podem conter *softwares* bastante avançados (TAURION, 2005).

Considerado um dos primeiros sistemas embarcados construídos, o AGC (*Apollo Guidance Computer*) foi fabricado no laboratório de instrumentação do MIT a pedido da NASA para ser utilizado em módulos espaciais. Embora simples, este sistema se mostrou bastante confiável, realizando diversos voos sem falhas, incluindo nove viagens até a lua e seis pousos lunares (MIT AEROASTRO, 2017). Desde então, seguindo um crescimento já previsto pela lei de Moore, a tecnologia empregada nestes dispositivos tem se desenvolvido em uma escala exponencial, ao passo que, atualmente, uma pequena placa de desenvolvimento que é comumente encontrada no mercado possui maior capacidade de processamento e memória do que o controlador dos módulos espaciais da década de 60.

O desenvolvimento destes sistemas, aliado ao avanço de *hardwares*, com maior capacidade e poder de processamento, demanda cada vez mais conhecimento por parte do programador. Desta forma, é importante encontrar meios de facilitar este processo de programação e torná-lo mais ágil e intuitivo, auxiliando não só processos que necessitem desta rapidez, seja em teste ou prototipagem, mas também estudantes e iniciantes que estão dando os primeiros passos na área e ainda estão aprimorando suas habilidades e conhecimentos na programação de sistemas embarcados.

Uma maneira de facilitar este processo é utilizar uma linguagem de programação com um nível mais alto de abstração, de forma que alguns processos sejam encobertos aos olhos do programador e automaticamente executados ao compilar o código (HERNYÁK, 2012). Assim, o programador pode focar suas atenções no desenvolvimento do processo em si, sem preocupar-se com a sintaxe de programação ou outros recursos que talvez não sejam tão importantes no momento. Um exemplo disso é a plataforma Arduino, que utiliza originalmente uma linguagem de programação baseada em C/C++ (ARDUINO 2017a), mas com simplificações que facilitam o aprendizado diminuem o tempo necessário para a programação.

Embora seja vantajoso para o programador realizar tarefas com maior facilidade em menos tempo, em algumas ocasiões, os métodos utilizados para estes fins acarretam no desperdício do poder computacional do *hardware*, ou na perda de características importantes para o processo, como a execução de tarefas periódicas em tempos definidos. Os chamados sistemas de tempo real são os que possuem restrições de tempo bastante precisas (BUTTAZZO, 2011), e muitas vezes a linguagem utilizada para a programação não permite o controle necessário das tarefas para que o sistema consiga executá-las de modo coerente e respeitando os requisitos temporais. Uma maneira de contornar este problema é a utilização de um sistema operacional de tempo real, que irá gerenciar os processos e recursos do processador de forma a aprimorar a eficiência da execução de tarefas e garantir o cumprimento de requisitos temporais (TANENBAUM, 2010). Assim, é conveniente perguntar-se: é possível implementar uma estrutura de

gerenciamento de tarefas em tempo real em um sistema embarcado a partir de um modelo em um nível mais alto de abstração?

A proposta deste trabalho é implementar um sistema de gerenciamento de tarefas em tempo real no ambiente de diagramas de blocos do Simulink, realizando a geração automática de código embarcado em C/C++ a partir do modelo construído. Esta aplicação, que pode ser considerada um sistema operacional básico, será implementada inteiramente a partir de blocos e estruturas de bibliotecas disponíveis para o Simulink, baseando-se em um sistema operacional escrito em C++ para a plataforma Arduino. O sistema será elaborado de forma a poder ser reutilizado posteriormente, com tarefas customizáveis, dentro de certas limitações.

Outra vantagem desta abordagem é a possibilidade de simulação das tarefas (pertinentes à aplicação foco do sistema) em conjunto com o sistema operacional necessário para gerência destas tarefas. Prática esta que muitas vezes é negligenciada nas análises, não permitindo uma otimização na utilização dos recursos computacionais

Posteriormente, um código embarcado contendo a rotina completa do sistema operacional e das tarefas implementadas pode ser gerado através do *Simulink Coder* (MATHWORKS, 2017a). Após a programação do sistema completo, o código gerado pode ser transferido para uma placa Arduino Uno, que foi escolhida por ter grande disponibilidade no mercado por um baixo custo, sendo assim de grande popularidade e aceitação do público, além de possuir capacidade suficiente para esta aplicação.

Para validar este sistema serão propostos diferentes conjuntos de tarefas, de modo a demonstrar a capacidade do sistema de executá-las de modo periódico e previsível.

1.1 Objetivos

O objetivo principal deste trabalho é realizar de maneira automatizada a programação de um sistema operacional simplificado de tempo real em um sistema embarcado, juntamente com as tarefas da aplicação, a partir do modelo do sistema em um nível maior de abstração.

Os objetivos específicos são:

- Implementar um sistema operacional simplificado no ambiente de programação do Simulink, baseando-se em um sistema escrito em C++ para a plataforma Arduino;
- Realizar a geração automática de um código embarcado a partir do sistema criado e a transferência para um microcontrolador Arduino Uno;
- Criar um método para a validação do sistema, de forma a verificar que a atuação do sistema de chamada de tarefas ocorre do modo previsível, de acordo com o período e a duração de cada tarefa.

1.2 Estrutura do trabalho

No Capítulo 2 é apresentada uma breve revisão bibliográfica, com os principais conceitos sobre sistemas embarcados e sistemas operacionais. No Capítulo 3 são apresentados os materiais utilizados, como o *hardware* e *software*, explicando algumas ferramentas utilizadas para a construção do sistema. No Capítulo 4 é mostrado o desenvolvimento do projeto, como a construção do sistema operacional foi elaborada com base em um código já escrito No Capítulo 5 são apresentados e analisados os testes de validação e os resultados obtidos após a implementação do sistema. No Capítulo 6 estão dispostas as principais conclusões deste trabalho e alguns pontos que servem como sugestões para trabalhos futuros.

2 Revisão Bibliográfica

Neste capítulo será apresentada uma breve revisão sobre os principais conceitos utilizados neste trabalho. Serão introduzidas as definições de sistemas embarcados e sistemas operacionais, bem como os seus princípios de funcionamento.

2.1 Sistemas embarcados

Segundo Heath (2003), um sistema embarcado é um dispositivo baseado em um microprocessador que é construído para controlar uma função ou uma série de funções específicas, não sendo projetado para ser programado pelo usuário final, da maneira como é programado um computador pessoal. Conforme Barr (2017), um sistema embarcado também pode ser parte de um conjunto maior de processos. Desta forma, tanto um eletrodoméstico quanto um sistema de controle antitravamento de freios em um automóvel podem ser considerados embarcados. Como o sistema é totalmente dedicado à função que desempenha, o projeto geralmente é otimizado para isso, reduzindo o tamanho da placa e simplificando demais propriedades, o que acarreta em uma redução no consumo de energia, característica importante em dispositivos portáteis.

Todo sistema embarcado contém algumas estruturas que o caracterizam, como o processador e o *software*. Para ter um programa salvo e poder acessá-lo, também se faz necessário uma memória. Além disso, também são necessárias entradas e saídas, que realizam a integração do sistema com os periféricos e, por sua vez, com o ambiente externo. As entradas geralmente são associadas a sensores, sinais de comunicação, botões e dispositivos de controle, enquanto as saídas são tipicamente associadas a displays, sinais de comunicação ou mudanças no ambiente físico por meio de atuadores (BARR, MASSA, 2009). A Figura 1 a seguir mostra a estrutura simplificada de um sistema embarcado genérico.

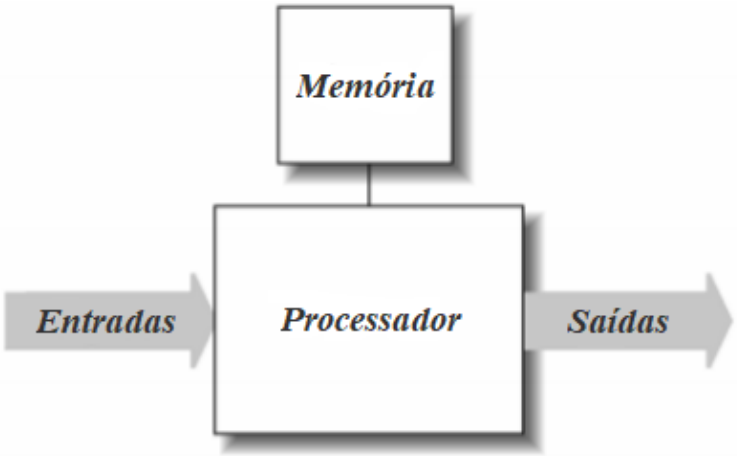

Figura 1 – Estrutura de um sistema embarcado genérico
Adaptado de Barr e Massa (2009)

2.1.1 Arduino

O Arduino é uma plataforma de prototipagem eletrônica de *hardware* e *software* livres, composta por uma placa com um microcontrolador e um ambiente de programação. O Arduino utiliza originalmente uma linguagem de programação baseada

em C/C++ (ARDUINO 2017b), que possui simplificações para auxiliar e agilizar a programação. O Arduino tornou-se popular pelo seu baixo custo de aquisição e grande disponibilidade no mercado, além da facilidade e versatilidade que propicia no aprendizado e confecção de sistemas eletrônicos diversos.

O fabricante oficial do Arduino disponibiliza diversos modelos, que se diferenciam no processador utilizado, número de entradas e saídas, além do tamanho das placas. Como todos os projetos, esquemas e códigos da plataforma são livres, há diversos modelos genéricos no mercado, geralmente com custo mais baixo do que as placas originais, além de diversos dispositivos derivados e acessórios.

2.2 Sistemas operacionais

Segundo Oliveira, Carissimi e Toscani (2009), um sistema operacional é uma camada de *software* que realiza a integração entre o *hardware* e os programas que executam tarefas para os usuários, sendo também responsável pelo acesso aos periféricos. Desta forma, o usuário não necessita conhecer detalhes do *hardware*, uma vez que o sistema operacional realiza o gerenciamento dos recursos.

O principal objetivo de um sistema operacional é tornar a utilização do computador mais eficiente e conveniente, com a abstração do *hardware* (plataforma computacional) utilizado para o programador. Uma utilização mais eficiente é obtida através de uma distribuição adequada de seus recursos entre os processos, ou seja, o sistema deve gerenciar a utilização dos componentes do *hardware* entre as tarefas solicitadas ao processador, como espaço na memória, tempo de processador, entre outros. Uma utilização mais conveniente é obtida escondendo-se do programador detalhes do *hardware* e dos periféricos, assim o programador não necessita saber como funcionam todos os processos que devem ser executados para realizar a tarefa solicitada, tornando assim a utilização mais confortável.

O princípio de funcionamento de um sistema operacional que atua em um sistema embarcado é similar ao dos executados em computadores, embora algumas simplificações possam ser aplicadas. Os sistemas embarcados não aceitam *softwares* instalados pelo usuário, ou seja, é garantido que nenhum programa não confiável seja executado no dispositivo, pois toda programação que será executada já estará na memória. Assim, não há necessidade de proteção entre as aplicações (TANENBAUM, 2010). Em outras palavras, como a aplicação final do sistema não será customizável, o projetista pode executar o gerenciamento das tarefas de modo mais direto, sem precisar se preocupar com aplicações que possam variar durante o tempo e necessitar diferentes demandas do processador e seus periféricos.

Sistemas operacionais de tempo real são caracterizados por terem o tempo como um parâmetro fundamental, onde os prazos para a execução de diferentes tarefas são bastante rígidos. Assim, dois tipos de sistemas de tempo real podem ser identificados: sistemas de tempo real crítico, os quais devem garantir a execução de determinadas tarefas em tempos específicos; e sistemas de tempo real não críticos, nos quais embora não seja desejável, o descumprimento de algum prazo não causará nenhum dano permanente (TANENBAUM, 2010).

Uma das propriedades mais importantes que um sistema de tempo real deve apresentar, segundo Buttazzo (2011), é a previsibilidade. O sistema deve poder prever a evolução das tarefas e garantir que as restrições temporais sejam atendidas. Entretanto, a confiabilidade desta previsão depende de diversos fatores, que envolvem desde a arquitetura do *hardware*, até o algoritmo e a linguagem de programação utilizados.

Existe uma grande diversidade de algoritmos para o escalonamento de tarefas em tempo real, utilizados para gerenciar a chamada das tarefas que se deseja executar no sistema. O escalonador estabelece uma ordem na chamada das tarefas, que pode estar associada a uma ordem de prioridade. Nestas condições, tarefas com maior prioridade têm preferência de execução perante tarefas com menor prioridade.

As seguintes classificações podem ser atribuídas aos algoritmos de escalonamento:

- Preemptivo ou não preemptivo – algoritmos preemptivos permitem que a tarefa sendo executada possa ser interrompida antes de seu término para a execução de outra tarefa mais prioritária;

- Estático ou dinâmico – algoritmos estáticos são aqueles em que as decisões de escalonamento são realizadas com base em parâmetros fixos, designadas às tarefas antes de suas execuções. Algoritmos dinâmicos tem seu escalonamento baseado em parâmetros que podem variar durante a evolução do sistema;

Um exemplo de algoritmo utilizado no escalonamento de tarefas periódicas é o chamado *Rate Monotonic* (RM), que é implementado partir de uma regra simples, que atribui prioridade mais alta para tarefas que possuem maior frequência de solicitação de execução, ou seja, com menor período. Como os períodos desejados para a chamada das tarefas são constantes, pode ser considerado um algoritmo estático, com prioridade fixa. A preempção, ou interrupção das tarefas durante a execução, pode ser implementada ou não, dependendo da estrutura e da lógica a ser adotada.

Atualmente, alguns trabalhos têm sido desenvolvidos com a proposta de facilitar a programação de sistemas embarcados e manter as características de execução de tarefas em tempo real. Um exemplo disso é o trabalho de Moster (2015), que realizou a implementação de um sistema operacional de código aberto no Simulink utilizando blocos do tipo *S-function*, que realizam a importação de códigos escritos em C/C++. Posteriormente, um código embarcado foi gerado automaticamente e transferido para um microcontrolador, sendo projetado para o uso em um sistema de controle de voo não tripulado.

3 Materiais e Métodos

Neste trabalho, utilizou-se principalmente o ambiente de programação do Matlab/Simulink para realizar implementação do sistema operacional. Além disso, também foi utilizada uma placa de desenvolvimento Arduino Uno para realizar a implementação e validação experimental do sistema criado.

3.1 Programação através do Simulink

O Simulink é um ambiente de programação em blocos integrado ao Matlab, no qual é possível utilizar diagramas de blocos para realizar simulações e construção de modelos (MATHWORKS, 2017b). A vantagem da utilização do Simulink frente ao Matlab está justamente na facilidade da programação, uma vez que os blocos utilizados podem esconder rotinas complicadas, para as quais seriam necessárias diversas linhas de código para obter o mesmo resultado. A seguir serão apresentadas algumas ferramentas e blocos do Simulink que serão utilizadas neste trabalho.

3.1.1 Function-Call Subsystem

O bloco *Function-Call Subsystem* é um bloco que, diferente dos subsistemas convencionais do Simulink, somente é executado quando recebe um sinal de controle com um evento do tipo *function-call* (MATHWORKS, 2017b). Desta forma, é possível realizar a execução de um conjunto de blocos em instantes desejados e com o número de iterações necessário. A chamada para este bloco pode ser gerada por um *Stateflow chart*, ou por blocos do tipo *Function-Call Generator* ou *S-funtion.*

3.1.2 Stateflow Chart

A biblioteca *Stateflow* do Simulink possibilita a implementação de lógicas combinatórias e sequenciais na forma de máquinas de estados e diagramas de fluxo (MATHWORKS, 2017b). É possível criar lógicas para cada estado e condições para as transições utilizando código na linguagem C ou do Matlab. Além disso, o *Stateflow Chart* possui suporte para a geração automática de código e permite a geração de eventos do tipo *Fucntion-Call*.

3.1.3 Simulink Coder

O *Simulink Coder* é uma ferramenta capaz de gerar e executar códigos nas linguagens C e C++ a partir de diagramas de blocos do Simulink. O código gerado pode ser inclusive utilizado em aplicações de tempo real (MATHWORKS, 2017a). O *Simulink Coder* possui diversas opções de parametrização, permitindo ao usuário escolher o ambiente alvo da programação, que pode ser um sistema embarcado microprocessado, e o tipo de placa que será utilizada, como o Arduino Uno, dentre outras opções.

3.1.4 Simulink Support Package for Arduino Hardware

O *Simulink Support Package for Arduino Hardware* (Pacote de suporte do Simulink para placas Arduino) é um pacote de ferramentas que permite a integração direta do ambiente de programação do Simulink com os microcontroladores Arduino, fornecendo suporte para a geração de código embarcado através do *Simulink Coder*. O pacote também inclui uma biblioteca com blocos configurados para acessar as entradas e saídas e realizar a comunicação com as placas (MATHWORKS, 2017d).

3.1.5 Custom C/C++ Target – Arduino

A biblioteca *Custom C/C++ Target* é semelhante à biblioteca instalada através do *Simulink Support Package for Arduino Hardware*, porém contém alguns blocos adicionais que implementam funções específicas da plataforma Arduino, como o bloco *millis*, que fornece o valor do tempo em milissegundos desde o início da execução do código. Este bloco foi utilizado para substituir o bloco *Digital Clock,* da biblioteca padrão do Simulink, que não apresentou o comportamento esperado, pois a sua contagem de tempo era interrompida quando uma tarefa externa ao sistema operacional era iniciada, gerando assim um bloqueio na temporização. A Figura 2 mostra os blocos contidos na biblioteca instalada.

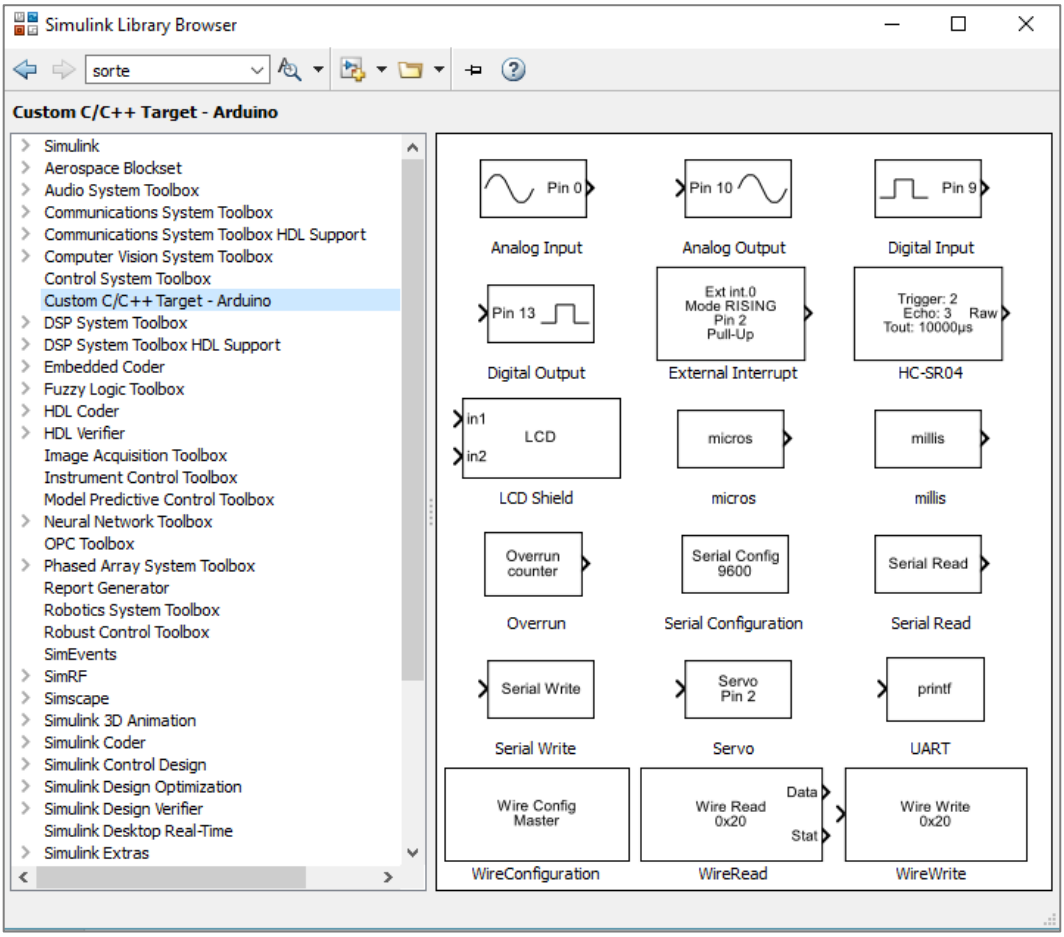

Figura 2 – Blocos disponíveis na biblioteca *Custom C/C+ Target - Arduino*
Fonte: autor

3.2 A Placa de desenvolvimento utilizada

Para realizar a implementação experimental do sistema criado, foi utilizada uma placa Arduino Uno. Esta placa possui as seguintes características, de acordo com o site do fabricante (Arduino, 2017b):

- Microcontrolador: ATmega328P, de 8 bits;
- Tensão de operação: 5V;
- 14 pinos de entrada/saída digital;
- 6 pinos de entrada analógica;
- Memória *Flash* de 32KB;
- Frequência de *Clock* de 16MHz.

A escolha desta placa é justificada pela alta disponibilidade no mercado, além de um baixo custo se comparada a placas de outros fabricantes. O modelo Uno é o mais recomendado para iniciantes, sendo assim o mais popular dentre as placas Arduino. A Figura 3 mostra uma fotografia da placa utilizada.

Figura 3 – Placa de desenvolvimento Arduino Uno
Fonte: Autor

4 Desenvolvimento do Trabalho

O primeiro passo na realização deste projeto foi a análise do código disponibilizado pelo professor Marcelo Götz, escrito na linguagem C/C++ para a plataforma Arduino, que será utilizado como base para a construção do sistema operacional no ambiente do Simulink.

O código em questão realiza a implementação de um sistema operacional simplificado em tempo real, com escalonamento não preemptivo, de forma a gerenciar o tempo em que diferentes tarefas são iniciadas. As tarefas podem ser escritas de acordo com a necessidade do programador, sendo preciso informar ao gerenciador apenas o período que se deseja iniciar a função.

O sistema operacional é dividido em três classes: *SchedulerClass*, *DispatcherClass* e *TaskClass*. A primeira é responsável pela ordenação temporal das tarefas, de acordo com a política de escalonamento do algoritmo *Rate Monotonic*. A segunda é responsável pelo chamamento da tarefa a ser executada (escolhida pelo escalonador). As tarefas da aplicação são criadas como objetos da classe *TaskClass*. Além disto, nas funções *loop()* e *setup()*, da arquitetura de software Arduino, são necessárias chamadas a métodos destas classes mencionadas. O código completo de cada classe pode ser consultado no Anexo A.

4.1 A Estrutura do Sistema Operacional

A base do funcionamento deste sistema operacional está na chamada de tarefas em tempos determinados, porém, a programação do Simulink não é determinística. Desta forma, foi necessário projetar uma estrutura que possibilite a execução de partes da programação somente quando houver uma solicitação do sistema principal. Assim, cada tarefa foi inserida em um bloco chamado *Function-Call Subsystem*, enquanto o sistema operacional foi inserido em um subsistema convencional.

A Figura 4 mostra a estrutura montada para a execução de três tarefas. O bloco RTOS contém a lógica de funcionamento do sistema operacional de tempo real, gerando sinais capazes de ativar a execução de um *Function-Call Subsystem*. Assim, cada saída do bloco RTOS é conectada a um subsistema com essa configuração, dentro do qual está construída a lógica para a tarefa. Neste caso, foram necessários três subsistemas, um para cada tarefa.

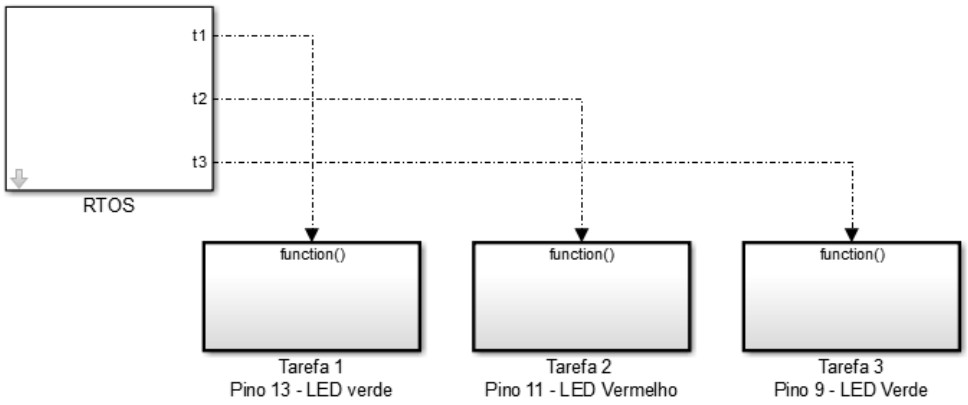

Figura 4 – Conexão principal do sistema operacional com as tarefas
Fonte: autor

A Figura 5 mostra as estruturas contidas no bloco RTOS. Percebe-se duas áreas principais, representando o escalonador (*scheduler*) e o expedidor (*dispatcher)*. Além disso, três blocos do tipo *Data Store Memory* foram utilizados para salvar variáveis que devem ser alteradas somente em determinados trechos do código e devem ser acessíveis tanto do escalonador quanto do expedidor.

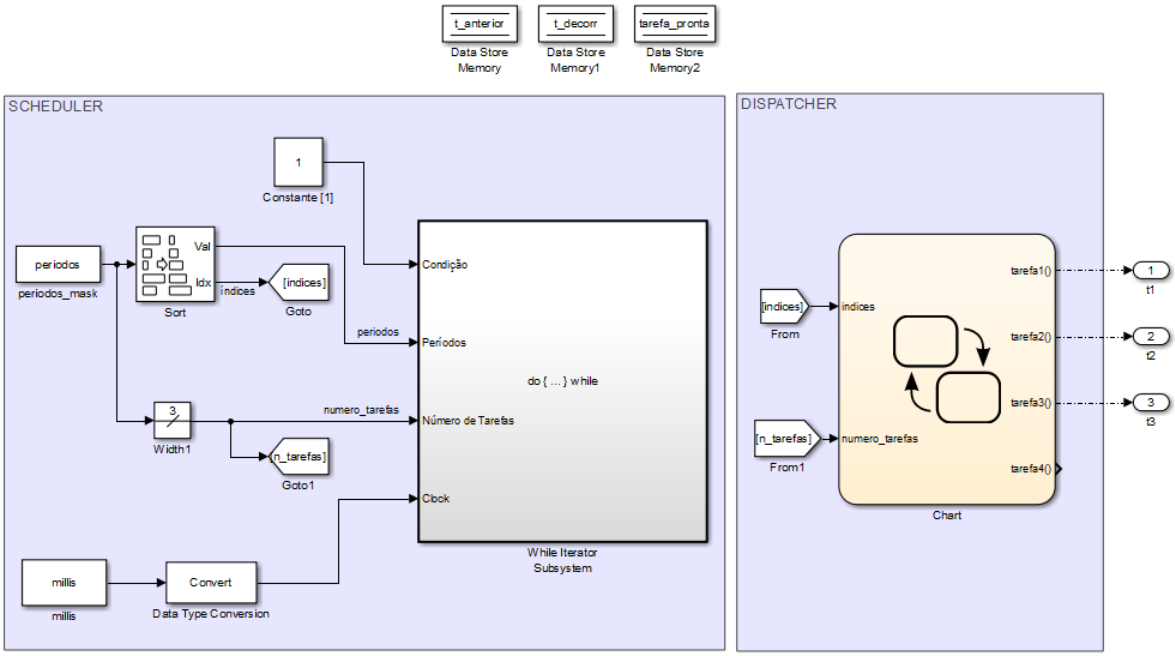

Figura 5 – Estrutura dentro do bloco RTOS
Fonte: autor

4.1.1 Implementação de tarefas

No código em C++, as tarefas são escritas em um arquivo separado, por exemplo, *Tasks.cpp*. Neste arquivo, cada tarefa é escrita separadamente dentro de uma função, enquanto que as informações relativas ao seu funcionamento serão indicadas através do método construtor da classe *TaskClass.* Neste construtor são indicados os nomes e os períodos de cada tarefa implementada. Além disso, o construtor possui outras duas variáveis que serão utilizadas internamente pelo sistema, uma indicando o tempo decorrido desde a última chamada desta função, e outra indicando se a tarefa correspondente está pronta ou não para ser executada.

A implementação das tarefas no Simulink é realizada independentemente do sistema operacional, utilizando necessariamente subsistemas do tipo *Function-call Subsystems,* que executam o diagrama de blocos construído em seu interior uma única vez, de acordo com o evento de chamada recebido através da entrada na parte superior do bloco.

Além da construção das tarefas, é necessário informar ao sistema operacional os períodos em que cada uma das tarefas deve ser chamada. Para isto foi criada uma máscara no bloco RTOS, conforme mostrado na Figura 6, na qual pode-se inserir os períodos desejados para as tarefas. Observa-se que para o correto funcionamento do sistema, os períodos devem ser inseridos com valores em milissegundos, em forma de vetor (entre colchetes e separados por vírgula) e na ordem em que aparecem as saídas do

bloco RTOS, ou seja, o primeiro período informado será para a saída t1, o segundo período para a saída t2 e o terceiro para a saída t3, conforme visualizado na Figura 6.

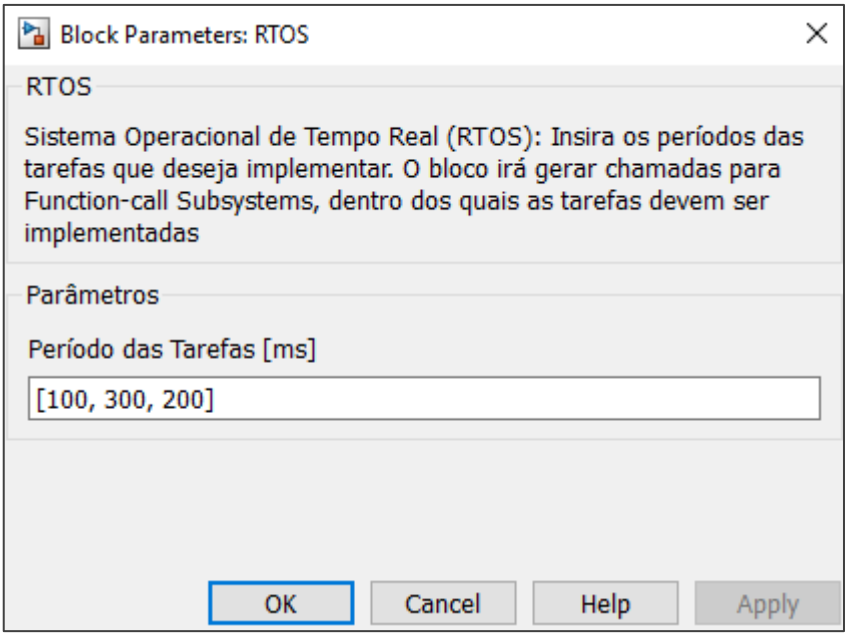

Figura 6 – Máscara do bloco RTOS
Fonte: autor

4.1.2 Escalonador (Scheduler)

O escalonador de processos é um subsistema do gerenciador de tarefas responsável por monitorar o tempo de execução do programa e decidir quando cada tarefa deverá ser chamada. Para isso, o escalonador deve ter uma base de tempo e estabelecer prioridades para as tarefas implementadas pelo usuário.

O sistema operacional implementado no código em C++ tem como característica a utilização de prioridades fixas, uma vez que utiliza o algoritmo de escalonamento conhecido como *Rate-Monotonic.* Através desta lógica, as prioridades são distribuídas de acordo com a duração do ciclo de cada tarefa, sendo tarefas com menor período atribuídas com prioridade mais alta.

A classe *SchedulerClass* possui métodos que realizam a implementação do algoritmo *Rate-Monotonic* e verificam a possibilidade de ativar as tarefas com o decorrer do tempo, respeitando a ordem de prioridade. Após a declaração do objeto desta classe, o método *InsertTask* é chamado para realizar a introdução de novas tarefas no escalonador. Este método é chamado na função *setup()* no código principal, uma vez para cada tarefa implementada, de maneira a ser executado somente na inicialização do sistema embarcado. O método recebe como parâmetro um ponteiro para o objeto anteriormente declarado para cada tarefa, através da classe *TaskClass*. Assim, o ponteiro é inserido em um vetor, caso o número máximo de tarefas não tenha sido ultrapassado, indicando os objetos de todas as tarefas inseridas.

A implantação desta classe no Simulink se dá através da utilização da máscara no bloco RTOS. Os períodos que são inseridos nesta janela são carregados pelo bloco *periodos_mask*, como detalhado na Figura 7. Desta forma, o vetor indicando as tarefas é

automaticamente criado pela inserção dos valores, sendo que a ordem de inserção corresponde a ordem em que estão dispostas as saídas no bloco RTOS.

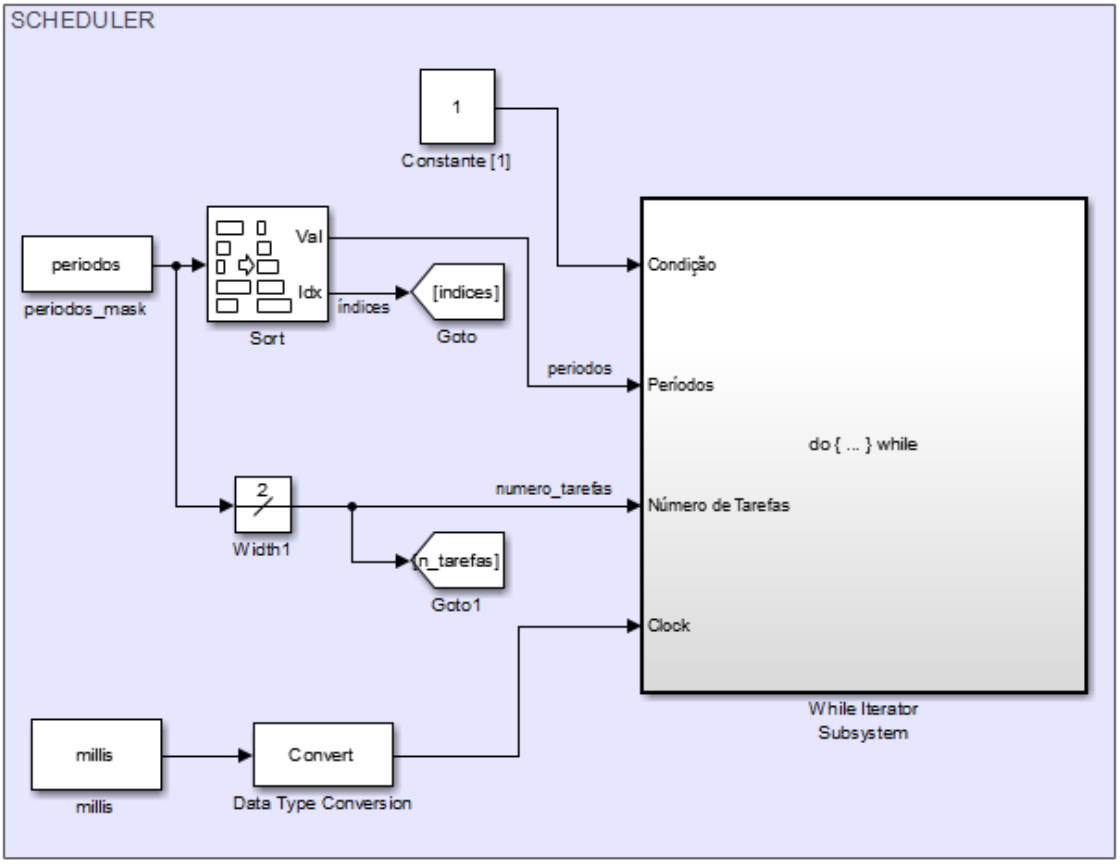

Figura 7 – Detalhamento da implementação em blocos do escalonador
Fonte: autor

Após a inserção das tarefas no código em C++, é chamado o método *Sort* do escalonador, que organiza as tarefas dentro do vetor em uma ordem crescente de período, o que equivale a ordem decrescente de prioridades, ou seja, a tarefa com menor período recebe a prioridade mais alta. No Simulink, este método é executado pelo bloco igualmente chamado *Sort*, que recebe o vetor de períodos inseridos através da máscara e os ordena de forma crescente. Além disso, este bloco produz uma saída chamada *Idx*, que representa a reorganização dos índices do vetor original após o ordenamento. Este vetor será utilizado posteriormente pelo expedidor para encontrar a porta de saída correta pela qual o sinal de chamada do tipo *Function-Call* deve ser enviado para ativar a tarefa correspondente.

O passo final na função *setup()* do código C++ é a chamada do método *Start* do escalonador, cuja função é salvar o tempo inicial de execução. No Simulink isto é realizado de maneira simplificada, colocando um estado inicial na variável tempo_anterior através de um bloco do tipo *Data Store Memory,* indicando um tempo igual a zero.

Iniciando o laço do micro controlador no código em C++, o método *Run* do escalonador é chamado em cada iteração. Este método verifica o tempo atual de execução através da função *millis()* do Arduino e compara com o tempo desde a última

execução do método, gerando assim uma diferença de tempo. O tempo atual é salvo na variável que salva o tempo anterior para que possa ser utilizado na próxima iteração. A diferença de tempo calculada é adicionada ao tempo decorrido desde a última chamada de cada função. Em seguida, o tempo decorrido de cada tarefa é comparado com o período que ela deve ser chamada, e caso seja maior, a tarefa é declarada como pronta para a execução e o tempo decorrido é zerado. Desta forma, a finalidade do método *Run* é verificar em cada instante de tempo quais tarefas devem ser executadas e declará-las prontas para execução.

Na programação no Simulink, o laço principal do programa do microcontrolador é construído com a utilização de um bloco *While Iterator Subsystem,* que implementa uma função de laço do tipo *do-while,* sendo a condição de parada deste laço declarada com valor 1 para que seja executada indefinidamente. Os parâmetros que são passados para dentro deste bloco são o vetor de períodos em ordem crescente, o número de tarefas desejadas, dado pelo tamanho do vetor de períodos e a base de tempo do sistema, gerada através do bloco *millis*, que possui a mesma funcionalidade da função utilizada no código em C++ e pode ser encontrada na biblioteca *Custom C/C++ Target – Arduino*. O bloco de laço e os parâmetros utilizados podem ser vistos na Figura 7.

A Figura 8 mostra a organização dentro do bloco de laço, no qual foram inseridos um subsistema e um bloco do tipo *for*.

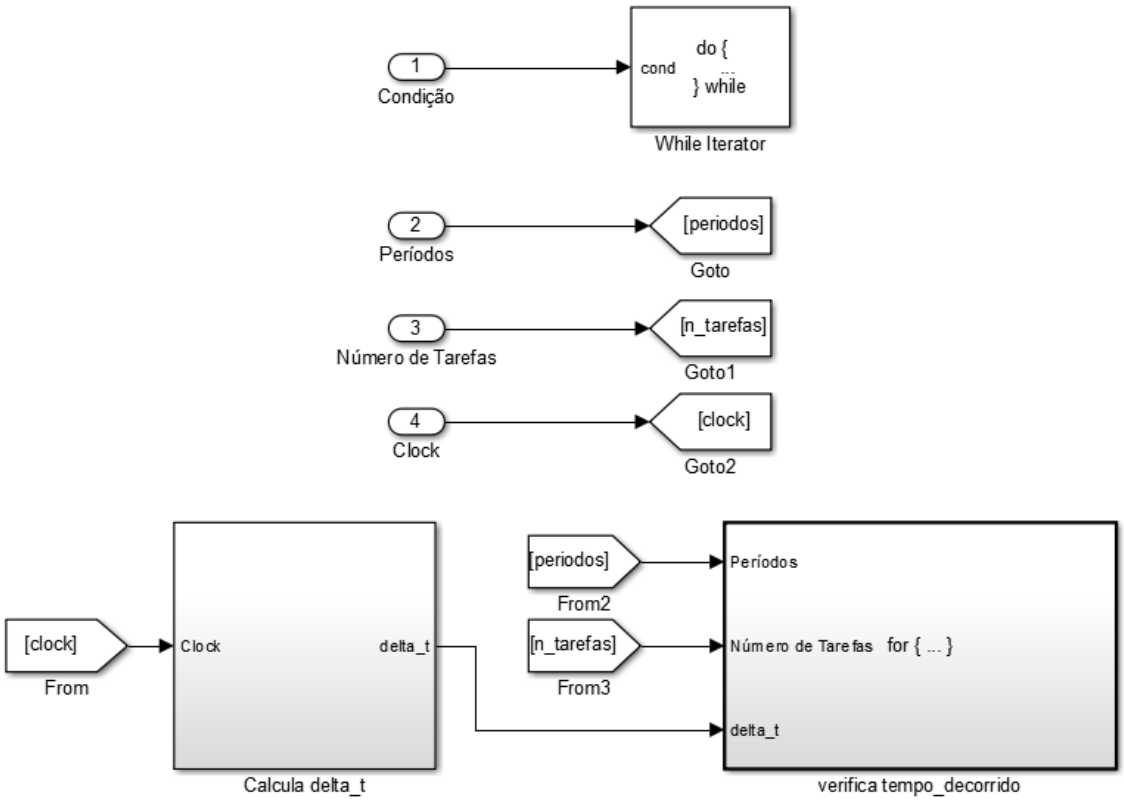

Figura 8 – Subsistemas contidos no bloco de laço *Do-While*
Fonte: autor

O primeiro subsistema, chamado *Calcula delta_t*, realiza a verificação do tempo atual e compara com o tempo anterior, armazenado em um bloco *Data Store Memory*. O teste

é realizado através de um bloco *If*, e caso o tempo atual seja maior ou igual ao tempo anterior, a variável *delta_t* é calculada como a diferença entre os dois tempos, sendo também adicionada à variável *tempo_anterior* para atualizar o valor para a próxima iteração. Esta atualização é realizada dentro de um bloco do tipo *MATLAB Function*, no qual é escrito um código na linguagem do Matlab que pode ser interpretado pelo Simulink. A Figura 9 mostra a construção do subsistema descrito.

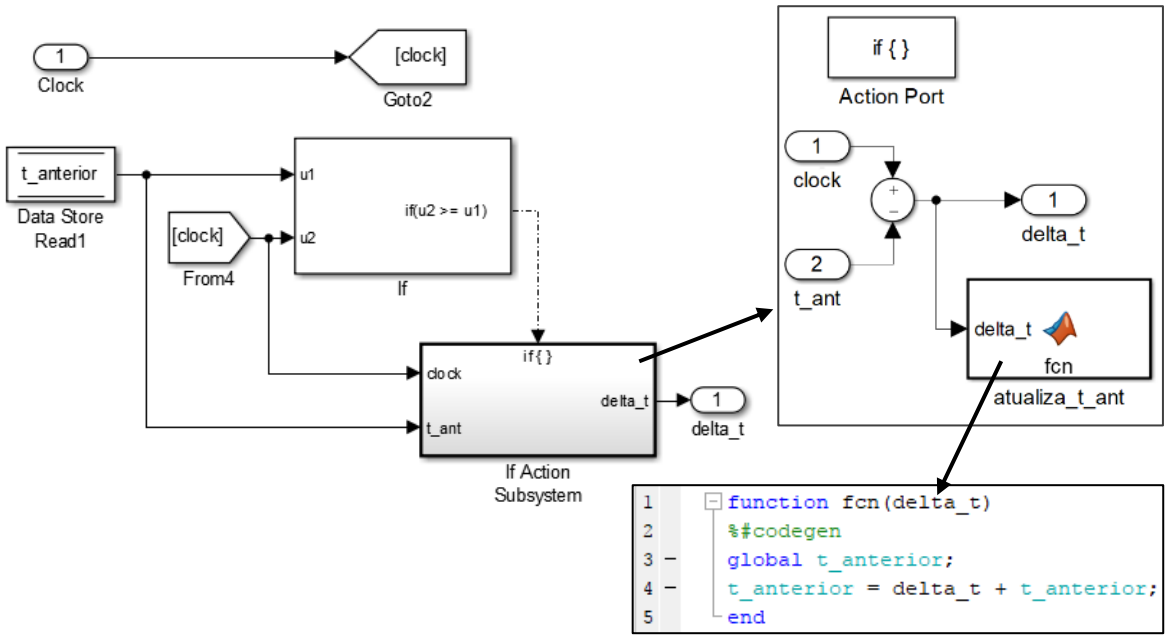

Figura 9 – Diagrama de blocos do subsistema *Calcula delta_t*
Fonte: autor

O segundo bloco, chamado verifica *tempo_decorrido*, implementa um laço do tipo *for*, que é executado uma vez para cada tarefa, realiza a atualização dos tempos decorridos desde a última execução de cada tarefa, comparando posteriormente com o período indicado. A Figura 10 mostra a organização interna deste bloco.

Inicialmente, o valor da diferença de tempo calculado no subsistema anterior é adicionado ao elemento da variável *t_decorr* correspondente à tarefa que está sendo avaliada, que é armazenada em forma de vetor em um bloco *Data Store Memory.* A ordem dos elementos deste vetor corresponde ao ordenamento realizado pelo bloco *Sort*, sendo o primeiro elemento relativo à tarefa com maior prioridade. Assim que o tempo decorrido desde a última execução é atualizado, é comparado com o período da tarefa correspondente em um bloco *If*. Caso o tempo decorrido seja maior ou igual ao período, a primeira condição é satisfeita, gerando a chamada para o bloco *If ok*, que irá zerar o valor do tempo decorrido desde a última execução e indicar que a tarefa correspondente está pronta para execução através da variável *tarefa_pronta*, também salva em forma de vetor em uma memória. Caso o tempo decorrido ainda não tenha atingido o tempo indicado no período, o bloco *else* será ativado, no qual o elemento da variável *t_decorr* será atualizado com o novo tempo calculado. A Figura 11 mostra a implementação realizada para os blocos *If ok* e *else*.

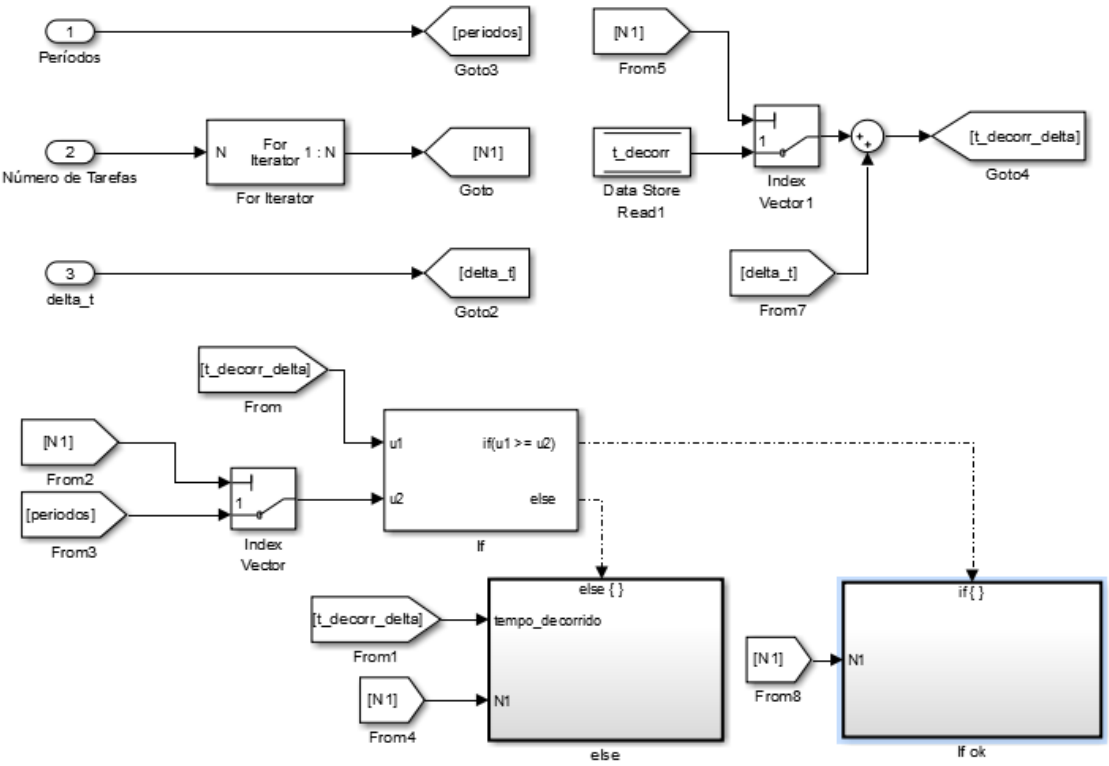

Figura 10 – Diagrama de blocos do laço *For - verifica tempo_decorrido*
Fonte: autor

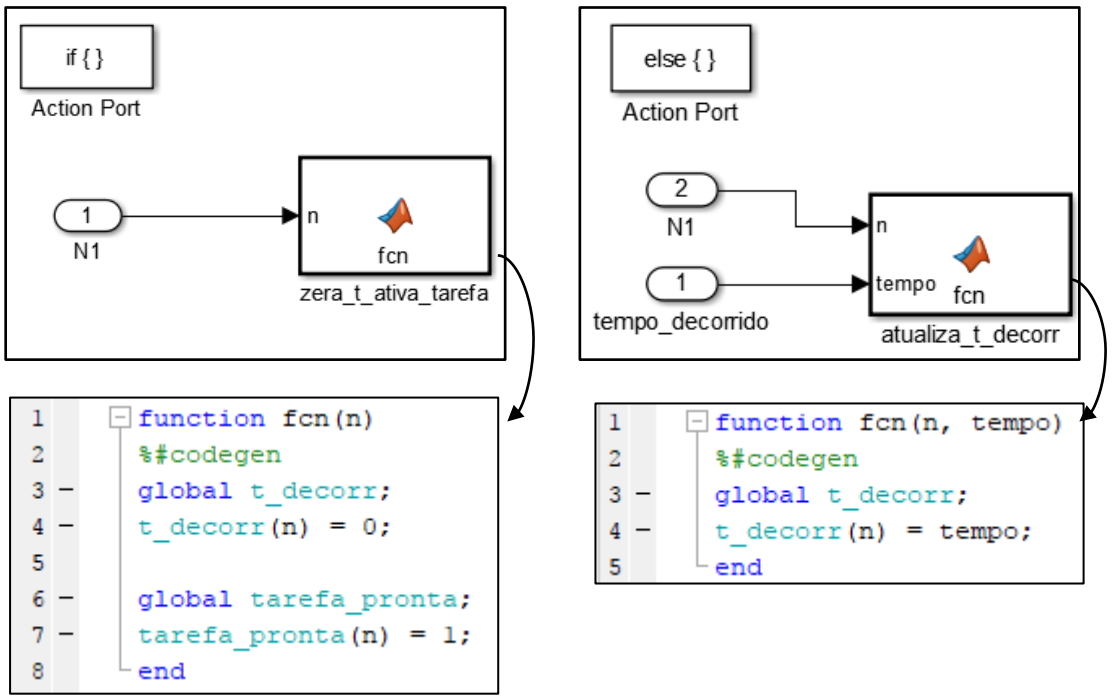

Figura 11 – Diagrama de blocos das condições if e else do teste contido no bloco *verifica tempo_decorrido*
Fonte: autor

4.1.3 Expedidor (Dispatcher)

O expedidor de processos é uma estrutura do sistema operacional que tem como função executar as tarefas que foram declaradas prontas para execução através do escalonador. No código em C++, o expedidor possui uma implementação, com apenas um método, além do seu construtor. O método *RunTask* é chamado no *loop* principal do programa, tendo como parâmetro um ponteiro indicando a próxima tarefa a ser executada. A tarefa é selecionada através do método *GetReadyTask* da classe *Scheduler.* Este método realiza uma varredura no vetor de tarefas inseridas, buscando por uma tarefa que esteja pronta para execução. Como este vetor já foi reorganizado por ordem de prioridade, a primeira tarefa detectada como pronta pelo método será indicada para a execução, visto que possui a maior prioridade.

A implementação do expedidor no Simulink foi realizada com a utilização de uma estrutura do tipo *Stateflow Chart*, que possibilita a construção de máquinas de estado no ambiente do Simulink. O principal motivo para a escolha desta estrutura foi a possibilidade de gerar eventos do tipo *Function-call*, que são necessários para a ativação das tarefas implementadas dentro de *Function-call Subsystems*. Desta forma, foi criada uma máquina de estados com dois estados, como pode ser visualizado na Figura 12. Um estado verifica e identifica as tarefas prontas e outro estado realiza a geração do evento que ativa a execução da tarefa correspondente.

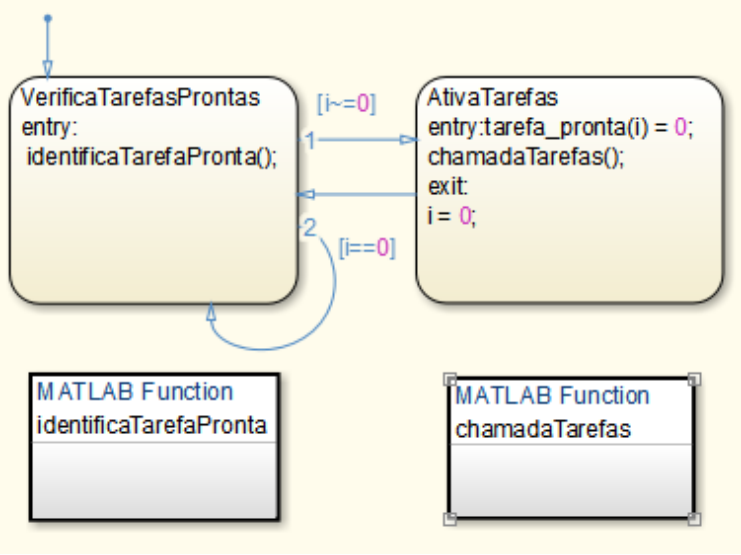

Figura 12 – *Stateflow Chart* que implementa a função do expedidor
Fonte: autor

Dentro de cada estado foi implementada uma função do tipo *MATLAB Function*, que permite adicionar um trecho de código a ser executado dentro do estado. A Figura 13 mostra os códigos adicionados nos dois estados.

```
function identificaTarefaPronta

if(sum(tarefa_pronta)>0)
    for(j=1:numero_tarefas)
        if(tarefa_pronta(j)==1)
            i = double(j);
            break;
        else
            i = 0;
        end
    end
end
end
```

```
function chamadaTarefas

switch indices(i)
    case 1
        tarefa_pronta(i) = 0;
        send(tarefa1);
    case 2
        tarefa_pronta(i) = 0;
        send(tarefa2);
    case 3
        tarefa_pronta(i) = 0;
        send(tarefa3);
    case 4
        tarefa_pronta(i) = 0;
        send(tarefa4);
end
end
```

Figura 13 – Códigos executados nos estados do *Stateflow Chart*
Fonte: autor

A máquina de estados é iniciada no estado *VerificaTarefasProntas*, no qual é chamada a função *identificaTarefaPronta*() logo que o estado é iniciado. Nesta função é verificada a existência de tarefas prontas para execução, checando a soma de todos os elementos do vetor *tarefa_pronta*. Neste vetor, tarefas que já atingiram o tempo para chamada recebem o valor 1 no elemento correspondente, enquanto as tarefas que ainda estão esperando recebem o valor zero. Assim, se a soma de todos os elementos for maior que zero, existe pelo menos uma tarefa pronta para a execução. Caso isto ocorra, é realizada uma varredura no vetor para identificar a primeira tarefa, na ordem de prioridade, que está pronta para execução. Assim que esta tarefa for encontrada, é atribuído à variável *i* o índice do vetor no qual esta tarefa está localizada. Caso nenhuma tarefa pronta seja detectada, o valor atribuído a *i* é zero.

Os eventos de saída do estado inicial testam o valor da variável *i*. Caso este valor seja zero, o evento retorna ao início da execução do mesmo estado, fazendo com que o programa execute novamente a lógica descrita anteriormente que identifica as tarefas prontas. Caso o valor de *i* seja diferente de zero, a execução passa para o estado *AtivaTarefas*, identificando que há uma tarefa selecionada para ser chamada imediatamente.

O estado *AtivaTarefas* contém uma *MATLAB Function* com o nome *chamadaTarefas* que é executada assim que o estado é ativado. Nesta função é identificado o valor do índice *i*, que corresponde à tarefa que deve ser executada. Entretanto, este valor pode não corresponder à porta de saída pela qual o evento *function-call* deve ser enviado, uma vez que as tarefas foram reordenadas de acordo com seu período. Para identificar a porta de saída correta, a função consulta o vetor índices, que é uma saída direta do bloco *Sort*, (Figura X) utilizado no ordenamento das tarefas. Este vetor possui a informação das posições originais dos períodos inseridos pelo usuário através da máscara do bloco RTOS. Assim, o valor de *índices(i)* indica a porta pela qual o evento de ativação de tarefas deve ser enviado para realizar a chamada da tarefa certa.

A geração dos eventos *function-call* é feita dentro do estado *AtivaTarefas* do *Stateflow Chart.* Para gerar os sinais de chamada, é necessário declarar cada evento como

uma saída do tipo *function-call*, indicando o nome e o número da porta de saída. Assim, a geração do evento é realizada através da função *send(),* como pode ser visto na Figura 13, as chamadas para as tarefas dentro da estrutura *Switch-Case*. Além das chamadas, esta função também reinicializa o elemento do vetor *tarefa_pronta* correspondente à tarefa que foi executada, para indicar que a chamada foi realizada e liberar as demais tarefas para a execução.

4.2 Geração de código embarcado

Após a implementação do sistema operacional e das tarefas que se deseja implementar, sendo este já validado por simulação, é possível realizar a geração de código na linguagem C/C++ e a transferência automática para o sistema embarcado utilizado. Para isto é utilizada a ferramenta *Simulink Coder.*

Para realizar a geração de código e a transferência para o microcontrolador, é necessário realizar o ajuste de alguns parâmetros, informando ao *software* a plataforma que está sendo utilizada. A Figura 14 mostra a janela de configuração dos parâmetros, que é encontrada no ambiente principal do Simulink, na aba *Code – C/C++ Code – Code Generation Options.*

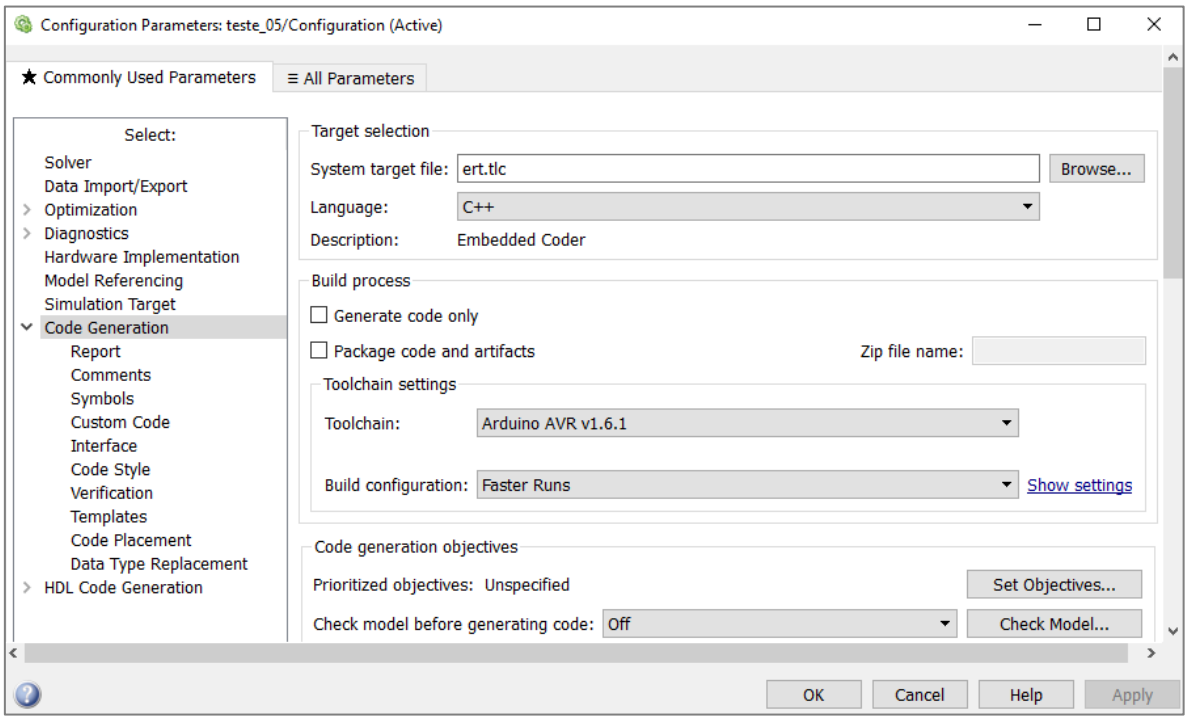

Figura 14 – Janela de configuração dos parâmetros do *Simulink Coder*
Fonte: Autor

Na janela mostrada na Figura 14 foram selecionadas as seguintes opções:

- aba *Code generation:*
    - selecionada a opção *ert.tcl* (*embedded coder*) no campo *System target file,* através do botão *Browse*. Esta opção indica ao *coder* que o código gerado será utilizado em um sistema embarcado de tempo real;

- no campo *Toolchain* foi selecionada a opção Arduino AVR v1.6.1.

- aba *Hardware Implementation*:

  - selecinonada a opção Arduino Uno no campo *Hardware board*, indicando a placa que receberá o código gerado. Selecionando esta opção, as configurações da placa são ajustadas automaticamente.

Após a configuração descrita acima, o código é gerado acionando o botão *Deploy to Hardware*, encontrado na tela principal de programação do Simulink. Após a geração de código e transferência para o microcontrolador, são gerados algumas informações e relatórios que permitem a visualização dos arquivos e códigos gerados, podendo também mapear cada bloco Simulink para o trecho do código C/C++ em que foi traduzido.

5 Resultados

Neste capítulo serão apresentados e analisados os testes realizados para validar o sistema construído e verificar seu correto funcionamento. Para isso, é necessário verificar se as chamadas de tarefas estão acontecendo de maneira correta. Da maneira que o sistema foi criado, espera-se que funcione em situações diferentes: na simulação do Simulink e quando transferido para um microcontrolador através do *Simulink Coder*, desde que o microcontrolador seja suportado pelo *Embedded C*oder e pela biblioteca *Custom C/C++ Target – Arduino* (este último, caso seja utilizada a plataforma Arduino).

Para uma verificação inicial do funcionamento na simulação, foi proposto um conjunto de tarefas simples, cuja única função é alterar o estado de uma saída, alternando entre os níveis lógicos alto e baixo, correspondente aos valores 1 e 0 na simulação. A Figura 15 mostra a implementação de blocos desta tarefa.

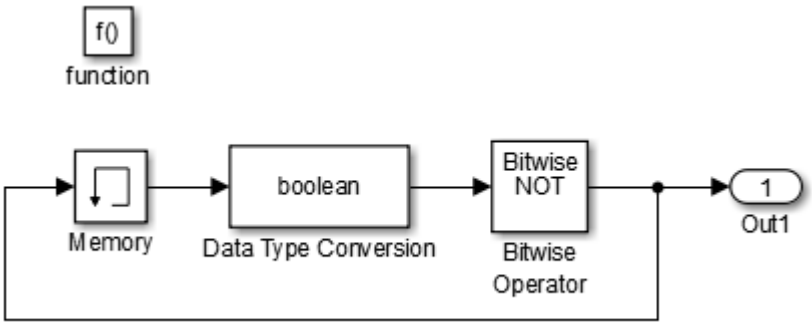

Figura 15 – Tarefa de mudança do estado da saída
Fonte - autor

O conjunto de blocos mostrado na Figura 15 foi implementado dentro de três blocos do tipo *function-call subsystem*, de forma a caracterizar três tarefas diferentes a serem chamadas pelo bloco RTOS, como ilustrado pela Figura 4. A única diferença entre as três tarefas foi o período de execução, estipulado em 100, 300 e 200 milissegundos para as tarefas 1, 2 e 3, respectivamente, declarado através da máscara do bloco RTOS.

As saídas das tarefas foram ligadas a um bloco *Scope*, para visualizar as mudanças de estado. A Figura 16 mostra os gráficos gerados pelas saídas. Pode-se perceber que as mudanças de estado de cada uma das saídas ocorrem aproximadamente em tempos múltiplos dos períodos inseridos. Assim, comprova-se que o sistema operacional realiza a chamada das tarefas no tempo desejado.

Além disso, é possível avaliar se o sistema está respeitando a ordem de prioridade, que deve ser maior para tarefas com menor período. Para o conjunto de tarefas implementado, a ordem decrescente de prioridade é: Tarefa 1, Tarefa 3 e Tarefa 2. Caso existam duas ou mais tarefas que sejam ativadas no mesmo instante de tempo, o sistema sempre escolherá aquela com a maior prioridade.

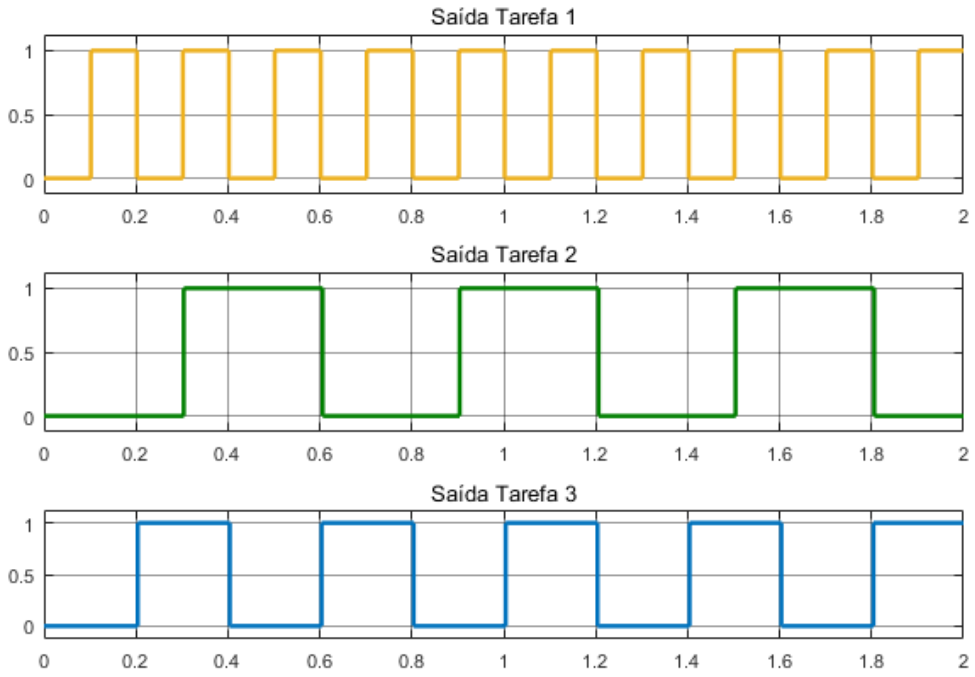

Figura 16 – Gráfico das saídas das tarefas gerado na simulação
Fonte: autor

A Figura 17 mostra em detalhe o comportamento das saídas quando o tempo é igual a 1800 milissegundos. Neste instante, as três tarefas são declaradas prontas para a execução, porém, apenas uma pode ser executada. Como a Tarefa 1 é prioritária diante das outras em função de seu menor período, é executada por primeiro. Em seguida, a Tarefa 3 é executada por possuir período e prioridade intermediários. Por fim, a Tarefa 2 é executada por possuir o período mais longo. Desta forma, verifica-se o correto seguimento da ordem de prioridade das tarefas, estipulada seguindo o algoritmo *Rate Monotonic*.

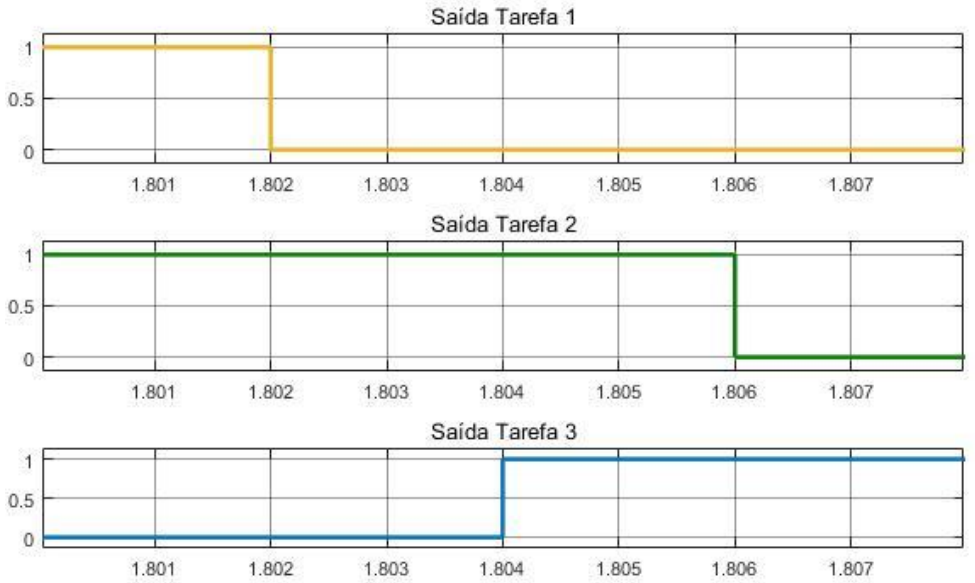

Figura 17 – Gráfico detalhado das saídas das tarefas na simulação no tempo a partir de 1800 milissegundos
Fonte: autor

Embora este teste seja válido para mostrar as chamadas de tarefas nos períodos estipulados, não é possível verificar o comportamento do sistema com atrasos, que podem acontecer devido ao tempo de processamento de cada tarefa. Uma vez iniciadas, as tarefas devem completar sua execução para liberar o uso do processador para o sistema operacional identificar e chamar uma próxima tarefa. Assim, um conjunto de tarefas com um custo computacional elevado pode não ser viável, pois o sistema não conseguirá atender todas as condições temporais demandadas.

Para verificar o comportamento do sistema operacional com tarefas que exijam maior tempo de processamento, foi construído o conjunto de blocos mostrado na Figura 18.

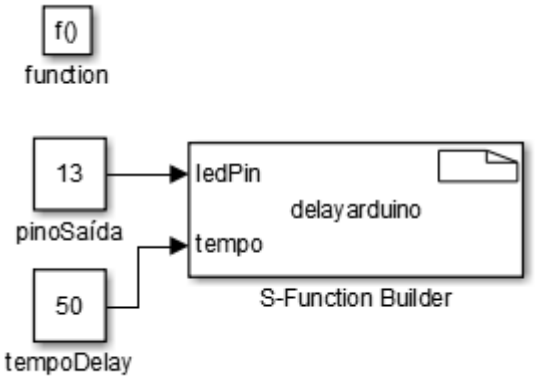

Figura 18 – Tarefa considerando um tempo computacional com a função *delay*
Fonte: autor

Nesta tarefa foi implementado um bloco do tipo *S-function*, que possibilita a inserção de código C/C++ no ambiente do Simulink. Neste bloco foi escrita uma função que executa um atraso através da função *delay()* do Arduino. Uma saída digital do Arduino foi colocada em nível alto antes e em nível baixo após o *delay*, para que se possa visualizar o tempo que o processador gastou para executar esta tarefa. Como a troca de nível lógico da saída demanda um tempo de processamento bastante baixo, o tempo total da tarefa pode ser considerado como o tempo inserido na função *delay.* Como a função *delay* não é compreendida como um comando válido no Matlab, a tarefa mostrada na Figura 18 não funciona de forma correta na simulação do Simulink. Entretanto, ela é compreendida pelo *Simulink Coder* e pelo Arduino, quando transferida para o microcontrolador. Assim, o funcionamento do sistema pode ser visualizado adquirindo os sinais das saídas digitais do Arduino através de um osciloscópio.

Utilizando a lógica detalhada no trabalho, é possível criar um conjunto de tarefas com períodos e tempos de processamento definidos. Desta forma, pode-se prever o comportamento do sistema operacional em conjunto com as tarefas da aplicação e analisar se o conjunto proposto é escalonável ou não. Para realizar esta verificação, foram criados dois conjuntos de tarefas, conforme pode ser visto nas tabelas 1 e 2.

O primeiro conjunto, mostrado na Tabela 1, é considerado escalonável, pois todas as tarefas podem ser executadas dentro de seu período, considerando que podem ocorrer

bloqueios em tarefas com maior prioridade em função do tempo de processamento das outras tarefas.

Tabela 1 – Conjunto de tarefas escalonável
Fonte: autor

Tarefa	Período (ms)	Tempo de processamento (ms)	Prioridade
T1	100	50	Alta
T2	300	70	Baixa
T3	200	30	Média

A Figura 19 mostra a previsão de comportamento do sistema para o conjunto de tarefas mostrado na Tabela 1, sendo que cada divisão representa um tempo de 10 milissegundos. Cada linha representa uma tarefa, sendo as barras verticais representantes do momento em que cada tarefa deve ser chamada. O período de chamada de uma tarefa também representa o tempo máximo que ela deve terminar sua execução, ou seja, o início de uma tarefa pode ser atrasado em função de um bloqueio, mas o término deve ocorrer antes que uma segunda chamada seja gerada. A Figura 19 mostra que todas as tarefas foram completadas em seu período, mostrando que o conjunto é escalonável. Na Tarefa 1, por exemplo, percebe-se que o tempo entre a primeira e a segunda chamada da Tarefa é de 150ms, porém ela é finalizada antes de completar o prazo máximo para sua execução, que é igual ao seu período.

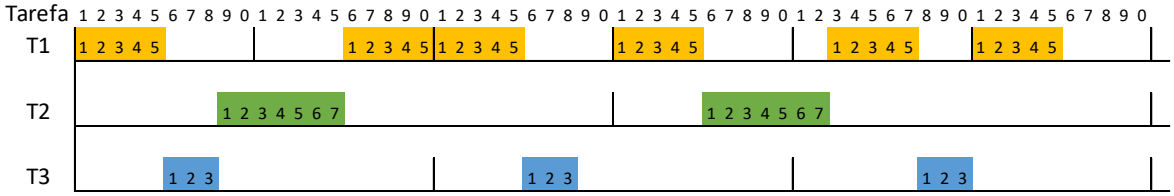

Figura 19 – Estimativa de comportamento do conjunto escalonável de tarefas
Fonte: autor

Desta forma, o conjunto de tarefas dado na Tabela 1 foi implementado no Simulink, inserindo os períodos através da máscara do bloco *RTOS* e o tempo de processamento no bloco *tempoDelay* no interior de cada tarefa. Em seguida, foi iniciada a geração de código através do *Simulink Coder*, e a transferência para o microcontrolador, realizada de forma automática. Os sinais gerados nos pinos de saída do Arduino foram adquiridos através de um osciloscópio.

A Figura 20 representa fotografias da tela do osciloscópio, mostrando as formas de onda geradas pelas saídas do Arduino. Foram necessárias duas imagens para representar o comportamento das três tarefas, uma vez que o osciloscópio utilizado possuía apenas dois canais de entrada. Nas imagens está representado o ponto de início (onde todas as tarefas estão prontas para serem executadas), para que se possa comparar com a estimativa mostrada na Figura 19. Percebe-se que o comportamento dos sinais segue a estimativa, e as medidas dos tempos dos períodos e intervalos mostrou-se coerente com os parâmetros utilizados.

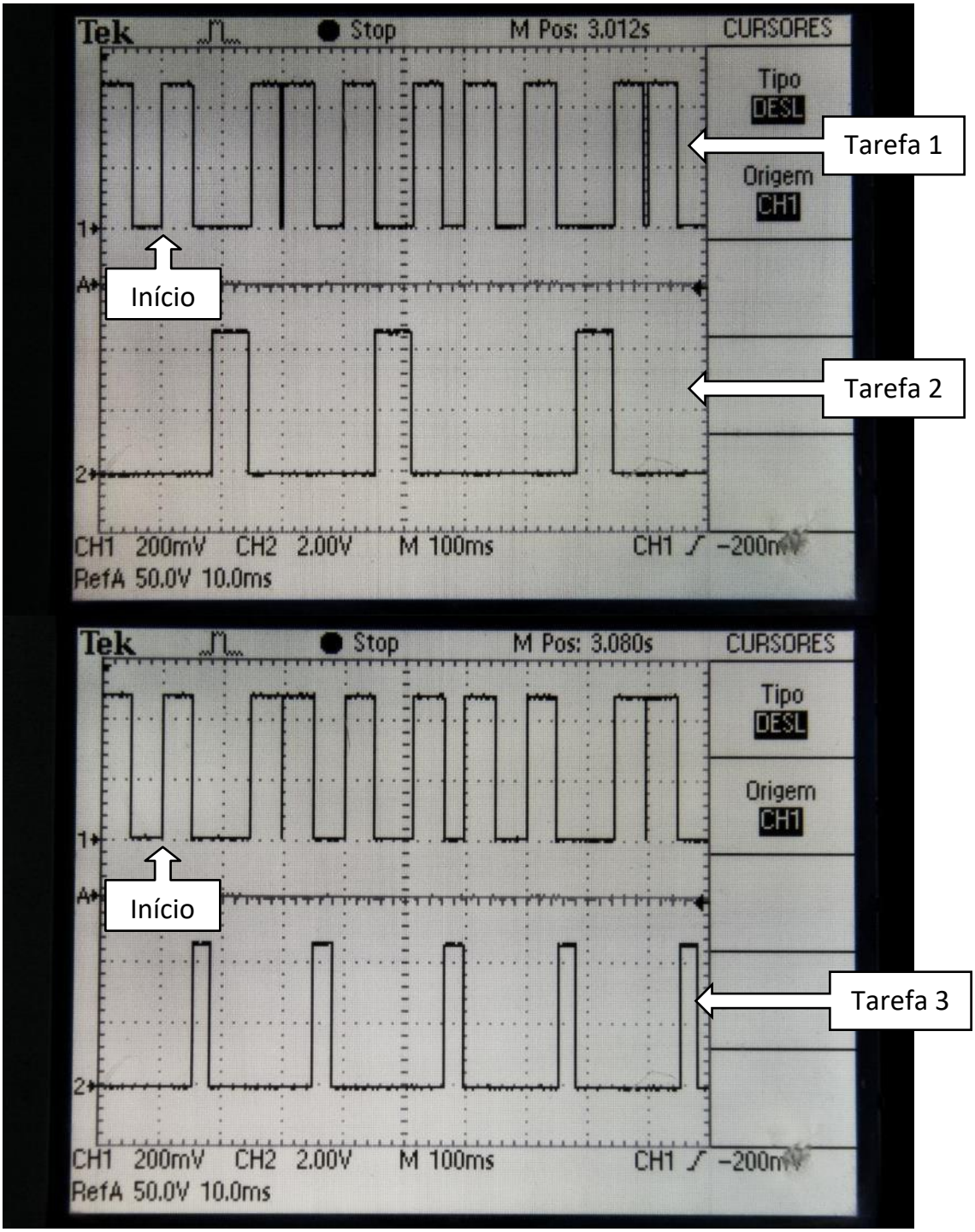

Figura 20 – Fotografias dos sinais medidos nas saídas do Arduino para o conjunto de tarefas escalonável
Fonte: autor

O segundo conjunto, mostrado na Tabela 2, é considerado não-escalonável, pois em determinados períodos, os requisitos temporais de algumas tarefas não são cumpridos. Comparando a Tabela 2 com a Tabela 1, nota-se que a única diferença está no tempo de processamento da Tarefa 3, que passou de 30 para 80 milissegundos.

Tabela 2 – Conjunto de tarefas não-escalonável
Fonte: autor

Tarefa	Período (ms)	Tempo de processamento (ms)	Prioridade
T1	100	50	Alta
T2	300	70	Baixa
T3	200	80	Média

A Figura 21 mostra a previsão de comportamento do sistema para o conjunto de tarefas mostrado na Tabela 2. Embora o período de chamada e as prioridades não tenham sido alteradas, a diferença no tempo de processamento da Tarefa 3 acaba gerando atrasos na execução. Pode-se notar que a Tarefa 2, mesmo não sofrendo alterações, não foi executada no segundo período do seu ciclo. A Tarefa 3 tem seu término atrasado no segundo período, o mesmo acontecendo com a Tarefa 1. Desta forma, classifica-se este conjunto como não escalonável através do algoritmo implementado por não ter seus requisitos temporais cumpridos.

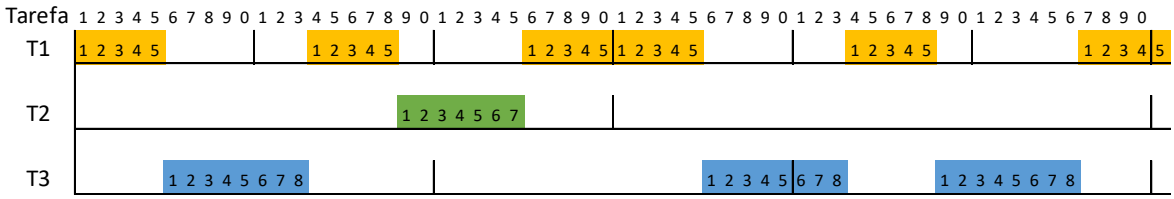

Figura 21 – Estimativa de comportamento do conjunto não-escalonável de tarefas
Fonte: autor

Analogamente ao caso anterior, o conjunto de tarefas dado na Tabela 2 foi implementado no Simulink e, posteriormente, transferido ao Arduino. A Figura 22 mostra os sinais adquiridos através do osciloscópio. Novamente percebe-se que o sistema implementado no microcontrolador respondeu da forma estimada, mostrando que o sistema operacional construído opera de forma previsível, e que, dependendo do período e do tempo de execução das tarefas, os requisitos temporais desejados não são atendidos, devido ao algoritmo utilizado.

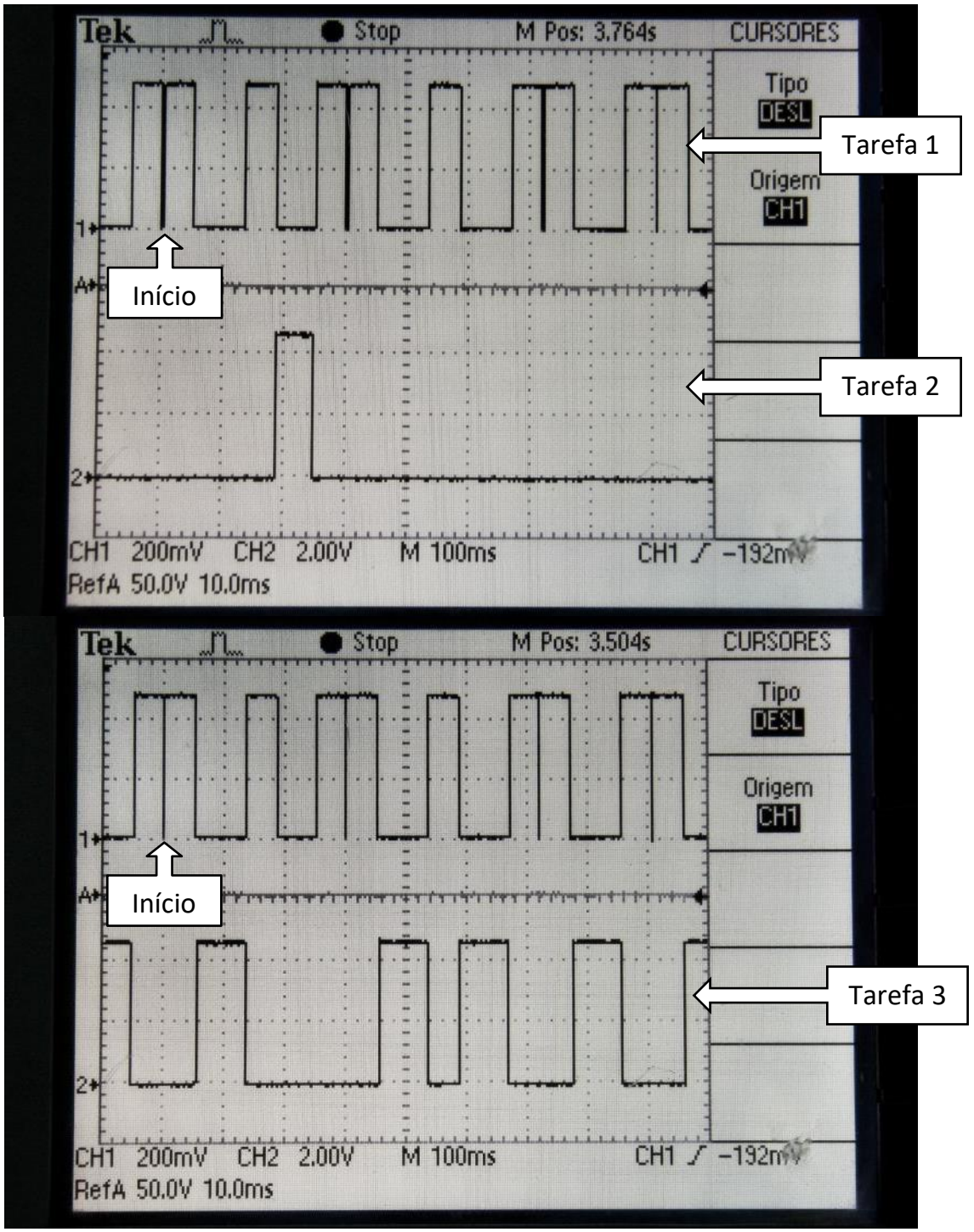

Figura 22 – Fotos dos sinais medidos nas saídas do Arduino para o conjunto de tarefas não-escalonável
Fonte: autor

Embora não tenha sido precisamente quantificado, o tempo necessário para a troca de contexto do sistema, ou seja, o período entre a execução de duas tarefas diferentes em sequência, é em torno de 2 milissegundos. Desta forma, dependendo do tempo de execução das tarefas, o período mínimo que pode ser utilizado fica em torno de 5 milissegundos.

6 Conclusões e Trabalhos Futuros

A partir da realização dos testes para validação do sistema e verificação dos resultados obtidos, conclui-se que o sistema operacional construído no ambiente de blocos do Simulink opera de maneira satisfatória, atendendo os requisitos mínimos especificados para seu funcionamento em tempo real, tanto na simulação quanto na implementação em *hardware.*

A utilização do sistema operacional construído apresenta alguns benefícios aos usuários. Com relação à não utilização de um sistema operacional, o uso desta estrutura fornece maior previsibilidade e garantia de que os requisitos temporais desejados na execução das tarefas sejam cumpridos. Além disto, permite facilitar a programação, não sendo necessário implementar estruturas de monitoramento de tempo para execução de tarefas periódicas. Outro ganho é a possibilidade de geração de código para outras plataformas, permitindo assim a sua portabilidade.

Comparando com o sistema operacional no qual este trabalho foi baseado, a utilização do Simulink frente à IDE do Arduino também fornece vantagens ao programador, visto que a programação em blocos disponibilizada é bastante intuitiva e simplificada, se comparada com a linguagem C++ utilizada no código original. Ou seja, o usuário pode desenvolver as tarefas da aplicação dentro do mesmo ambiente Simulink. Além disso, o novo sistema possibilita uma utilização mais versátil, podendo ser aplicado inclusive na própria simulação, fornecendo uma previsão de seu funcionamento no próprio Simulink antes da implantação em um sistema embarcado. Estes fatores acarretam em uma experiência de programação mais simples e rápida.

A programação através do Simulink mostrou-se bastante versátil no decorrer do desenvolvimento deste trabalho, não só pela diversidade de blocos disponíveis em suas bibliotecas, mas também pelas diferentes ferramentas e estruturas que o *software* oferece. Além dos blocos convencionais, foi possível utilizar estruturas do tipo *stateflow chart*, que possibilita a criação de máquinas de estado, e blocos que realizam a importação de códigos em linguagens C/C++ e Matlab, fornecendo a correta integração com o restante do diagrama de blocos. Além disso, a maior parte dos blocos e estruturas disponíveis são suportadas pela ferramenta *Simulink Coder*, que possibilita a geração e transferência de código embarcado diretamente para um microcontrolador.

Apesar de apresentar diversas vantagens na sua utilização, o sistema operacional implementado através do Simulink apresenta alguns pontos que podem não ser desejáveis para algumas aplicações. O código gerado pode ocupar mais espaço na memória do que ocuparia se fosse escrito diretamente na linguagem C/C++. Porém, isto é compensado quando se considera o ganho que o usuário tem em se preocupar apenas com o problema em um nível mais alto de abstração, e não no nível de linhas de código. Outro ponto que limita as aplicações do sistema é a impossibilidade do gerenciamento de tarefas aperiódicas, como no caso de interrupções do processador.

Alguns pontos observados como possíveis melhorias do sistema serão indicados a seguir para possíveis estudos futuros:

- Tornar o bloco RTOS totalmente independente da plataforma alvo. Atualmente, por exemplo, a função "*millis",* específica da plataforma de *software* Arduino, é utilizada como base temporal;

- Implementar um sistema preemptivo, que permita interrupção da tarefa para a execução de outra com maior prioridade, habilitando assim sistema na presença de tarefas críticas (que necessitem de requisitos temporais mais rígidos);

- Implementar a possibilidade de gerenciamento de tarefas aperiódicas, como por exemplo, no atendimento de tarefas associadas a interrupções do processador, as quais não são periódicas.

7 Referências

ARDUINO. **Arduino Uno**. Disponível em: <https://store.arduino.cc/usa/arduino-uno-rev3>. Acesso em: 21 set. 2017b.

ARDUINO. **What is Arduino?**. Disponível em: <https://www.arduino.cc/en/guide/introduction>. Acesso em: 01 out. 2017a.

BARR GROUP. **Embedded Systems Glossary**. Disponível em: <https://barrgroup.com/embedded-systems/glossary>. Acesso em: 25 set. 2017

BARR, Michael; MASSA, Anthony. **Programming Embedded Systems With C and GNU Development Tools**. 2 ed. O'Reilly, 2009. 336 p.

BUTTAZZO, Giorgio C. **Hard Real-Time Computing Systems**: Predictable Scheduling Algorithms and Applications. 3 ed. Springer, 2011. 521 p.

HEATH, Steve. **Embedded Systems Design**. 2 ed. Oxford: Newnes, 2003. 430 p.

HERNYÁK, Zoltán. **High-Level Programming Languages II**: Object-Oriented Programming in practice, 2012. 296 p.

MATHWORKS. **Matlab & Simulink: Simulink User's Guide**. Edição. Natick, MA, 2017b. 4090 p.

MATHWORKS. **Simulink Coder**. Disponível em: <https://www.mathworks.com/products/simulink-coder.html>. Acesso em: 03 out. 2017a.

MATHWORKS. **Simulink Support Package for Arduino Hardware**. Disponível em: <https://www.mathworks.com/help/supportpkg/arduino/index.html>. Acesso em: 03 out. 2017d.

MATHWORKS. **Stateflow**. Disponível em: <https://www.mathworks.com/help/stateflow/>. Acesso em: 04 dez. 2017c.

MATHWORKS. **Simulink - Dynamic System Simulation for Matlab: Writing S-Functions**. 3 ed. Natick, MA, 1998. 210 p.

MIT AEROASTRO. **MIT and Navigating the Path to the Moon**. Disponível em: <http://web.mit.edu/aeroastro/news/magazine/aeroastro6/mit-apollo.html>. Acesso em: 01 out. 2017.

MOSTER, Joseph E. Integration of a Real-Time Operating System into Automatically Generated Embedded Control System Code. Raleigh, North Carolina, 2015.

OLIVEIRA, Rômulo S.; CARISSIMI, Alexandre S.; TOSCANI, Simão S. **Sistemas Operacionais-Vol. 11: Série Livros Didáticos Informática UFRGS**. Bookman Editora, 2009

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. Prentice-Hall, 2010.

TAURION, CEZAR. **Software Embarcado – A Nova Onda da Informática**. Brasport, 2005.

8 Anexo A – Código original em C++

Código em C++ desenvolvido pelo professor Marcelo Götz para a plataforma Arduino, no qual o sistema operacional foi baseado:

Sketch_sep23a.ino

```
#include "TaskClass.h"
#include "SchedulerClass.h"
#include "DispatcherClass.h"

#include "Tasks.h"

SchedulerClass Scheduler;

DispatcherClass Dispatcher;

TaskClass Task1(Task_1_code, 1000);
TaskClass Task2(Task_2_code, 1500);
TaskClass Task3(Task_3_code, 500);

void setup() {
   // put your setup code here, to run once:

   Scheduler.InsertTask (&Task1);
   Scheduler.InsertTask (&Task2);
   Scheduler.InsertTask (&Task3);

   Scheduler.Sort();

   Scheduler.Start();
}

void loop() {
   // put your main code here, to run repeatedly:
   TaskClass* f;

   Scheduler.Run();

   f = Scheduler.GetReadyTask();

   if (f != NULL)
   {
      // Exist a task to run
      Dispatcher.RunTask(f);
   }
}
```

SchedulerClass.cpp

```
#include "SchedulerClass.h"

const unsigned long m_last_ = 0;
#define M_LAST (m_last_ - 1)

static inline unsigned long timing() { return millis(); }
//static inline unsigned long timing() { return micros(); }

// Constructor
```

```
10 SchedulerClass::SchedulerClass(void)
11 {
12    nTasks = 0;
13 }
14
15 unsigned char SchedulerClass :: getMaxNTasks(void)
16 {
17    return nTasks;
18 }
19
20 int SchedulerClass :: InsertTask (TaskClass *ptrTask)
21 {
22    //int ret;
23
24    // check if there is room free for inclusion
25 if (nTasks >= _MAX_N_TASKS)
26 {
27    return -1;
28 }
29
30    // if room free, insert and return number of tasks
31    ptrTaskList[nTasks] = ptrTask;
32    //ret = nTasks;
33    nTasks++;
34
35    return nTasks;
36 }
37
38 void SchedulerClass :: Start (void)
39 {
40    time_prev = timing();
41 }
42
43 boolean SchedulerClass :: Run (void)
44 {
45    unsigned long time_now;
46    unsigned long delta_time;
47    boolean ret;
48
49    int i;
50
51    time_now = timing();
52    ret = false;
53
54    if (time_now == time_prev)
55    return ret;
56
57    if (time_now > time_prev)
58    {
59       delta_time = (time_now > time_prev);
60    }
61    else // (time_now < time_prev) // will enter here not so often,
just in overflow cases.
62    {
63       delta_time = ((M_LAST - time_prev) + time_now);
64    }
65
66    time_prev = time_now;
67
68    // update all elapsed time now and check if it is ready;
69    for (i = 0; i < nTasks; i++)
70    {
71       if (ptrTaskList[i]->period != 0)
72       {
```

```
    73          ptrTaskList[i]->elapsed_time += delta_time;
    74          if (ptrTaskList[i]->elapsed_time >= ptrTaskList[i]-
>period)
    75          {
    76             ptrTaskList[i]->elapsed_time = 0;
    77             ptrTaskList[i]->ready = true;
    78             ret = true;
    79          }
    80       }
    81    }
    82
    83    return ret;
    84 }
    85
    86 TaskClass* SchedulerClass :: GetReadyTask(void)
    87 {
    88    TaskClass* f = NULL;
    89    int i;
    90
    91    for (i = 0; i < nTasks; i++)
    92    {
    93       if (ptrTaskList[i]->ready)
    94       {
    95          f = ptrTaskList[i];
    96          break;
    97       }
    98    }
    99
    100   return f;
    101 }
    102
    103 // method to sort task list in decreasing priority.
    104 void SchedulerClass :: Sort(void)
    105 {
    106   int i, j;
    107   TaskClass* pTask;
    108
    109   for(i=0; i<(nTasks-1); i++)
    110   {
    111      for(j=0; j<(nTasks-(i+1)); j++)
    112      {
    113         if(ptrTaskList[j] > ptrTaskList[j+1])
    114         {
    115            pTask = ptrTaskList[j];
    116            ptrTaskList[j] = ptrTaskList[j+1];
    117            ptrTaskList[j+1] = pTask;
    118         }
    119      }
    120   }
    121 }
```

SchedulerClass.h

```
1 #ifndef SCHEDULERCLASS_H
2 #define SCHEDULERCLASS_H
3
4 #include <Arduino.h>
5
6 #include "TaskClass.h"
7
8 #ifndef _MAX_N_TASKS
```

```
 9 #define _MAX_N_TASKS 16 // Maximum number of tasks
10 #endif
11
12 class SchedulerClass
13 {
14     public:
15     SchedulerClass();
16
17     // ponteiro para uma lista de objetos tipo TaskClass
18     TaskClass *ptrTaskList[_MAX_N_TASKS];
19
20     // numero de Tarefas existentes
21     unsigned char nTasks;
22
23     // informa o numero de tarefas maximo
24     unsigned char getMaxNTasks(void);
25
26     // insere tarefa na lista
27     int InsertTask (TaskClass *ptrTask);
28
29     // Start data and timing. Need to be called right before loop
execution.
30     void Start (void);
31
32     // store previous time value
33     unsigned long time_prev;
34
35     // sort Task list in descending priority order (highest priority
to lowest priority)
36     void Sort (void);
37
38     // verify if period is over. If yes, set as ready.
39     boolean Run(void);
40
41     // get task with higher priority amoung ready tasks
42     TaskClass* GetReadyTask(void);
43
44     private:
45
46 };
47
48 #endif
```

DispatcherClass.cpp

```
 1 #include "DispatcherClass.h"
 2
 3 // Constructor
 4 DispatcherClass :: DispatcherClass()
 5 {
 6
 7 }
 8
 9 void DispatcherClass :: RunTask (TaskClass* f)
10 {
11    if (f == NULL)
12    return;
13
14    f->task_function();
15    f->ready = false;
16 }
```

DispatcherClass.h

```
#ifndef DISPATCHERCLASS_H
#define DISPATCHERCLASS_H

#include <Arduino.h>

#include "TaskClass.h"

class DispatcherClass
{
   public:
   // constructor
   DispatcherClass();

   // execute task:
   void RunTask(TaskClass* f);

   private:

};
#endif
```

TaskClass.cpp

```
#include "TaskClass.h"

// Task Constructor
TaskClass :: TaskClass(ptr_task_function f, unsigned long p)
{
    ready = false;

    task_function = f;

    elapsed_time = 0;

    period = p;
}
```

TaskClass.h

```
#ifndef TASKCLASS_H
#define TASKCLASS_H

#include <Arduino.h>

//typedef std::function<void(void)> timer_callback;
typedef void(*ptr_task_function)(void);

class TaskClass
{
public:

   // Constructor
   TaskClass(ptr_task_function f, unsigned long p);

   // indicates if task is ready to run
```

```
17    boolean ready;
18
19    // Task period. If 0 indicates an aperiodic task.
20    unsigned long period;
21
22    // Elapsed time unit since last execution.
23    unsigned long elapsed_time;
24
25    // Task to be executed in each period ou once.
26    ptr_task_function task_function;
27
28 private:
29
30 };
31
32 #endif
```

Tasks.cpp

```
1  #include "Tasks.h"
2
3  void Task_1_code (void)
4  {
5     char x;
6
7     x = 1;
8  }
9
10 void Task_2_code (void)
11 {
12    char x;
13
14    x = 1;
15 }
16
17 void Task_3_code (void)
18 {
19    char x;
20
21    x = 1;
22 }
```

Tasks.h

```
1  #ifndef TASKS_H
2  #define TASKS_H
3
4  void Task_1_code (void);
5
6  void Task_2_code (void);
7
8  void Task_3_code (void);
9
10 #endif
```