UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Bruno Fetter

Proposta de um Sistema Operacional Tempo-Real para plataforma Arduino

Porto Alegre

2014

Bruno Fetter

Proposta de um Sistema Operacional Tempo-Real para plataforma Arduino

Trabalho de Diplomação apresentado ao Departamento de Engenharia Elétrica da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como parte dos requisitos para obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Dr. Marcelo Götz

Porto Alegre

2014

AGRADECIMENTOS

Agradeço ao meu professor e orientador Dr. Marcelo Götz pela sugestão de projeto, pelo seu acompanhamento durante todo seu desenvolvimento e pelo tempo e dedicação prestados sempre que necessário.

Agradeço ao Prof. Dr. João Manoel Gomes da Silva Jr. pela sua visão e dedicação buscando a internacionalização da formação oferecia pela UFRGS, proporcionando oportunidades de intercâmbio junto a renomadas universidades estrangeiras.

Agradeço aos meus familiares e amigos, pelo apoio incansável durante toda a extensão deste curso de graduação.

RESUMO

A programação de aplicações nos processadores ATMega utilizados na plataforma do Arduino é facilitada pelo software disponível juntamente com esta plataforma, devido à sua linguagem de mais fácil acesso e disponibilidade de bibliotecas com funções e rotinas pré-programadas de fácil utilização. Porém, existe ainda a necessidade de um modelo estruturado de gerenciamento de recursos do processador, como tempo de processamento e periféricos. O sistema operacional tempo-real desenvolvido e aqui documentado visa atender esta necessidade, sendo utilizado para modelagem de tarefas periódicas e processos esporádicos, através de um escalonador preemptivo de prioridade fixa, tendo em vista requisitos de aplicações tempo-real. Este sistema garante o determinismo temporal, podendo ser utilizado para tarefas cujos períodos de execução não sejam inferiores a unidade do milissegundo, sob certas condições

Palavras-chave: Sistema Operacional Tempo-Real, Sistemas Embarcados, Arduino, Escalonador.

ABSTRACT

The programming of applications in the processors ATMega used in the Arduino Platforms is simplified by the software available within this platform, due to it's easy to use programming language and the availability of straight forwards pre-programmed libraries with functions and procedures. However, there is still need of a structured model to manage processors resources, like processing time and peripherals. The Real-Time Operating System developed and here documented aims to fulfill this need, being used to model periodic and non-periodic tasks through a fixed-priority preemptive scheduler with real-time constraints. This system assures a deterministic behavior and can be used to manage procedures with execution periods above one millisecond, under certain conditions.

Key words: Real-Time Operating System, Embedded Systems, Arduino, Scheduler

LISTA DE FIGURAS

Figura 1 – Arduino UNO 18
Figura 2 – Interface do Sistema Operacional com o Usuário 20
Figura 3 – Rotina para inicialização de tarefas 24
Figura 4 – Diagrama de estados do Sistema Operacional 26
Figura 5 – Estrutura de armazenamento de tarefas 28
Figura 6 – Rotina de criação de tarefa 29
Figura 7 – Rotina de organização de tarefas por prioridade 29
Figura 8 – Tempo de Ciclo Principal 30
Figura 9 – Rotinas para o cálculo do Tempo de Ciclo Principal 31
Figura 10 – Rotina para o cálculo do número de execuções de cada tarefa por ciclo 31
Figura 11 – Rotina de configuração do Timer responsável pela interrupção interna 32
Figura 12 – Rotina de Verificação de Tarefa 33
Figura 13 – Contexto de execução de um processador 34
Figura 14 – Rotinas de troca de contexto para um processador da família ATmega 35
Figura 15 – Rotina de preempção de tarefas 36
Figura 16 – Rotina de execução de tarefa 36
Figura 17 – Rotina de inicialização do Sistema Operacional 37
Figura 18 – Rotinas das tarefas para as simulações 39
Figura 19 – Resposta esperada para um escalonamento de três tarefas periódicas sem preempção 40
Figura 20 – Rotina para escalonamento de três tarefas periódicas sem preempção 41
Figura 21 – Escalonamento do RTOS desenvolvido para três tarefas periódicas sem preempção 41
Figura 22 – Período de execução da tarefa 1 42
Figura 23 – Resposta esperada para um escalonamento de duas tarefas periódicas com preempção 43
Figura 24 – Rotina para escalonamento de duas tarefas periódicas com preempção 43
Figura 25 – Escalonamento do RTOS desenvolvido para duas tarefas periódicas com preempção 44
Figura 26 – Medição temporal da saída da segunda tarefa em sua segunda execução 44
Figura 27 – Resposta esperada para um escalonamento de três tarefas periódicas com preempção 45
Figura 28 – Rotina para escalonamento de três tarefas periódicas com preempção 46
Figura 29 – Escalonamento do RTOS desenvolvido para três tarefas periódicas com preempção 46
Figura 30 – Medição temporal da saída da terceira tarefa em sua primeira execução 47
Figura 31 – Medição temporal da saída da terceira tarefa em sua segunda execução 47
Figura 32 – Resposta esperada para um escalonamento de duas tarefas periódicas e um processo esporádico, sem preempção 49
Figura 33 – Rotina para escalonamento de duas tarefas periódicas e um processo esporádico, sem preempção 49

Figura 34 – Escalonamento do RTOS desenvolvido para duas tarefas periódicas e um processo esporádico, sem preempção **50**
Figura 35 – Resposta esperada para um escalonamento de duas tarefas periódicas e um processo esporádico, com encadeamento de preempções **51**
Figura 36 – Rotina para escalonamento de duas tarefas periódicas e um processo esporádico, com encadeamento de preempções **52**
Figura 37 – Escalonamento do RTOS desenvolvido para duas tarefas periódicas e um processo esporádico, com encadeamento de preempções **52**
Figura 38 – Medição temporal da saída da terceira tarefa em sua primeira execução **53**

LISTA DE TABELAS

Tabela 1 – Especificações técnicas do Arduino UNO **18**
Tabela 2 – Propriedades das tarefas para o primeiro estudo de caso..**40**
Tabela 3 – Propriedades das tarefas para o segundo estudo de caso **43**
Tabela 4 – Propriedades das tarefas para o terceiro estudo de caso..**45**
Tabela 5 – Propriedades das tarefas para o quarto estudo de caso..**48**
Tabela 6 – Propriedades das tarefas para o quinto estudo de caso **50**

SUMÁRIO

1. INTRODUÇÃO 11

1.1. Contextualização e Problemática 11

1.2. Objetivos e Metodologia 12

1.3. Organização do Texto 13

2. REVISÃO BIBLIOGRÁFICA 14

2.1. Sistemas operacionais 14

2.2. Sistemas operacionais embarcados 15

2.3. Sistemas de tempo-real 16

2.4. A plataforma Arduino 17

3. DEFINIÇÃO DA INTERFACE E FUNCIONALIDADES 19

3.1. Interface com o Usuário 19

3.2. Definição das funcionalidades do Sistema Operacional 21

3.3. Argumentos necessários à criação de tarefas 23

4. FUNCIONAMENTO DO SISTEMA OPERACIONAL TEMPO-REAL 25

4.1. Macro-visão do Sistema 25

4.2. Rotinas de armazenamento de tarefas e ordenamento 27

4.3. Tempo de Ciclo Principal 30

4.4. Rotinas de interrupção por timer e verificação de tarefa 32

4.5. Rotinas de preempção e troca de contexto 34

4.6. Execução de tarefa 36

4.7. Inicialização do RTOS 37

4.8. Execução do RTOS 37

5. ESTUDOS DE CASO 38

5.1. Escalonamento temporal e cumprimento de deadlines 40

5.2. Duas tarefas com preempção 42

5.3. Três tarefas com preempção 45

5.4. Duas tarefas periódicas e um processo esporádico, sem preempção 48

5.5. Duas tarefas periódicas e um processo esporádico, com encadeamento de preempções 50

5.6. Limites temporais de interrupção e limitações em período 53

6. CONCLUSÕES 55

REFERÊNCIAS BIBLIOGRÁFICAS 56

ANEXO A – CÓDIGO COMPLETO DO RTOS 57

1. INTRODUÇÃO

1.1. Contextualização e Problemática

Desde o aparecimento dos primeiros microprocessadores na década de 70, diversos foram os avanços tecnológicos nesta área. São inumeráveis as aplicações e os sistemas de hoje cujo funcionamento depende de um microprocessador. Graças a sua miniaturização, possibilitada pela evolução da microeletrônica, eles tem sido embarcados em sistemas, aumentando ainda mais sua gama de aplicação, dando assim origem ao termo Sistemas Embarcados [1]. O cotidiano está cercado de sistemas embarcados, que algumas vezes passam até despercebidos. Exemplos vão desde calculadoras, relógios, controles remotos, aparelhos de televisão e jogos eletrônicos até sistemas mais complexos como sistemas de controle utilizados em automóveis, aviões, entre outros.

Utilizados previamente em um contexto profissional e acadêmico, os desenvolvimentos e pesquisas nestes sistemas, sejam em hardware e software, contribuíram para uma alta penetração destes processadores no mercado. Isto despertou também um grande interesse da parte dos *hobbystas* e criadores amadores, utilizando os microprocessadores como plataforma de criação e desenvolvimento das mais variadas aplicações.

Neste sentido, a plataforma de desenvolvimento Arduino [2] foi criada. Extremamente versátil, ela consiste de uma placa de prototipagem eletrônica com um hardware de fácil utilização, voltada para aplicações interativas de automação. Juntamente com este hardware, é disponibilizado um software com compilador na forma de um IDE (do inglês, Integrated Development Environment). Este software, em conjunto com as bibliotecas disponíveis, proporciona diversas funções básicas para esta plataforma, visando facilitar sua programação e interação com o usuário, tais como funções de leitura e escrita nas portas seriais, operações e comparações aritméticas e bit a bit, operações com ponteiros, gerenciamento de interrupções, entre outras. Esta plataforma encontra seu espaço em uma imensa gama de aplicações profissionais, amadoras ou acadêmicas.

Porém, como em todo microprocessador embarcado, esta plataforma ainda carece de um modelo estruturado para gerenciamento de tarefas e desenvolvimento de aplicações em um contexto mais complexo. O gerenciamento dos diversos recursos de um processador, como entradas e saídas, espaços em memória e

tempo do processador não são triviais para programadores que não possuem um profundo conhecimento do hardware no qual estão trabalhando.

Historicamente, o conceito de Sistema Operacional surgiu da necessidade de uma unidade central em gerenciar estes recursos entre as diversas tarefas solicitadas pelo usuário, criando uma camada intermediária entre o hardware e o software e servindo como base para criação e desenvolvimento de aplicações, sem a necessidade de um conhecimento aprofundado de hardware da parte do programador.

Paralelamente aos sistemas operacionais, a computação em tempo-real tem um papel crucial em sistemas que necessitam um controle computacional mais rigoroso. Estes sistemas devem garantir que a resposta seja processada e apresentada dentro de restrições temporais, chamadas de *deadlines*. Sistemas de automação, sistemas com atuadores e sistemas de controle são exemplos de sistemas em tempo-real: eles coletam informação do ambiente em sua volta, processam e devolvem a resposta dentro do deadline pré-estabelecido por suas condições de contorno.

1.2. Objetivos e Metodologia

É neste contexto que encontra-se o interesse no desenvolvimento de um Sistema Operacional Tempo-Real (RTOS, do inglês, *Real-Time Operating System*) para Arduino. A premissa deste sistema é de facilitar ainda mais o desenvolvimento de aplicações finais nesta plataforma, garantindo um sistema de gerenciamento de tempo de processador e cumprimento de deadlines especificados para cada tarefa, bem como prover uma hierarquia de prioridades para diferentes funções no mesmo código, sempre atendendo a requisitos temporais.

O presente projeto tem por objetivo a criação de um Sistema Operacional Tempo-Real para Arduino, para ser utilizado juntamente com seu ambiente de desenvolvimento na forma de uma biblioteca. Inserido neste contexto, serão abordadas as etapas de especificação deste desenvolvimento, baseando-se em uma análise das possíveis operações e requisitos de tal sistema e a interação destes recursos com o usuário final, visando uma maior versatilidade e facilidade de integração de processos complexos nesta plataforma, bem como confiabilidade em execução em tempo-real.

1.3. Organização do Texto

A presente documentação do projeto desenvolvido será estruturada da seguinte maneira:

- Revisão bibliográfica dos conteúdos necessários para boa compreensão deste trabalho:
  - Sistemas Operacionais;
  - Sistemas Operacionais Embarcados;
  - Sistemas Tempo-Real;
  - Plataforma Arduino.
- Análise das funcionalidades necessárias do ponto de vista do usuário, baseado nas possíveis aplicações para este sistema;
- Criação e desenvolvimento do RTOS, explicando seu funcionamento através de rotinas e funções;
- Estudos de caso que comprovam seu funcionamento;
- Limitações do sistema;
- Conclusões.

2. REVISÃO BIBLIOGRÁFICA

Nesta seção, será feita uma revisão bibliográfica dos conteúdos necessários para uma boa compreensão deste projeto, abordando-se aspectos gerais de sistemas operacionais, sistemas tempo-real, sistemas embarcados e da plataforma para a qual será desenvolvido o RTOS.

2.1. Sistemas operacionais

Em linhas gerais, o Sistema Operacional é o software responsável pelo gerenciamento de todos os recursos de hardware e software de um processador. Ele fornece serviços gerais de gerenciamento destes recursos de hardware e uma base sobre a qual todas as outras aplicações podem ser escritas. Sua definição formal, porém, foge do trivial e torna-se mais complexa à medida que se aprofunda nas camadas de um sistema.

No hardware puro, o desenvolvimento de uma aplicação se tornaria extremamente complexo e árduo se o programador tivesse que se preocupar especificamente com os recursos do sistema, como, por exemplo, o funcionamento dos discos rígidos, os protocolos de leituras em memória e os valores de registradores em particular. É dentro deste contexto que surgiram os primeiros sistemas operacionais, como uma camada intermediaria de software entre o hardware puro e as aplicações finais para gerenciar todas as partes do sistema e apresentar ao usuário uma interface de mais fácil criação e utilização.

A parte mais importante de um Sistema Operacional é o Escalonador. Ele é responsável pelo **escalonamento de processos**, seguindo algum algoritmo pré-determinado, definido pelo tipo de tarefas e aplicação na qual este processador será utilizado. Para execução deste algoritmo, o sistema deve manter uma estrutura de armazenamento para cada processo, com informações como seu status atual, um ponteiro que aponta para seu código em memória, seu período de execução (se aplicável), sua prioridade, entre outros.

A diferença fundamental entre algoritmos de escalonamento está na capacidade ou não de **preempção**. Preempção de um processo é o ato de interromper

temporariamente sua execução, favorecendo a execução de um processo de maior prioridade. A mudança entre processos é chamada de **troca de contexto**, e envolve o salvamento em memória dos registradores e da posição de leitura do código de forma que, após a execução do processo de maior prioridade, não existam conflitos na retomada de execução do processo que foi anteriormente preemptado.

Pode-se então dividir os algoritmos de escalonamento em dois tipos principais:

- **Escalonamento preemptivo**

  Neste tipo de escalonamento, o escalonador permite que um processo seja interrompido durante sua execução. Dependendo do tipo de sistema operacional, isto pode ser feito automaticamente pelo escalonador ou em determinados pontos de execução das tarefas, onde são buscadas tarefas de maior prioridade prontas para serem executadas. Após a interrupção do processo, ocorre a troca de contexto, garantindo que, na retomada do processo preemptado, a execução volte exatamente ao mesmo ponto anteriormente interrompido. Cabe ressaltar que múltiplas trocas de contexto e preempções podem ocorrer simultaneamente, causando um encadeamento de salvamentos e restaurações de contexto, sem gerar problemas na execução do sistema.

- **Escalonamento não preemptivo**

  Como o nome indica, neste tipo de escalonamento não existe a interrupção de um processo em execução para troca de contexto. Uma vez iniciado um processo, o processador só iniciará outro quando o primeiro estiver concluído.

2.2. Sistemas operacionais embarcados

Um sistema embarcado é um sistema de computação dentro de um sistema elétrico ou mecânico de maior porte. Contrariamente aos computadores de uso geral, criados para atender a uma grande gama de aplicações diferentes, os sistemas embarcados possuem uma limitação em termos de recursos, sendo utilizados para aplicações específicas. Devido a isto, eles podem ser projetados com tamanhos inferiores, otimizando sua confiabilidade e desempenho.

O sistema operacional para tais processadores embarcados tende a ser primitivo e com funcionalidades limitadas, abrindo mão de diversos recursos que os sistemas não-embarcados possuem. Eles são desenhados para serem compactos, eficientes na utilização dos recursos e confiáveis, cobrindo apenas o escopo das tarefas para as quais são criados. Normalmente não possuem interface com o usuário, limitando-se à execução do que foi previamente programado.

2.3. Sistemas de tempo-real

Sistemas de tempo-real estão ligados, imperativamente, a sistemas de computação que monitoram, respondem e/ou controlam um ambiente externo, através de sensores, atuadores e outras interfaces. Diferentemente de sistemas convencionais, estes sistemas necessitam ler, processar e enviar informações dentro de uma asserção temporal imposta pelo próprio meio no qual ele está inserido, adicionando uma variável temporal em sua operação.

Um sistema de computação que está sujeito a tais restrições temporais impostas pelo comportamento de tempo real do mundo externo com o qual faz interface é chamado de *sistema de tempo-real*. Estes sistemas buscam um comportamento determinístico executando as suas tarefas, possibilitando um maior controle e previsão sobre os tempos de execução. Diversos são os sistemas de hoje que possuem tais requerimentos, como por exemplo, um sistema de controle de veículos, sistemas de controle de processos para usinas de energia, sistemas de monitoramento de pacientes em um hospital, entre outros. Sistemas embarcados estão sempre sujeitos a estas restrições devido à sua interação com o ambiente no qual ele está inserido.

Nestes sistemas, a preocupação está focada principalmente na arquitetura do *software*. Em sistemas de tempo-real, existe uma preocupação não somente com sua capacidade de produzir a resposta adequada, mas também com sua capacidade de prover esta resposta satisfazendo asserções temporais, que podem envolver tempos relativos e absolutos [3]. A asserção mais comum em tais sistemas é o *deadline*, um limite sobre o tempo no qual a computação deve completar-se. Qualitativamente, estes sistemas podem ser diferenciados em sistemas de tempo-real *estritos (hard)* e sistemas de tempo-real *tolerantes (soft)*. Se uma restrição temporal em um sistema estrito é violada, o sistema, imperativamente, falha. Por

outro lado, sistemas tolerantes podem ser considerados bem-sucedidos mesmo se algumas de suas restrições não são atendidas.

2.4. A plataforma Arduino

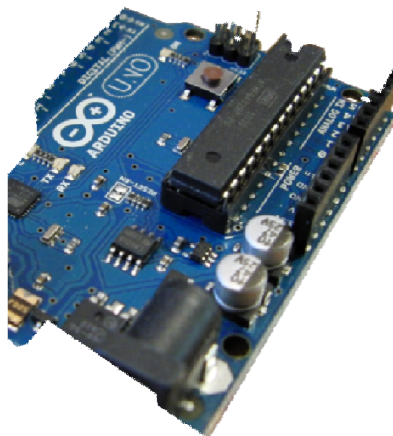
A plataforma Arduino é composta por um processador da marca Atmel, entradas e saídas digitais programáveis - entre as quais algumas podem ser usadas como PWM – entradas e saídas analógicas e uma interface USB para gravação do código [2]. O modelo do processador (normalmente da família ATMega) e o número de entradas e saídas depende do modelo do Arduino utilizado, proporcionando versatilidade na escolha da plataforma adequada para cada aplicação. Complementarmente, existe disponível comercialmente uma grande quantidade de módulos com funções especificas para serem acoplados ao Arduino, chamados de *Shields*, como por exemplo, módulos WiFi, Bluetooth, GSM e outros, criando assim mais uma infinidade de possibilidades de aplicações para tal plataforma, sem a preocupação de escrever protocolos de comunicação.

O software disponível juntamente com esta plataforma facilita a programação do processador, tendo uma linguagem de programação baseada em *Wiring [2]*, muito semelhante ao C/C++, não sendo necessário, portanto, conhecimento das diretivas em Assembly nem as diretivas de configuração do compilador associadas ao processador. Juntamente com isto, são disponibilizadas diversas bibliotecas com funções básicas que proporcionam maior facilidade na criação de programas: funções de leitura e escrita em serial, operações lógicas e aritméticas, leitura e escrita em entradas analógicas e digitais, controle temporal, entre outros.

Dos diferentes modelos de Arduino existentes, o Arduino UNO foi utilizado para os testes e estudos realizados neste projeto (Figura 1). Ele possui um número razoável de entradas e saídas e um processador que atende a maior parte das necessidades impostas pelos usuários desta plataforma. Suas especificações estão listadas na Tabela 1.

Devido à compatibilidade existente entre os diferentes modelos, em nível de utilização de processadores da família AVR, o sistema operacional desenvolvido não está apenas restrito ao Arduino UNO, sendo compatível com os diferentes modelos desta plataforma.

Figura 2 - Arduino UNO



Fonte – [2]

Tabela 1 – Especificações técnicas do Arduino UNO

Microcontrolador	ATMega328
Tensão de operação	5 V
Tensão de alimentação	6 - 20 V
Pinos entrada e saída digitais	14, sendo que 6 podem ser usadas como PWM
Pinos entrada e saída analógica	6
Corrente por pino I/O	40 mA
Memória Flash	32 KB, sendo 0,5 KB usados pelo Bootloader
SRAM	2 KB
EEPROM	1 KB
Clock	16 MHz
Interrupções externas	2
Comunicação serial	UART TTL (5V)

Fonte – [2]

3. DEFINIÇÃO DA INTERFACE E DAS FUNCIONALIDADES

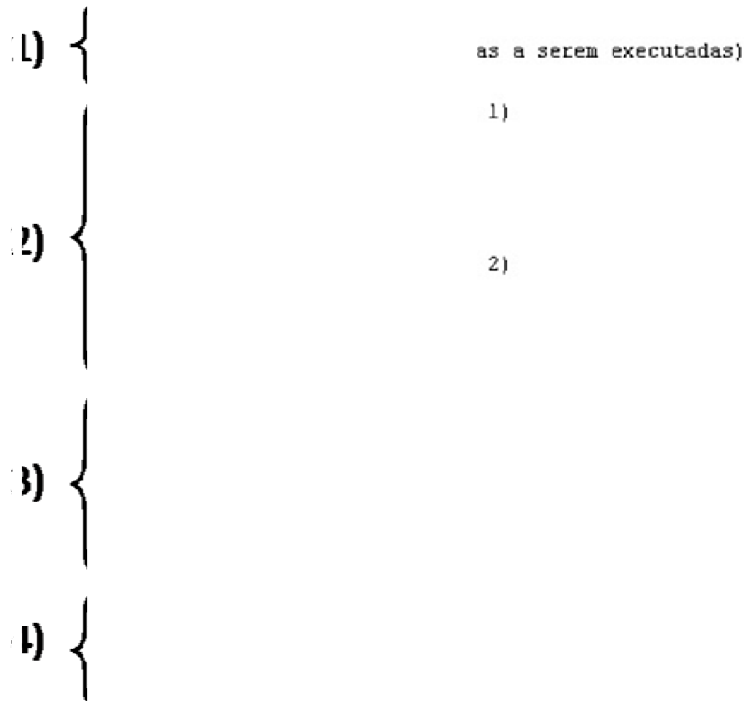
Neste capítulo, será definida a estrutura em que este sistema será organizado e a interface com o usuário, juntamente com as funcionalidades do RTOS e os argumentos iniciais que devem ser passados ao sistema para seu bom funcionamento. Ele será dividido em três partes: primeiramente, será apresentada a estrutura de programação deste sistema do ponto de vista do usuário, exemplificando a declaração de processos e a inicialização deste sistema. Esta primeira definição é importante, pois dela depende a forma como este sistema será desenvolvido. Em seguida, serão definidas as funcionalidades que este sistema terá, qual seu tipo de escalonamento e para que tipos de processos ele deve ser utilizado. Por fim, serão apresentados os argumentos que devem ser passados juntamente com a criação de uma tarefa pelo usuário.

3.1. Interface com o usuário

Mesmo proporcionando uma interface mais completa e intuitiva do que um processador comum, a plataforma Arduino necessita ainda de uma estrutura para gerenciamento de recursos e desenvolvimentos de aplicações em nível acadêmico ou profissional. Esta estrutura deve criar uma interface que facilita a programação do Arduino visando aplicações embarcadas, proporcionando funções de sistema operacional e garantindo o bom desempenho em tempo-real do processador.

Sendo um sistema embarcado, procura-se um sistema operacional com alto desempenho e voltado para aplicações extremamente específicas. O usuário deve então ser capaz de declarar, na IDE do Arduino, os processos que devem ser executados. Ele deve associar a estes processos uma série de argumentos pré-determinados, garantindo um bom poder de decisão ao escalonador no momento de seu funcionamento. Então, vista pelo lado de programação de aplicações finais, esta interface está ilustrada na Figura 2. Pode-se perceber que esta estrutura apresentada é a forma mais simples de programar um sistema operacional neste ambiente de desenvolvimento.

Figura 2 – Interface do Sistema Operacional com o Usuário



Então, a programação feita pelo usuário restringe-se à utilização das 4 estruturas apresentadas na Figura 2:

1. Nesta primeira parte, a biblioteca deve ser inclusa no código e declarado o número de tarefas que farão parte deste programa.

2. As rotinas de cada tarefa devem, então, ser declaradas. É importante ressaltar que as rotinas devem ser únicas para cada tarefa e sua quantidade igual ao número de tarefas definidas no começo do código. O argumento de cada rotina pode ser utilizado para passar valores iniciais às tarefas.

3. As tarefas que foram declaradas nas rotinas anteriores devem ser inicializadas no sistema operacional, vinculando-as com argumentos necessários a seu funcionamento. Estes argumentos são uma função direta do tipo e das operações que este sistema operacional irá realizar, sendo discutidos em detalhes no decorrer deste capítulo. Após isto, deve ser

chamada a rotina de inicialização do sistema operacional. Seu funcionamento será discutido em detalhes no Capítulo 4.

4. Aqui, o sistema operacional começa seu funcionamento e roda indefinidamente, utilizando os parâmetros definidos anteriormente e as variáveis inicializadas. Bem como sua rotina de inicialização, ele será discutido em detalhes no Capítulo 4.

A seguir, parte-se para uma análise do ponto de vista da necessidade das aplicações para definir quais os serviços que este sistema operacional irá fornecer ao Arduino.

3.2. Definição das funcionalidades do Sistema Operacional

Tendo em vista a interface proposta na seção anterior, o próximo passo é a determinação das funcionalidades que o sistema operacional irá exercer, bem como seu tipo de escalonamento e divisão de tarefas. Para isto, será feita uma análise das possíveis aplicações para as quais esta plataforma será utilizada.

O Arduino é uma plataforma voltada para aplicações embarcadas, sendo utilizado como forma de monitoramento e atuação no ambiente em sua volta. A modelagem de um sistema operacional para estas aplicações deve, então, ser feita na forma de **tarefas periódicas**. Estes processos são normalmente utilizados para monitoramento sistemático, verificação de entradas e saídas ou amostragem de informação a partir de sensores por um intervalo longo de tempo. Por exemplo, pode-se empregar um processo periódico para ler a temperatura de um sensor a cada 50 milissegundos, varrer um teclado a cada 20 milissegundos e atualizar uma saída a cada 10 milissegundos.

É importante ter em mente que nem todos os processos para aplicações embarcadas podem ser periódicos; algumas vezes eles dependem de um evento ou de um acionamento externo (leitura de sensor, chegada de dados em algum barramento, entre outros) para serem executados. Desta forma, deve-se criar a possibilidade de utilização, juntamente com as tarefas periódicas, de **processos esporádicos**. Estes processos devem ser ativados por interrupções externas na

placa do Arduino, nos pinos previamente programados para isto. É importante salientar que a execução de um processo esporádico não é automaticamente executada na ocorrência de um evento externo, ele é apenas adicionado à fila do escalonador e será executado conforme sua prioridade, sendo submetido então às mesmas restrições que tarefas periódicas.

Como o Arduino possui apenas uma unidade de processamento, sabe-se que em qualquer instante de tempo, apenas uma tarefa pode estar sendo executada. O usuário deve ser capaz de definir, através de uma ordem pré-estabelecida, qual tarefa tem prioridade sobre as demais. Fisicamente, isto se traduz em processos com importância superior aos outros: por exemplo, a leitura de um sensor é mais importante que a atualização de um display. Então, deve ser possível ao usuário definir uma ordem de prioridade para as tarefas que ele declarou, organizando-as hierarquicamente. Na modelagem de sistemas embarcados, o usuário tem um bom conhecimento sobre as tarefas que serão executadas no processador, bem como a importância relativa de cada uma delas com relação às outras. Portanto, foi escolhido o uso de processos de **prioridade fixa**. Esta prioridade deve ser única e imutável durante toda a execução da aplicação, criando uma hierarquia que facilita o escalonador na divisão do tempo de processamento entre tarefas diferentes.

O algoritmo utilizado no escalonador deve ser capaz de definir qual tarefa deverá ser executada em cada instante de tempo, baseando seu algoritmo na prioridade dos processos. Quando mais de uma tarefa está pronta para ser executada em um determinado instante de tempo, o escalonador optará sempre por executar a tarefa de maior prioridade. Quando o sistema está executando uma tarefa e outra de maior prioridade fica pronta para ser executada, deve haver uma **troca de contexto** para a preempção da tarefa de menor prioridade e execução da tarefa de maior prioridade. Um escalonador com esta propriedade é chamado de **escalonador preemptivo**.

A partir desta análise, pode-se definir o tipo de funcionalidades que o RTOS deverá executar e suas propriedades:

- Sistema de **prioridade fixa;**
- Processos **periódicos** com período definido e **processos esporádicos** gerenciados por interrupção externa;
- Escalonamento **preemptivo;**

3.3. Argumentos necessários à criação de tarefas

Baseando-se nesta interface de utilização e funcionalidades definidas, o próximo passo é a determinação dos argumentos que o usuário deve passar ao inicializar as tarefas em função do tipo de sistema operacional que será desenvolvido e de seu funcionamento. Para um sistema com as propriedades listadas anteriormente, os argumentos que devem ser passados na declaração de funções, como mostrado na estrutura 3 da Figura 2, são:

1. **Período:** toda tarefa deve ser associada a um período de execução. Por padrão, este período deve ser inserido em milissegundos. No caso de processos esporádicos, deve-se inserir um período igual a zero.

2. **Prioridade:** as tarefas devem ser declaradas com prioridades relativas únicas e decrescentes. Sendo N o número de tarefas, aquela de prioridade 1 será a mais importante e a de prioridade N menos importante. Esta declaração deve ser compatível com o número de tarefas definido na inicialização do RTOS

3. **Ponteiro da rotina:** Um argumento do tipo ponteiro deve ser associado a cada tarefa para indicar em que posição da memória se encontra sua rotina de execução.

4. **Argumento inicial:** Algumas tarefas podem utilizar um valor inicial para sua execução, podendo ser inicializado nesta rotina ou adicionado em algum outro ponto do código. Com isto, é possível a troca de informações entre diferentes tarefas.

Então, a estrutura de inicialização de tarefas apresentada na Figura 2 pelo índice (3) pode ser redefinida como a estrutura apresentada na Figura 3,utilizando-se como exemplo um RTOS de duas tarefas:

Figura 3 – Rotina para inicialização de tarefas

```
void setup()
{
  CreateTask(Periodo 1, Prioridade 1, Rotina 1, Argumento 1);
  CreateTask(Periodo 2, Prioridade 2, Rotina 2, Argumento 2);
  OSSetup();
}
```

Conhecendo-se as funcionalidades que este sistema irá executar e a forma como o usuário define as funções e seus argumentos, pode-se prosseguir para a parte fundamental deste projeto: o desenvolvimento e análise do funcionamento do Sistema Operacional de Tempo-Real.

4. FUNCIONAMENTO DO SISTEMA OPERACIONAL TEMPO-REAL

Baseando-se nas propriedades e nos parâmetros definidos no capitulo precedente, o objetivo desta seção é de descrever o desenvolvimento e o funcionamento do RTOS. Este desenvolvimento será apresentado de uma forma sequencial, facilitando a compreensão de seu funcionamento. Primeiramente, será apresentado uma macro visão deste sistema, na forma de um diagrama de estados, para uma compreensão global do algoritmo utilizado e atuação do escalonador. Após, aprofundando-se no código, serão apresentadas as diversas subrotinas que compõe esse sistema operacional, bem como na forma em que elas se integram e o sistema as organiza, resultando em um sistema com as propriedades definidas na seção precedente. Cabe ressaltar que, neste capítulo, não será feita uma análise exaustiva de todo o código implementado, sendo apresentadas apenas as rotinas fundamentais para seu funcionamento, omitindo-se declarações de variáveis e algumas rotinas menos importantes. Para uma análise detalhada do código, vide Anexo A.

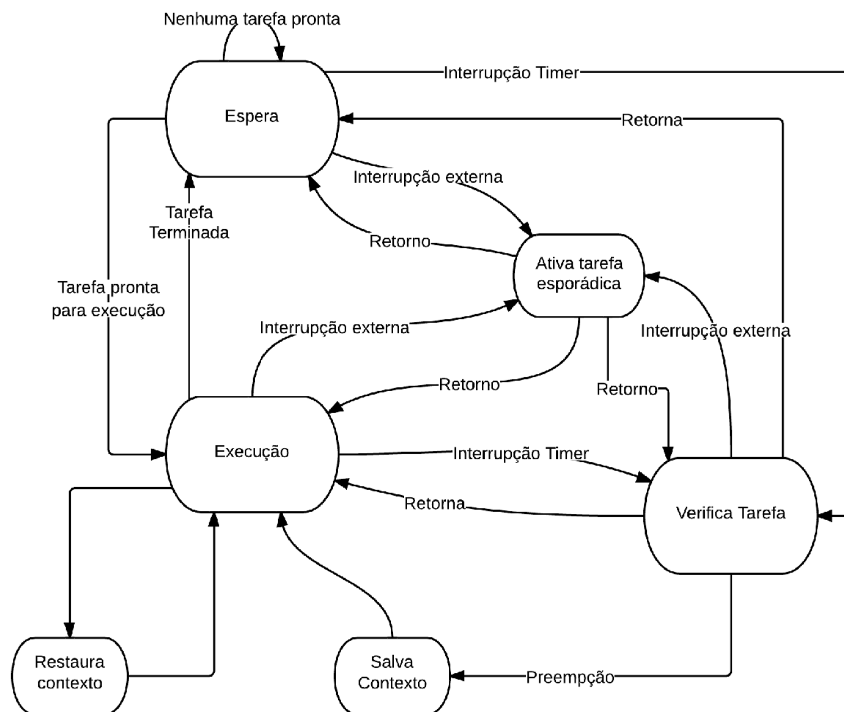
4.1. Macro-visão do sistema

Pode-se representar o sistema operacional desenvolvido através de um diagrama de estados simplificado, apresentado na Figura 4. Este diagrama omite os detalhes dos estados, concentrando-se no funcionamento global deste sistema. As particularidades das suas rotinas e execuções, porém, são relativamente mais complexas e discutidas no decorrer deste capítulo. Analisando-se brevemente cada estado que compõe este diagrama, pode-se entender como este sistema funciona:

- **Espera**

  Neste estado, o escalonador varre a lista de tarefas, por ordem de prioridade, em busca de uma tarefa que está pronta para execução. Quando ele a encontra, ocorre uma mudança de estado e o sistema passa a executar esta tarefa. Tanto as interrupções externas quanto internas estão habilitadas, fazendo com que o sistema mude de estado na ocorrência de alguma delas.

Figura 4 – Diagrama de estados do Sistema Operacional



- **Executa**

Quando o escalonador encontra uma tarefa pronta para ser executada, o sistema passa à sua execução. Uma vez terminada, o escalonador retoma o controle e entra em estado de espera novamente. De forma similar ao estado anterior, as interrupções também estão habilitadas.

- **Verifica tarefa**

Este estado é o responsável por dizer ao escalonador quando uma tarefa está pronta para execução ou não. Ele é acessado através de uma interrupção interna por timer, em uma frequência pré-definida.

Se for necessário, ele também é responsável pela preempção da tarefa atualmente em execução e troca de contexto, chamando as rotinas adequadas.

- **Ativa tarefa esporádica**

  Nas aplicações que utilizam processos esporádicos, este estado é acessado através da interrupção externa associada a este processo. Ele é responsável por deixar pronto para execução o processo assim definido. Isso não quer dizer, necessariamente, que a tarefa esporádica será executada imediatamente após esta ativação: ela será apenas colocada na lista de tarefas prontas para serem executadas e será tratada hierarquicamente pelo escalonador.

- **Salva contexto**

  É o estado responsável pela rotina de salvamento de contexto quando uma tarefa é preemptada. As interrupções são desabilitadas, pois esta parte de código é crítica e deve ser executada de maneira atômica.

- **Restaura contexto**

  Similarmente ao salvamento de contexto, aqui é chamada a rotina para restaurar o contexto e retomada de execução da tarefa que foi anteriormente preemptada.

Tendo em mente esta macro-visão do sistema operacional, vamos agora entrar nos detalhes do seu funcionamento, explicando as particularidades das rotinas que o compõe e como elas interagem garantindo o seu bom funcionamento.

4.2. Rotinas de armazenamento de tarefas e ordenamento

Na seção precedente, foi definido que o usuário deverá passar quatro parâmetros iniciais ao declarar uma tarefa: seu período, sua prioridade relativa, o ponteiro que guarda a posição na memória da rotina e um argumento inicial que poderá ser usado no decorrer da tarefa. Juntamente com isso, o sistema deve armazenar um flag que indica se a tarefa está pronta para execução (valor lógico 1) ou não (valor lógico 0). É através deste flag que o escalonador sabe quando uma tarefa está pronta ou não, e seu controle é feito através do estado de verificação de tarefa, que será discutido em detalhes posteriormente.

Para armazenar todos estes dados e associá-los a uma tarefa específica, foi necessária a criação de uma estrutura de armazenamento. Na Figura 5, a estrutura, chamada de TLM (do inglês, Task List Manager), foi criada com os diversos argumentos que devem ser passados a uma tarefa na sua inicialização, seja pelo usuário ou pelo próprio sistema operacional. São através destas variáveis que o sistema diferencia as tarefas e as gerencia segundo suas prioridades e períodos.

Figura 5 – Estrutura de armazenamento de tarefas

```
typedef struct tc
{
        uint8_t pid;                // ID do processo
        uint8_t prio;               // Prioridade única decrescente, sendo 1 a mais alta
        unsigned long per;          // Período, em ms
        unsigned int flag;          // Flag de execução: 1 = Tarefa pronta para ser executada;
                                    //                   0 = Tarefa não está pronta para execução
        unsigned int inter;         // Bit que define se a tarefa é periódica (0) ou ativada por interrupção (1)
        unsigned int interId;       // ID que define a posição da tarefa de interrupção da estrutura
        void (*taskptr)(void*);     // Ponteiro que aponta para a rotina de execução da tarefa
        void *arg;                  // Argumento passado para a tarefa na sua inicialização
} TLM;                              // TLM = Task List Manager
TLM _tasks[TASKS];                  // Inicialização da estrutura na forma de um vetor
```

Então, criando-se um vetor de estruturas TLM, pode-se armazenar as diversas tarefas e suas propriedades através de índices crescentes e únicos para cada processo.

Associando-se então a função definida anteriormente na Figura 2 com o vetor criado pode-se facilmente armazenar as propriedades das tarefas passadas pelo usuário com posições neste vetor, como visto na Figura 6.

Antes da execução do sistema, convém ao escalonador organizar as tarefas dentro desta estrutura, seguindo uma ordem decrescente de prioridade. Esta lista é ordenada por um método de flutuação, também chamado de *bubble sort* [4]. A ideia deste algoritmo é percorrer o vetor diversas vezes, colocando em cada passagem a tarefa de maior prioridade no começo da estrutura. A inconveniência deste algoritmo de triagem é sua complexidade, sendo de Ordem Quadrática (n^2). Visto que ele é executado apenas na inicialização do sistema operacional, sua execução não influencia o desempenho final do sistema. A rotina de organização é apresentada na Figura 7.

Figura 6 – Rotina de criação de tarefa

```
void CreateTask(unsigned long period, int priority ,void (*rptr)(void *), void *arg)
{
  _tasks[_procCount].per = period;
  _tasks[_procCount].prio = priority;
  _tasks[_procCount].taskptr = rptr;
  _tasks[_procCount].pid = _procCount;
  _tasks[_procCount].arg = arg;
  if (period == 0)
  {
    numInter=_procCount;
    codeInt = true;
    _tasks[_procCount].inter = 1;
    _tasks[_procCount].interId = numInter;
    intCount++;
  }
  else
  {
    _tasks[_procCount].inter = 0;
    _tasks[_procCount].interId = NULL;
  }

  _procCount++;


}
```

Figura 7 – Rotina de organização de tarefas por prioridade

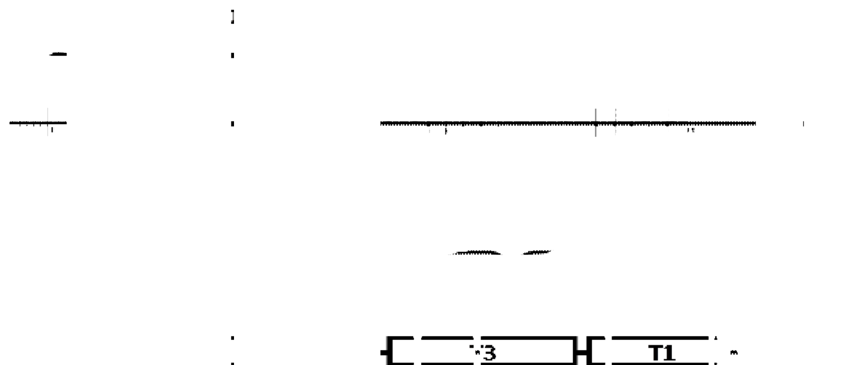
```
void Ordena()
{
  TLM prov;
  for (int i = 0; i< TASKS-1; i++)
   {
     for (int j=i+1; j< TASKS; j++)
     {
       if (_tasks[j].prio < _tasks[i].prio)
       {
         prov.per = _tasks[i].per;
         prov.prio = _tasks[i].prio;
         prov.taskptr = _tasks[i].taskptr;
         prov.pid =   _tasks[i].pid;
         prov.arg = _tasks[i].arg;
         prov.inter = _tasks[i].inter;
         prov.interId = _tasks[i].interId;

         _tasks[i].per = _tasks[j].per ;
         _tasks[i].prio = _tasks[j].prio;
         _tasks[i].taskptr = _tasks[j].taskptr;
         _tasks[i].pid = _tasks[j].pid;
         _tasks[i].arg = _tasks[j].arg;
         _tasks[i].inter = _tasks[j].inter;
         _tasks[i].interId = _tasks[j].interId;

         _tasks[j].per = prov.per ;
         _tasks[j].prio = prov.prio;
         _tasks[j].taskptr = prov.taskptr;
         _tasks[j].pid = prov.pid;
         _tasks[j].arg = prov.arg;
         _tasks[j].inter = prov.inter;
         _tasks[j].interId = prov.interId;
       }
     }
   }
}
```

4.3. Tempo de Ciclo Principal

Devido à necessidade de tarefas periódicas determinada na seção precedente e às limitações físicas do Arduino em interrupções externas, a maior parte dos processos será composta de tarefas periódicas. O principal objetivo do escalonador é assegurar que todas as tarefas satisfaçam seus *deadlines.* Para alcançar este objetivo, deve-se predefinir, antes da execução do sistema operacional, uma intercalação de execução de processos que produzam um escalonamento viável de execução. Como as tarefas modeladas são periódicas, o escalonador também o será. Os processos esporádicos deverão então ser executados entre os processos intercalados, seguindo sua ordem de prioridade. Este enfoque cíclico é muito popular entre os projetistas de sistemas de tempo-real, porque é simples, gera um sistema de execução eficiente e produz comportamento altamente previsível [3].

Para isto, deve-se alocar blocos de execução para o processador de tal forma que os deadlines e períodos de todos os processos sejam satisfeitos. Esta divisão é chamada de *escalonamento principal*, sendo o tempo mínimo para execução deste bloco chamado de **tempo ciclo principal (TCP),** representado na Figura 8 [3]. O ciclo principal conterá, pelo menos, um período de cada processo. Assumindo-se que os processos são iniciados ao mesmo tempo, o tempo de ciclo principal será o mínimo múltiplo comum entre os períodos dos processos. O MMC é a única e correta possibilidade para este tempo, porque é o menor tempo repetível dentro do qual todos os processos podem executar pelo menos uma vez.

Figura 8 – Tempo de Ciclo Principal



Então, faz-se necessária uma rotina para o cálculo do mínimo múltiplo comum entre os períodos das tarefas. Em sistemas computacionais, este cálculo é feito interativamente entre pares de números, utilizando-se o máximo divisor comum entre eles, através de uma fatoração simultânea. De mesma forma, faz-se necessária uma rotina para o calculo do número de execuções de cada tarefa dentro de um ciclo principal. Tais rotinas são apresentadas nas Figuras 9 e 10, respectivamente.

Figura 9 – Rotinas para o cálculo do Tempo de Ciclo Principal

```
unsigned long mdc(unsigned long a, unsigned long b)
{
  unsigned long t;
  while (b!= 0){
    t = b;
    b = a % b;
    a = t;
  }
  return a;
}

unsigned long mmc(unsigned long a, unsigned long b)
{
  unsigned long t;
  t = a*b/mdc(a,b);
  return t;
}

unsigned long TempoCicloPrincipal ()
{
unsigned long a = 1;
  for (int i =0; i<TASKS; i++)
  {
    if (_tasks[i].per != 0)
    {
    a = mmc(a, _tasks[i].per);
    }
  }
  return a;
}
```

Figura 10 – Rotina para o cálculo do número de execuções de cada tarefa por ciclo

```
void NumeroProcessos()
{

    for (int i =0; i<TASKS; i++)
    {
      if (_tasks[i].per !=0)            // Não aplicável à tarefas não periódicas
      {
      numExec[i] = TCP/_tasks[i].per;    // Define o numero de execuções de uma tarefa dentro do ciclo principal
      numProc = numProc + numExec[i];    // Numero de execuções totais dentro do ciclo principal
      }
    }
}
```

4.4. Rotinas de interrupção por timer e verificação de tarefa

Prosseguindo com o desenvolvimento deste sistema, deve-se definir o funcionamento da interrupção temporal, a forma como ela verifica se existe uma tarefa pronta para ser executada e como ela verifica se é necessária ou não a preempção da tarefa atual e uma mudança de contexto.

Nos processadores da família ATmega, existem 3 temporizadores (timers) internos que possibilitam uma utilização de interrupção [5]. Neste sistema operacional, foi escolhido o timer1, único no processador de 16 bits, possibilitando o uso de frequências de interrupção superiores. A frequência de interrupção deste timer foi definida experimentalmente em 7,8 kHz (uma interrupção a cada 0,128 ms, aproximadamente), como será discutido em detalhes no próximo capítulo. A rotina de configuração deste timer está representada na Figura 11.

Figura 11 – Rotina de configuração do Timer responsável pela interrupção interna

```
void ConfiguraTimer()
{

  noInterrupts();
  TCCR1A = 0;                              // Modo de operação normal
  TCCR1B = 0;                              // Timer parado

  timer1_counter = 65534;                  // Preload timer na frequência desejada: 65536-(16MHz/1024/7,8KHz)

  TCNT1 = timer1_counter;                  // Preload timer
  TCCR1B |= (1 << CS12) | (1 << CS10);     // Definição do prescaler = 1024  e inicio do timer
  TIMSK1 |= (1 << TOIE1);                  // Habilita interrupção com overflow do timer
  interrupts();


}
```

Neste modo de funcionamento, a cada overflow do timer será chamada uma rotina de atendimento à interrupção. Esta rotina, por sua vez, recarrega o timer e chama a rotina de verificação de tarefa para determinar se algum processo está pronto para ser executado. Sua verificação é baseada em algumas variáveis: tempo absoluto no qual o ciclo atual foi iniciado, tempo absoluto atual do sistema, número de execuções da tarefa no ciclo e número de execuções restantes da tarefa neste ciclo. O sistema realiza então a comparação, tarefa a tarefa, dada pela Equação 1.

$$t_{atual} \geq [t_{início\ ciclo} + T * (N_{exec} - N_{rest})] \quad (1)$$

onde t_{atual} é o tempo absoluto do sistema no instante de tempo da comparação, $t_{início\ ciclo}$ é o tempo no qual o ciclo principal foi iniciado, T é o período da tarefa em comparação, N_{exec} é o número de execuções totais desta tarefa por ciclo e N_{rest} é o número de execuções restantes desta tarefa neste ciclo.

Quando esta desigualdade for verdadeira, a tarefa que está sendo verificada estará pronta para execução. Isto garante que cada tarefa seja executada uma vez dentro de cada um de seus períodos compreendidos dentro do ciclo principal. Para ilustrar este funcionamento, consideramos uma tarefa que é repetida quatro vezes dentro do ciclo principal. Seu período de execução é, então, quatro vezes menor que este tempo de ciclo. Em cada uma de suas execuções, a tarefa deve respeitar os limites de cada período. No início do ciclo, o termo $(N_{exec} - N_{rest})$ da Equação 1 é nulo, e o sistema deixa a tarefa pronta para execução. Após a primeira execução da tarefa, o termo N_{rest} terá seu valor diminuído de uma unidade. Então, esta desigualdade será verdadeira novamente quando o tempo absoluto de execução for maior que o tempo de início mais uma vez o período da tarefa. De forma sucessiva, esta comparação garante que o sistema deixará as tarefas prontas para execução no instante de tempo correto.

Juntamente com esta verificação, esta rotina é responsável por verificar se existe a necessidade de preempção da tarefa que está sendo atualmente executada. Esta rotina é apresentada na Figura 12.

Figura 12 – Rotina de Verificação de Tarefa

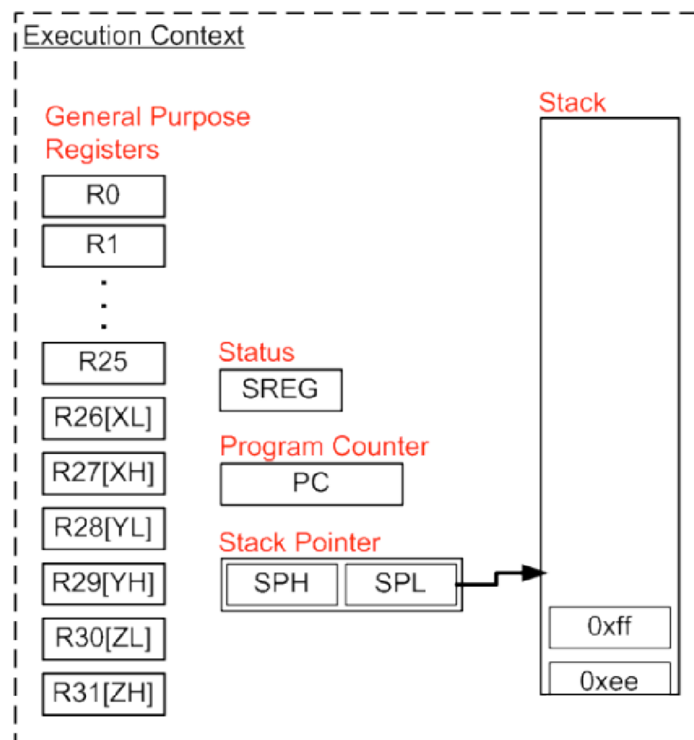
```
void VerificaTarefa()
{
  for (int i = 0; i< TASKS; i++)
  {
    ticks = millis();
    if ((_tasks[i].flag == 0) && (ticks > (start + ((_tasks[i].per)*(numExec[i] - numTask[i])))) && (_tasks[i].inter == 0))
    {
      _tasks[i].flag = 1;
      if ((taskexec == true) && (i < _running))     // Se existe uma tarefa sendo executada e a prioridade da tarefa que teve
      {                                             // seu flag ativado é maior, chama rotina de preempção
        PreempExec (i);
      }
    }
  }

}
```

4.5. Rotinas de preempção e troca de contexto

Durante a execução de uma tarefa, ela utiliza diversos recursos do processador, como registradores, memória RAM e ROM. Estes recursos compõem o contexto da tarefa [7], como representado na Figura 13. A rotina de preempção, então, deve salvar este contexto e executar a tarefa de maior prioridade, retomando este contexto posteriormente. Em um processador da família ATMega, fazem parte do contexto [5]:

- 32 registradores de uso geral;
- Registrador de status, que contém informação sobre as operações aritméticas executadas recentemente, como flag de número negativo, carry, zero, entre outros;
- Contador do programa, necessário para retomada da tarefa na instrução que seria executada imediatamente antes de sua suspensão;
- Os dois registradores do stack pointer.

Figura 13 – Contexto de execução de um processador



Fonte – [6]

Para as rotinas de troca de contexto nos processadores AVR, não se pode evitar códigos em Assembly. Tais rotinas estão apresentadas na Figura 14. Observando-se a rotina de salvamento de contexto, percebe-se que o contexto é salvo em uma lista do tipo LIFO (Last In First Out), utilizando as diretivas *push* e *pop* em Assembly. Em sua linha (2) o registrador de status SREG é armazenado no registrador R0 para ser posteriormente salvo em memória. Na linha (3), as interrupções são desabilitadas, pois esta é uma parte crítica do código que deve ser executada sequencialmente. Nas linhas (37) a (40), o stack pointer é salvo através da variável pxCurrentTCB e na linha (41) as interrupções são ativadas novamente. O processo inverso ocorre na rotina de restauração de contexto.

Figura 14 – Rotinas de troca de contexto para um processador da família ATmega

```
     #define portSAVE_CONTEXT()\
     asm volatile (\
(1)  "push r0                 \n\t"\
(2)  "in r0, __SREG__         \n\t"\
(3)  "cli                     \n\t"\
(4)  "push r0                 \n\t"\
(5)  "push r1                 \n\t"\
(6)  "clr r1                  \n\t"\
(7)  "push r2                 \n\t"\
(8)  "push r3                 \n\t"\
(9)  "push r4                 \n\t"\
(10) "push r5                 \n\t"\
(11) "push r6                 \n\t"\
(12) "push r7                 \n\t"\
(13) "push r8                 \n\t"\
(14) "push r9                 \n\t"\
(15) "push r10                \n\t"\
(16) "push r11                \n\t"\
(17) "push r12                \n\t"\
(18) "push r13                \n\t"\
(19) "push r14                \n\t"\
(20) "push r15                \n\t"\
(21) "push r16                \n\t"\
(22) "push r17                \n\t"\
(23) "push r18                \n\t"\
(24) "push r19                \n\t"\
(25) "push r20                \n\t"\
(26) "push r21                \n\t"\
(27) "push r22                \n\t"\
(28) "push r23                \n\t"\
(29) "push r24                \n\t"\
(30) "push r25                \n\t"\
(31) "push r26                \n\t"\
(32) "push r27                \n\t"\
(33) "push r28                \n\t"\
(34) "push r29                \n\t"\
(35) "push r30                \n\t"\
(36) "push r31                \n\t"\
(37) "in r26, __SP_L__        \n\t"\
(38) "in r27, __SP_H__        \n\t"\
(39) "sts pxCurrentTCB+1, r27 \n\t"\
(40) "sts pxCurrentTCB, r26   \n\t"\
(41) "sei    \n\t" : :);
```

```
     #define portRESTORE_CONTEXT()\
     asm volatile (\
(1)  "cli    \n\t"\
(2)  "out __SP_L__, %A0       \n\t"\
(3)  "out __SP_H__, %B0       \n\t"\
(4)  "pop r31                 \n\t"\
(5)  "pop r30                 \n\t"\
(6)  "pop r29                 \n\t"\
(7)  "pop r28                 \n\t"\
(8)  "pop r27                 \n\t"\
(9)  "pop r26                 \n\t"\
(10) "pop r25                 \n\t"\
(11) "pop r24                 \n\t"\
(12) "pop r23                 \n\t"\
(13) "pop r22                 \n\t"\
(14) "pop r21                 \n\t"\
(15) "pop r20                 \n\t"\
(16) "pop r19                 \n\t"\
(17) "pop r18                 \n\t"\
(18) "pop r17                 \n\t"\
(19) "pop r16                 \n\t"\
(20) "pop r15                 \n\t"\
(21) "pop r14                 \n\t"\
(22) "pop r13                 \n\t"\
(23) "pop r12                 \n\t"\
(24) "pop r11                 \n\t"\
(25) "pop r10                 \n\t"\
(26) "pop r9                  \n\t"\
(27) "pop r8                  \n\t"\
(28) "pop r7                  \n\t"\
(29) "pop r6                  \n\t"\
(30) "pop r5                  \n\t"\
(31) "pop r4                  \n\t"\
(32) "pop r3                  \n\t"\
(33) "pop r2                  \n\t"\
(34) "pop r1                  \n\t"\
(35) "pop r0                  \n\t"\
(36) "sei                     \n\t"\
(37) "out __SREG__, r0        \n\t"\
(38) "pop r0 \n\t": : "r" (pxCurrentTCB));
```

Uma vez definida as estruturas de salvamento de contexto, a Figura 15 apresenta a rotina utilizada na preempção das tarefas.

Figura 15 – Rotina de preempção de tarefas

```
void PreempExec (unsigned int a)

{
  portSAVE_CONTEXT()
  RunTask(a);
  portRESTORE_CONTEXT()
}
```

4.6. Execução de tarefa

Para a execução das tarefas, faz-se necessário uma função que execute a parte de código correspondente a cada tarefa, como pode ser visto na estrutura (2) da Figura 2. Esta função deve ter como argumento a posição da tarefa que deve ser executada no vetor da estrutura de armazenamento. Na inicialização de cada tarefa, foi armazenado nesta estrutura um ponteiro que contém a posição da memória na qual está armazenada a rotina responsável pela tarefa. Então, o sistema operacional deve ser capaz de selecionar qual tarefa deve ser executada através do argumento e buscar sua rotina de execução na memória através do ponteiro armazenado em sua estrutura. Esta rotina é apresentada na Figura 16

Figura 16 – Rotina de execução de tarefa

```
inline void RunTask(int a)
{

  taskexec = true;                          // Tarefa em execução
  (*_tasks[a].taskptr)(_tasks[a].arg);      // Rotina que chama (executa) a tarefa
   if (_tasks[a].per != 0)
   {
   numTask[a]--;                            // Diminui o numero de execuções da tarefa no TCP, se periódica
   }
  _tasks[a].flag = 0;                       // Baixa o flag de execução da tarefa
   taskexec = false;                        // Tarefa termina sua execução

}
```

A particularidade da rotina para execução da tarefa é sua chamada *inline.* Com este comando, garantimos que o compilador, no momento da tradução das diretivas em C para linguagem de máquina, insira o código da função no ponto do código onde ela deve ser chamada. Desta forma, garante-se um tempo de resposta mais

rápido quando chamamos esta função, em troca de um espaço maior utilizado na memória de programa.

4.7. Inicialização do RTOS

Como as rotinas de Organização, Tempo de Ciclo Principal, Número de Processos e Configuração do Timer devem ser chamadas antes do funcionamento do sistema operacional, define-se então, como mencionado na estrutura (3) da Figura 2, a rotina de inicialização deste sistema operacional. Ela é responsável pela chamada das funções mencionadas, devendo ser chamado na estrutura **voidsetup()** do Arduino. Ele é apresentado na Figura 17.

Figura 17 – Rotina de inicialização do Sistema Operacional

```
void OSSetup ()
{

  Ordena();                     // Chama rotina para ordenar tarefas por prioridade
  TCP = TempoCicloPrincipal() ; // Chama rotina para o calculo do tempo de ciclo principal
  NumeroProcessos();            // Calcula o número de execução de cada tarefa por ciclo
  ConfiguraTimer();             // Configura o Timer responsável pela verificação de tarefas e preempção

}
```

4.8. Execução do RTOS

Com a utilização das rotinas definidas anteriormente e da interrupção por timer responsável pela verificação e ativação dos flags das tarefas, o funcionamento do escalonador propriamente dito torna-se relativamente simples. No início de cada ciclo principal, ele deixa pronta para a execução todas as tarefas periódicas. Então, ele entra em um loop com a duração do ciclo principal, onde ele varre inumeráveis vezes, por ordem de prioridade, toda a lista de tarefas. Quando ele encontra uma para ser executada, ele chama a rotina de execução da tarefa, como mostrada anteriormente na Figura 16. Uma vez completa a execução, o escalonador volta a varrer a lista até o fim do ciclo principal.

5. ESTUDOS DE CASO

Este capítulo é dedicado à apresentação de demonstrações e estudos de caso envolvendo exemplos de programas desenvolvidos com o RTOS criado, com o objetivo final de comprovar que este sistema atende as premissas estabelecidas no capitulo dois e garante a execução de tarefas em tempo-real.

No total, serão apresentados cinco estudos de caso, divididos em seções. No início de cada uma delas, serão especificados os objetivos que deverão ser atendidos com aquele estudo em particular e a forma como o estudo será conduzido, bem como a resposta esperada de um sistema operacional na forma de um diagrama de Gant, para fins de comparação com a resposta experimental do sistema.

Comprovando-se o bom funcionamento do sistema, será conduzido um estudo do limite de resposta do mesmo, isto é, até que ordem de grandeza temporal este sistema atende os requisitos especificados. Para isto, será feito um estudo da frequência de interrupção para verificação de tarefas, determinando-se qual a maior frequência que pode ser utilizada sem comprometer o funcionamento do sistema. Após, serão diminuídos gradativamente os períodos e tempos de execução de tarefas, procurando-se atingir o limiar de seu funcionamento.

Para tais demonstrações, serão utilizadas algumas saídas digitais do Arduino, ligadas aos diferentes canais de um osciloscópio digital. Cada tarefa do sistema é associada a uma destas saídas, respeitando-se a ordem de canais do osciloscópio: a saída correspondente à tarefa um será conectada ao canal 1 e assim sucessivamente. Desta forma, para simular a execução de uma determinada tarefa, o sistema coloca sua respectiva saída em um estado HIGH (5V) no começo de sua execução e coloca-o num estado LOW (0V) no final de sua execução. A duração de cada tarefa é simulada através da função *delay()*, inclusa na biblioteca padrão do Arduino. Esta função é baseada em contagens de variável, e não baseada em unidades temporizadoras do processador. A quantidade de tempo especificada em seu argumento, em milissegundos, modela o tempo de computação da tarefa [3]. No evento de uma interrupção (seja ela interna ou externa) durante a execução desta função, ela interrompe sua contagem de variáveis e atende a rotina de interrupção. Quando o código retorna desta rotina, a contagem continua do instante de tempo em

que foi suspensa. Desta forma, a saída de uma tarefa preemptada ficará no estado HIGH por um intervalo de tempo igual a sua execução, especificada na função *delay(),* mais o tempo em que esta tarefa foi suspensa para execução de uma tarefa de maior prioridade. Com isto, pode-se perceber a preempção das tarefas de uma forma mais clara: quando o escalonador interrompe a execução de uma tarefa e troca de contexto, a saída desta respectiva tarefa preemptada continuada no estado HIGH até que esta tarefa seja completa após a retomada de seu contexto.

Para estes estudos, serão utilizadas, no máximo, três tarefas. Para cada estudo será utilizado um conjunto diferente de valores para período e duração, visando atender os objetivos especificados para cada ensaio. As rotinas para estas simulações são apresentadas na Figura 18.

Figura 18 – Rotinas das tarefas para as simulações

```
void task1(void *p)
{
  digitalWrite(5, HIGH);
  delay (Duracao1);
  digitalWrite(5, LOW);
}

void task2(void *p)
{
  digitalWrite(6, HIGH);
  delay (Duracao2);
  digitalWrite(6, LOW);
}

void task3(void *p)
{
  digitalWrite(7, HIGH);
  delay (Duracao3);
  digitalWrite(7, LOW);
}
```

5.1. Escalonamento temporal e cumprimento de deadlines

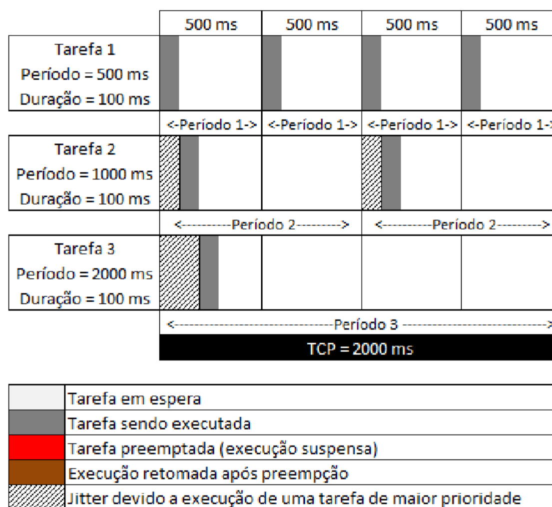
O objetivo deste primeiro estudo é a verificação do escalonamento principal em um sistema operacional que não necessita preempção, sendo responsável por escalonar temporalmente três tarefas de curta duração em um ciclo principal. Neste estudo, pode-se observar a hierarquia respeitada pelo escalonador na execução de tarefas, bem como o cumprimento de seus períodos e deadlines especificados. Para este estudo, as propriedades de cada tarefa estão apresentadas na Tabela 2.

Tabela 2 – Propriedades das tarefas para o primeiro estudo de caso

<table>
<tr><th>Tarefa</th><th>Período (ms)</th><th>Duração (ms)</th><th>Prioridade relativa</th><th>Execuções no TCP</th><th>TCP (ms)</th></tr>
<tr><td>1</td><td>500</td><td>100</td><td>1</td><td>4</td><td rowspan="3">2000</td></tr>
<tr><td>2</td><td>1000</td><td>100</td><td>2</td><td>2</td></tr>
<tr><td>3</td><td>2000</td><td>100</td><td>3</td><td>1</td></tr>
</table>

Escalonando-se temporalmente estas tarefas com suas respectivas propriedades, a resposta esperada deste sistema é apresentada na Figura 19.

Figura 19 – Resposta esperada para um escalonamento de três tarefas periódicas sem preempção



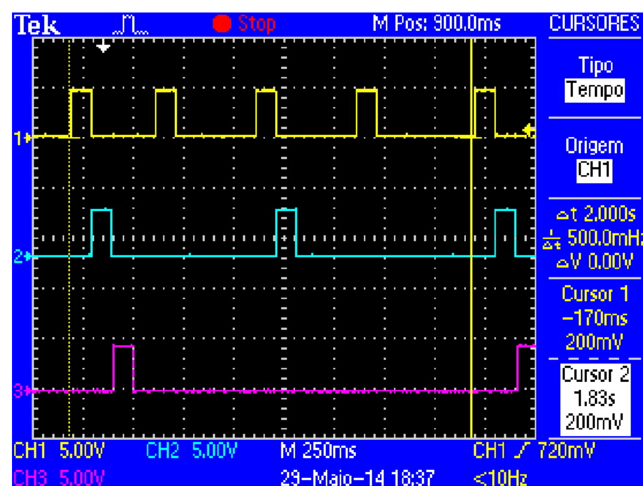
Então, declarando-se a rotina de cada tarefa como especificado na Figura 18, a estrutura principal do programa escrito na IDE do Arduino é apresentada na Figura 20.

Figura 20 – Rotina para escalonamento de três tarefas periódicas sem preempção

```
void setup()
{

  pinMode(5, OUTPUT);                     // Saída responsável pela Tarefa 1
  pinMode(6, OUTPUT);                     // Saída responsável pela Tarefa 2
  pinMode(7, OUTPUT);                     // Saída responsável pela Tarefa 3

  Duracao1 = 100;
  Duracao2 = 100;
  Duracao3 = 100;

  CreateTask(500, 1 ,task1, NULL);        // Criação das tarefas
  CreateTask(1000, 2 ,task2, NULL);
  CreateTask(2000, 3 ,task3, NULL);

  OSSetup();                              // Chama setup do RTOS

}

void loop()
{

   OSRun();                               // Executa o RTOS

}
```

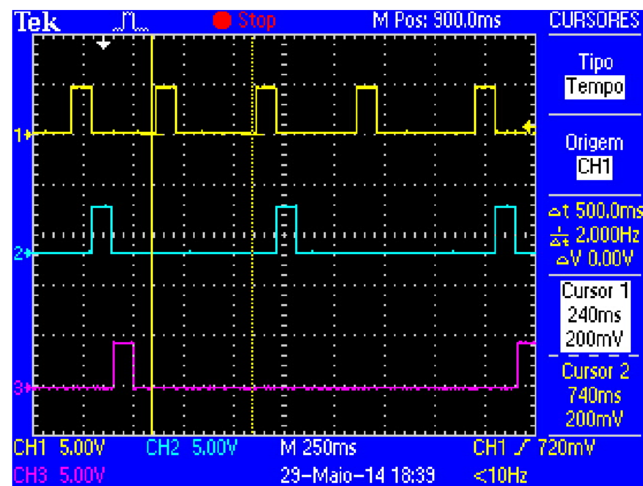
Conectando-se os canais do osciloscópio digital nas saídas do Arduino responsáveis por cada tarefa (pinos digitais 5, 6 e 7, correspondentes aos bits 5, 6 e 7 da porta D do microcontrolador), se obtém o gráfico da Figura 21. Através do posicionamento dos cursores nesta figura, pode-se perceber que a duração do ciclo principal corresponde à duração prevista e este sistema escalona hierarquicamente as tarefas por ordem de prioridade.

Figura 21 – Escalonamento do RTOS desenvolvido para três tarefas periódicas sem preempção



Analisando-se a tarefa de maior prioridade (canal 1), percebe-se, através da Figura 22, que seu período de execução é estritamente respeitado. Na execução das tarefas 2 e 3, existe um pequeno intervalo de tempo entre o começo do seu período e sua real execução, devido à execução da tarefa de maior prioridade.

Figura 22 – Período de execução da tarefa 1



Então, comparando-se o gráfico esperado para esta simulação (Figura 19) com o resultado obtido experimentalmente (Figura 21), pode-se concluir que o sistema desenvolvido é capaz de hierarquizar as tarefas e escaloná-las temporalmente segundo os critérios definidos no capítulo dois, para tarefas periódicas sem a necessidade de preempção.

5.2. Duas tarefas com preempção

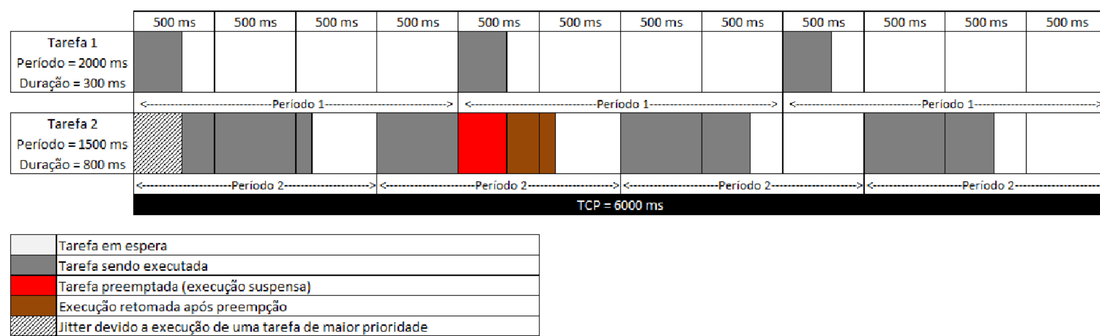
Sabendo que o sistema responde dentro do esperado na execução de um ciclo principal onde não existe a necessidade de preempção, o objetivo desta seção é de demonstrar que o sistema também responde dentro das premissas estabelecidas quando existe uma necessidade de preempção, utilizando-se como exemplo duas tarefas com durações diferentes. As propriedades das tarefas utilizadas estão listadas na Tabela 3.

Tabela 3 – Propriedades das tarefas para o segundo estudo de caso

Tarefa	Período (ms)	Duração (ms)	Prioridade relativa	Execuções no TCP	TCP (ms)
1	2000	300	1	3	6000
2	1500	800	2	4	

Para estas tarefas, a resposta esperada do sistema é representada pela Figura 23.

Figura 23 – Resposta esperada para um escalonamento de duas tarefas periódicas com preempção



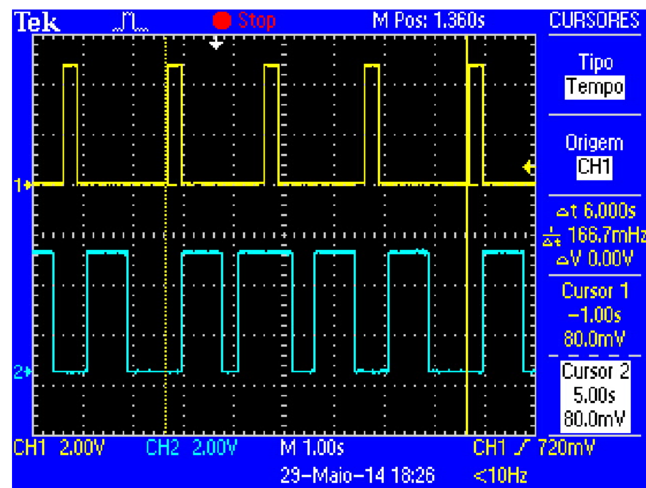
Com as rotinas das tarefas estabelecidas na Figura 18, o código da interface do Arduino que foi utilizado é representado na Figura 24.

Figura 24 – Rotina para escalonamento de duas tarefas periódicas com preempção

```
void setup()
{

  pinMode(5, OUTPUT);               // Saída responsável pela Tarefa 1
  pinMode(6, OUTPUT);               // Saída responsável pela Tarefa 2

  Duracao1 = 300;
  Duracao2 = 800;

  CreateTask(2000, 1 ,task1, NULL);   // Criação das tarefas
  CreateTask(1500, 2 ,task2, NULL);

  OSSetup();                        // Chama setup do RTOS

}

void loop()
{

   OSRun();                         // Executa o RTOS

}
```

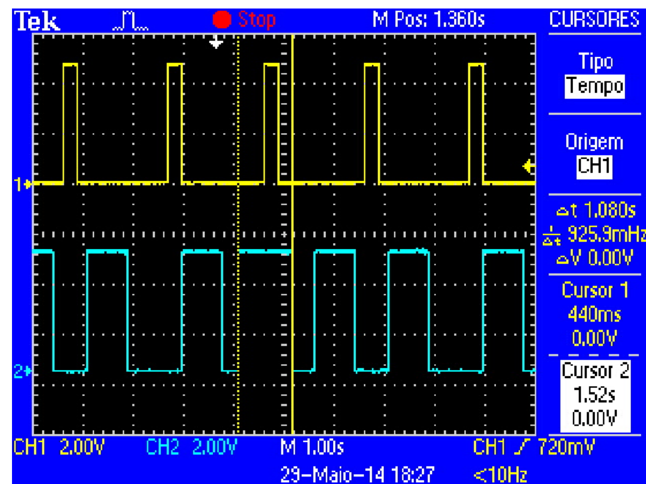
De forma similar à demonstração anterior, conectando-se os canais dos osciloscópios às saídas responsáveis por cada tarefa, se obtém a resposta representada pela Figura 25. Os cursores representam o ciclo principal deste sistema. Pode-se perceber que a duração em que a saída correspondente à tarefa 2 está no estado HIGH não é constante, visto que existe uma preempção da mesma para execução da tarefa 1.

Figura 25 – Escalonamento do RTOS desenvolvido para duas tarefas periódicas com preempção



Medindo-se a duração na qual esta saída fica em seu estado HIGH, durante sua segunda execução, obtém-se aproximadamente 1100 milissegundos (Figura 26). Isto corresponde ao tempo de execução da segunda tarefa mais o tempo de execução da primeira tarefa.

Figura 26 – Medição temporal da saída da segunda tarefa em sua segunda execução



Então, comparando-se a resposta experimental obtida com a resposta prevista pela Figura 23, pode-se afirmar que este sistema comporta-se dentro do esperado quando duas tarefas são escalonadas de modo que, em determinado momento, exista uma preempção e troca de contexto pelo sistema operacional.

5.3. Três tarefas com preempção

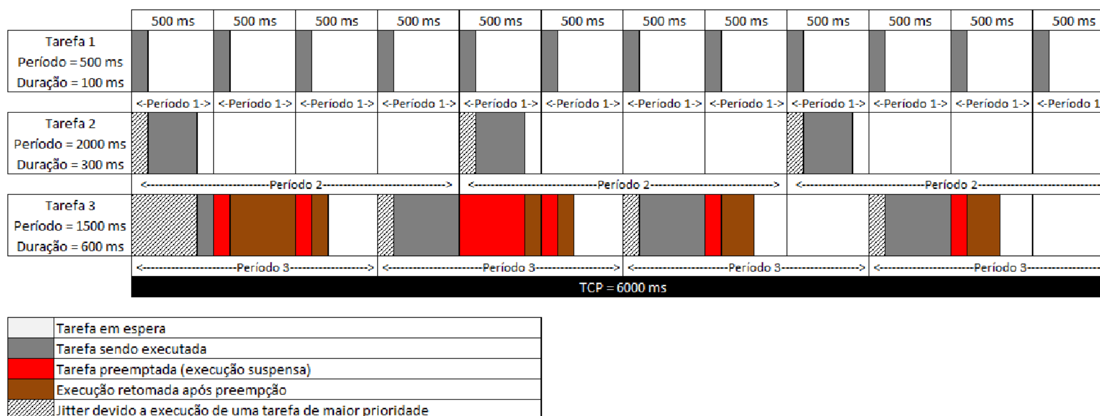
De forma similar à seção precedente, o objetivo deste estudo de caso é analisar o funcionamento do sistema operacional diante de um escalonamento que exige preempção, desta vez com três tarefas ao invés de duas. Suas propriedades estão listadas na Tabela 4.

Tabela 4 – Propriedades das tarefas para o terceiro estudo de caso

Tarefa	Período (ms)	Duração (ms)	Prioridade relativa	Execuções no TCP	TCP (ms)
1	500	100	1	12	6000
2	2000	300	2	3	
3	1500	600	3	4	

Com este escalonamento e duração de tarefas, a resposta esperada é apresentada na Figura 27.

Figura 27 – Resposta esperada para um escalonamento de três tarefas periódicas com preempção



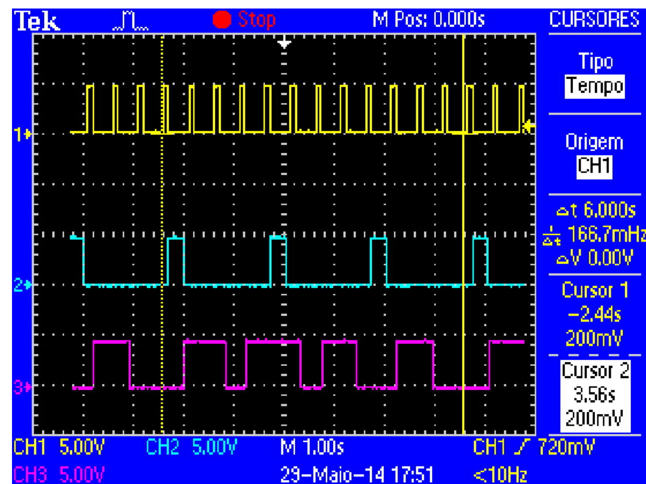
As declarações nas estruturas do Arduino para o funcionamento desta simulação são dadas pela Figura 28.

Figura 28 – Rotina para escalonamento de três tarefas periódicas com preempção

```
void setup()
{

  pinMode(5, OUTPUT);                     // Saída responsável pela Tarefa 1
  pinMode(6, OUTPUT);                     // Saída responsável pela Tarefa 2
  pinMode(7, OUTPUT);                     // Saída responsável pela Tarefa 3

  Duracao1 = 100;
  Duracao2 = 300;
  Duracao3 = 600;

  CreateTask(500, 1 ,task1, NULL);        // Criação das tarefas
  CreateTask(2000, 2 ,task2, NULL);
  CreateTask(1500, 3 ,task3, NULL);

  OSSetup();                              // Chama setup do RTOS

}

void loop()
{

   OSRun();                               // Executa o RTOS

}
```

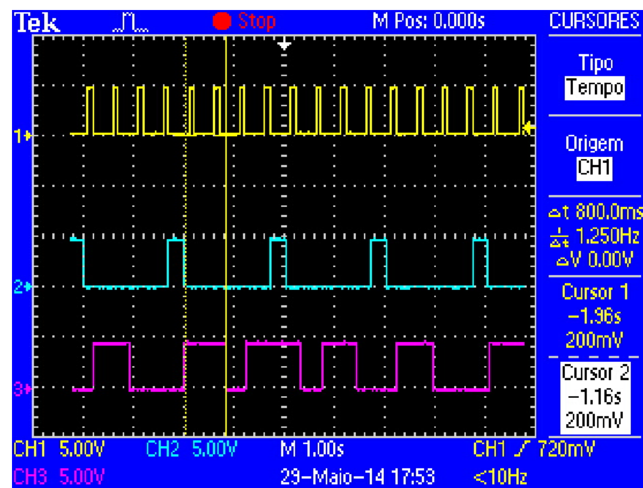
A resposta obtida no osciloscópio através da conexão das saídas digitais do Arduino nos seus canais é dada pela Figura 29, onde os cursores representam o ciclo de escalonamento principal.

Figura 29 – Escalonamento do RTOS desenvolvido para três tarefas periódicas com preempção



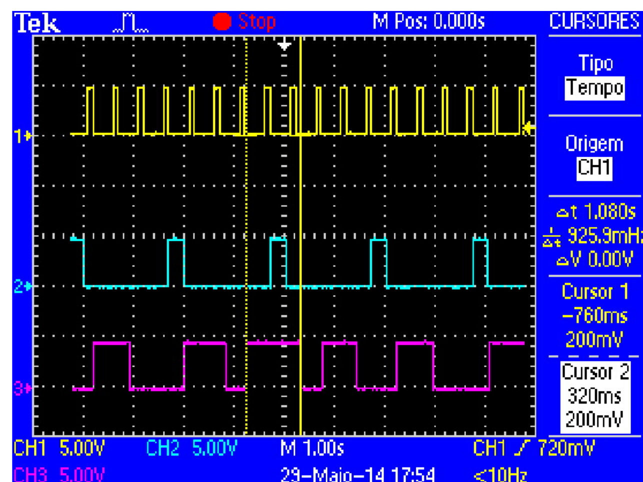
Para verificar se a preempção da terceira tarefa ocorre como o esperado, mede-se a duração que sua respectiva saída fica no estado HIGH nas diferentes execuções. Sabe-se que esta duração deve ser igual à duração original desta tarefa mais a duração das tarefas que foram executadas através de uma troca de contexto. Na sua primeira execução, representada pelos cursores da Figura 30, percebe-se que sua saída fica em HIGH por 800 ms, que é igual ao tempo de execução desta tarefa (600 ms) mais as duas execuções da tarefa 1 (2 x 100 ms).

Figura 30 – Medição temporal da saída da terceira tarefa em sua primeira execução



De forma similar, na Figura 31, sua saída fica no estado HIGH por aproximadamente 1100 ms, correspondendo aos 600 ms da execução da tarefa 3 mais os 300 ms de execução da tarefa dois e mais duas vezes os 100 ms da tarefa 1.

Figura 31 – Medição temporal da saída da terceira tarefa em sua segunda execução



Portanto, comparando-se com o diagrama esperado, pode-se afirmar que este sistema atende as premissas de preempção e troca de contexto durante um ciclo de escalonamento principal, como determinado no capítulo dois.

5.4. Duas tarefas periódicas e um processo esporádico, sem preempção

O objetivo deste ensaio é verificar o comportamento do sistema na presença de um processo esporádico juntamente com tarefas periódicas. Deve ser possível verificar neste estudo que este processo será colocado na fila de processos prontos pelo acionamento de uma interrupção externa, sendo sujeito à mesma hierarquia que os processos periódicos já escalonados pelo processador. Na tabela, estão presentes as propriedades das tarefas que farão parte deste estudo.

Tabela 5 – Propriedades das tarefas para o quarto estudo de caso

<table>
<tr><th>Tarefa</th><th>Período (ms)</th><th>Duração (ms)</th><th>Prioridade relativa</th><th>Execuções no TCP</th><th>TCP (ms)</th></tr>
<tr><td>1</td><td>1000</td><td>400</td><td>1</td><td>3</td><td rowspan="3">3000</td></tr>
<tr><td>2</td><td>Esporádico</td><td>100</td><td>2</td><td>-</td></tr>
<tr><td>3</td><td>1500</td><td>400</td><td>3</td><td>2</td></tr>
</table>

Tem-se por objetivo ativar a interrupção externa em algum instante de tempo entre 2000 e 2400 milissegundos após o início do ciclo principal. Desta forma, deve ser possível observar que o processo foi colocado na fila durante a terceira execução da primeira tarefa, mas só foi executado após o término da mesma, devido à sua menor prioridade. Com este escalonamento de tarefas e ativação da interrupção externa, a resposta esperada é dada pela Figura 32.

A configuração das estruturas do Arduino para atender as especificações da Tabela é dada pela Figura 33.

Utilizando então os respectivos canais do osciloscópio para as tarefas, obtém-se a Figura 34, onde os cursores delimitam o ciclo principal.

Figura 32 – Resposta esperada para um escalonamento de duas tarefas periódicas e um processo esporádico, sem preempção

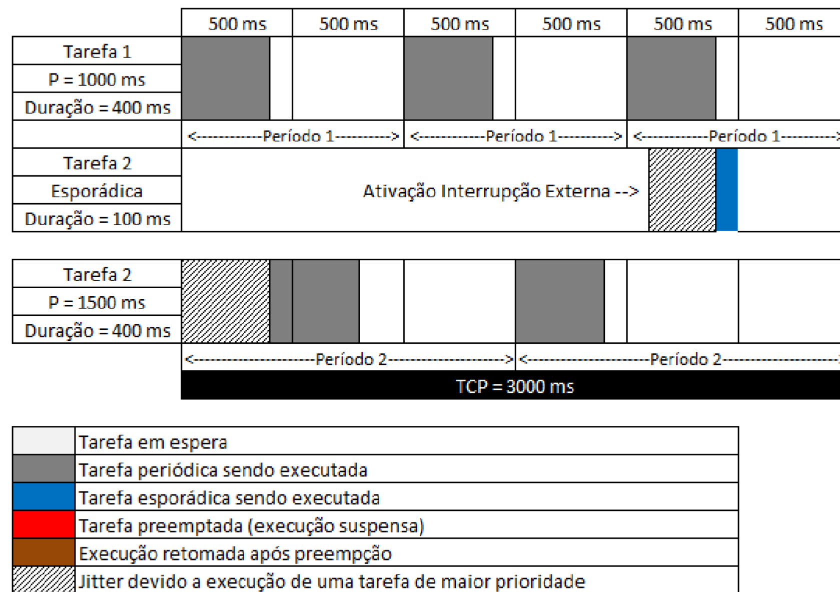
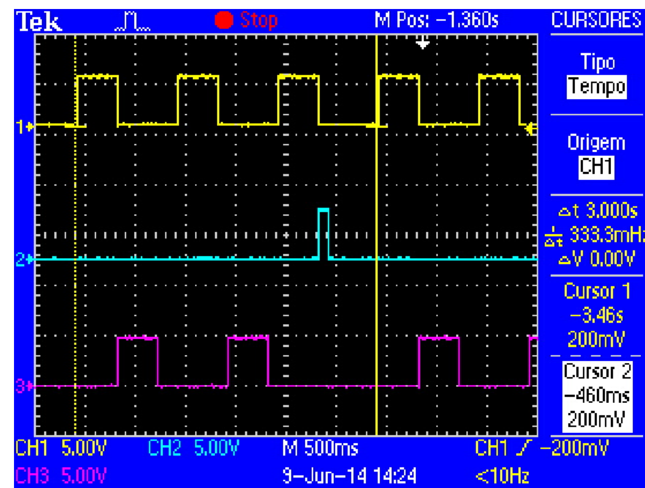


Figura 33 – Rotina para escalonamento de duas tarefas periódicas e um processo esporádico, sem preempção

```
void setup()
{

  pinMode(5, OUTPUT);                   // Saída responsável pela Tarefa 1
  pinMode(6, OUTPUT);                   // Saída responsável pela Tarefa 2
  pinMode(7, OUTPUT);                   // Saída responsável pela Tarefa 3

  Duracao1 = 400;
  Duracao2 = 100;
  Duracao3 = 400;

  CreateTask(1000, 1 ,task1, NULL);      // Criação das tarefas
  CreateTask(0, 2 ,task2, NULL);
  CreateTask(1500, 3 ,task3, NULL);

  OSSetup();                            // Chama setup do RTOS

}

void loop()
{

   OSRun();                             // Executa o RTOS

}
```

Figura 34 – Escalonamento do RTOS desenvolvido para duas tarefas periódicas e um processo esporádico, sem preempção



Pode-se afirmar, então, que este sistema atende às premissas definidas anteriormente para um escalonamento de processos periódicos com ativação de um processo esporádico por interrupção externa.

5.5. Duas tarefas periódicas e um processo esporádico, com encadeamento de preempções

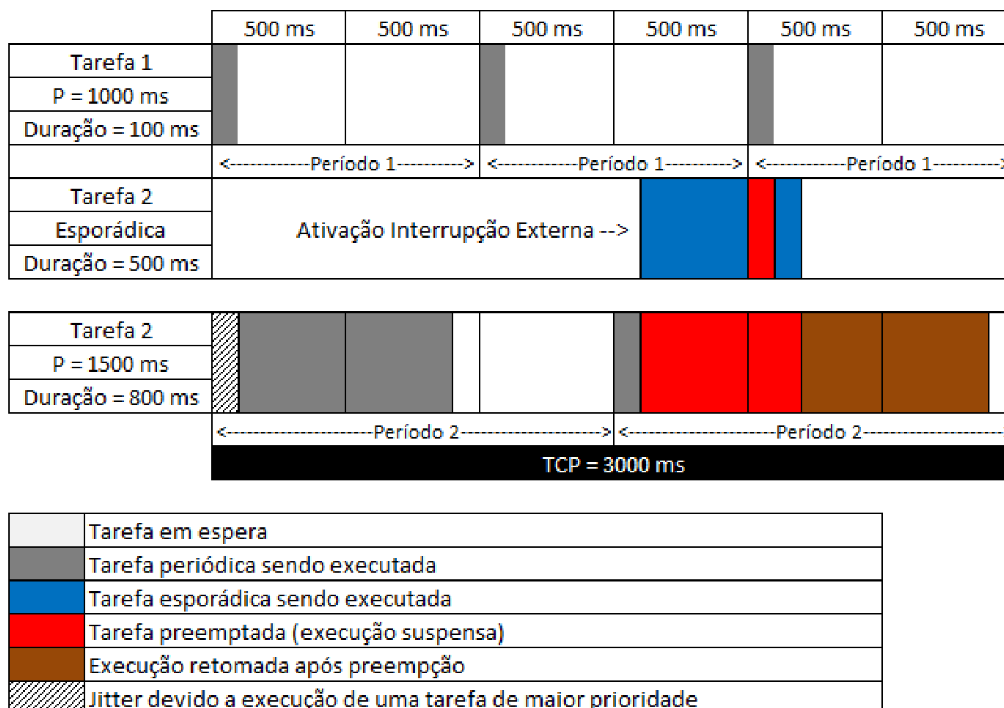
O objetivo desta simulação é comprovar o funcionamento do sistema na execução de duas tarefas periódicas e um processo esporádico, escalonados de tal forma que exista tanto uma preempção de uma tarefa periódica pelo processo esporádico como do processo esporádico por uma tarefa periódica. Juntamente com isto, tem-se por objetivo verificar o comportamento do sistema quando mais de uma rotina de troca de contexto é encadeada. Para isto, foram definidas as propriedades dos processos listados na Tabela 6.

Tabela 6 – Propriedades das tarefas para o quinto estudo de caso

Tarefa	Período (ms)	Duração (ms)	Prioridade relativa	Execuções no TCP	TCP (ms)
1	1000	100	1	3	3000
2	-	500	2	-	
3	1500	800	3	2	

Escalonando-se temporalmente estas tarefas e ativando a interrupção externa em algum momento no intervalo de tempo entre 2000 e 2500 ms após o inicio do ciclo principal, a resposta esperada do sistema é dada pela Figura 35. Neste estudo, percebe-se que, se a interrupção for ativada durante a execução de uma tarefa de menor prioridade, ocorre uma troca de contexto e o sistema passa a executar este processo esporádico. Antes da finalização deste processo, porém, o sistema deixa a tarefa 1 pronta para execução, ocorrendo uma preempção do processo esporádico para execução desta tarefa de maior prioridade. Desta forma, duas trocas de contexto são encadeadas. A restauração do contexto ocorre, então, de forma sequencial e progressiva: inicialmente, apenas o contexto do processo esporádico é restaurado, terminando assim sua execução e restaurando o contexto da tarefa 3, terminando também sua execução.

Figura 35 – Resposta esperada para um escalonamento de duas tarefas periódicas e um processo esporádico, com encadeamento de preempções



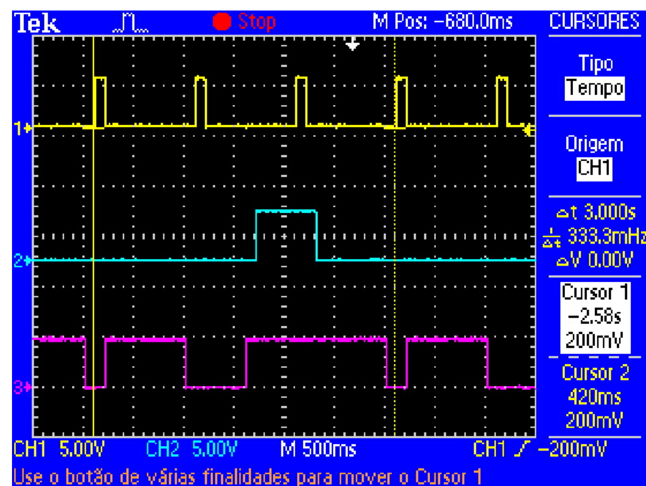
Utilizando as propriedades da Tabela, define-se, na interface do Arduino, o funcionamento do sistema, dado pela Figura 36.

Figura 36 – Rotina para escalonamento de duas tarefas periódicas e um processo esporádico, com encadeamento de preempções

```
void setup()
{

  pinMode(5, OUTPUT);                    // Saída responsável pela Tarefa 1
  pinMode(6, OUTPUT);                    // Saída responsável pela Tarefa 2
  pinMode(7, OUTPUT);                    // Saída responsável pela Tarefa 3

  Duracao1 = 100;
  Duracao2 = 500;
  Duracao3 = 800;

  CreateTask(1000, 1 ,task1, NULL);       // Criação das tarefas
  CreateTask(0, 2 ,task2, NULL);
  CreateTask(1500, 3 ,task3, NULL);

  OSSetup();                             // Chama setup do RTOS

}

void loop()
{

   OSRun();                              // Executa o RTOS

}
```

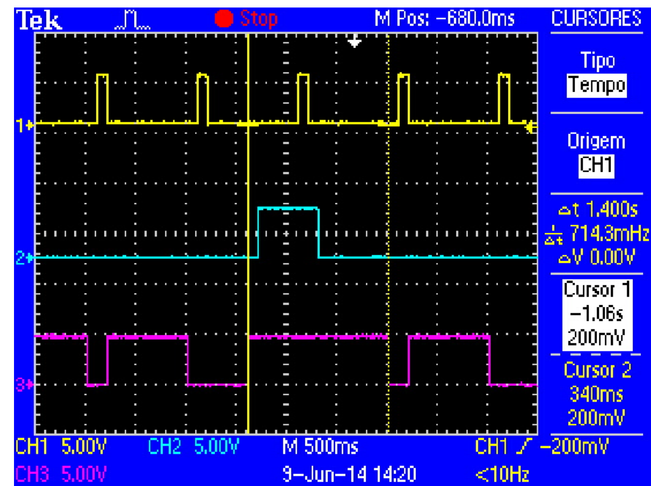
Então, utilizando-se as saídas digitais conectadas ao osciloscópio e ativando-se a interrupção externa no intervalo de tempo definido, obtém-se a Figura 37.

Figura 37 – Escalonamento do RTOS desenvolvido para duas tarefas periódicas e um processo esporádico, com encadeamento de preempções



Medindo-se a duração em que a saída correspondente à tarefa 3 fica ativada (Figura 38), pode-se verificar que ela corresponde à duração da tarefa 3 (800 ms) mais a duração da tarefa 2 (500 ms) mais a duração da tarefa 1(100 ms), comprovando então o funcionamento do sistema na preempção associada à processos esporádicos e o encadeamento de preempções.

Figura 38 – Medição temporal da saída da terceira tarefa em sua primeira execução



6. LIMITAÇÕES DO SISTEMA

O objetivo deste capítulo é descrever as principais limitações do sistema, definindo o escopo de aplicações para as quais este sistema pode ser utilizado.

6.1. Limitação na frequência de verificação de tarefa

Até então, os testes realizados utilizaram períodos e tempos de execução da ordem de centenas de milissegundos, com a interrupção temporal para verificação de tarefas não sendo superior a 1 kHz.

A forma como os timers são configurados em processadores da família ATMega não permite a utilização de um espectro de frequências continuas: apenas um número finito de frequências de interrupção pode ser utilizado. Testando-se estas diferentes frequências através da interrupção temporal, dadas pela configuração do timer1 do Arduino, foi determinado que a maior frequência que garante o bom funcionamento deste sistema é de 7,8 kHz. Acima desta frequência, a interrupção por timer compromete o funcionamento deste RTOS, pois o intervalo entre duas interrupções consecutivas torna-se da mesma ordem de grandeza do que a execução de sua rotina de atendimento.

6.2. Limitação no período dos processos utilizados

A fim de testar o limiar temporal abaixo do qual o sistema não pode funcionar, foram utilizadas duas tarefas periódicas cujos períodos e tempos de execução foram diminuídos gradativamente, verificando se a resposta do sistema possuía um comportamento adequado. Experimentalmente, este limiar foi obtido na ordem da unidade do milissegundo. As rotinas intrínsecas ao sistema (verificação de tarefa, acionamento de flag, varredura de lista, entre outras) possuem um tempo de execução desta ordem de grandeza, não sendo possível garantir o funcionamento deste sistema para tarefas que possuem um período inferior a 1 milissegundo.

6.3. Limitação em número de tarefas

A limitação do número de tarefas que este sistema pode processar é dada pelo espaço em memória do processador. As diferentes plataformas Arduino possuem diferentes processadores da família ATMega, cujos tamanhos de memórias podem ser acessadas através de [5]. Existe um compromisso entre o número de tarefas executadas em um processador e o tamanho de suas rotinas de execução. Por exemplo, podem-se utilizar diversas tarefas cujas rotinas ocupem pouco espaço na memória, ou poucas tarefas cujas rotinas são extensas na memória do processador. É responsabilidade do usuário garantir que o limite de memória do processador não seja excedido pelas tarefas em conjunto com suas rotinas.

6.4. Limites de escalonamento

Deve existir um compromisso da parte do usuário na declaração de tarefas, seus períodos e seu tempo de execução. Por exemplo, este sistema não funcionará corretamente se forem utilizadas tarefas que não podem ser escalonadas temporalmente devido ao seu tempo de execução. Conhecendo-se seus períodos e tempos de execução, existem diversos algoritmos que podem ser utilizados para verificar se as tarefas podem ser escalonadas ou não. O mais comum deles é o Escalonamento Taxa Monotônica (RM, do inglês *Rate Monotonic).* Este algoritmo é dito ótimo entre os escalonamentos preemptivos de prioridade fixa, quando as prioridades forem diretamente proporcionais às suas frequências de execução (tarefa com maior frequência terá a maior prioridade). Ou seja, nenhum outro algoritmo da mesma classe pode escalonar um conjunto de tarefas que não seja escalonável pelo RM [7].

.

7. CONCLUSÕES

O sistema desenvolvido apresentou bom desempenho nas simulações realizadas, sendo capaz de atender todos os requisitos definidos baseados em aplicações de sistemas embarcados. Este sistema garante uma resposta adequada para sistemas que possuem tarefas cujos períodos não são inferiores à unidade do milissegundo, sendo suficiente para a maioria das aplicações embarcadas desta plataforma. Sua limitação em tarefas é dada pelo espaço em memória do processador, sendo um compromisso entre número de tarefas e o espaço em memória ocupado por suas rotinas.

Também, deve existir um compromisso entre o usuário e o escalonador, de modo que exista um escalonamento possível para as tarefas declaradas e seus períodos de execução.

Durante os testes deste sistema, não foram encontrados problemas na utilização de funções das bibliotecas do Arduino. Porém, estas não foram testadas de forma completa e exaustiva, podendo, em certos casos, causar problemas na execução deste sistema. Ainda, deve-se tomar cuidado com o gerenciamento de interrupções: se estas forem desabilitadas por alguma rotina ou procedimento, não será possível ao sistema verificar se existem tarefas prontas para execução e, consequentemente, poderá acarretar na falha em seu funcionamento.

REFERÊNCIAS BIBLIOGRÁFICAS

1. MARWEDEL P. ***Embedded System Design: Embedded Systems Foundation of Cyber-Physical Systems*.** 2011
2. ARDUINO, ***Reference website*.** <http://arduino.cc/>. Acessado em 12 de abril de 2014
3. SHAW A. C. **Sistemas e Software de Tempo Real.** 2003
4. KNUTH D., ***The Art of Computer Programming, Volume 3: Sorting and Searching***. 1997
5. ATMEL, ***8-bit Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash datasheet*.** Disponível em <http://www.atmel.com/ Images/doc8161.pdf>. Acessado em 12 de abril de 2014
6. FreeRTOS. **The AVR Context.** Disponível em <http://www.freertos.org/ implementation/a00016.html>. Acessado em 03 de maio de 2014.
7. LIU C.L., LAYLAND W. ***Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*.** 1973
8. SILBERSCHATZ A., GALVIN P. B., GAGNE G. ***Operating System Concepts***. 2002
9. BUTTAZZO G. C. ***Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*.** 2011

ANEXO A – CÓDIGO COMPLETO DO RTOS

```
#include <Arduino.h>
#include <avr/interrupt.h>
#include <stdlib.h>

// Definição do numero de tarefas a serem executadas
#define  TASKS  2

// Macros para salvar e restaurar o contexto
#define portSAVE_CONTEXT()\
asm volatile (\
"push r0        \n\t"\
"in r0, __SREG__        \n\t"\
"cli    \n\t"\
"push r0        \n\t"\
"push r1        \n\t"\
"clr r1   \n\t"\
"push r2        \n\t"\
"push r3        \n\t"\
"push r4        \n\t"\
"push r5        \n\t"\
"push r6        \n\t"\
"push r7        \n\t"\
"push r8        \n\t"\
"push r9        \n\t"\
"push r10       \n\t"\
"push r11       \n\t"\
"push r12       \n\t"\
"push r13       \n\t"\
"push r14       \n\t"\
"push r15       \n\t"\
"push r16       \n\t"\
"push r17       \n\t"\
"push r18       \n\t"\
"push r19       \n\t"\
"push r20       \n\t"\
"push r21       \n\t"\
"push r22       \n\t"\
"push r23       \n\t"\
"push r24       \n\t"\
"push r25       \n\t"\
"push r26       \n\t"\
"push r27       \n\t"\
"push r28       \n\t"\
"push r29       \n\t"\
"push r30       \n\t"\
"push r31       \n\t"\
"in r26, __SP_L__       \n\t"\
"in r27, __SP_H__       \n\t"\
"sts pxCurrentTCB+1, r27        \n\t"\
"sts pxCurrentTCB, r26 \n\t"\
"sei    \n\t" :
:
);
```

```
#define portRESTORE_CONTEXT()\
asm volatile (\
"cli        \n\t"\
"out __SP_L__, %A0    \n\t"\
"out __SP_H__, %B0    \n\t"\
"pop r31          \n\t"\
"pop r30          \n\t"\
"pop r29          \n\t"\
"pop r28          \n\t"\
"pop r27          \n\t"\
"pop r26          \n\t"\
"pop r25          \n\t"\
"pop r24          \n\t"\
"pop r23          \n\t"\
"pop r22          \n\t"\
"pop r21          \n\t"\
"pop r20          \n\t"\
"pop r19          \n\t"\
"pop r18          \n\t"\
"pop r17          \n\t"\
"pop r16          \n\t"\
"pop r15          \n\t"\
"pop r14          \n\t"\
"pop r13          \n\t"\
"pop r12          \n\t"\
"pop r11          \n\t"\
"pop r10          \n\t"\
"pop r9 \n\t"\
"pop r8 \n\t"\
"pop r7 \n\t"\
"pop r6 \n\t"\
"pop r5 \n\t"\
"pop r4 \n\t"\
"pop r3 \n\t"\
"pop r2 \n\t"\
"pop r1 \n\t"\
"pop r0 \n\t"\
"sei        \n\t"\
"out __SREG__, r0\n\t"\
"pop r0 \n\t":
:
"r" (pxCurrentTCB));

typedef struct tc
{
  uint8_t pid;                  // ID do processo
  uint8_t prio;                 // Prioridade única decrescente, sendo 1 a mais alta
  unsigned long per;            // Período, em ms
  unsigned int flag;            // Flag de execução: 1 = Tarefa pronta para ser executada;
                                //                   0 = Tarefa não está pronta para execução
  unsigned int inter;           // Bit que define se a tarefa é periódica (0) ou ativada por interrupção (1)
  unsigned int interId;         // ID que define a posição da tarefa de interrupção da estrutura
  void (*taskptr)(void*);       // Ponteiro que aponta para a rotina de execução da tarefa
  void *arg;                    // Argumento passado para a tarefa na sua inicialização
}
TLM;                            // TLM = Task List Manager
TLM _tasks[TASKS];              // Inicialização da estrutura na forma de um vetor
```

```
// Declaração das variáveis globais
unsigned char _procCount=0;
unsigned long TCP;
unsigned int _running = 0;
unsigned long numExec[TASKS];
unsigned long numTask[TASKS];
unsigned long start = 0;
unsigned long ticks = 0;
unsigned int numProc = 0;
unsigned int numInter = 0;
unsigned long inttimer = 0;
unsigned long intCount[2] = 0;
boolean codeInt = false;
boolean taskexec = false;
int timer1_counter;
unsigned long pxCurrentTCB;
unsigned int Duracao1, Duracao2, Duracao3;

// Rotina de criação de tarefa
void CreateTask(unsigned long period, int priority ,void (*rptr)(void *), void *arg)
{
  _tasks[_procCount].per = period;
  _tasks[_procCount].prio = priority;
  _tasks[_procCount].taskptr = rptr;
  _tasks[_procCount].pid = _procCount;
  _tasks[_procCount].arg = arg;
  if (period == 0)
  {
    numInter[intCount]=_procCount;
    codeInt = true;
    _tasks[_procCount].inter = 1;
    _tasks[_procCount].interId = numInter[intCount];
    intCount++;
  }
  else
  {
    _tasks[_procCount].inter = 0;
    _tasks[_procCount].interId = NULL;
  }

  _procCount++;

}

// Rotina de ordenamento de tarefas
void Ordena()
{
  TLM prov;
  for (int i = 0; i< TASKS-1; i++)
  {
    for (int j=i+1; j< TASKS; j++)
    {
      if (_tasks[j].prio < _tasks[i].prio)
      {
        prov.per = _tasks[i].per;
        prov.prio = _tasks[i].prio;
        prov.taskptr = _tasks[i].taskptr;
        prov.pid =   _tasks[i].pid;
        prov.arg = _tasks[i].arg;
        prov.inter = _tasks[i].inter;
```

```
        prov.interId = _tasks[i].interId;

        _tasks[i].per = _tasks[j].per ;
        _tasks[i].prio = _tasks[j].prio;
        _tasks[i].taskptr = _tasks[j].taskptr;
        _tasks[i].pid = _tasks[j].pid;
        _tasks[i].arg = _tasks[j].arg;
        _tasks[i].inter = _tasks[j].inter;
        _tasks[i].interId = _tasks[j].interId;

        _tasks[j].per = prov.per ;
        _tasks[j].prio = prov.prio;
        _tasks[j].taskptr = prov.taskptr;
        _tasks[j].pid = prov.pid;
        _tasks[j].arg = prov.arg;
        _tasks[j].inter = prov.inter;
        _tasks[j].interId = prov.interId;
      }
    }
  }
}

// Funções para calculo do maximo divisor comum e minimo multiplo comum, necessarias para o TCP
unsigned long mdc(unsigned long a, unsigned long b)
{
  unsigned long t;
  while (b!= 0){
    t = b;
    b = a % b;
    a = t;
  }
  return a;
}

unsigned long mmc(unsigned long a, unsigned long b)
{
  unsigned long t;
  t = a*b/mdc(a,b);
  return t;
}

// Calculo do TCP
unsigned long TempoCicloPrincipal ()
{
  unsigned long a = 1;
  for (int i =0; i<TASKS; i++)
  {
    if (_tasks[i].per != 0)
    {
      a = mmc(a, _tasks[i].per);
    }
  }
  return a;
}
```

```
// Calculo numero de execuçoes em ciclo e numero de processos
void NumeroProcessos()
{

  for (int i =0; i<TASKS; i++)
  {
    if (_tasks[i].per !=0)            // Não aplicável à tarefas não periódicas
    {
      numExec[i] = TCP/_tasks[i].per;    // Define o numero de execuções de uma tarefa dentro do ciclo
principal
      numProc = numProc + numExec[i];    // Numero de execuções totais dentro do ciclo principal
    }
  }
  if (codeInt==true)
  {
    numExec[numInter]=255;
  }
}

// Configuração de timer
void ConfiguraTimer()
{

  noInterrupts();
  TCCR1A = 0;                            // Modo de operação normal
  TCCR1B = 0;                            // Timer parado

  timer1_counter = 65534;                 // Preload timer na frequência desejada: 65536-
                                          //(16MHz/1024/7,8KHz)

  TCNT1 = timer1_counter;                 // Preload timer
  TCCR1B |= (1 << CS12) | (1 << CS10);     // Definição do prescaler = 1024  e inicio do timer
  TIMSK1 |= (1 << TOIE1);                 // Habilita interrupção com overflow do timer
  interrupts();

}

// Execução de tarefa
inline void RunTask(int a)
{

  taskexec = true;                      // Tarefa em execução
  (*_tasks[a].taskptr)(_tasks[a].arg);    // Rotina que chama (executa) a tarefa
  if (_tasks[a].per != 0)
  {
    numTask[a]--;                        // Diminui o numero de execuções da tarefa no TCP, se periódica
  }
  _tasks[a].flag = 0;                   // Baixa o flag de execução da tarefa
  taskexec = false;                     // Tarefa termina sua execução

}


// Verificação de tarefa
void VerificaTarefa()
{
  for (int i = 0; i< TASKS; i++)
  {
    ticks = millis();
```

```
    if ((_tasks[i].flag == 0) && (ticks > (start + ((_tasks[i].per)*(numExec[i] - numTask[i])))) &&
(_tasks[i].inter == 0))
    {
      _tasks[i].flag = 1;
      if ((taskexec == true) && (i < _running))
      {
        PreempExec (i);
      }
    }
  }

}

// Rotina de preempção
void PreempExec (unsigned int a)

{
  portSAVE_CONTEXT()
    RunTask(a);
  portRESTORE_CONTEXT()
  }

   // Rotina de atendimento à interrupção externa
  void InterruptRoutine0 ()
  {

    _tasks[numInter[0]].flag = 1;
    _running = 0;
    delay (100);

  }

void InterruptRoutine1 ()
  {

    _tasks[numInter[1]].flag = 1;
    _running = 0;
    delay (100);

  }


// Rotina de atendimento à interrupção por timer
ISR(TIMER1_OVF_vect)
{
  TCNT1 = timer1_counter;
  VerificaTarefa();
}

void OSSetup ()
{

  Ordena();                      // Chama rotina para ordenar tarefas por prioridade
  TCP = TempoCicloPrincipal() ;  // Chama rotina para o calculo do tempo de ciclo principal
  NumeroProcessos();             // Calcula o número de execução de cada tarefa por ciclo
  ConfiguraTimer();              // Configura o Timer responsável pela verificação de tarefas e preempção
  if (intCount==1)           // Se existem processos esporádicos, ativa interrupções externas
  {
    attachInterrupt (0, InterruptRoutine0, LOW);
  }
```

```
  else if (intCount==2)
  {
    attachInterrupt (0, InterruptRoutine0, LOW);
    attachInterrupt (1, InterruptRoutine0, LOW);
  }
}

void OSRun()
{

  for (int i=0 ; i<TASKS ; i++)
  {
    numTask[i] = numExec[i];
    if (_tasks[i].per != 0)
    {
      _tasks[i].flag = 1;       // Seta o flag das tarefas periódicas para 1 no começo do ciclo
    }
  }

  _running = 0;
  start = millis();
  ticks = millis();

  int count = numProc;

  while (ticks < (start + TCP))
  {
    if ((numTask[_running] > 0) && (_tasks[_running].flag == 1))
    {
      RunTask(_running);
      if (_tasks[_running].inter==0)
      {
        count--;
      }
      _running++;
    }
    else
    {
      _running++;
    }

    ticks = millis();

    if (_running == TASKS && count != 0)
    {
      _running = 0;
    }

    if (count == 0 )
    {
      ticks = millis();
    }

  }

}
```