Uma avaliação do sistema operacional FreeRTOS na plataforma Arduino Uno

Marco Aurélio Spohn
Universidade Federal da Fronteira Sul
Chapecó, SC - Brasil
marco.spohn@uffs.edu.br

Felipe Chabatura Neto
Universidade Federal da Fronteira Sul
Chapecó, SC - Brasil
felipechabat@gmail.com

Leonardo Tironi Fassini
Universidade Federal da Fronteira Sul
Chapecó, SC - Brasil
leehtironi@gmail.com

ABSTRACT

Um sistema operacional em tempo real (*Real Time Operating System*, RTOS) oferece mecanismos à execução de tarefas com restrições de tempo, entitulados requisitos de tempo real da aplicação. A execução de aplicações com características de tempo real em ambientes com limitações de recursos de *hardware* (*e.g.*, processamento e memória) torna o desenvolvimento dessas aplicações mais desafiador ou, até mesmo, impossível. Nesse trabalho, avalia-se o FreeRTOS, um sistema operacional de tempo real voltado a sistemas embarcados, de código aberto e livre de licença comercial. Todavia, objetivou-se avaliar o sistema em uma plataforma de *hardware* minimalista, o Arduino UNO, muito empregado na prototipação de sistemas embarcados. Como principal resultado, além de propiciar melhor entendimento acerca dos limites de um RTOS em uma plataforma limitada em termos de recursos de processamento e de memória, prevê-se o perfil das aplicações que eventualmente possam ser desenvolvidas no conjunto avaliado.

CCS Concepts

•Computer systems organization → Real-time operating systems; •Software and its engineering → Requirements analysis; Software design engineering;

Keywords

Sistemas de tempo real; FreeRTOS; Arduino Uno

1. INTRODUÇÃO

Um sistema operacional em tempo real (*Real Time Operating System*, RTOS) oferece mecanismos que, quando utilizados apropriadamente, garantem que um conjunto de tarefas sejam concluídas dentro de certos limites de tempo, entitulados requisitos de tempo real da aplicação [5]. Ou seja, a corretude do sistema não depende apenas dos resultados que ele produz, mas do tempo em que esses resultados são produzidos. Todos os prazos estipulados devem ser cumpridos, independente das circunstâncias, o que atribui aos RTOS a característica de previsibilidade. Garantir essa previsibilidade pode inclusive tornar o sistema mais lento [8].

Um modelo de tarefas de tempo real apresenta características peculiares. Na linha do tempo, tem-se os seguintes instantes de referência: i) instante de chegada da tarefa; ii) instante de liberação; iii) tempo de execução (com início e conclusão bem definidos); e iv) *deadline* da tarefa. Quando a tarefa finaliza antes de seu *deadline*, tem-se uma folga de tempo; caso contrário, um atraso. O tempo de resposta é dado como a diferença entre o tempo de conclusão e o tempo de chegada. Em termos de propriedades temporais das tarefas no sistema, pode-se destacar os seguintes tipos de tarefas: periódicas (ativadas em intervalos de tempo bem definidos), esporádicas (instantes de chegada não são conhecidos) e aperiódicas (sabe-se nada em relação às ativações das tarefas). Essas propriedades temporais são o ponto de partida para a análise de escalonabilidade do sistema. Nesse contexto, os sistemas em tempo real podem ser classificados pela rigidez de seus requisitos de tempo real como:

- *Hard*: caso os requisitos de tempo não sejam alcançados, inutilizam totalmente o sistema. Como exemplos, pode-se mencionar os sistemas de controle de motores de aeronaves e os sistemas de acionamento de *airbags* em veículos automotores.

- *Firm*: caso os requisitos de tempo não sejam alcançados, tornam o resultado inútil, mas não comprometem todo o sistema. Como exemplo, pode-se citar sistemas de previsão de tempo e os sistemas de decisões do mercado de ações da bolsa de valores.

- *Soft*: não causam grave dano caso os requisitos de tempo não sejam alcançados, apenas impactam diretamente no desempenho do sistema. Como exemplo, pode-se citar aplicações de transmissão multimídia e aplicações de comunicação, onde perder uma certa porcentagem dos dados gera ruído ou desconforto, mas não vai tornar a aplicação totalmente inutilizável.

Em alguns cenários específicos, como no caso de sistemas embarcados, também há necessidade de se prever mecanismos que dêem suporte a aplicações com requisitos de tempo real. Nesses ambientes, comumente se emprega plataformas de *hardware* que apresentam recursos limitados de processamento e armazenamento. Adicionalmente, muitos desses sistemas tem fonte limitada de energia (*i.e.*, são alimentados por baterias). Como principal resultado, geralmente se

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

observa sistemas que executam uma única aplicação embarcada, sem sequer utilizar um sistema operacional.

Esse trabalho apresenta uma avaliação inicial de um sistema operacional de tempo real voltado a sistemas embarcados, de código aberto e livre de licença comercial. Todavia, objetiva-se avaliar o sistema em uma plataforma de *hardware* minimalista, muito empregada na prototipação de sistemas embarcados. Como principal resultado, além de propiciar melhor entendimento acerca dos limites de um RTOS em uma plataforma de *hardware* limitada, ter-se-á uma noção clara do perfil de aplicações que eventualmente possam ser desenvolvidas nessas plataformas. Destaca-se que não se objetiva uma análise extensiva de desempenho, tampouco comparativa a outras plataformas de *hardware* com diferentes microcontroladores.

O restante desse artigo está organizado da seguinte forma: na Seção 2 descreve-se os principais componentes do FreeRTOS, enquanto a Seção 3 apresenta, brevemente, a plataforma de *hardware* utilizada nesse trabalho (*i.e.*, Arduino UNO); a Seção 4 é reservada para a avaliação e análise do FreeRTOS no Arduino UNO; e, para encerrar, a Seção 5 apresenta as conclusões mais relevantes desse trabalho.

2. FREERTOS

O FreeRTOS [4] é um sistema operacional em tempo real gratuito e de código livre, desenvolvido na linguagem C. Ele serve como base para o desenvolvimento de aplicações em sistemas embarcados que precisam cumprir requisitos de tempo real. Esse sistema é voltado para aplicações essencialmente embarcadas que, geralmente, executam em plataformas baseadas em microcontroladores ou pequenos microprocessadores, apresentando características de *soft real time requirements* e *hard real time requirements*.

A distribuição oficial do FreeRTOS possui uma série de arquivos fontes que são compartilhados por todas as combinações de compiladores e arquiteturas para as quais o sistema está disponível. Adicionalmente, há arquivos fontes específicos para cada conjunto de compilador e arquitetura.

Atualmente, o FreeRTOS está na versão 10.2.1; entretanto, este trabalho foi realizado tomando como objeto de estudo a versão 8.2.3 do sistema. Focou-se nesta versão devido a ser a primeira adaptação independente do FreeRTOS para o Arduino Uno e por não haver uma versão oficial para esta plataforma. Ou seja, esse *port* não faz parte da distribuição oficial, sendo realizado por Phillip Stevens que o desponibilizou em um repositório público em seu *github* [10]. Posteriormente, o *port* foi disponibilizado nas bibliotecas da plataforma Arduino. Em síntese, a utilização de uma versão mais antiga do FreeRTOS não compromete a avaliação e análise realizadas nesse trabalho.

Entre as diversas alterações que acorreram até a versão atual, destaca-se que, desde a versão 9.0.0, o FreeRTOS suporta alocação estática de alguns dos objetos do *kernel* (*e.g.*, tarefas, filas, semáforos e *timers*). O FreeRTOS suporta quatro modos de gerenciamento de memória; entretanto, na versão para Arduino Uno, apenas um dos esquemas de gerenciamento de memória previstos na versão completa está disponível.

2.1 Tarefas

No FreeRTOS, as aplicações são organizadas como coleções de tarefas semelhantes a *threads*[1] e independentes entre si. Cabe ao *kernel*, mais especificamente ao escalonador, decidir qual tarefa executará de acordo com as prioridades definidas pelo desenvolvedor. No caso de um processador ou microcontrolador com apenas um núcleo (como é o caso do *ATmega328P*, no Arduino Uno) apenas uma tarefa executa por vez.

Para cada tarefa, na memória, são alocados uma pilha e um TCB (*Task Control Block*, ou bloco de controle da tarefa) onde ficam armazenadas informações de controle da tarefa, tais como a prioridade da tarefa e um ponteiro para o início da pilha

Existem vários estados nos quais uma tarefa pode estar em um dado momento de execução da aplicação: executando, pronta, suspensa e bloqueada. Quando a tarefa está executando, ela está utilizando o processador e outros recursos do sistema para executar suas instruções. Tarefas que estão no estado pronto estão disponíveis para o escalonador, mas não estão sendo executadas. No estado bloqueado, a tarefa não está disponível para o escalonador. Neste caso, ela pode estar à espera de um evento temporal, tal como aguardando por uma determinada quantidade de tempo, ou um evento de sincronização como, por exemplo, a chegada de uma mensagem em uma fila. No estado suspensa, a tarefa também não está disponível para o escalonador, mas só pode ser colocada e retirada desse estado através de chamadas de uma função específica da API (*Application Programming Interface*, ou interface de programação de aplicações) do sistema.

O FreeRTOS possui diversos algoritmos de escalonamento que variam de acordo com configurações que definem a utilização, ou não, de preempção e fatiamento de tempo. Caso a preempção esteja ativada, o escalonador irá imediatamente trocar a tarefa que está executando atualmente caso outra tarefa com maior prioridade fique disponível. Isso significa que a tarefa que estava sendo executada é movida para o estado pronto involuntariamente (sem ceder lugar nem ficar bloqueada por outro motivo). Esse evento é conhecido como troca de contexto. O fatiamento de tempo diz respeito ao compartilhamento de tempo entre tarefas com a mesma prioridade. A cada intervalo de tempo predeterminado, o escalonador troca a tarefa que está executando com outra, contanto que haja pelo menos uma outra tarefa com a mesma prioridade no estado de pronta. Essa intervenção e ativação do escalonador ocorre mediante uma interrupção de relógio, sempre após o decorrer de uma fatia de tempo pré-estabelecida. O algoritmo mais comum de escalonamento é o que combina preempção e fatiamento de tempo.

Outra modalidade disponível é o escalonamento priorizado, preemptivo sem fatias de tempo. Neste caso, emprega-se quase o mesmo sistema que o escalonamento com fatias de tempo, a diferença está no fato que não há fatias de tempo, ocorrendo a troca de contexto nas situações em que uma tarefa de maior prioridade mudar para o estado pronto ou a tarefa que está executando bloquear. Desta forma, antecipa-se menos trocas de contexto e uma menor carga de processamento do escalonador; entretanto, poderá aumentar a discrepância entre a quantidade de tempo de execução de cada tarefa.

Adicionalmente, há a modalidade de escalonamento cooperativo, onde as tarefas executam até que explicitamente

[1] *Threads* são as menores sequências de instruções que podem ser gerenciadas independentemente pelo escalonador do sistema operacional.

liberem o uso do processador ou, alternativamente, entrem no estado bloqueado.

2.2 Notificações de tarefa

Notificações de tarefa é um mecanismo de comunicação direta entre tarefas, eliminando a necessidade de uma estrutura intermediária como, por exemplo, uma fila. Cada tarefa tem uma variável de notificação, que consiste em um valor inteiro de 32 *bits*. Além disso, cada tarefa tem um estado de notificação, que pode assumir um dos seguintes valores:

- ***eNotWaitingNotification***: A tarefa não está esperando uma notificação.
- ***eWaitingNotification***: A tarefa está esperando uma notificação.
- ***eNotified***: A tarefa recebeu uma notificação.

Ao notificar uma tarefa, diversas opções podem ser especificadas para se alterar o valor de notificação da tarefa alvo:

- ***eNoAction***: A tarefa recebe o evento da noticação, mas o valor de notificação não é alterado.
- ***eSetBits***: Ativa alguns bits do valor de noticação através de uma operação OR.
- ***eIncrement***: O valor de notificação da tarefa alvo é incrementado em um.
- ***eSetValueWithOverwrite***: Atribui ao valor de notificação da tarefa alvo o valor passado como parâmetro, incondicionalmente.
- ***eSetValueWithoutOverwrite***: Atribui ao valor de notificação da tarefa alvo o valor passado como parâmetro somente se a tarefa não tenha uma notificação pendente; caso contrário, a notificação falha e não surte efeito.

Por não necessitar nenhum objeto auxiliar/intermediário, o recurso de notificações de tarefas tem um custo consideravelmente menor de memória RAM, além de ser significativamente mais rápido. Entretanto, este recurso possui uma série de limitações que o fazem não ser o meio de comunicação mais apropriado em alguns casos. Dentre as principais limitações estão: não é possível enviar eventos e dados para um serviço de interrupção; não é possível notificar mais de uma tarefa simultaneamente; não é possível armazenar vários dados como, por exemplo, em um *buffer*; e, também não é possível que uma tarefa espere no estado de bloqueada até que outra tarefa com uma notificação pendente fique disponível para ser notificada.

2.3 *Timers*

Timers são usados para agendar a execução de uma determinada ação ou para repetir uma determinada ação periodicamente. Eles estão sob o controle do *kernel* e não possuem relação com os *timers* de *hardware*. *Timers* que somente executam uma ação apenas uma vez são chamados de *timers one-shot*, e podem ser reiniciados manualmente. *Timers* que executam a cada período de tempo são denominados *timers auto-reload* e, a cada vez que finalizam a ação programada, são reinicializados [4].

Existem dois estados possíveis para os *timers*: o estado *dormant*, que é quando ele existe e pode ser referenciado, mas que não está executando e, portanto, não será disparado em momento algum; e o estado *running*, indicando que ele será executado após um tempo de espera programado, quando será realizada uma ação pré-estabelecida [4].

2.4 Interrupções

Geralmente é necessário que alguma ação seja executada caso um evento ocorra, sendo a funcionalidade que capta esses eventos denominada de interrupção [4]. Por exemplo, um sensor de temperatura pode ser programado para reagir caso a temperatura do ambiente atingir determinado valor, ocasião na qual comunica o evento gerando uma interrupção.

A depender do sistema e das respectivas aplicações, deve-se responder a diversos eventos e, consequentemente, tratar várias interrupções, desencadeando processamento e uso de recursos que dependem das ações envolvidas no processo. O FreeRTOS não limita como esse processamento deve ocorrer; entretanto, recomenda ao desenvolvedor manter o tratamento da interrupção o mais breve possível e repassando para tarefas convencionais a continuidade e finalização do tratamento associado ao evento que desencadeou a interrupção.

As interrupções também possuem uma prioridade, todavia diferente em comparação a prioridade atribuída a uma tarefa. A prioridade de uma tarefa sempre será menor que a prioridade de uma interrupção; ou seja, a tarefa de maior prioridade ainda assim será deixada de lado caso uma interrupção com a menor prioridade possível ocorra. Isso se deve ao fato que uma interrupção é tratada pelo *hardware*, enquanto que uma tarefa é, essencialmente, controlada por *software* [4].

2.5 Controle de recursos

Em sistemas multitarefa, pode ocorrer de uma tarefa tentar acessar um recurso que está sendo utilizado por outra tarefa, acarretando em possíveis estados inconsistentes do sistema. Isso ocorre pois não é necessário que uma tarefa libere seus recursos ao ser removida da CPU quando ocorre uma troca de contexto.

O FreeRTOS possui métodos para gerenciar o acesso concorrente aos recursos. Um desses métodos permite a definição de uma região crítica, garantindo que apenas uma tarefa execute o código delimitado pela região. Isso é garantido pois todas as interrupções (ou interrupções com prioridades menores que uma estabelecida) são desabilitadas ao ingressar em uma região crítica. Como o escalonador somente executará a troca de contexto quando ocorre a interrupção que sinaliza a necessidade de realizar a troca, a tarefa corrente continuará executando até que ela saia da região crítica [4].

Em uma granularidade diferenciada, o sistema oferece o mecanismo de *mutexes* para gerenciar o acesso concorrente a recursos, não exigindo que se desabilite as interrupções mais do que o mínimo necessário. Um *mutex* acaba operando como uma variável de controle, deixando a cargo do desenvolvedor gerenciar apropriadamente o controle de acesso aos recursos compartilhados.

Por fim, um recurso pode ter seu acesso controlado por uma tarefa *gatekeeper*, que é uma tarefa que tem acesso exclusivo ao recurso (semelhante ao mecanismo de *spooler*). Um exemplo clássico seria uma tarefa *gatekeeper* para a impressora, sendo a única que gerencia a fila de impressão.

2.6 Filas

Filas são um recurso extremamente versátil e amplamente usado por todo o sistema operacional e aplicações no geral. Elas provêem um mecanismo de comunicação entre duas tarefas, ou tarefas e serviços de interrupção (serviços de interrupção se referem ao tratamento de eventos que acontecem no ambiente onde o sistema atua, como leituras vindo de um sensor, comandos vindos de um teclado ou pacotes chegando por um periférico de rede).

Filas são normalmente usadas como *buffers* FIFO (*First-In First-Out*), onde os dados são escritos no final e lidos do começo da estrutura. O FreeRTOS provê uma série de recursos para que filas possam ser utilizadas por diversas tarefas, tanto para leitura quanto para escrita. Um exemplo de utilização de uma fila na comunicação dentro do sistema, é uma fila onde uma tarefa escreve valores recebidos pelo serviço de interrupção de um determinado sensor/dispositivo e outra tarefa que aguarda por essas escritas para que possa realizar a leitura desses valores e realizar o processamento desejado.

2.7 Recursos de memória

Existem três tipos de memória no Arduino Uno [1]:

- Memória *Flash*: onde as instruções do programa ficam armazenadas, sendo também executadas diretamente dessa memória.
- Memória SRAM[2]: onde ficam os dados durante a execução do programa.
- Memória EEPROM: utilizada para guardar informações de longo prazo, como dados fixos e *strings*.

Até versões mais recentes do FreeRTOS, os objetos do *kernel* como filas, semáforos e tarefas só podiam ser alocados dinamicamente, em tempo de execução. Dessa forma, o sistema aloca memoria RAM a cada vez que um objeto é criado e libera a cada vez que um objeto é removido. A partir da versão 9.0.0 do sistema, é possível alocar os objetos dinamicamente em tempo de execução ou estaticamente (*i.e.*, durante a compilação).

No FreeRTOS, alocações dinâmicas de memória podem ser realizadas utilizando as funções *malloc()* e *free()* da linguagem C. Como esse meio de alocação de memória pode não ser adequado para todos os cenários, a distribuição oficial oferece cinco esquemas de gerenciamento de memória, que dizem respeito às políticas e mecanismos a serem empregados na alocação de memória nos procedimentos de criação e liberação de objetos.

Os cinco esquemas de alocação diferem entre si em aspectos como, por exemplo, a forma como é organizado o espaço ocupado pela *heap* de memória e a maneira como o espaço livre é reaproveitado quando um objeto é excluído. A *heap 1*, como assim é chamada, é extremamente simples, implementando apenas funções de alocação, sem possibilidade de liberação de espaço ocupado. A *heap 2* implementa liberação de memória e utiliza um algoritmo para garantir que os blocos de memória escolhidos para serem alocados tenham o tamanho mais próximo possível do requisitado. A *heap 3*, por sua vez, implementa as funções de alocação e liberação de memória da linguagem C (*i.e.*, *malloc()* e *free()*). Já a *heap 4* é similar à segunda variante, agregando também a combinação de blocos livres adjacentes, o que diminui a fragmentação da memória. Todos os esquemas, com exceção do terceiro, alocam estaticamente um vetor contínuo na memória para assumir a função da *heap*. Finalmente, a *heap 5* que, apesar de operar de maneira semelhante à *heap 4*, mapeia previamente as regiões da memória que serão utilizadas, dispensando assim, a necessidade delas serem contínuas. No caso específico do *port* para o Arduino Uno, o único esquema de gerenciamento de memória disponível é a *heap 3*.

[2]SRAM é a sigla para *Static Random Acess Memory. É equivalente a memória RAM, mas mantém os bits de dados na memória enquanto o dispositivo estiver ligado, diferente da RAM, que é periodicamente atualizada.*

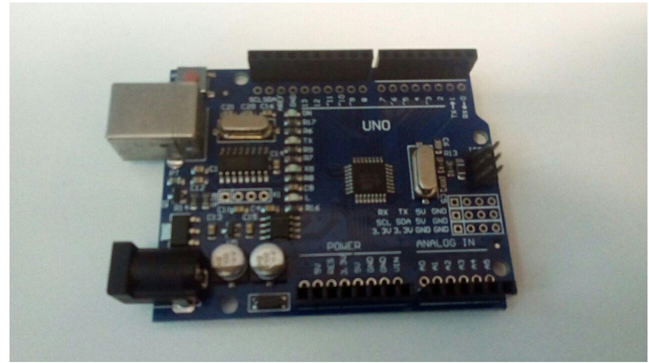

Figure 1: A placa Arduino Uno (*Release* 3, R3).

Microcontrolador	ATmega328P
Velocidade do *clock*	16 MHz
Memória *Flash*	32 KB [3]
SRAM	2 KB

Table 1: Especificações técnicas básicas do Arduino UNO.

3. ARDUINO UNO

Esta sessão apresenta informações relevantes acerca do *hardware* utilizado neste trabalho: a placa microcontroladora Arduino Uno [2].

O Arduino UNO é uma placa microcontroladora de *design* aberto. Ela possui 14 pinos de entrada e saída, onde diversos dispositivos podem ser conectados como, por exemplo, um sensor captando a temperatura ambiente para fazer algum tipo de processamento, ou um braço robótico que está sendo controlado pelo programa executando no *hardware*. A placa opera numa voltagem de $5V$ e pode ser alimentada por uma fonte externa, bateria ou por uma entrada USB tipo B, que também é utilizada para a carga (*upload*) do código executável ao ligar a placa em um computador.

Apesar de ser versátil e possibilitar a criação de uma infinidade de aplicações, o Arduino Uno é um componente de *hardware* bastante limitado. A tabela 1 mostra algumas das principais especificações da placa.

3.1 Ambientes de desenvolvimento

A plataforma Arduino possui duas IDEs (*Integrated development environment* ou ambiente de desenvolvimento integrado) para o desenvolvimento e *upload* de programas para as placas microcontroladoras. Um dos ambientes é *online*, só necessitando a instalação de um *plug-in*. O outro é um pro-

grama que pode ser instalado em qualquer computador com Mac OS, Microsoft *Windows* ou Linux. Ambos os ambientes possuem essencialmente os mesmos recursos e funcionalidades.

A linguagem usada para desenvolver programas para as placas Arduino é um conjunto de comandos das linguagens C e C++, além de comandos específicos da plataforma. Depois de pronto, o código é compilado e enviado para a placa através de uma conexão serial, estabelecida por um cabo USB tipo B entre a placa Arduino e o dispositivo no qual a IDE está sendo executada. Uma vez que o código é totalmente copiado para a placa, ele começa a executar imediatamente e a saída serial pode ser monitorada através de uma funcionalidade do próprio ambiente.

Nos ambientes de desenvolvimento, existe um recurso que permite facilmente instalar uma série de bibliotecas que oferecem uma grande variedade de funcionalidades. Dentre as bibliotecas disponíveis, tem-se o *port* do FreeRTOS (alternativamente, pode-se baixar a biblioteca do repositório original, em formato compactado, instalando-a diretamente deste arquivo).

4. AVALIAÇÃO E ANÁLISE

Até onde pôde-se constatar, não há outros trabalhos focados na análise de aspectos de desempenho dos serviços do FreeRTOS no Arduino Uno. Algumas pesquisas mais próximas (*e.g.*, [11, 7]), concentram-se na proposição e análise de aplicações na mesma plataforma ou, adicionalmente, descrevem os recursos avançados do sistema (*e.g.*, suporte ao desenvolvimento multitarefa).

Durante a análise do código fonte do *port* para Arduino Uno do FreeRTOS, documentou-se o funcionamento de várias funções, bem como diversas estruturas de dados do sistema. Essa documentação auxiliar está disponível publicamente via repositório da plataforma Zenodo [6].

Esta Seção apresenta uma análise mais detalhada acerca dos componentes e funcionalidades alvos deste trabalho, tendo-se também a intenção de realizar uma avaliação das plataformas em questão.

4.1 Gerenciamento de memória

Como mencionado anteriormente, o *port* para Arduino Uno do FreeRTOS possui apenas uma opção de gerenciamento de memória: as funções da biblioteca padrão *malloc()* e *free()*. Para entender as implicações deste esquema de gerenciamento de memória para a escalabilidade e desempenho de aplicações, é necessário primeiro entender como ele funciona.

O Arduino Uno utiliza uma biblioteca de funções padrões da linguagem C adaptadas para dispositivos AVR[4], denominada *AVR Libc* [3]. A maioria dos dispositivos AVR, assim como o Arduino Uno, possuem uma quantidade limitada de memória RAM. Esta memória é organizada em áreas/sessões da seguinte forma:

- ***.data*:** Sessão onde ficam as variáveis inicializadas e/ou com dados estáticos.
- ***.bss*:** Sessão onde ficam as variáveis globais ou estáticas não inicializadas.
- ***heap*:** Espaço ocupado pelas alocações dinâmicas de memória.
- ***stack* (pilha):** Espaço usado para chamar sub-rotinas e armazenar variáveis locais.

No Arduino Uno, assim como em outras arquiteturas AVR, não há proteção de memória. O *layout* padrão de memórias RAM contempla a sessão *.data* no início da memória, seguido da sessão *.bss* (Figura 2). A pilha inicia no topo (endereço mais alto) da memória, enquanto a *heap* inicia após o fim da sessão *.bss*, avançando em direção contrária à pilha. Dessa forma, não há possibilidade de colisão entre a *heap* e a *.bss*, mas permanece o risco da pilha e da *heap* colidirem. Esse risco é mais eminente quanto maior for a demanda por alocações dinâmicas ou espaço na pilha, seja esta resultante da alocação de variáveis locais ou chamadas recursivas de funções.

O Gerenciador de Memória utilizado pela *AVR Libc* não mantém nenhuma estrutura de dados separada para manter o controle dos blocos livres de memória. Para isso, ele implementa uma lista encadeada de blocos livres na própria *heap*. Cada bloco de memória livre é precedido de quatro *bytes* contendo o ponteiro para o próximo bloco livre e o tamanho do bloco, respectivamente. Além disso, o endereço do espaço alocado retornado por uma chamada de *malloc()* é precedido de dois *bytes* que indicam o tamanho do bloco ocupado.

Ao receber uma chamada de alocação de memória (*malloc()*), a lista encadeada é percorrida em busca de um bloco que atenda perfeitamente a solicitação de tamanho. Caso esse bloco exista, ele será desconectado da lista e retornado para quem chamou a função. Caso contrário, utiliza-se o bloco de tamanho mais próximo à requisição. Neste caso, o bloco é particionado em dois: um deles é retornado a quem chamou a função, enquanto o outro permanece na lista. Caso o bloco encontrado seja apenas dois *bytes* maior que o requisitado, esses dois *bytes* serão inclusos na requisição, já que não seria possível criar um novo bloco livre. Na hipótese de não haver blocos livres capazes de atender a solicitação, faz-se uma tentativa de estender a *heap*. Na situação de não existir memória suficiente para atender à solicitação, o valor *NULL* é retornado para o chamador da função.

Ao receber uma chamada de liberação de memória (*free()*), tenta-se juntar o bloco sendo liberado com possíveis blocos livres adjacentes e, desta forma, reduzir a fragmentação de memória. Espaços livres que surgem ao longo da *heap* no transcorrer de alocações e liberações acabam não sendo utilizados por não satisfazerem as requisições de novas alocações, causando desperdício de memória. Quando o bloco na borda superior da *heap* é liberado, o tamanho da *heap* também diminui.

No FreeRTOS, a função *malloc()*, da biblioteca *AVR Libc*, é encapsulada por outra função denominada *pvPortMalloc()*. Quando acontece uma alocação dinâmica de memória, é necessário garantir que a operação seja segura (*i.e.*, *thread-safe*[5]); ou seja, deve-se garantir que não ocorram acessos indevidos à memória antes do processo de alocação finalizar. Para implementar essa segurança, o FreeRTOS cria uma sessão crítica, suspendendo o escalonador (via função

[4] AVR™ é uma família de microcontroladores desenvolvidos desde 1996 pela Atmel, atualmente de propriedade da empresa *Microchip Technology* (`www.microchip.com`).

[5] Uma aplicação é considerada *thread-safe* quando não causa problemas ao ser executada em um cenário de múltiplas *threads* (ou tarefas, no caso do *FreeRTOS*).

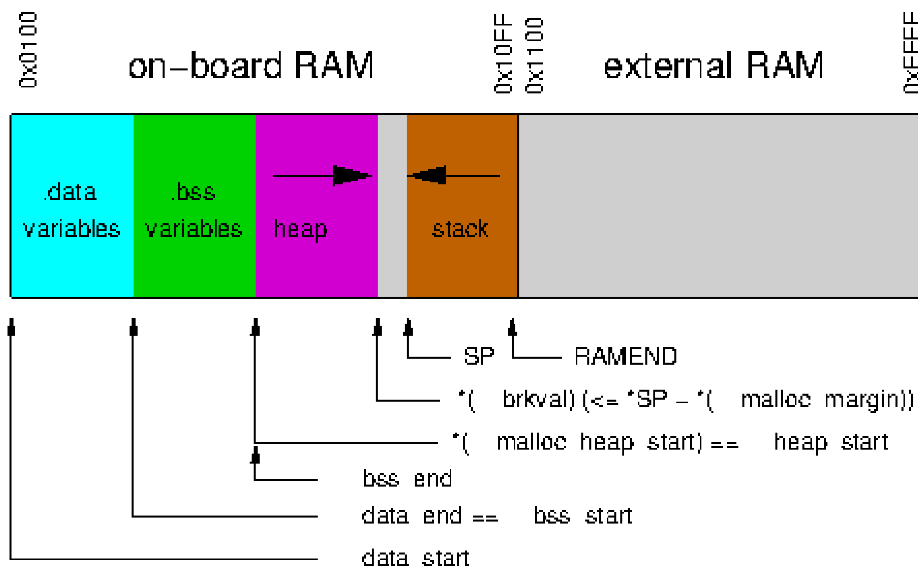

Figure 2: Mapa do *layout* da memória RAM (adaptado de [3]).

vTaskSuspendAll()) antes do início da alocação e retomando-o apenas quando a alocação estiver concluida.

Enquanto o escalonador está suspenso, trocas de contexto não ocorrem; porém, as interrupções não são desabilitadas. Os *ticks*[6] ficarão pendentes até que o escalonador seja retomado através de uma chamada à função *xTaskResumeAll()*. Depois que o escalonador é suspenso, uma chamada à função *malloc()* é realizada. O retorno da chamada é salvo e será o endereço para o início do bloco de memória alocado em caso de sucesso e *NULL* caso contrário.

Depois que a alocação é concluida, o funcionamento do escalonador é retomado através de uma chamada da função *xTaskResumeAll()*. Primeiramente, a função entra numa região crítica, desabilitando globalmente as interrupções através de uma chamada à função *taskENTER_CRITICAL()*. É possível que, enquanto o escalonador esteve suspenso, alguma tarefa tenha sido removida de uma lista de evento[7] através de uma rotina de serviço de interrupção (ISR). Se este for o caso, a tarefa associada ao evento terá sido adicionada à lista de tarefas que estão pendentes para ficarem prontas (*xPendingReadyList*). Uma vez que o escalonador é retomado, essas tarefas devem ser movidas para a lista de prontas apropriada. Caso alguma dessas tarefas tenha prioridade maior que aquela que está sendo executada no momento, uma troca de contexto é marcada como pendente. Caso hajam *ticks* pendentes referentes ao tempo em que o escalonador esteve suspenso, eles devem ser processados a partir desse momento, para garantir que não hajam erros no contador de *ticks* e tarefas atrasadas sejam continuadas no tempo correto. Durante o processamento dos *ticks* pendentes, é possível que uma troca de contexto seja marcada como pendente também, uma vez que o processamento de um *tick* pode desbloquear tarefas com maior prioridade que a atual. Depois de processados os *ticks* pendentes, caso haja uma troca de contexto pendente, ela é realizada contanto que o uso de preempção esteja habilitado (*i.e.*, *configUSE_PREEMPTION* seja 1). Por fim, a função sai da sessão crítica com uma chamada de *taskEXIT_CRITICAL()*.

Após concluído o procedimento de retomada do escalonador, a função *pvPortmalloc()* analisa o retorno da função *malloc()*, retornando o endereço para o bloco de memória alocado em caso de sucesso, ou NULL caso contrário. A implementação da função *pvPortMalloc()* também permite que, caso o retorno da função *malloc()* seja *NULL*, uma função *hook*[8] seja executada. Para tanto, a definição da constante *configUSE_MALLOC_FAILED_HOOK* deve ser verdadeira e a função (*vApplicationMallocFailedHook()*) deve ter sido previamente implementada pelo desenvolvedor da aplicação.

Destaca-se que o tempo de execução da função *malloc()* impacta diretamente no tempo necessário para retomada do escalonador. Considerando-se que o processo de alocação dinâmica é complexo, espera-se que o custo (tempo) varie de acordo com uma série de fatores, tais como quantidade de memória requisitada, histórico de alocações, número de tarefas executando e o estado atual do sistema.

4.2 Gerenciamento de tarefas

Cada tarefa possui na memória uma pilha e um bloco de controle (TCB). O tamanho da pilha é determinado no momento da criação da tarefa, variando entre o mínimo de 85 *bytes* (difinido pela constante *configMINIMAL_STACK_SIZE*) até o máximo possível sem que ocorra uma *heap overflow*[9].

Para melhor compreensão do espaço ocupado por cada tarefa na memória, é preciso analisar os elementos que compõe os blocos de controle de cada tarefa, além de como ocorre a criação das tarefas.

A Tabela 2 mostra os principais campos da estrutura de

[6] Cada ciclo de processamento do microcontrolador.

[7] Listas onde tarefas esperam pela ocorrência de um evento. Geralmente, o evento relacionado está associado a uma rotina de serviço de interrupção.

[8] Funções *hook* permitem adicionar funcionalidades ao sistema em resposta a determinados eventos.

[9] Quando há uma tentativa de fazer uma alocação maior que o espaço disponível na *heap*, ocorre um transbordamento da *heap*, ou *heap overflow*.

bloco de controle (TCB) de uma tarefa, definida como *TCB_t*. Por motivos de clareza e simplicidade, alguns campos referentes aos recursos ausentes no *port* para Arduino Uno (*e.g.*, proteção de memória), recursos de *debugging* e/ou que dizem respeito ao uso de bibliotecas de terceiros não foram considerados nessa análise.

A seguir, descreve-se cada um dos principais campos que compõe o TCB:

- ***pxTopOfStack***: Ponteiro para a localização do último item colocado na pilha da tarefa.
- ***xGenericListItem***: A lista referenciada por este item denota o estado atual da tarefa (pronta, bloqueada, suspensa).
- ***xEventListItem***: Caso a tarefa esteja em uma lista de eventos, ela será referenciada por esse item.
- ***uxPriority***: Prioridade da tarefa. O valor mínimo é 0.
- ***pxStack***: Ponteiro para o começo da pilha da tarefa.
- ***pcTaskName***: Nome descritivo da tarefa, para facilitar na depuração.
- ***uxBasePriority***: Caso a funcionalidade de *mutexes* esteja ativada, guarda a última prioridade atribuída à tarefa. Esse valor é usado pelo mecanismo de herança de prioridades.
- ***uxMutexesHeld***: Caso a funcionalidade de *mutexes* esteja ativada, guarda a quantidade de *mutexes* segurados pela tarefa.
- ***ulNotifiedValue***: Caso a funcionalidade de notificações de tarefa esteja ativada, guarda o valor de notificação da tarefa, um inteiro de 32 *bits*.
- ***eNotifyState***: Guarda o estado de notificação da tarefa, pode ser um dentre os seguintes valores: (*eNotWaitingNotification*, *eWaitingNotification*, *eNotified*).

O tamanho padrão do TCB é de 41 *bytes*, sem habilitar nenhuma funcionalidade de depuração e/ou recursos de terceiros ou aumentar o tamanho máximo do nome da tarefa. Em contrapartida, o tamanho mínimo do TCB é de 27 *bytes*. Esses valores não incluem os 2 *bytes* provenientes da alocação dinâmica, apenas o tamanho da estrutura.

4.2.1 Criação de tarefas

A função *xTaskGenericCreate* é responsável por criar tarefas: inicialmente, o espaço necessário para o TCB e para a pilha da nova tarefa é solicitado através de uma chamada à função *prvAllocateTCBAndStack()*. Esta função determina, de acordo com a arquitetura, o sentido de crescimento da pilha (*i.e.*, dos endereços mais baixos aos mais altos, ou o contrário). Caso a pilha cresça no sentido dos endereços mais altos, o TCB é alocado antes da pilha. Caso contrário, o TCB é alocado depois da pilha. Essa verificação garante que a pilha nunca cresça na direção do TCB, de modo a poder sobrescrevê-lo. No caso do *port* para Arduino Uno, a pilha cresce em direção aos endereços mais baixos e, tanto a alocação da pilha como a do TCB, são executadas via função *pvPortMalloc()* e, em consequência disso, a pilha e o TCB ocupam cada um 2 *bytes* a mais na memória. Caso as alocações sejam bem sucedidas, o endereço do começo da pilha é atribuído ao campo *pxStack* do TCB e o endereço deste é passado como retorno da função. Caso contrário, qualquer espaço utilizado é liberado, e a função retorna *NULL*.

Depois de alocados a pilha e o TCB, os demais campos do TCB são inicializados através de uma chamada à função *prvInitialiseTCBVariables()*. Esta função inicializa todos os campos do TCB ainda não inicializados como, por exemplo, o nome da tarefa (quando houver), a prioridade desta e os itens de lista. Depois disso, a pilha da tarefa é inicializada (via função *pxPortInitialiseStack()*) de modo a parecer como se a tarefa já estivesse sendo executada; porém, como se estivesse sido interrompida pelo escalonador. O início da função da tarefa é atribuído ao endereço de retorno. Caso tenha sido informado um endereço para armazenar o *handle* da tarefa, ele é copiado para tal.

Em uma etapa seguinte, as estruturas globais sãos atualizadas para se adequarem à criação da nova tarefa. Para isso, desabilita-se as interrupções através de uma chamada à função *taskENTER_CRITICAL()*, garantindo-se acesso exclusivo às listas de tarefas durante as atualizações. Caso seja a primeira tarefa a ser criada, necessita-se inicializar as listas de tarefas via uma chamada à função *prvInitialiseTaskLists()*. Caso não haja outra tarefa marcada como atual ou o escalonador ainda não esteja sendo executado ou, ainda, a tarefa sendo criada tenha a maior prioridade dentre as demais, ela é atribuída como tarefa atual.

Finalmente, atribuí-se *pdPASS* à variável de retorno da função, indicando sucesso na criação da nova tarefa. A tarefa inicia seu ciclo na lista de tarefas prontas e, caso seja a única de mais alta prioridade, o escalonador procede à mudança de contexto para a tarefa recém criada.

4.2.2 Máximo alocável de tarefas

Sabendo exatamente quanto cada tarefa ocupa de espaço na memória, como descrito nas sessões anteriores, é possível determinar o mínimo de memória necessária para execução de determinada aplicação. Porém, para ilustrar as funcionalidades e limitações do *port* do FreeRTOS no Arduino Uno, avaliou-se o máximo de tarefas que podem ser criadas concomitantemente ao máximo de pilha alocável por tarefa. Para determinar esses valores, testou-se o maior tamanho de pilha possível para as tarefas criadas sem, todavia, resultar em *heap overflow*. Na plataforma de *hardware* em questão, quando ocorre *heap overflow*, um LED de *status* da placa pisca com um intervalo de 100 ms [10]. O teste foi realizado utilizando as configurações padrões do *port* do FreeRTOS para Arduino Uno disponibilizado nas bibliotecas da plataforma Arduino.

A Tabela 3 relaciona a quantidade de tarefas criadas com o máximo de tamanho de pilha que é possível de ser alocado. Os valores contidos na terceira coluna da tabela não levam em conta o *overhead* produzido pela alocação dinâmica de cada pilha e TCB. Esse teste foi realizado utilizando as configurações padrões do *port* do FreeRTOS para Arduino Uno.

4.2.3 Latência média da troca de contexto entre tarefas

Segundo [9], sistemas operacionais em tempo real podem ser caracterizados por dois requisitos básicos: pontualidade e confiabilidade. Para satisfazer esses requisitos, são utilizados alguns princípios, tais como *multitasking* e comunicação eficiente entre tarefas. Para analisar a utilização desses princípios como garantia dos requisitos do sistema operacional, dois testes foram propostos por Sacha: O primeiro mede a

Campo	Tipo	Tamanho (bytes)	Obrigatório	Habilitado por Padrão
pxTopOfStack	*stackType_t**	2	**Sim**	-
xGenericListItem	*ListItem_t*	10	**Sim**	-
xEventListItem	*ListItem_t*	10	**Sim**	-
uxPriority	*UBaseType_t*	1	**Sim**	-
pxStack	*StackType_t**	2	**Sim**	-
pcTaskName	*char[]*	8	**Não**	**Sim**
uxBasePriority	*UBaseType_t*	1	**Não**	**Sim**
uxMutexesHeld	*UBaseType_t*	1	**Não**	**Sim**
ulNotifiedValue	*uint32_t*	4	**Não**	**Sim**
eNotifyState	*eNotifyValue*	2	**Não**	**Sim**

Table 2: Principais campos do TCB de cada tarefa.

Quantidade de tarefas	Tamanho máximo da pilha (bytes)	Pilhas + TCB's (*bytes*)
1	1297	1338
2	626	1334
3	402	1329
4	290	1324
5	223	1320
6	178	1314
7	146	1309
8	122	1304
9	104	1305
10	89	1300

Table 3: Tamanho máximo de pilha alocável por tarefa de acordo com a quantidade de tarefas.

velocidade das trocas de contexto entre tarefas, o segundo mede a agilidade na transferência de dados entre tarefas. O primeiro teste é foco desta seção, enquanto o segunto é apresentado na Seção 4.3.

O tempo consumido pela troca de contexto entre duas tarefas é um *overhead*[10] naturalmente produzido por qualquer sistema operacional que funcione com *multitasking*. Além disso, este tempo está presente nos meios de comunicação entre tarefas, uma vez que é necessária a troca de contexto entre a tarefa que envia a informação e a que recebe. Portanto, este tempo é uma medida importante, que ajuda a caracterizar o desempenho do sistema operacional como um todo [9].

Descrição do Teste: Nesse teste, duas tarefas executam simultaneamente: uma principal e uma secundária. Primeiro, a tarefa principal lê o tempo atual[11] e executa uma quantidade determinada de iterações num laço vazio. Então, ela lê o tempo atual novamente, calcula a diferença e divide pela quantidade de iterações. O resultado é a duração de tempo de uma iteração do laço na tarefa principal. Depois disso, a tarefa principal lê o tempo novamente e então executa um laço com o mesmo número de iterações do anterior, mas dessa vez, a cada iteração, ela entrega a CPU para a tarefa secundária. A tarefa secundária, por sua vez, tem como única função devolver a CPU para a tarefa principal. Esse processo continua até que o laço da tarefa principal termine.

[10]Excesso de processamento necessário para executar determinada tarefa.

[11]A leitura do tempo é feita através da chamada da função *micros()*, segundo a documentação da API do Arduino, em placas com microcontroladores de 16MHz, como é o caso do Arduino Uno, a função tem resolução de 4 microssegundos.

Núm. de Iterações	Tempo Médio da Troca de Contexto (μs)
10000	7,74
20000	7,74
30000	7,74

Table 4: Latência média na troca de contexto entre tarefas.

Quando isso ocorre, a tarefa principal lê o tempo mais uma vez e então calcula a diferença do tempo inicial com o atual, dividido pelo número de iterações (para se obter o tempo por iteração) dividido por dois (por serem duas tarefas, consideradas idênticas). Esse é o tempo de uma iteração do laço somado ao tempo da troca de contexto. Por fim, para se obter o tempo da troca de contexto, basta subtrair o tempo da iteração no laço da tarefa principal, calculado anteriormente.

Para evitar que o tempo medido nesse teste abranja outras atividades do sistema operacional, tal como execução periódica de certas tarefas, elas foram criadas com o maior nível de prioridade possível, sendo que a tarefa principal tem prioridade maior que a secundária, para que a secundária não execute até o momento que a principal lhe entregue a CPU.

A Tabela 4 apresenta os resultados obtidos para o tempo médio de troca de contexto entre tarefas conforme o número de iterações do código teste descrito na Figura 3. A variação na quantidade de amostras não resultou em diferença no tempo médio das trocas de contexto, o que significa um resultado positivo em termos de previsibilidade e confiabilidade do sistema.

4.3 Gerenciamento de filas

Como mencionado anteriormente, as filas são um meca-

```
1  void vMainTask(){
2   volatile uint32_t count = LOOP_COUNT;
3   uint32_t t1, t2;
4   float tx, ty, TS;
5
6   t1 = micros();
7   do {
8   } while(count--);
9   t2 = micros();
10  tx = ((float)t2 - t1)/(float)LOOP_COUNT;
11  count = LOOP_COUNT;
12  t1 = micros();
13  do {
14    taskYIELD(); // força a mudança de contexto
15  } while(count--);
16  t2 = micros();
17  ty = (t2 - t1)/(float)LOOP_COUNT/2.0;
18  TS = ty - tx;
19  Serial.print(TS);
20  vTaskDelete(); // remove/finaliza essa tarefa
21 }
22
23 void vInterweavingTask(){
24  do {
25   taskYIELD(); // força a mudança de contexto
26  } while(1);
27 }
```

Figure 3: Teste para troca de contexto no FreeRTOS.

nismo amplamente utilizado pelo sistema operacional, servindo como *buffers* de dados e participando no processo de comunicação entre tarefas ou de tarefas com serviços de interrupção. No FreeRTOS, a estrutura de filas também é utilizada para implementar a funcionalidade de *mutexes*.

A Tabela 5 mostra os principais campos da estrutura de fila, definida como *Queue_t*. Novamente, por questões de clareza, omitiram-se campos referentes a funcionalidades de depuração.

A seguir, descrevem-se os principais campos que compõe a estrutura das filas:

- ***pcHead***: Ponteiro para o ínicio da área de armazenamento da fila.
- ***pcTail***: Ponteiro para o *byte* no final da área de armazenamento da fila. Um *byte* a mais que o necessário é alocado para armazenar os itens da fila, ele é usado como marcador.
- ***pcWriteTo***: Ponteiro para o próximo espaço livre na área de armazenamento.
- ***pcReadFrom/uxRecursiveCallCount***: Como estes dois itens nunca coexistem, usa-se uma *union*[12] para economizar RAM. Pode ser um ponteiro para o último lugar de onde um item enfileirado foi lido ou um contador para o número de vezes que um *mutex* recursivo foi recursivamente pego (*i.e.,* quando a estrutura é usada como um *mutex*).
- ***xTasksWaitingToSend***: Lista das tarefas que estão bloqueadas esperando para escrever nessa fila. As tarefas são armazenadas em ordem de prioridade.
- ***xTasksWaitingToReceive***: Lista das tarefas que estão bloqueadas esperando para ler dessa fila. As tarefas são armazenadas em ordem de prioridade.
- ***uxMessagesWaiting***: Número de itens atualmente na fila.
- ***uxLength***: Tamanho da fila em capacidade total de itens, não *bytes*.
- ***uxItemSize***: Tamanho de cada item da fila em *bytes*.
- ***xRxLock***: Armazena o número de itens removidos da fila enquanto a fila estava bloqueada. Quando a fila não está bloqueada, vale *queueUNLOCKED*, que é definida como -1.
- ***xTxLock***: Armazena o número de itens adicionados à fila enquanto a fila estava bloqueada. Quando a fila não está bloqueada, vale *queueUNLOCKED*, que é definida como -1.
- ***pxQueueSetContainer***: Caso o recurso de *"queue sets"* esteja ativado, aponta para o conjunto a qual essa fila pertence. *"Queue sets"* são um recurso do FreeRTOS para que uma tarefa desbloqueie ao conseguir ler de mais de uma fila, por exemplo.

A partir da análise dos campos listados acima e seus respectivos tamanhos, pode-se determinar que o tamanho da estrutura de uma fila, por padrão, é de 31 *bytes*. O tamanho máximo da estrutura é de 33 *bytes*, quando a funcionalidade de conjuntos de fila (*queue sets*) está habilitada.

[12] *Unions* são tipos de dados especiais da linguagem C que permitem armanezar diferentes tipos de dados no mesmo espaço de memória. Apenas um dos elementos da *union* pode existir em dado instante de tempo.

Campo	Tipo	Tamanho (bytes)	Obrigatório	Habilitado por Padrão
pcHead	*int8_t**	2	**Sim**	-
pcTail	*int8_t**	2	**Sim**	-
pcWriteTo	*int8_t**	2	**Sim**	-
pcReadFrom/ uxRecursiveCallCount	*int8_t* / UBaseType_t*	2	**Sim**	-
xTasksWaitingToSend	*List_t*	9	**Sim**	-
xTasksWaitingToReceive	*List_t*	9	**Sim**	-
uxMessagesWaiting	*UBaseType_t*	1	**Sim**	-
uxLength	*UBaseType_t*	1	**Sim**	-
uxItemSize	*UBaseType_t*	1	**Sim**	-
xRxLock	*UBaseType_t*	1	**Sim**	-
xTxLock	UBaseType_t	1	**Sim**	-
pxQueueSetContainer	QueueDefinition*	2	**Não**	**Não**

Table 5: Principais campos da estrutura de fila.

Este tamanho só diz respeito a estrutura de controle da fila, o tamanho da área de armazenamento pode ser calculado multiplicando-se a capacidade máxima da fila (*uxLenght*) pelo tamanho de cada item (*uxItemSize*) e somando-se um byte extra, alocado para ser utilizado como marcador. Apenas uma alocação dinâmica é realizada para criar a estrutura da fila e sua área de armazenamento, reduzindo o espaço consumido pelo gerenciador de memória.

4.3.1 Criação de filas

A função *xQueueGenericCreate* é responsável pela criação das filas. As filas são compostas por dois elementos localizados sequencialmente na memória: A estrutura da fila em si e a área de armazenamento, onde ficam guardados os itens da fila. O tamanho da área de armazenamento é determinado pelo número de itens que a fila suporta multiplicado pelo tamanho de cada item. O espaço total ocupado pela fila (estrutura mais a área de armazenamento) é alocado sequencialmente na memória, através de uma chamada a *pvPortMalloc*.

Depois da alocação do espaço ocupado pela fila, calcula-se o valor da variável *pcHead*, que passa a apontar para o início da área de armazenamento. Os campos da estrutura da fila *uxQueueLength* e *uxItemSize* tem seus respectivos valores atribuídos e a fila é configurada para um estado inicial através de uma chamada da função *xQueueGenericReset*. Esta função é responsável por atribuir às outras variáveis da estrutura da fila seus valores padrão. Como ela pode ser chamada para uma fila já existente, a função entra na região crítica (via uma chamada a*taskENTER_CRITICAL()*), pois tarefas podem ser desbloqueadas durante o procedimento: havendo uma tarefa esperando para escrever na fila, a tarefa já pode ser desbloqueada, pois a fila ficará vazia. Na finalização da execução da função *xQueueGenericReset*, retorna-se o endereço (*i.e.*, *handle*) da fila ao chamador.

4.3.2 Máximo de filas e tarefas

De maneira análoga à avaliação para determinar a quantidade máxima de tarefas, realizou-se uma avaliação relativa à quantidade máxima de filas. Como o tamanho das filas varia de acordo com a sua capacidade máxima e com o tamanho de cada item, fixou-se cada fila com cinco itens de quatro *bytes* cada. O tamanho da pilha de cada tarefa ficou definido como sendo o mínimo possível (*i.e.*, 85 *bytes*). Igualmente à análise anterior, o recurso de detecção de *heap overflow* foi empregado para determinar o limite de filas alocáveis. O teste foi realizado utilizando as configurações padrões do *port* do FreeRTOS para Arduino Uno disponibilizado nas bibliotecas da plataforma Arduino. A Tabela 6 apresenta os resultados obtidos.

Quantidade de Tarefas	Quantidade Máxima de Filas
1	22
2	19
3	17
4	15
5	12
6	10
7	7
8	5
9	3
10	0

Table 6: Máximo de filas por quantidade de tarefas.

4.3.3 Latência média na troca de mensagens entre tarefas

A maioria dos mecanismos para comunicação entre tarefas podem ser divididos em duas categorias: síncronos e assíncronos. O FreeRTOS dispõe de dois mecanismos principais para esse fim: filas, que podem ser usadas como *buffers* assíncronos, e notificações de tarefas, um mecanismo síncrono que permite comunicação direta entre duas tarefas. Como os dois mecanismos possuem características e aplicações distintas, pode-se tornar difícil fazer um teste quantitativo das duas. Para contornar esse problema, propõe-se um teste constituído da troca de pares de mensagens entre duas tarefas, que pode ser facilmente implementado com qualquer ferramenta. O tempo a ser medido no seguinte teste corresponde a toda a operação: o envio da mensagem para a primeira tarefa, a troca de contexto e o recebimento da mensagem pela segunda tarefa. Segundo [9], essa é a menor porção da operação que pode ser observada na vida real.

Descrição do teste: Duas tarefas fazem parte do teste, executando simultaneamente: uma tarefa cliente e uma tarefa servidor. A tarefa cliente lê o tempo/hora atual e então faz uma quantia predeterminada de trocas de mensagens

```
void vClient(){
  count = LOOP_COUNT;
  t1 = micros();
  do {
    // envia mensagem ao servidor (bloqueia até enviar)
    xQueueSend(fila1, message, portMAX_DELAY);
    // recebe mensagem do servidor (bloqueia até receber)
    xQueueReceive(fila2, message, portMAX_DELAY);
  } while(count--);
  t2 = micros();
 TM = (t2-t1)/(double)LOOP_COUNT/2.;
  Serial.println(TM);
}

void vServer(){
   do {
     // recebe mensagem do cliente (bloqueia até receber)
     xQueueReceive(fila1, message, portMAX_DELAY);
     // envia mensagem ao cliente (bloqueia até enviar)
     xQueueSend(fila2, message, portMAX_DELAY);
   } while(1);
}
```

Figure 4: Teste para comunicação entre tarefas usando filas no FreeRTOS.

```
void vClient(){
  count = LOOP_COUNT;
  t1 = micros();
  do {
    // notifica o servidor
    xTaskNotify(tServer, NUM, eSetValueWithOverwrite);
    // aguarda até ser reciprocamente notificado
    xTaskNotifyWait(0, 0, &val, portMAX_DELAY);
  } while(count--);
  t2 = micros();
 TM = (t2-t1)/(double)LOOP_COUNT/2.;
  Serial.println(TM);
  vTaskDelete(NULL);
}

void vServer(){
  do {
    // aguarda notificação do cliente
    xTaskNotifyWait(0, 0, &val, portMAX_DELAY);
    // retorna notificação ao cliente
    xTaskNotify(tClient, NUM, eSetValueWithOverwrite);
  } while(1);
}
```

Figure 5: Teste para comunicação entre tarefas usando notificações de tarefa no FreeRTOS.

Número de iterações (amostras)	Tamanho da mensagem (bytes)	Tempo de transferência (μs)
100000	1	94,35
100000	10	103,29
100000	50	143,57
100000	100	193,73
100000	250	344,61
100000	500	338,52

Table 7: Duração média na transferência de mensagens via filas.

Número de iterações (amostras)	Tamanho da mensagem (bytes)	Tempo de transferência (μs)
100000	4	50,28

Table 8: Duração média na transferência de mensagens via notificações de tarefas.

com a tarefa servidor que, simplesmente, recebe a mensagem do cliente e responde. Depois que as trocas de mensagem acabam, a tarefa cliente lê novamente o tempo/hora, calcula a diferença com o tempo inicial, divide pelo número de iterações e então divide por dois, obtendo o tempo médio equivalente a apenas uma troca de mensagem.

As Tabelas 7 e 8 apresentam os resultados referentes aos tempos médios de envio de mensagens entre tarefas utilizando filas e notificações de tarefas, respectivamente. Apesar de suas limitações, o mecanismo de troca de mensagens via notificação de tarefas é mais eficiente do que a troca baseada em filas. Isto é justificado pelo fato do mecanismo de notificação não envolver qualquer outro objeto/estrutura auxiliar no processo de comunicação (*i.e.*, envolve apenas manipulação de campos do próprio TCB da tarefa).

Observa-se que a correspondência entre o tamanho da mensagem e o tempo de transferência utilizando filas é fortemente não linear. Essa característica já havia sido salientada por Krzysztof Sacha, quando propôs o mesmo teste em seu trabalho [9].

5. CONCLUSÕES

Apesar dos recursos de processamento e memória serem limitados no Arduino Uno, constatou-se que há margem para executar o sistema operacional FreeRTOS contemplando aplicações com múltiplas tarefas. A quantidade de tarefas fica dependente da necessidade de espaço para a pilha de cada tarefa, sendo possível executar até 10 tarefas com no máximo 89 bytes de pilha cada uma.

A mudança de contexto entre tarefas também se mostrou eficiente em termos de atraso, ficando abaixo de 8 μs. O baixo custo se deve, sobretudo, ao fato do FreeRTOS suportar apenas uma aplicação com múltiplas tarefas, facilitando os ajustes necessários quando da mudança de contexto.

A comunicação entre tarefas também pode ser outro fator agregado ao atraso na execução da aplicação. Quando empregando filas para comunicação entre tarefas, observou-se que o atraso aumenta à medida que o tamanho da mensagem aumenta, mas com comportamento não linear para mensagens com tamanho entre 250 e 500 bytes. De qualquer forma, os atrasos observados estão sempre abaixo dos 350 μs. A comunicação entre tarefas é mais rápida quando se emprega o mecanismo de notificação de tarefas, com atrasos na ordem de 50 μs. No entanto, este mecanismo permite mensagens de apenas 4 bytes. Em síntese, esse mecanismo sempre será preferido quando se puder priorizar atraso em detrimento ao volume de dados transmitidos.

Para ampliar o entendimento dos limites desse conjunto de plataformas, avaliou-se também a combinação entre a quantidade possível de tarefas e de filas simultaneamente. Como esperado, consegue-se instanciar menos filas à medida que se aumenta o número de tarefas simultâneas: consegue-se avançar de um patamar de 22 filas para uma única tarefa até 3 filas com 9 tarefas.

Em síntese, o desenvolvimento de uma aplicação no cenário em questão deverá manter a quantidade de tarefas ao mínimo necessário, privilegiando a comunicação entre tarefas baseada em notificação de tarefas. Manter também a quantidade e tamanho das filas dentro do estritamento necessário, utilizando-se mensagens com tamanho máximo bem definido. Dessa forma, pode-se estimar com boa precisão os atrasos intrínsecos aos mecanismos envolvidos no processo.

6. REFERENCES

[1] A. AG. Memory. Disponível em: `https://www.arduino.cc/en/Tutorial/Memory`, 2018. Acesso em: 01/03/2019.

[2] Arduino AG. Arduino uno rev3. Disponível em: `https://store.arduino.cc/arduino-uno-rev3`, 2018. Acesso em: 01/03/2019.

[3] AVR-Libc. Memory areas and using malloc. Disponível em: `https://www.nongnu.org/avr-libc/user-manual/malloc.html`, 2016. Acesso em: 01/03/2019.

[4] R. Barry. Mastering the freertos real time kernel-a hands on tutorial guide. *Real Time Engineers Ltd*, 2016.

[5] G. Buttazzo. *Hard Real-Time Computing Systems.* Springer US, 3 edition, 2011.

[6] F. Chabatura and L. Tironi Fassini. Descrições de Funções e Estruturas da Versão 8.2.3 do FreeRTOS para o Arduino Uno. `https://doi.org/10.5281/zenodo.1580268`, 2018.

[7] D. Mitrovic and S. Randic. Arduino platform capabilities in multitasking environment. In *7th International Scientific Conference Technics and Informatics in Education*, pages 304–309. University of Kragujevac, Faculty of Technical Sciences Cacak, Serbia, May 2018.

[8] F. Rammig, M. Ditze, P. Janacik, T. Heimfarth, T. Kerstan, S. Oberthuer, and K. Stahl. Basic concepts of real time operating systems. In *Hardware-dependent Software*, pages 15–45. Springer, 2009.

[9] K. M. Sacha. Measuring the real-time operating system performance. In *Real-Time Systems, 1995. Proceedings., Seventh Euromicro Workshop on*, pages 34–40. IEEE, 1995.

[10] P. Stevens. Arduino freertos library. Disponível em: <`https://github.com/feilipu/Arduino_FreeRTOS_Library`>, 2018. Acesso em: 01/03/2019.

[11] H. YANIK, E. UYSAL, and A. ELEWI. Multitasking driver assistance system using arduino uno. In *2018 International Conference on Artificial Intelligence and Data Processing (IDAP)*, pages 1–6, Sep. 2018.