Análise do impacto da reprogramação do clock do microcontrolador ATmega328P na previsibilidade do FreeRTOS no Arduino Uno

Rodrigo Miguel Tomazi
Universidade Federal da Fronteira Sul
Chapecó - SC - Brazil
rodrigo.tomazi@outlook.com

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

ABSTRACT

O FreeRTOS é um sistema operacional de tempo real, de código livre, voltado para sistemas embarcados. O Arduino Uno é uma plataforma de projeto aberto para prototipação de *hardware*. Apesar de suas intrínsecas limitações, o Arduino Uno comporta a execução do FreeRTOS. Ajustes na configuração de microcontroladores podem trazer benefícios em termos de consumo de energia em detrimento da degradação de desempenho. Esse trabalho apresenta uma análise detalhada do impacto da reprogramação do *clock* do microcontrolador ATmega328P na previsibilidade do FreeRTOS no Arduino Uno. Os principais recursos do sistema foram criteriosamente avaliados através de casos de teste. Os resultados demonstram que há, na maioria dos casos, uma relação linear entre ajustes no *clock* da CPU e os atrasos dos serviços. A depender dos requisitos temporais da aplicação, pode-se obter reduções significativas no consumo de energia ao se ajustar o *clock* da CPU apropriadamente.

CCS Concepts

•Computer systems organization → Real-time operating systems;

Keywords

FreeRTOS; Arduino Uno; peformance analysis; ATmega328P

1. INTRODUÇÃO

Sistemas Operacionais (SOs) embarcados são projetados para dispositivos com recursos limitados de processamento, armazenamento e energia, geralmente executando funções específicas [6]. Segundo Tanenbaum [13], a principal propriedade desses sistemas é a certeza de que nenhum outro *software*, a não ser o originalmente carregado na memória (*i.e.*, *firmware*), será executado nesses dispositivos, facilitando o projeto e a proteção do sistema. Como os recursos do *hardware* geralmente são bastante limitados, os maiores desafios para o desenvolvimento de sistemas embarcados é a gerência adequada dos recursos e do consumo de energia que, segundo Simonovic [10], tem se tornado uma grande preocupação em sistemas embarcados.

Um Sistema Operacional em Tempo Real (*Real Time Operating System*, RTOS) diferencia-se dos demais por prover mecanismos que garantem o tratamento de eventos com prazos delimitados [9]. Ou seja, a principal característica de um RTOS é a previsibilidade, pois todos os prazos devem ser cumpridos independente das circunstância, onde a corretude do sistema não depende somente dos seus resultados, mas também do tempo em que esses resultados são produzidos.

Os sistemas RTOS podem ser divididos em categorias. Tanenbaum [13] divide-os em duas categorias: sistemas de tempo real críticos, que devem garantir que uma ação ocorrerá no tempo determinado ou dentro de um intervalo de tempo; e os sistemas de tempo real não críticos, onde uma certa taxa de atrasos ou perda de prazos é aceitável, uma vez que isso não causará danos permanentes, apesar de não ser o desejável.

Já Christofferson [4], divide os RTOS em três categorias de acordo com os tipos de aplicações:

- *Hard real-time applications*: prazos são cruciais e se não forem alcançados inutilizam totalmente o sistema, podendo-se causar danos irreversíveis (*e.g.*, sistemas de controle de motores de carros);
- *Firm real-time applications*: caso o prazo não seja cumprido, não há comprometimento total do sistema, apenas produção de resultados inúteis (*e.g.*, sistemas de processamento de sinais de rádio);
- *Soft real-time applications*: caso os prazos não forem alcançados, não há danos graves, apenas acarretando degradação de desempenho (*e.g.*, servidores de rede, onde a perda parcial do dados vai apenas diminuir a qualidade do serviço, mas a aplicação vai continuar utilizável até certa taxa de perdas).

Esse trabalho tem seu foco no FreeRTOS [3], um RTOS de código aberto para sistemas embarcados. Como plataforma de *hardware*, tem-se a atenção voltada ao Arduino Uno, comumente adotada na prototipação de *hardware* e desenvolvimento de aplicações embarcadas. Trata-se de uma plataforma bastante limitada em recursos de processamento e armazenamento, exigindo-se máxima atenção à otimização de recursos.

Nesse contexto, explora-se o FreeRTOS no Arduino Uno considerando ajustes no *clock* do microcontrolador ATmega328p: a redução no *clock* da CPU resulta em redução no consumo de energia em detrimento da degradação de desempenho. Tem-se então como principal objetivo e contribuição, apresentar casos de teste para avaliar o impacto de tais ajustes na previsibilidade do SO nessa

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.

plataforma. A depender dos impactos previstos, pode-se ajustar o modo de operação do microcontrolador garantindo-se um menor consumo de energia e o atendimento dos requisitos temporais de uma aplicação.

O restante desse artigo está organizado da seguinte forma: a Seção 2 apresenta uma breve descrição do FreeRTOS e seus principais serviços; a Seção 3 aborda aspectos essenciais da plataforma Arduino Uno; a Seção 4 trata dos casos de teste para avaliação de desempenho, enquanto a Seção 5 apresenta os resultados e uma análise destes; a Seção 6 discorre sobre alguns dos trabalhos relacionados; e, por final, a Seção 7 aborda as principais conclusões desse trabalho.

2. FREERTOS

O FreeRTOS [3] é um RTOS de código aberto, focado em microcontroladores e pequenos microprocessadores, desenvolvido originalmente pela *Real Time Engineers Ltd.* e, atualmente, mantido pela *Amazon*™.

Esse RTOS é um sistema multitarefas; ou seja, ele permite a criação de múltiplas tarefas independentes e com prioridades distintas. O escalonamento de tarefas é baseado na prioridade destas, onde as tarefas com maior prioridade são executadas primeiro, deixando-se a cargo do desenvolvedor impedir situações de inanição (*starvation*). A priorização das tarefas pode ser bastante útil caso um sistema possua, simultaneamente, tarefas críticas e não críticas, atribuindo-se prioridades mais altas às tarefas críticas.

Trata-se de um RTOS com código aberto desenvolvido em linguagem C, suportando uma ampla gama de plataformas de *hardware*. A disponibilidade do código permite adequações a plataformas não suportadas oficialmente, como é o caso do *port* desenvolvido por [12] (disponibilizado na plataforma *GitHub*) para a plataforma de prototipação Arduino™Uno.

A seguir, apresenta-se os principais recursos do FreeRTOS no seu *port* para o Arduino Uno.

2.1 Gerenciamento de memória

Para versões posteriores à 9.0.0, os objetos do *kernel* podem ser alocados dinamicamente (durante a execução do sistema) ou estaticamente (durante sua compilação). A alocação estática pode se mostrar necessária, sobretudo, em plataformas com memória limitada, pois o gerenciamento de uma *heap* demandaria para si uma fatia significativa da memória total.

Na versão completa do FreeRTOS, há cinco implementações disponíveis para alocação dinâmica de memória, deixando a cargo do desenvolvedor a sua adoção, ou não. No entanto, no caso específico do *port* para o Arduino Uno, tem-se apenas a variante *"Heap_3"*. Ela disponibiliza duas funções: ***pvPortMalloc***, para alocar uma quantidade de *bytes* na memória, e a ***vPortFree***, para liberar um espaço na memória. Essas funções são versões seguras, pois suspendem, temporariamente, o escalonador de tarefas, evitando-se que outras tarefas recebam a CPU durante a alocação ou liberação da memória.

2.2 Tarefas

As tarefas são implementadas em forma de funções do tipo *void* e recebem como parâmetro um ponteiro também do tipo *void*. Múltiplas tarefas podem ser instanciadas a partir de uma mesma função de entrada. Assume-se nenhuma função de retorno das tarefas, sendo geralmente executadas em um ciclo (*loop*) infinito e, quando não são mais necessárias, devem ser excluídas via uma chamada de sistema específica.

Para criar uma tarefa exitem duas chamadas de sistema: a ***xTaskCreate*** e a ***xTaskCreateStatic***, esta última utilizada apenas quando empregada a alocação estática de memória para a tarefa durante a compilação. Para que seja possível fazer um gerenciamento apropriado das tarefas, tem-se os seguintes estados para as tarefas:

- Bloqueada: a tarefa está aguardando pelo retorno de uma chamada de sistema bloqueante (*e.g.*, um tempo de espera programado, aguardando por uma mensagem em uma fila, bloqueio em um *mutex* ou semáforo);
- Pronta: a tarefa está pronta e aguardando para ser executada pelo processador;
- Em execução: a tarefa está sendo executada pelo processador;
- Suspensa: a tarefa é suspensa por requisição explícita do usuário, deixando esse estado somente via uma chamada específica solicitada por outra tarefa.

O escalonador de tarefas padrão do FreeRTOS é baseado em múltiplas filas, uma para cada nível de prioridades estabelecidas, selecionando-se sempre primeiro as tarefas que possuem prioridades mais elevadas; portanto, para se evitar a inanição de tarefas de baixa prioridade, a aplicação deve garantir que as tarefas de mais alta prioridade não permaneçam sempre no estado pronto. A troca de tarefas na CPU é chamada de troca de contexto, o que envolve salvar o contexto (*i.e.*, estado atual da CPU) da tarefa corrente e recuperar o contexto salvo da tarefa selecionada para ocupar a CPU.

2.3 Filas

Uma fila é uma estrutura de armazenamento de dados onde tarefas ou rotinas de serviço de interrupção (*Interrupt Service Routine*, ISR) inserem e/ou obtêm dados. As filas, como o nome já sugere, tem como característica principal a ordem de entrada e saída dos dados: novos dados são inseridos ao final da fila, enquanto a extração ocorre a partir do início da fila. No entanto, o FreeRTOS oferece também a possibilidade de inserção no início da fila (comportamento típico de pilhas).

O FreeRTOS permite também a criação de conjuntos de filas, onde é possível integrar várias filas ou semáforos. Isso permite que uma tarefa possa receber dados de diferentes filas, ou semáforos, sem a necessidade de ficar sondando cada uma individualmente: retorna-se um identificador para acessar diretamente a fila ou semáforo que contém dados.

As funções para manipulação de filas são: criação; inserção (tanto no final como no início da fila); retirada; leitura sem retirada; e número de itens presentes na fila. Caso uma tarefa tentar retirar um elemento da fila e a mesma estiver vazia, a tarefa pode entrar em estado bloqueado por um tempo determinado ou indeterminado. O mesmo pode ocorrer caso uma tarefa tentar inserir um elemento em uma fila cheia.

2.4 Temporizadores de software

Um temporizador de *software* permite a execução de uma função, denominada de “função de retorno de chamada do temporizador”, após transcorrido um tempo pré-determinado ou periodicamente. Geralmente uma função de retorno é breve, sempre executada até o seu fim sem bloquer. Para isso, dentro da função de retorno devem ser utilizadas as funções seguras do FreeRTOS[1].

Existem dois tipos de temporizadores: os temporizadores únicos, que executam a função de retorno uma única vez e não reiniciam

[1]Funções seguras são funções que não entram no estado bloqueado. A maioria dos recursos do FreeRTOS implementam também versões seguras das suas funções, as quais adotam na sua nomenclatura o sufixo *FromISR*.

automaticamente, mas podem ser reiniciados manualmente; e os temporizadores de recarregamento automático, os quais são automaticamente reinicializados a cada vez que expiram, executando a função de retorno de forma periódica.

2.5 Gerenciamento de interrupções

O tratamento de uma interrupção envolve a execução de uma ISR. Recomenda-se que o código da ISR seja o mais breve possível e, caso seja necessário, a continuação do tratamento da interrupção poderá ser repassada a uma tarefa convencional. Vale ressaltar que dentro do código de uma ISR deverão ser empregadas as funções seguras do FreeRTOS, pois eventuais bloqueio impediriam o retorno da própria ISR.

As interrupções são controladas por *hardware* e, assim como as tarefas, também possuem prioridades; porém, uma tarefa, por mais alta que seja sua prioridade, nunca irá ser executada enquanto houver interrupções para tratar, por mais baixas que sejam suas prioridades. Portanto, em havendo muitas interrupções consecutivas e/ou com código muito extenso, pode ser que tarefas que tratam interrupções mais prioritárias e pretéritas não sejam executadas no tempo esperado. Para que seja possível o encaminhamento do tratamento das interrupções para as tarefas, o FreeRTOS implementa alguns recursos para sincronização como, por exemplo, os semáforos.

2.6 Semáforos binários

Um semáforo binário é um recurso que pode ser empregado para sincronização e que, no FreeRTOS, é tratado como um tipo especial de fila a qual pode conter, no máximo, um elemento; portanto, ela está vazia ou cheia. Para os semáforos, não importa o que há na fila, mas se há ou não um elemento nela.

Um semáforo binário pode ser utilizado para que, por exemplo, uma rotina de interrupção comunique a uma tarefa convencional a ocorrência de determinada interrupção, delegando à tarefa a responsabilidade de continuar o tratamento da interrupção. Neste cenário, a rotina de interrupção é responsável por carregar o semáforo, enquanto a tarefa aguarda bloqueada à espera da sinalização no respectivo semáforo.

Trata-se de um recurso relativamente simples e capaz de sinalizar um único evento; ou seja, caso uma interrupção aconteça consecutivamente e a tarefa ainda tem uma notificação pendente, ela não tomará conhecimento de uma interrupção subsequente, pois o semáforo já está cheio. Isso pode, ou não, ser um problema dependendo da aplicação mas, de qualquer forma, pode ser contornado empregando-se os semáforos de contagem.

2.7 Semáforos de contagem

Os semáforos de contagem são semelhantes aos semáforos binários, tendo-se como única diferença o fato deles serem tratados como filas especiais de tamanho variável. Com eles, torna-se possível sinalizar à tarefa que processa uma interrupção o número de ocorrências de uma determinada interrupção.

2.8 Gerenciamento de recursos

Em um sistema multitarefas onde várias tarefas compartilham um mesmo recurso como, por exemplo, os periféricos e regiões de memória, torna-se necessária a implementação de gerenciadores de recursos, evitando-se resultados inconsistentes. Nesse caso, necessita-se de mecanismos para a exclusão mútua, garantindo-se que apenas uma tarefa tenha acesso ao recurso compartilhado. O código que envolve o acesso aos recursos compartilhados é conhecido como região crítica. O FreeRTOS disponibiliza dois mecanismos para controle no acesso à região crítica:

- Suspensão do escalonador: impede que haja troca de contexto e outra tarefa assuma o processador; ou seja, essa será a única tarefa em execução até que ela determine a retomada o escalonador. Vale ressaltar que esse mecanismo não desativa as interrupções; ou seja, ISRs ainda poderão acessar a região crítica.
- *Mutex*: funciona como uma permissão para poder acessar a região crítica. Para que uma tarefa possa acessar a região crítica, torna-se necessário primeiro adquirir o *mutex* e, caso este não esteja disponível, a tarefa solicitante bloqueia. Uma tarefa deve sempre entregar o *mutex* ao sair da região crítica, para que outra tarefa possa obtê-lo. Caso a tarefa não devolva o *mutex*, as tarefas que estão bloqueadas à sua espera poderão entrar em *deadlock*[2] e, caso ela mesma tentar adquirir novamente o *mutex*, também entrará em *deadlock* (exceção somente no caso de *mutexes* recursivos). Vale a ressalva que o controle de acesso à região crítica é de inteira responsabilidade do desenvolvedor da aplicação.

2.9 Grupos de eventos

Um grupo de eventos é gerenciado via um mapa de *bits* (*bitmap*), armazenado em uma única variável: quando o valor do *bit* for igual a 1 sinaliza a ocorrência de um determinado evento, e 0 do contrário. Um grupo de eventos também pode ser empregado na sincronização e comunicação entre tarefas e ISR.

Uma tarefa ou ISR pode notificar a ocorrência de um evento atribuindo valor 1 ao bit correspondente ao evento, enquanto outras tarefas ficam bloqueadas à espera do respectivo evento. Caso uma tarefa verifique o grupo de eventos e o evento de interesse ainda não tenha ocorrido, ala pode aguardar em estado bloqueado até que o evento ocorra ou por um tempo pré-determinado.

2.10 Notificações de tarefas

Diferente dos recursos vistos até agora, que permitem comunicação entre tarefas através de objetos intermediários, as notificações de tarefas possibilitam a comunicação direta entre as tarefas ou entre ISR e tarefas. As notificações de tarefas ocorrem via uma variável de notificação/controle presente em cada tarefa.

Trata-se de um recurso de comunicação mais rápido que os demais; porém, há limitações: i) o volume de informação compartilhado está limitado ao tamanho da variável de notificação; ii) não é possível enviar uma notificação de uma tarefa para uma ISR; iii) apenas uma tarefa pode ser notificada por vez; e, iv) diferente das filas, a tarefa de envio não poderá entrar em estado bloqueado caso haja notificação pendente na tarefa de recebimento.

3. ARDUINO UNO

O Arduino Uno [1] é uma placa de prototipação de *hardware* de projeto aberto, baseada no microcontrolador ATmega328P™. Possui 14 pinos de entrada e saída digitais e 6 pinos de entrada analógica, uma conexão USB, que permite a comunicação entre o computador e a placa, possibilitando a gravação do *software* na memória do microcontrolador. Há também um conector de fonte de energia[3] e um botão de reset. O microcontrolador ATmega328P™é programável e empregado, sobretudo, no desenvolvimento de sistemas com uma única aplicacão embarcada (Tabela 1 mostra as principais especificações do microcontrolador).

[2] Quando dois ou mais processos ficam em estado bloqueado aguardando uns pelos outros.

[3] A placa pode ser alimentada tanto pelo conector de energia quanto pela conexão USB.

[4] Sendo que 0,5 KB estão reservados para o *bootloader* (código responsável por fazer a inicialização do sistema).

Table 1: Especificações básicas do ATmega328P™ [2].

Memória *Flash*	32 KB [4]
Memória SRAM	2 KB
Memória EEPROM	1 KB
Clock	16 MHz
Tensão de operação	2.7V à 5.5V

Table 2: Fator de divisão do *clock* do ATmega328p™ [2].

Fator de divisão	*Clock* da CPU
1	16 MHz
2	8 MHz
4	4 MHz
8	2 MHz
16	1 MHz
32	500 kHz
64	250 kHz
128	125 kHz
256	62,5 kHz

3.1 IDE e Monitor Serial

O arduino disponibiliza seu próprio ambiente de desenvolvimento integrado (*Integrated Development Environment*, IDE) para programação das placas microcontroladoras. A linguagem utilizada é um conjunto de comandos da linguagem C, C++ e alguns comandos específicos. A IDE está disponível tanto para *desktop* quanto *online*, oferecendo a possibilidade de compilaçao e carga (*upload*) do aplicativo direto para a placa. A IDE *online* permite salvar os códigos na nuvem e compartilhá-los; porém, para fazer o *upload* para a placa é necessário instalar um *plugin* específico.

As duas IDE's contêm praticamente os mesmos recursos e permitem a instalação de bibliotecas externas como, por exemplo, o *port* do FreeRTOS para o Arduino Uno. Além disso, fornecem um monitor serial, que é uma ferramenta utilizada para a comunicação entre o computador e a placa via porta USB.

3.2 CLKPR e Consumo de Energia

O microprocessador ATMega328P™possui um registrador denominado *Clock Prescale Register* (CLKPR), que pode ser usado para alterar o relógio (*clock*) da CPU e, consequentemente, o seu consumo de energia. O CLKPR divide o *clock* por valores específicos chamados fatores de divisão, os quais estão discriminados na Tabela 2.

O *datasheet* [2] não apresenta dados sobre o consumo de energia para todas as frequências de *clock*; porém, ele discrimina o consumo referente às frequências de 4, 8 e 16 MHz (vide Tabela 3), onde o consumo médio é obtido com a placa operando a uma temperatura de 25°C.

A biblioteca *avr-libc* [5] [5] permite a alteração do *clock* a partir da função *void clock_prescale_set (clock_div_t __x)*, a qual recebe como parâmetro *__x* o fator de divisão, onde este pode assumir os seguintes valores:

```
typedef enum
{
    clock_div_1 = 1,
    clock_div_2 = 2,
    clock_div_4 = 4,
    clock_div_8 = 8,
    clock_div_16 = 16,
    clock_div_32 = 32,
    clock_div_64 = 64,
    clock_div_128 = 128
} clock_div_t;
```

[5]Incluída via <*avr/power.h*>.

Table 3: Consumo (corrente) para algumas taxas de *clock* do ATmega328p™ [2].

Clock	Tensão	Consumo médio(25°C)	Consumo máximo
4 MHz	3V	1,5 mA	2,4 mA
8 MHz	5V	5,2 mA	10 mA
16 MHz	5V	9,2 mA	14 mA

4. CASOS DE TESTE DE DESEMPENHO

Para Sacha [8], os requisitos essenciais em sistemas de tempo real são a pontualidade e a confiabilidade, que podem ser atendidos aplicando-se os princípios de multitarefas, comunicação e sincronização eficiente de tarefas e manipulação eficiente de tempo e eventos.

Ao empregar ajustes ao CLKPR com o propósito de diminuir o consumo de energia, o atraso nos serviços do FreeRTOS pode aumentar e, consequentemente, afetar requisitos temporais da aplicação. Com o propósito de mensurar tais impactos, cenários de testes foram projetados para mensurar o atraso envolvido na criação e manipulação de recursos, na troca de contexto e temporizadores de *software*. Para garantir o isolamento necessário aos testes, durante sua execução não há tarefas ou rotinas alheias às envolvidas diretamente na avaliação. Os arquivos de todos os testes estão disponíveis publicamente na plataforma Zenodo[6].

4.1 Criação e exclusão dos recursos

Os testes de criação e exclusão seguiram o modelo de código ilustrado na Figura 1, que apresenta o esqueleto da tarefa principal, responsável por medir o tempo de criação e exclusão dos recursos. O tipo da variável *xRecurso* é relativo a cada recurso testado e representa a referência ao mesmo. As variáveis *inicio* e *fim* são empregadas para armazenar os instantes de tempo antes e após a criação ou remoção do recurso. A remoção somente é executada caso a criação tenha sido bem sucedida. O processo se repete pelo número de iterações estabelecido e, ao final, os valores médios são obtidos e enviados via porta serial para armazenamento.

4.2 Manipulação dos recursos

4.2.1 Comunicação entre tarefas

Segundo Sacha [8], duas ações devem ser executadas entre o início do envio de uma mensagem e o final do seu recebimento: a entrega da mensagem pelo SO e a alternância das tarefas. Ele sugere que se deve medir o tempo total da operação, pois corresponde a menor fração que pode ser efetivamente observada, e a sua discriminação em partes é irrelevante para o desenvolvedor de uma aplicação.

Os testes projetados para analisar a comunicação entre tarefas seguem o modelo de Sacha [8]. A Figura 2 apresenta o esqueleto segundo esse modelo, centrado em duas tarefas com a mesma prioridade. Há uma tarefa principal que tem como objetivo calcular o tempo médio de envio de uma mensagem: em um laço finito,

[6]Referência omitida durante o processo de revisão em concordância com o processo de revisão *double blindedreview*.

```
void vTarefaPrincipal(void *){
  unsigned int inicio = 0, fim = 0, i = ITERACOES;
  float mediaTempoCriacao = 0.0, mediaTempoExclusao = 0.0;

  do{
    //Declaração das variáveis necessárias de acordo com cada recurso
    inicio = micros();
    //Espaço para executar a função de criação do recurso
    fim = micros();

    if(xRecurso != NULL){
      mediaTempoCriacao += (float)(fim - inicio);
      inicio = micros();
      //Espaço para executar a função de exclusão do recurso
      fim = micros();
      mediaTempoExclusao += (float)(fim - inicio);
      xRecurso = NULL;
    }else
      i++;
  }while(i--);

  mediaTempoCriacao /= (float)ITERACOES;

  mediaTempoExclusao /= (float)ITERACOES;

  Serial.print("Tempo de criação: ");
  Serial.println(mediaTempoCriacao);
  Serial.print("Tempo de exclusão: ");
  Serial.println(mediaTempoExclusao);
  vTaskDelete(NULL);
}
```

Figure 1: Modelo de implementação dos testes de criação e exclusão dos recursos

executa-se o envio de uma mensagem para uma tarefa secundária e, em seguida, aguarda-se resposta desta. A tarefa secundária permanece em um laço infinito, à espera de uma mensagem da tarefa primária. Logo após receber uma mensagem da tarefa primária, a tarefa secundária envia a mesma mensagem de volta à tarefa primária. Após a execução de todas as iterações programadas na tarefa principal, obtem-se o tempo médio agregado referente ao envio e recepção (*i.e.*, ida e volta), bastando calcular a média aritmética para discriminar o tempo médio de envio de uma mensagem.

Os casos de teste de comunicação englobam os seguintes recursos:

- **Filas**. Duas filas são utilizadas: a *xFila1*, empregada para mandar a mensagem à tarefa secundária (pela tarefa primária) e a *xFila2*, empregada para mandar a mensagem à tarefa primária (pela tarefa secundária).
- **Grupo de eventos**. Dois grupos de eventos foram usados. No mesmo esquema das filas, o *xGrupoDeEventos1* é manipulado pela tarefa principal e aguardado pela tarefa secundária; enquanto o *xGrupoDeEventos2* é manipulado pela tarefa secundária e aguardado pela tarefa primária, realizando-se assim a comunicação referente a ocorrência dos respectivos eventos.
- **Notificação de tarefas**. Diferente dos recursos anteriores, a comunicação ocorre sem entidades intermediárias, necessitando-se apenas ter conhecimento dos identificadores das tarefas envolvidas na comunicação. Existem dois modelos de notificação de tarefas:
  - O modelo básico: a tarefa é apenas notificada da ocorrência de um evento, não sendo possível enviar dados para a tarefa. Esse modelo é implementado usando a função *xTaskNotifyGive* para enviar a notificação e *ulTaskNotifyTake* para receber a notificação.
  - O modelo configurável: é implementado via função *xTaskNotify* para enviar uma notificação e a função *xTaskNotifyWait* para receber a notificação. Esse modelo é configurável porque contempla vários modos de ação:
    - ∗ *eNoAction* - Executa a mesma função do modelo básico, apenas informando a tarefa que há uma notificação, mas sem alterar o valor da respectiva variável de notificação da tarefa.
    - ∗ *eSetBits* - Realiza um *OR* lógico *bit* a *bit* entre o valor atual de notificação da tarefa e o valor passado no parâmetro da função de notificação (*i.e.*, *ulValue*).
    - ∗ *eIncrement* - O valor da variável de notificação da tarefa de recebimento é incrementado em 1 (o parâmetro *ulValue* não é empregado).
    - ∗ *eSetValueWithoutOverwrite* - Caso a tarefa de recebimento esteja com uma notificação pendente, nenhuma ação será executada; caso contrário, o valor de notificação da tarefa de recebimento receberá o valor passado no parâmetro *ulValue*.
    - ∗ *eSetValueWithOverwrite* - O valor da variável de notificação da tarefa recebe o valor passado em *ulValue*, independente de a tarefa de recebimento ter ou não uma notificação pendente.

4.2.2 Comunicação entre ISR e tarefas

Caso uma ISR se torne muito complexa, recomenda-se passar a carga extra de processamento a uma tarefa convencional. Para que a ISR sinalize a tarefa correspondente, os recursos de comunicação entre tarefas e semáforos podem ser empregados. Para Sacha [8], a operação de sinalização é constituída pelas instruções de geração de sinal, entrega de sinal, captura de sinal e alternância de tarefas.

Os casos de teste para analisar a sinalização de uma ISR a uma tarefa seguem o modelo/esqueleto apresentado na Figura 2: há uma tarefa primária que fica sempre no estado bloqueado à espera de uma sinalização da ISR, uma tarefa secundária que é responsável por reproduzir a interrupção de *hardware* via *software*, e uma ISR que, logo após ser ativada, sinaliza a tarefa primária. A ISR registra o instante de tempo antes de enviar o sinal à tarefa que, por sua vez, registra o instante de tempo após o recebimento do sinal. O ciclo de sinalização é repetido várias vezes para se obter um tempo médio de sinalização. Destaca-se que a tarefa secundária deve ter prioridade inferior à tarefa primária, a qual, por padrão, permanece no estado bloqueado à espera da sinalização da ISR.

Para a realização dos casos de teste, foram empregados os seguintes recursos:

- **Fila**. Utilizou-se uma única fila, empregada para enviar um sinal da ISR à tarefa primária.
- **Grupo de eventos**. Um grupo de eventos foi utilizado de forma análoga à fila (*i.e.*, para enviar um sinal da ISR à tarefa primária).
- **Notificação de tarefas**. A tarefa primária é sinalizada/desbloqueada via notificação de tarefas pela ISR. Os dois modelos suportados foram analisados.
- **Semáforos**. Como o semáforo é um mecanismo de autorização/desbloqueio, ele também pode ser empregado como meio para sinalizar a ocorrência de um evento. Semelhante aos cenários anteriores, a tarefa principal aguarda no semáforo à espera de um desbloqueio pela ISR. Foram testados os semáforos binários e contadores.

```
void vTarefaPrincipal(void *){
  volatile uint32_t i = ITERACOES;
  uint32_t inicio = 0, fim = 0;
  float mediaTempo = 0.0;

  inicio = micros();
  do{
    //Envia a mensagem para a tarefa secundária
    //Aguarda pela mensagem da tarefa secundária
  }while(i--);
  fim = micros();

  mediaTempo = float(fim - inicio) / (float)ITERACOES;
  mediaTempo /= 2.;

  Serial.println(mediaTempo);
  vTaskDelete(NULL);
}
```

```
void vTarefaSecundaria(void *){
  do{
    //Aguarda pela mensagem da tarefa primária
    //Envia a mensagem para a tarefa primária
  }while(1);
}
```

Figure 2: Modelo de implementação dos testes de comunicação entre tarefas

```
uint32_t somaTempo = 0;
void vTarefaPrincipal(void *){
  volatile uint32_t i = ITERACOES;

  do{
    //Aguarda pelo sinal da ISR
    somaTempo += micros();
  }while(i--);

  float mediaTempo = float(somaTempo) / (float)ITERACOES;

  Serial.println(mediaTempo);
  vTaskDelete(NULL);
}
```

```
void vTarefaSecundaria(void *){
  do{
    digitalWrite(pinoInterrupcao, LOW);
    digitalWrite(pinoInterrupcao, HIGH);
    taskYIELD();
  }while(1);
}
```

```
static void ulTrataInterrupcaoISR(void){
    BaseType_t xHigherPriorityTaskWoken = pdTRUE;
    somaTempo -= micros();
    //Envia o sinal para a tarefa primária
}
```

Figure 3: Modelo de implementação dos testes de sinalização de uma ISR a uma tarefa

```
EventGroupHandle_t xGrupoDeEventos;
#define BIT_TAREFA (1UL << b)

void vTarefaSincronizacao(void *pvParameters){
  volatile uint32_t i = ITERACOES;
  uint32_t inicio = 0, fim = 0;
  float mediaTempo = 0.0;

  EventBits_t uxBitDaTarefa = (EventBits_t) pvParameters;

  EventBits_t uxTodosBitsDeSicronizacao;
  for(unsigned long b = 0; b < QNT_TAREFAS; b++)
    uxTodosBitsDeSicronizacao |= BIT_TAREFA;

  xEventGroupSync(xGrupoDeEventos, uxBitDaTarefa, uxTodosBitsDeSicronizacao, portMAX_DELAY);

  inicio = micros();
  do{
    xEventGroupSync(xGrupoDeEventos, uxBitDaTarefa, uxTodosBitsDeSicronizacao, portMAX_DELAY);
  }while(i--);
  fim = micros();

  xEventGroupSync(xGrupoDeEventos, uxBitDaTarefa, uxTodosBitsDeSicronizacao, portMAX_DELAY);

  mediaTempo = float(fim - inicio) / (float)ITERACOES;

  vTaskSuspendAll();
  Serial.println(mediaTempo);
  xTaskResumeAll();
  vTaskDelete(NULL);
}
```

Figure 4: Implementação do teste de sincronização de tarefas

4.2.3 *Sincronização de tarefas*

A sincronização de tarefas foi avaliada utilizando a chamada *xEventGroupSync*, que é baseada em grupo de eventos. Esse mecanismo permite que múltiplas tarefas sincronizem suas ações: quando uma tarefa executa a chamada à função de sincronização, permanece em estado bloqueado (por tempo indeterminado ou por um tempo limite de espera) até que todas as demais tarefas do mesmo grupo executem a função de sincronização.

A Figura 4 apresenta o modelo de tarefa para os casos de teste de sincronização. A tarefa atribui à variável *uxBitDaTarefa* o valor recebido por parâmetro, representando o *bit* da tarefa no grupo de eventos. A variável *uxTodosBitsDeSicronizacao* contém os *bits* de todas as tarefas envolvidas na sincronização. Inicialmente, cada tarefa bloqueia à espera de todas as demais: o próprio grupo de eventos é utilizado para preparar todas as tarefas para ingressar no laço de medida de tempo de sincronização simultaneamente. O procedimento de sincronização entre todas as tarefas se dá repetidas vezes para se obter uma amostragem de medidas. Após o laço, o tempo médio de um ciclo de sincronização é calculado baseado no número de amostras (*i.e.*, iterações do laço).

4.2.4 *Aquisição e entrega de mutex e semáforos*

Desenvolveu-se casos de teste para avaliar o tempo de aquisição e entrega de *mutex*, semáforo binário e semáforo contador. A Figura 5 apresenta o modelo de código para os testes. A tarefa entra em um laço que executa a chamada de aquisição e devolução do recurso, registrando-se os respectivos atrasos envolvidos. Após a execução do laço repetidamente, calcula-se os tempos médios de aquisição e devolução do respectivo recurso.

4.3 Troca de contexto

O atraso decorrente da alternância de tarefas na CPU (*context switch*) é uma métrica de desempenho relevante para qualquer SO multitarefa. Para medição desse atraso, utilizou-se o modelo proposto por Sacha [8], o qual está ilustrado no código da Figura 6.

O caso de teste possui duas tarefas, uma primária e uma secundária, ambas com a mesma prioridade. A tarefa principal é responsável por registrar o atraso. Primeiramente, ela mede o tempo médio de execução de um laço vazio. Em seguida, mede-se o tempo médio da troca de contexto: em cada iteração do laço, a tarefa principal voluntariamente entrega a CPU (via chamada de sistema *taskYIELD*), forçando o escalonador da CPU a alternar para a terefa secundária que, por sua vez, tem como único objetivo devolver, imediatamente, a CPU para a tarefa principal. Portanto, para cada troca de contexto realizada entre a tarefa principal e a secundária, outra é realizada em contrapartida. O cálculo final é realizado dividindo-se o tempo total pela quantidade de iterações e dividindo esse resultado por 2 (*i.e.*, devido às duas trocas de contexto em cada iteração). No entanto, deve-se subtrair desse resultado o tempo médio de execução do laço em si, obtendo-se como resultado o tempo médio de uma troca de contexto.

Figure 5: Modelo de implementação dos testes de aquisição e entrega do *mutex* e dos semáforos

```
void vTarefaPrincipal(void *){
  volatile uint32_t i = ITERACOES;
  uint32_t inicio = 0, fim = 0;
  float mediaTempoAquisicao = 0.0, mediaTempoEntrega = 0.0;

  do{
    inicio = micros();
    //Adquire o recurso
    fim = micros();
    mediaTempoAquisicao += float(fim - inicio);
    inicio = micros();
    //Entrega o recurso
    fim = micros();
    mediaTempoEntrega += float(fim - inicio);
  }while(i--);

  mediaTempoAquisicao /= (float)ITERACOES;

  mediaTempoEntrega /= (float)ITERACOES;

  Serial.print("Aquisicao: ");
  Serial.print(mediaTempoAquisicao);
  Serial.print(" - Entrega: ");
  Serial.println(mediaTempoEntrega);
  vTaskDelete(NULL);
}
```

4.4 Temporizadores de software

A Figura 7 mostra a implementação da função que é executada pelo temporizador cada vez que ele expirar. A variável *TEMPO_TICKS* armazena o tempo do temporizador em *ticks*[7]. O temporizador empregado nesse caso de teste foi um de recarregamento automático, o qual executa a função de retorno *FuncaoRetorno*. A função de retorno tem como objetivo enviar para o monitor serial o tempo interno do sistema em microssegundos, para que se possa registrar a diferença de tempo entre as execuções consecutivas do temporizador. Após 51 execuções da função (*i.e.*, 50 amostras de medidas de tempo entre as execuções do temporizador), o temporizador é parado, o intervalo de tempo é incrementado em 1 *tick* e o temporizador é reiniciado.

Tarefas auxiliares (*vTarefaX*) foram instanciadas com o objetivo de disputar a CPU com o *daemon* do FreeRTOS[8], com o intuito de analisar como os temporizadores se comportam caso o *daemon* consiga acesso a CPU fora do prazo de expiração dos temporizadores.

5. RESULTADOS E ANÁLISE

[7]Interrupção de *hardware* que ocorre de forma periódica. Todas as funções que envolvem tempo no FreeRTOS são expressadas em *ticks*. Há chamadas do sistema que permitem converter um quantitativo de *ticks* em tempo real e vice-versa.

[8]Tarefa do sistema, instanciada pelo FreeRTOS, responsável por controlar a expiração dos temporizadores de *software* e a execução da função de retorno.

```
void vTarefaPrincipal(void *){
  volatile uint32_t i = ITERACOES;
  uint32_t inicio = 0, fim = 0;
  float mediaTempo = 0.0, mediaTempoLaco = 0.0;

  inicio = micros();
  do{
  }while(i--);
  fim = micros();

  mediaTempoLaco = float(fim - inicio) / float(ITERACOES);

  i = ITERACOES;

  inicio = micros();
  do{
    taskYIELD();
  }while(i--);
  fim = micros();

  mediaTempo = ((float(fim - inicio) / (float)ITERACOES)
               / 2.0);
  mediaTempo = mediaTempo - mediaTempoLaco;

  Serial.println(mediaTempo);
  vTaskDelete(NULL);
}
```

```
void vTarefaSecundaria(void *){
  do{
    taskYIELD();
  }while(1);
}
```

Figure 6: Implementação do teste da troca de contexto

```
volatile uint16_t TEMPO_TICKS, i;
TimerHandle_t xTimer;

void FuncaoRetorno(TimerHandle_t xTimer){
  Serial.println(micros());
  i++;
  if(i >= 51){
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xTimerStopFromISR(xTimer, &xHigherPriorityTaskWoken);
    i = 0;
    TEMPO_TICKS++;
    Serial.print("Tempo : ");
    Serial.print(TEMPO_TICKS * portTICK_PERIOD_MS);
    Serial.println(" ms");
    xTimerChangePeriodFromISR(xTimer, TEMPO_TICKS, &xHigherPriorityTaskWoken);
  }

}
```

```
void vTarefaX(void *){
  for( ;; );
}
```

Figure 7: Implementação do teste dos temporizadores de *software*

Table 4: Tempo de criação e exclusão das filas

CRIAÇÃO DE FILAS						
Itens	Tamanho	Tempo médio (μs)				
		16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
1	1 byte	12,33	27,18	54,70	108,67	218,56
	8 bytes	12,25	27,30	54,94	109,73	221,50
	16 bytes	12,28	27,50	55,06	109,57	221,18
8	1 byte	12,28	27,32	54,94	110,11	221,66
	8 bytes	12,28	27,50	55,06	109,38	220,86
	16 bytes	12,30	26,95	55,42	109,06	221,31
16	1 byte	12,28	27,32	55,06	109,38	220,86
	8 bytes	12,30	26,97	55,41	109,09	221,31
	16 bytes	12,27	27,39	55,2	109,41	222,08
EXCLUSÃO DE FILAS						
Itens	Tamanho	Tempo médio (μs)				
		16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
1	1 byte	2,34	7,68	17,01	38,27	74,37
	8 bytes	2,37	7,68	17,39	36,93	72,70
	16 bytes	2,32	7,68	17,41	37,38	74,24
8	1 byte	2,35	7,70	17,39	37,92	72,83
	8 bytes	2,32	7,68	17,41	37,63	74,62
	16 bytes	2,79	7,27	17,04	38,24	73,15
16	1 byte	2,32	7,70	17,41	37,63	74,62
	8 bytes	2,79	7,30	16,86	38,75	73,15
	16 bytes	2,39	7,66	17,2	37,57	73,41

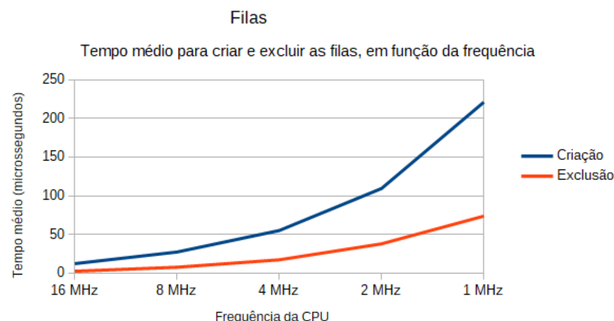

Figure 8: Tempo de criação e exclusão de filas

Table 5: Tempo de criação e exclusão do *mutex* e dos semáforos

Operação	Recurso	Tempo médio (μs)				
		16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
Criação	*Mutex*	21,01	44,23	88,60	181,73	379,07
	S. binário	12,30	26,74	55,28	108,99	220,80
	S. contador	12,24	27,34	55,17	109,22	221,57
Exclusão	*Mutex*	2,60	7,56	16,99	37,98	84,99
	S. binário	2,89	7,76	17,10	38,43	76,99
	S. contador	2,36	7,62	17,25	37,41	74,24

Os casos de teste foram realizados com a versão 10.2.0-3 do FreeRTOS executando no Arduino Uno Rev3, com frequências da CPU iguais a 16 MHz (*clock_div_1*), 8 MHz (*clock_div_2*), 4 MHz (*clock_div_4*), 2 MHz (*clock_div_8*) e 1 MHz (*clock_div_16*). As frequências mais baixas, alcançadas por *clock_div_32*, *clock_div_64* e *clock_div_128*, apesar de estarem disponíveis, não são suportadas pelo Arduino Uno. Há uma constante interna do FreeRTOS, identificada como *clockCyclesPerMicrosecond*, representando a quantidade de ciclos de *clock* por microssegundo, que é obtida do resultado da frequência da CPU dividida por 1.000.000. Como a divisão é realizada entre dois inteiros, frequências menores que 1 MHz arredondam a constante para 0 e, sendo que essa constante é empregada como divisor em outras definições, ela acabaria levando a uma divisão por 0.

Todos os casos de teste referentes a criação e manipulação dos recursos foram realizados dentro de tarefas, com o número de iterações igual a 1000 (inclusive o caso de teste referente a troca de contexto). Os resultados também estão publicamente disponíveis na plataforma Zenodo[9].

5.1 Criação e exclusão de recursos

5.1.1 Filas

A criação e exclusão das filas foram testadas com uma quantidade de elementos iguais a 1, 8 e 16 e com tamanho dos elementos iguais a 1, 8 e 16 bytes. O resultado dos testes, apresentados na Tabela 4, mostram que, como era de se esperar, o número de itens e o tamanho destes não influencia no tempo de criação e exclusão das filas, pois envolve procedimentos de alocação e liberação de memória com atrasos praticamente constantes.

O gráfico da Figura 8 mostra a relação do tempo com a frequência. Pode-se ver que a frequência da CPU impacta diretamente no tempo de criação e exclusão dos itens. Para a criação, o tempo dobra a cada divisão da frequência com exceção aos 8 MHz, que tem um aumento um pouco maior do que o dobro em relação à frequência de 16 MHz. Na exclusão, observou-se um aumento mais rápido na alternância entre as frequências mais altas, com o atraso na frequência de 8 MHz três vezes maior em relação à frequência de 16 MHz; porém, a relação é aproximadamente linear ao alternar entre frequências adjacentes inferiores a 8 MHz.

5.1.2 Mutexes e Semáforos

A Tabela 5 lista os tempos de criação e exclusão dos *mutexes*, semáforos binários e semáforos contadores. Como esses recursos são, internamente, implementados usando filas, os tempos obtidos são semelhantes. Todas as funções de criação desses recursos envolvem a instanciação de filas genéricas. Há, no entanto, uma particularidade envolvendo *mutex*: a criação deste envolve a inclusão de um item na fila, representando simbolicamente a presença do *mutex*; portanto, a sua criação é mais custosa quando comparada aos semáforos. A função de exclusão é a mesma para os três recursos, internamente executada via função de exclusão das filas, resultando em valores semelhantes destas.

5.1.3 Grupos de Eventos

Os resultados para a criação e exclusão de grupos de eventos estão listados na Tabela 6. Na instanciação do recurso, mantém-se o comportamento de avanço linear nos atrasos à medida que se reduz a frequência, observando-se uma maior diferença entre as frequências de 16 e 8 MHz, com um aumento de aproximadamente $2,5$ vezes. Já na exclusão, observa-se um comportamento mais linear e com valores superiores à criação. Dentre os recursos testados nesse trabalho, os grupos de eventos foram os únicos que apresentaram um tempo de exclusão maior que o tempo de criação. Isso ocorre pois, durante a exclusão, ao contrário das filas, o grupo de eventos desbloqueia todas as tarefas que estavam eventualmente aguardando por um dos eventos do grupo.

5.1.4 Tarefas

Ao instanciar uma tarefa, deve-se definir o tamanho da sua pilha. Para os casos de teste realizados, foram consideradas pilhas com 32, 64, 128, 256 e 512 bytes. Os resultados (vide Tabela 7) indicam que o tamanho da pilha impacta diretamente o tempo de criação

[9]Referência omitida durante o processo de revisão em concordância com o processo de revisão *double blindedreview*.

Table 6: Tempo de criação e exclusão dos grupos de eventos

Operação	Tempo médio (μs)				
	16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
Criação	5,31	13,03	25,09	52,22	110,40
Exclusão	10,41	23,27	48,11	98,69	204,10

Table 7: Tempo de criação e exclusão das tarefas

Operação	Pilha	Tempo médio (μs)				
		16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
Criação	32 bytes	49,48	100,14	202,90	409,54	830,85
	64 bytes	61,38	125,86	255,02	512,38	1038,34
	128 bytes	86,34	174,09	350,78	704,77	1428,03
	256 bytes	134,45	271,10	544,03	1092,93	2210,37
	512 bytes	231,69	465,03	932,18	1873,38	3779,52
Exclusão	32 bytes	14,91	30,76	64,14	131,42	274,69
	64 bytes	14,84	31,14	63,39	130,75	267,97
	128 bytes	14,73	31,10	63,81	132,06	274,24
	256 bytes	14,63	30,48	64,32	131,52	274,18
	512 bytes	15,06	30,92	63,58	132,29	272,7

das tarefas: apesar da memória da pilha já estar reservada, a função *memset* é invocada durante a criação para reescrever todos os bytes da memória da pilha com uma espécie de marca d'água do próprio FreeRTOS. Já no procedimento de exclusão das tarefas, os resultados apontam que o tamanho da pilha não influencia no atraso de exclusão. Em particular, o gráfico na Figura 9 apresenta a evolução do atraso em relação ao tamanho da pilha, considerando-se o processador operando à frequência de 16 MHz.

5.2 Manipulação de recursos

5.2.1 Comunicação entre tarefas

Para realizar a comunicação entre as tarefas, foram aplicadas filas, grupo de eventos e notificação de tarefas.

Filas.

Foram testadas mensagens com tamanhos de 1, 2, 4, 8, 16, 32, 64 e 128 *bytes*. As mensagens enviadas pelas tarefas primária e secundária tem o mesmo tamanho. Os resultados (vide Tabela 8) mantêm o padrão de redução linear do atraso com o aumento da frequência. O tamanho da mensagem também impactou no atraso, uma vez que cada operação envolve a cópia da mensagem duas vezes pelo SO: uma no envio, da tarefa para a pilha, e outra na entrega, em sentido oposto.

Grupos de eventos.

A Tabela 9 lista os resultados do tempo de comunicação entre tarefas a partir de um grupo de eventos. A configuração padrão do

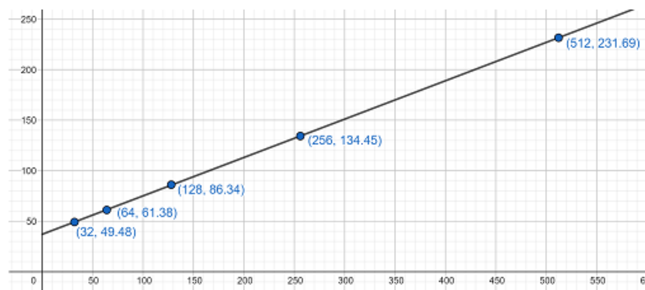

Figure 9: Tempo de criação das tarefas, de acordo com o tamanho da pilha com o processador operando em 16 MHz

Table 8: Tempo de envio de uma mensagem entre tarefas a partir de uma fila

Tamanho	Tempo médio (μs)				
	16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
1 byte	95,33	190,45	380,57	760,06	1460,70
2 bytes	96,38	192,21	381,02	765,09	1510,53
4 bytes	98,08	195,75	390,56	785,49	1587,97
8 bytes	102,11	204,61	407,98	809,20	1640,29
16 bytes	110,39	220,43	440,08	874,80	1746,72
32 bytes	126,38	252,21	504,19	1008,08	1996,26
64 bytes	158,52	316,58	635,95	1249,92	2495,68
128 bytes	223,06	445,46	891,33	1749,76	3494,08

Table 9: Tempo de envio de uma mensagem entre tarefas a partir de um grupo de eventos

Tempo médio (μs)				
16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
79,30	158,65	318,03	629,81	1250,05

FreeRTOS disponibiliza um grupo de eventos com 8 *bits*; ou seja, pode-se mandar uma mensagem de até um *byte* entre as tarefas, uma vez que é possível obter o valor do grupo de eventos. A evolução do atraso é semelhante ao observado no caso das filas; no entanto, quando comparado com estas, o atraso é, em média, 16% inferior (considerando cenários equivalentes, com mensagens de um *byte*).

Notificação de tarefas.

A Tabela 10 contém os resultados referentes aos dois modelos de comunicação via notificação de tarefas, confirmando que esse mecanismo oferece a maneira mais rápida de comunicação entre tarefas. Para ambos os modelos, o SO utiliza a mesma função para enviar a notificação, sendo que no modelo básico é atribuído o valor *eIncrement* para o parâmetro *eNotifyAction*. A única diferença entre os dois modelos está na função de recebimento da notificação, razão pela qual há uma pequena diferença no atraso. No modelo configurável, como era de se esperar, o modo *eNoAction* é o mais rápido pois, ao contrário dos demais, ele não altera o valor da variável de notificação da tarefa.

5.2.2 Comunicação entre ISR e tarefas

Para que uma ISR possa comunicar um evento a uma tarefa, foram testados os mesmos recursos de comunicação entre tarefas, inclusive os semáforos binários e os semáforos contadores.

Filas.

O comportamento observado (vide Tabela 11) foi semelhante ao da comunicação entre tarefas; porém, com valores de atraso

Table 10: Tempo de envio de uma mensagem entre tarefas, usando notificações de tarefas

MODELO BÁSICO					
	Tempo médio (μs)				
	16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
	48,77	97,44	194,94	390,62	783,10
MODELO CONFIGURÁVEL					
eNotifyAction	**Tempo médio (μs)**				
	16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
eNoAction	50,76	101,53	203,48	407,06	806,94
eSetBits	52,05	104,08	208,45	416,8	832,9
eIncrement	52,05	104,12	208,52	416,53	833,18
eSetValueWithoutOverwrite	51,47	102,92	206,05	410,5	831,74
eSetValueWithOverwrite	52,18	102,5	205,02	409,06	830,08

Table 11: Tempo de envio de uma mensagem da ISR para a tarefa, usando filas

Tamanho	Tempo médio (μs)				
	16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
1 byte	56,83	114,50	231,12	467,04	944,19
2 bytes	57,77	116,56	235,04	474,94	961,25
4 bytes	59,82	120,61	243,28	490,78	994,37
8 bytes	63,89	128,67	259,49	523,26	1055,62
16 bytes	71,92	144,82	291,6	588,48	1188,67
32 bytes	88,07	177,11	356,34	716,96	1448,77
64 bytes	120,3	241,64	485,31	976,32	1970,3
128 bytes	184,75	370,54	744,83	1494,08	3007,30

Table 12: Tempo de envio de uma mensagem da ISR para uma tarefa, usando grupo de eventos

Tempo médio (μs)				
16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
154,46	309,86	621,71	1250,85	2510,91

menores, uma vez que as funções de inserção invocadas pela ISR não necessitam desabilitar as interrupções e o escalonador. Em geral, observa-se uma variação no atraso independente do tamanho da mensagem, com uma redução de, aproximadamente, 38, 75, 148, 284 e 545 microssegundos, nas respectivas frequências de 16, 8, 4, 2 e 1 MHz.

Grupos de eventos.
Analisando a Tabela 12, pode-se observar um padrão de evolução do atraso semelhante aos casos anteriores. No entanto, em termos absolutos, os atrasos são aproximadamente o dobro daqueles registrados quando a comunicação via grupos de eventos ocorre entre tarefas.

Notificação de tarefas.
Assim como na comunicação entre tarefas, esse mecanismo também foi o que apresentou o menor atraso (vide Tabela 13). Diferente das funções chamadas pelas tarefas, as versões do modelo básico e do modelo configurável invocadas pela ISR são implementadas separadamente. De forma semelhante ao observado nos resultados das filas, a transferência de mensagens entre a ISR e a tarefa foi mais rápida do que a transferência entre tarefas, pois nas funções invocadas pela ISR não há necessidade de controlar o acesso à região crítica.

Semáforos.
Como antecipado (vide Tabela 14), sabendo-se que as funções invocadas para aquisição e entrega de semáforos são, essencialmente, a mesma para ambos os modelos, era de se esperar que o atraso

Table 13: Tempo para ISR enviar uma notificação de tarefas

MODELO BÁSICO					
	Tempo médio (μs)				
	16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
	39,60	80,11	162,10	328,45	667,39
MODELO CONFIGURÁVEL					
eNotifyAction	**Tempo médio (μs)**				
	16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
eNoAction	39,10	78,98	160,14	323,36	657,60
eSetBits	40,23	81,31	164,83	333,57	677,5
eIncrement	40,33	81,81	164,93	334,5	680,32
eSetValueWithoutOverwrite	39,99	80,86	163,78	331,39	672,83
eSetValueWithOverwrite	39,86	80,58	162,98	330,53	671,42

Table 14: Tempo para ISR sinalizar à tarefa, usando semáforos

Semáforo	Tempo médio (μs)				
	16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
Binário	44,97	90,84	183,76	371,49	752,64
Contador	44,98	90,84	183,60	371,49	752,58

Table 15: Tempo para sincronização de tarefas, usando grupo de eventos

Tarefas	Tempo médio (μs)				
	16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
2	103,52	207,28	416,22	833,36	1688,26
3	183,53	367,49	736,94	1480,64	2988,84
4	265,85	532,42	1067,31	2146,42	4359,59
5	350,39	702,09	1406,99	2828,42	5730,07
6	437,25	875,64	1754,93	3526,57	7127,36
7	526,34	1054,20	2113,29	4249,00	8610,22

também fosse praticamente idêntico.

5.2.3 *Sincronização de tarefas*

Conforme descrito anteriormente, os grupos de eventos fornecem uma função que permite realizar a sincronização entre tarefas. Essa funcionalidade foi testada empregando 2, 3, 4, 5, 6 e 7 tarefas, todas com a mesma prioridade. O atraso observado (vide Tabela 15) também evoluiu linearmente com a alteração do *clock*. Como era antecipado, o tempo de sincronização cresce à medida que se aumenta o número de tarefas. Em particular, para a CPU operando em uma frequência de 16 MHz, o atraso aumenta, aproximadamente, 80 μs à medida que se acrescenta uma nova tarefa ao grupo (vide Figura 10).

5.2.4 *Aquisição e entrega de mutex e semáforos*

O último teste de manipulação de recursos trata-se de verificar o atraso das funções de aquisição e entrega do *mutex*, do semáforo binário e do semáforo contador. Os resultados (vide Tabela 16) acompanham o padrão observado para os outros recursos. Os três recursos empregam, essencialmente, a mesma função para aquisição e entrega; porém, pode-se perceber que o atraso no *mutex* é levemente superior ao atraso dos semáforos. Vale relembrar que há algumas particularidades no tratamento de *mutexes* como, por exemplo, a possibilidade da recursividade.

5.3 Troca de contexto

O tempo consumido pela troca de contexto também teve cresci-

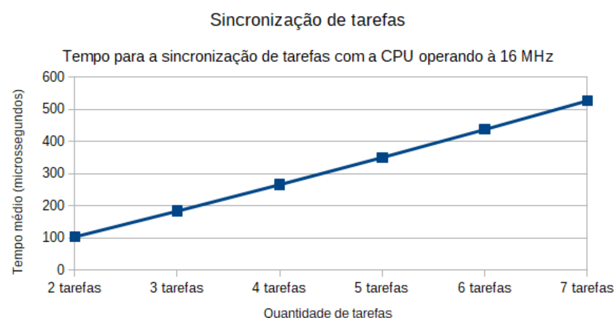

Figure 10: Tempo para sincronização de tarefas

Table 16: Tempo para aquisição e entrega do *mutex* e dos semáforos

Operação	Recurso	Tempo médio (μs)				
		16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
Aquisição	Mutex	7,71	16,32	34,45	78,50	164,35
	S. binários	6,25	12,66	27,04	60,38	128,64
	S. contadores	6,17	12,70	26,83	60,19	128,90
Entrega	Mutex	12,43	26,61	53,34	110,18	230,85
	S. binários	10,61	22,88	48,06	98,18	212,03
	S. contadores	10,68	23,15	48,06	98,75	208,96

Table 17: Tempo para troca de contexto

Tempo médio (μs)				
16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
15,92	31,99	63,99	128,56	260,80

mento exponencial à medida que a frequência é reduzida. A Tabela 17 mostra os resultados desse atraso. É importante salientar que independente da frequência da CPU, um *tick* ocorre a cada 16 ms no FreeRTOS então, em um sistema operando à 1 MHz, 1,63% do tempo de execução das tarefas é gasto com troca de contexto. Já com o sistema operando à 16 MHz esse tempo se torna praticamente insignificante, com o consumo de apenas 0,1% do tempo.

5.4 Temporizadores de software

Os temporizadores foram testados com valores variando de 16 a 400 ms, em cenários com nenhuma[10], 1, 2, 4 e 8 tarefas do tipo *vTarefaX*, objetivando-se disputar a CPU com a tarefa *daemom* do FreeRTOS que é responsável pela execução dos temporizadores. Como enfatizado anteriormente, os valores dos temporizadores são definidos como múltiplos de *tick* (*i.e.*, múltiplos de 16 ms), sendo este independente da frequência da CPU. Caso o valor do temporizador não seja múltiplo de 16 ms, atribui-se um valor múltiplo do *tick* imediatamente inferior ao solicitado (*e.g.*, se o valor passado for entre 17 e 31 ms, adota-se o valor de 16 ms).

Observou-se uma defasagem na execução dos temporizadores, dependente da frequência da CPU e da quantidade de tarefas. Para a CPU operando em 16 MHz e sem tarefas concorrentes, os temporizadores apresentaram um atraso de aproximadamente $9,9\%$, diminuindo à medida que a frequência era reduzida, chegando-se a um atraso de $8,75\%$ à frequência de 1 MHz. Essa diferença não foi notada na presença de tarefas concorrentes, independente da quantidade destas e da frequência da CPU, mantendo-se com uma defasagem uniforme de, aproximadamente, 9%.

Em geral, os resultados apresentaram baixa dispersão, com exceção de alguns casos: obteve-se um desvio padrão elevado para a execução com 2 tarefas e temporizador de 16 ms, 4 tarefas e temporizadores de 16, 32 e 48 ms e, mais acentuado, com 8 tarefas e temporizadores de 16, 32, 48, 64, 80, 96 e 112 ms (vide Figura 11). Esse comportamento anômalo resulta do fato do *daemon* (responsável pela execução dos temporizadores) não ter obtido a CPU a tempo.

A Figura 12 ilustra um diagrama que representa um cenário de execução das tarefas na CPU (não necessariamente na ordem apresentada), onde cada *vTi* refere-se a uma das quatro tarefas do tipo *vTarefaX* (*i.e.*, há o *daemon* mais quatro tarefas). A cada *tick* do sistema (*i.e.*, interrupção do *timer*), com período de 16 ms, o escalonador de tarefas é ativado e alterna a CPU para a próxima tarefa; portanto, o *daemon* é executado a cada 64 ms, aproximadamente. Neste caso, no intervalo de tempo correspondente a 64 ms, um temporizador de 16 ms expirou 4 vezes: quando o *daemon* assumir a CPU, o temporizador é tratado 4 vezes consecutivamente, impactando significativamente na dispersão das medidas, uma vez que o intervalo entre essas 4 execuções do temporizador é bem reduzido.

[10] Ou seja, era instanciada apenas a tarefa com o *timer* programado.

6. TRABALHOS RELACIONADOS

Mitrovic e Randic [14] exploram os recursos de suporte a multitarefa do FreeRTOS no Arduino. O enfoque é a nível de serviços, em como as funções disponíveis no SO podem ser empregadas para se desenvolver uma aplicação multitarefa. Aborda-se também o ambiente de desenvolvimento (IDE) disponível para o Arduino. No entanto, os autores não apresentam nenhuma análise de funcionamento e desempenho do FreeRTOS.

Yanik *et al.* [7] avaliam os benefícios de se empregar soluções multitarefas em sistemas baseados em microcontroladores. Em especial, foca-se no projeto e desenvolvimento de um sistema de assistência ao motorista via uma aplicação multitarefa embarcada em uma plataforma baseada no Arduino Uno. O objetivo principal consiste em comparar o desempenho da aplicação no Arduino Uno utilizando três sistemas operacionais: FreeRTOS, OS48 e Simulink. Os resultados indicam que as soluções avaliadas são mais efetivas e flexíveis quando empregada a abordagem multitarefa.

Spohn *et al.* [11] realizaram uma análise de desempenho do FreeRTOS na plataforma Arduino Uno, focando-se no limite de instanciação de tarefas e tamanho máximo da pilha, quantidade máxima de filas e tarefas, análise do atraso de comunicação via filas e notificação de tarefas e tempo de troca de contexto. No entanto, a metodologia empregada não contempla nenhuma forma de reprogramação do microcontrolador (*e.g.*, alteração do *clock* da CPU), tampouco os demais recursos do FreeRTOS.

7. CONCLUSÃO

A partir dos casos de teste executados, observou-se que, em geral, os atrasos medidos variam linearmente com a frequência da CPU. Dentre os recursos avaliados, as tarefas apresentaram o maior atraso de criação e exclusão. O *mutex*, o recurso com o segundo maior atraso de criação, enquanto que o grupo de eventos apresentou o segundo maior atraso em operações de exclusão. Já na comunicação entre tarefas, a utilização de filas foi o recurso mais demorado, em contraposição com a notificação de tarefas, a mais célere. Na comunicação entre ISR e tarefas, o grupo de eventos foi o recurso com o maior atraso, enquanto a notificação de tarefas foi também a mais eficiente.

Constatou-se que a precisão dos temporizadores está limitada à interrupção do *timer* (*i.e.*, *tick* de 16 ms), independente da frequência de operação da CPU. Se o valor de um temporizador, em *ticks*, for maior ou igual ao número de tarefas com a mesma prioridade da *daemon*, nenhuma anormalidade foi identificada. No entanto, para valores inferiores, algumas frequências da CPU podem coibir ou tornar irregular a execução dos temporizadores.

Caso o propósito do sistema não seja prejudicado pela diminuição da frequência da CPU, a economia de energia (apresentada na Tabela 3) é extremamente vantajosa para frequência de 4 MHz: apesar do atraso do sistema aumentar quatro vezes comparado à frequência de 16 MHz, o consumo de energia é, aproximadamente, 6 vezes menor. Em contrapartida, para o processador operando em 8 MHz, dobra-se o atraso mas não se reduz à metade o consumo de energia.

8. REFERENCES

[1] A. AG. Arduino uno rev3, 2019.

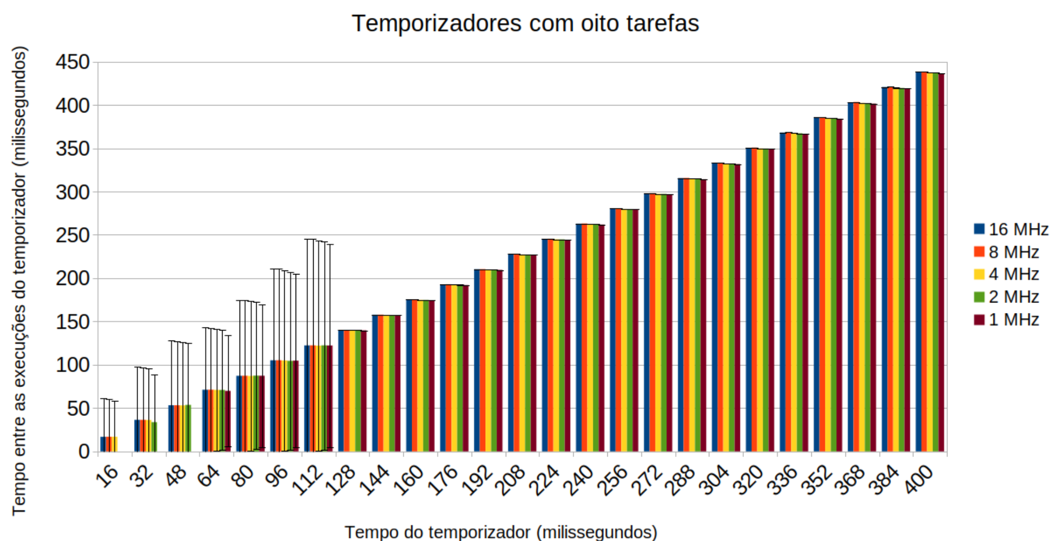

Figure 11: Temporizadores: cenário com 8 tarefas

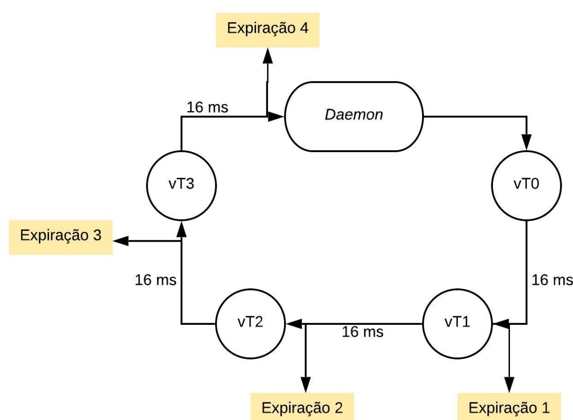

Figure 12: Esquema de execução das tarefas

[2] Atmel Corporation, San Jose, CA 95110 USA. *ATmega328P*, 2015. Rev.: 7810D–AVR–01/15.

[3] R. BARRY. *Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide*. Real Time Engineers Ltd, 2016.

[4] M. Christofferson. ways to improve performance in embedded linux systems. In *Korea Linux Forum*, 2013.

[5] A. Libc. Power reduction management, 2018.

[6] C. A. Maziero. Sistemas operacionais: Conceitos e mecanismos. *Livro aberto. Acessível em: `http://www.lcvdata.com/ads/so-livro.pdf`*, 2017.

[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] 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.

[9] C. A. M. d. SANTOS. Sistema dinâmico de economia de energia em rtos. 2017.

[10] M. Simonović and L. Saranovac. Power management implementation in freertos on lm3s3748. *Serbian Journal of Electrical Engineering*, 10(1):199–208, 2013.

[11] M. A. Spohn, F. C. Neto, and L. T. Fassini. Uma avaliação do sistema operacional freertos na plataforma arduino uno. *Revista de Sistemas e Computação*, 9(2), 2019.

[12] P. Stevens. Arduino_freertos_library, 2019.

[13] A. S. Tanenbaum and H. Bos. *Sistemas operacionais modernos*. Pearson Education do Brasil, São Paulo, 4 edition, 2016. tradução Jorge Ritter; revisão técnica Raphael Y. de Camargo.

[14] 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.