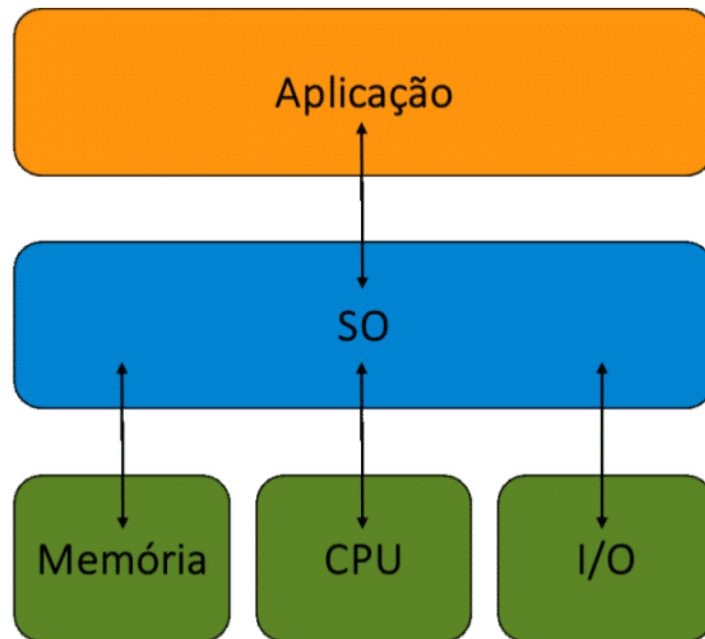


Desenvolvendo um RTOS: Introdução

Por **Rodrigo Almeida** - 16/11/2015

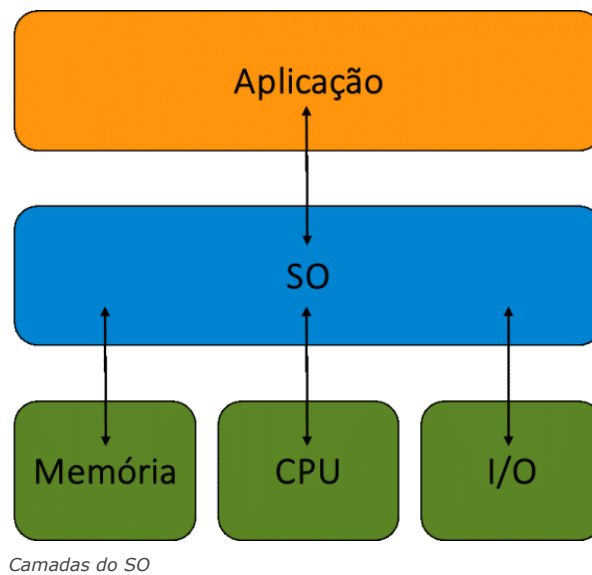


Este post faz parte da série [Desenvolvendo um RTOS](#). Leia também os outros posts da série:

- [Desenvolvendo um RTOS: Introdução](#)
- [Desenvolvendo um RTOS: processos e tarefas](#)
- [Desenvolvendo um RTOS: Utilizando buffer circular como gestor de processos](#)

Nesta série de artigos o objetivo será abordar os conceitos de programação que envolvem a criação e operação de um sistema de tempo real, RTOS. Iremos começar com um pouco de teoria falando sobre o que são os sistemas operacionais, suas responsabilidades e onde eles se encaixam no ecossistema embarcado. Os artigos estão sendo co-escritos com o [Marcelo Barros](#), professor da UFU em sistemas embarcados.

Um sistema operacional é um conjunto de código que tem como função principal gerenciar os recursos do sistema. Em geral os principais recursos a serem gerenciados são o processador, a memória e os dispositivos de entrada e saída.



Dentro do sistema operacional existem diversos trechos de códigos especializados em atender uma determinada função: interfaces gráficas, drivers de dispositivos, sistemas de arquivos, engines de processamento 3D, etc. Esses códigos visam simplificar a criação das aplicações, permitindo que os programadores se preocupem menos com o hardware e mais com as funcionalidades de seu projeto.

No entanto, no ambiente embarcado existem diversas restrições que podem impedir ou dificultar o uso de um sistema operacional convencional. Em algumas situações pode ser necessário acessar diretamente algum periférico por questão de velocidade; noutras a quantidade de memória ou capacidade de processamento impede a utilização das diversas camadas de software; ou ainda a plataforma de hardware utilizada não possui a implementação do sistema operacional desejado. Estes impedimentos são mais comuns em sistemas embarcados de médio ou baixo custo.

Talvez o maior impedimento no uso de sistemas operacionais em alguns ambientes embarcados seja a capacidade de atendimento de tarefas que exijam tempo real. Por tempo real definimos atividades que possuem um prazo no qual elas devem começar e terminar. Atrasar essas atividades podem trazer problemas ou danos físicos ao sistema.

Visando atender essas restrições foram desenvolvidos os sistemas operacionais de tempo real. Estes sistemas são relativamente simples e possuem o foco voltado para atender as questões de tempo das aplicações. A garantia de tempo é realizada no kernel do sistema. A tabela a seguir apresenta alguns exemplos de sistemas operacionais e seus requisitos de memória.

Sistema Operacional	Consumo de Flash (mínimo)	Consumo de RAM (mínimo)
VxWorks	> 75.000	-
FreeRTOS	> 6.000	> 800
uC/OS	> 5.000	-
uOS	> 2.000	> 200
BRTOS	> 2.000	~ 100

O kernel é uma camada de software do SO responsável por gerenciar e coordenar a execução dos processos segundo um critério. O critério utilizado pode ser a

prioridade, a criticidade de uma tarefa, uma sequência de execução pré-definida ou o tempo de re-execução da atividade, visando atingir tempo real.

O kernel pode ser dividido em dois grandes grupos: preemptivos ou cooperativos (não-preemptivos). No kernel cooperativo cada atividade é executada do começo ao fim, de modo que a nova atividade só se inicia quando a anterior termina. Já um kernel preemptivo possui a capacidade de pausar a atividade atual para começar uma segunda atividade. Quando a segunda atividade termina, ou esgota seu tempo de execução, o kernel retorna para a primeira atividade, exatamente no ponto que havia parado.

Os kernels cooperativos são a evolução das arquiteturas baseadas em **máquina de estado**. Uma das restrições da abordagem baseada em máquinas de estado é que o modelo de execução do código é fixo. As mudanças na operação do programa exige uma nova análise da máquina de estado, com um novo projeto e implementação, uma vez que os estados são, em geral, fixos. Não é simples, com a placa em execução, modificar seu comportamento base. Para adicionar ou remover atividades é necessário reescrever o código, recompilar e regravar o dispositivo. Um kernel cooperativo remove essa limitação, permitindo que as tarefas possam ser criadas ou removidas com o sistema em funcionamento.

A base de funcionamento do kernel é a execução de atividades. Essas atividades em geral são definidas por um conjunto de código que poderá, ou não, ser executado, dependendo das condições percebidas pelo kernel.

No próximo artigo iremos apresentar uma definição de processo e como podemos executar um trecho de código que, a princípio, não conhecemos em tempo de compilação. Esta habilidade formará a base do nosso kernel cooperativo.

Outros artigos da série

[Desenvolvendo um RTOS: processos e tarefas >>](#)

 Facebook 182

 Twitter 2

 Google+ 10

 LinkedIn 146

Este post faz da série **Desenvolvendo um RTOS**. Leia também os outros posts da série:

- [Desenvolvendo um RTOS: Introdução](#)
- [Desenvolvendo um RTOS: processos e tarefas](#)
- [Desenvolvendo um RTOS: Utilizando buffer circular como gestor de processos](#)

NEWSLETTER



Receba os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.



E-mail

CADASTRAR E-MAIL

Fique tranquilo, também não gostamos de spam.

Desenvolvendo um RTOS: Introdução por **Rodrigo Almeida** . Esta obra está licenciado com uma Licença **Creative Commons Atribuição-NãoComercial-Compartilhual** 4.0 Internacional .

Rodrigo Almeida

<http://sites.google.com/site/rmaalmeida> 

Professor da Universidade Federal de Itajubá onde leciona sobre programação embarcada, sistemas operacionais e desenvolvimento de produtos eletrônicos. Pesquisa na área de sistemas críticos desenvolvendo técnicas para melhoria de confiabilidade e segurança.



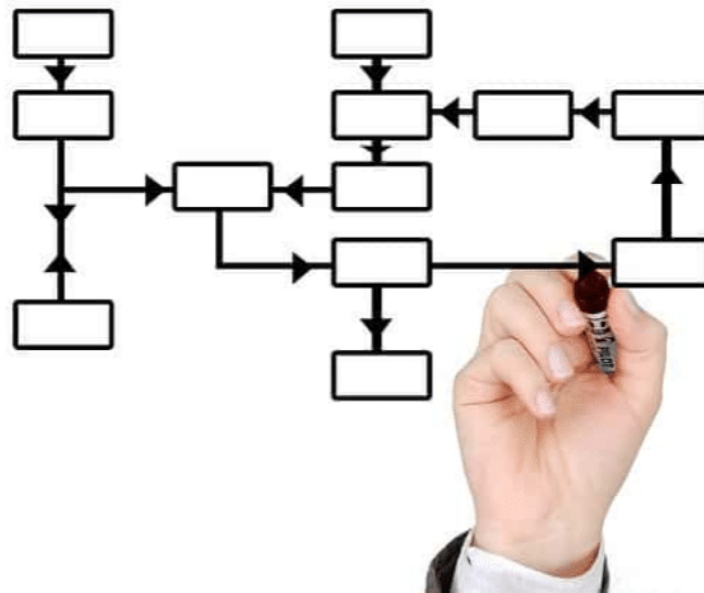
Este site utiliza cookies. Ao usá-lo você concorda com nossos Termos de Uso.

[Saiba mais.](#)

Continuar

Desenvolvendo um RTOS: processos e tarefas

Por **Rodrigo Almeida** - 19/11/2015

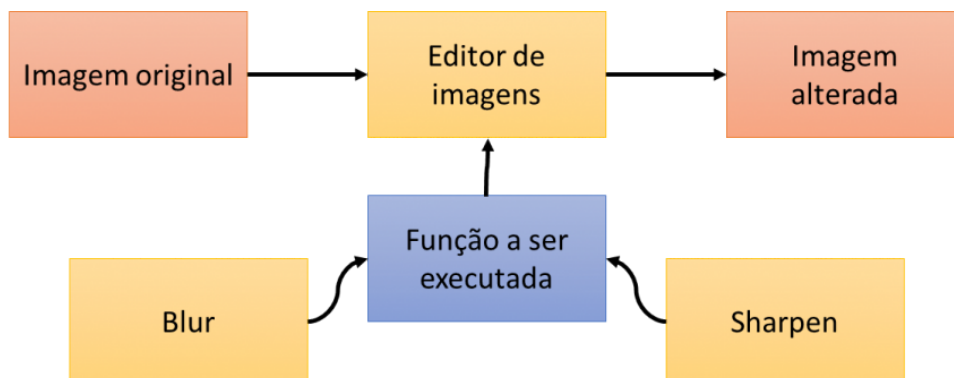


Este post faz parte da série [Desenvolvendo um RTOS](#). Leia também os outros posts da série:

- [Desenvolvendo um RTOS: Introdução](#)
- [Desenvolvendo um RTOS: processos e tarefas](#)
- [Desenvolvendo um RTOS: Utilizando buffer circular como gestor de processos](#)

Continuando a série em parceria com [Marcelo Barros](#), chegamos ao segundo artigo sobre o processo de criação de um kernel cooperativo, definindo processos e tarefas. Para efeito de simplificação, utilizaremos a linguagem C voltada para desktop. Deste modo todos, independente do hardware utilizado, podem replicar as atividades.

Em algumas situações queremos que o programa possa escolher qual função deseja executar, por exemplo num editor de imagens: queremos poder escolher entre usar a função blur ou função sharpen na imagem.



Um exemplo de implementação deste sistema pode ser dado pelo código a seguir, onde **image** é uma variável matricial que representa uma imagem.

```
1 // implementação das funções de tratamento de imagem
2 image Blur(image nImg){}
3 image Sharpen(image nImg){}
4
5 image imgEngine(image nImg, int opt){
6     image temp;
7     //checagem de consistência da imagem
8     switch(opt){
9         case 1:
10            temp = Sharpen(nImg);
11            break;
12         case 2:
13            temp = Blur(nImg);
14            break;
15     }
16     return temp;
17 }
```

O problema com essa abordagem é que a estrutura é fixa. Se forem adicionadas novas funcionalidades, a função *imgEngine* tem que ser alterada.

Um outro modo de implementar esse código trazendo uma maior flexibilidade é utilizar os ponteiros de função.

Os ponteiros de função são variáveis que, como ponteiros, armazenam um endereço. A diferença é que esse endereço é de uma função, e não de uma variável. A declaração de uma variável do tipo ponteiro de função, em linguagem C, é bastante diferente da declaração de uma variável normal, sendo muito parecida com a declaração de um protótipo de função. Uma variável com o nome *ponteiroTeste*, que aponta para uma função que recebe uma imagem e retorna uma imagem, pode ser declarado como:

```
1 //declaração da variável ponteiroTeste
2 imagem (*ponteiroTeste)(imagem nImg);
3 ponteiroTeste = 0xF879; //apontando para o endereço de uma função conhecida previamente.
```

Por se tratar de um ponteiro é necessário de-referenciar a variável antes de chamar a função, por exemplo:

```
1 imagemAlterada = (*ponteiroTeste)(imagemOriginal);
```

Notar que após a de-referência são passados os parâmetros como numa função qualquer. Além da passagem por parâmetro a função pode ser armazenada.

É mais comum criar uma definição de tipo para simplificar o uso destes ponteiros no código. Reescrevendo a função *imgEngine()*, com a estrutura de ponteiros de função temos:

```
1 //declaração do tipo ponteiro para função
2 typedef imagem (*ptrFunc)(imagem nImg);
3
4 // implementação das funções de tratamento de imagem
5 image Blur(image nImg){}
6 image Sharpen(image nImg){}
7
8 //chamado pelo editor de imagens
9 image imgEngine (ptrFunc nFuncao, imagem nImg){
10     imagem temp;
11     temp = (*nFuncao)(nImg);
12     return temp;
13 }
```

Como podemos perceber pelo código a função *imgEngine()* recebe dois parâmetros: um é a função e o outro a imagem que será processada. Essa função tem que ser do tipo *ptrFunc*, ou seja, ela deve receber um parâmetro imagem e retornar uma variável imagem.

A atribuição de uma função a um ponteiro de função, ou passagem por parâmetros como vimos no exemplo, só pode ser feita se ambas as funções tiverem a mesma assinatura (os mesmos parâmetros e o mesmo retorno). Tanto a função **Blur()** quanto a função **Sharpen()** obedecem este quesito. Por isso é possível executar o código a seguir:

```
1 //...
2 imagem nImagem = recebeImagemCamera();
3 nImagem = imgEngine (Blur, nImagem);
4 nImagem = imgEngine (Sharpen, nImagem);
5 //...
```

As funções **Blur()** e **Sharpen()** são passadas como se fossem variáveis para serem usadas dentro da função **imgEngine()**.

A grande vantagem desta estrutura é que agora, num mesmo local (**imgEngine()**), é possível executar diferentes funções, sem precisar conhecer previamente quem ela é. Isto permite inclusive que a placa receba, por exemplo, via comunicação serial, o código binário de uma nova funcionalidade e o armazene num buffer na memória. Logo em seguida passamos o endereço de início do buffer para a função **imgEngine()** e o código será executado. Um processo muito similar a esse acontece quando realizamos o download de um aplicativo e o executamos no celular.

Process, thread ou task?

Na área de ciências da computação definimos um **thread** como uma unidade de código executável que pode ser gerenciada de modo independente por um gestor (**scheduler** ou **kernel**). A principal diferença entre processos e threads é o compartilhamento de memória. Dois processos só conseguem se comunicar com sistemas dedicados de passagem de mensagens pois as memórias que ocupam são isoladas. Já as **threads** compartilham a região de memória que se encontram.

Em sistemas operacionais voltados para microcontroladores mais simples, principalmente aqueles que não possuem MMU, pode não ser possível implementar um processo, de modo que todas as atividades são por definição tarefas. Para evitar confusões é comum utilizar a nomenclatura **task** para definir uma aplicação/trecho de código executável por um sistema operacional embarcado.

Como o objetivo desta série de artigos é explicar o funcionamento de um kernel, bem como o processo de construção do mesmo, as palavras processo e tasks poderão ser utilizadas com o mesmo significado.

Por fim vamos ao que interessa, como criar um processo (ou task)?

A implementação mais simples de um processo é a criação de uma função. Executar o processo significa realizar a chamada da função. Deste modo podemos adaptar o código utilizado para a criação de um editor de imagens para criar um "executor" de funções.

```
1 //declaração do tipo ponteiro para função
2 typedef void (*process)(void);
3
4 //Executor de funções
5 void exec(process Processo){
6     (*Processo)();
7 }
8 //criando o processo 1
```

```
9 void proc1(void) {
10     printf("Executando o processo 1\n");
11 }
12 //criando o processo 2
13 void proc2(void) {
14     printf("Executando o processo 2\n");
15 }
16 //executando os processos
17 int main(int argc, char **argv){
18     printf("Inicio\n");
19     exec(proc1);
20     exec(proc2);
21     printf("\nFim\n");
22     return 0;
23 }
```

O código acima utiliza a função `exec()` para executar os processos já criados. O próximo passo é conseguir gerenciar os processos de modo que seja possível controlar e modificar sua ordem de execução. Para isso precisamos de uma estrutura que consiga armazenar referências para os processos, adicionar essas referências e executar os processos armazenados na estrutura. Isso fica para o próximo artigo.

Outros artigos da série

<< Desenvolvendo um RTOS: Introdução Desenvolvendo um RTOS: Utilizando buffer circular como gestor de processos >>

 Facebook 114

 Twitter 0

 Google+ 8

 LinkedIn 106

Este post faz da série [Desenvolvendo um RTOS](#). Leia também os outros posts da série:

- [Desenvolvendo um RTOS: Introdução](#)
- [Desenvolvendo um RTOS: processos e tarefas](#)
- [Desenvolvendo um RTOS: Utilizando buffer circular como gestor de processos](#)

NEWSLETTER

Receba os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.

E-mail


CADASTRAR E-MAIL

Fique tranquilo, também não gostamos de spam.



Desenvolvendo um RTOS: processos e tarefas por [Rodrigo Almeida](#) . Esta obra está licenciado com uma Licença [Creative Commons Atribuição-NãoComercial-SemDerivações 4.0 Internacional](#) .

Rodrigo Almeida

<http://sites.google.com/site/rmaalmeida> 

Professor da Universidade Federal de Itajubá onde leciona sobre programação embarcada, sistemas operacionais e desenvolvimento de produtos eletrônicos. Pesquisa na área de sistemas críticos desenvolvendo técnicas para melhoria de confiabilidade e segurança.



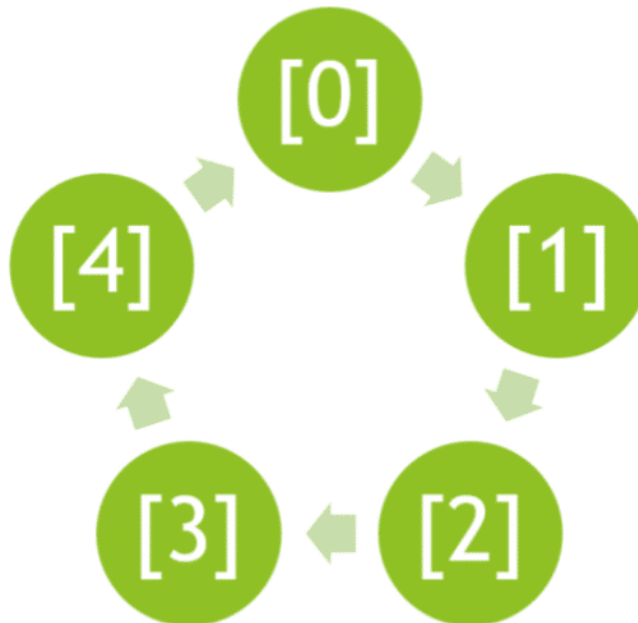
Este site utiliza cookies. Ao usá-lo você concorda com nossos Termos de Uso.

[Saiba mais.](#)

Continuar

Desenvolvendo um RTOS: Utilizando buffer circular como gestor de processos

Por **Rodrigo Almeida** - 10/12/2015



ÍNDICE DE CONTEÚDO [MOSTRAR]

Este post faz parte da série [Desenvolvendo um RTOS](#). Leia também os outros posts da série:

- [Desenvolvendo um RTOS: Introdução](#)
- [Desenvolvendo um RTOS: processos e tarefas](#)
- Desenvolvendo um RTOS: Utilizando buffer circular como gestor de processos

Nos primeiros artigos falamos sobre o que é um sistema operacional e como descrever/implementar um processo. Neste artigo eu e o [Marcelo](#) vamos falar sobre como desenvolver uma estrutura para gestão destes processos, ou seja, como criar o próprio kernel.

Para gerenciar a execução dos processos é necessário armazenar suas referências de algum modo. Existem diversos modos de se criar uma estrutura de dados para armazenamento/gerenciamento de informações. Visando equilibrar a simplicidade de operação com recursos disponíveis optamos por utilizar um buffer circular.

Buffers são regiões de memória que servem para armazenar dados temporários. Os buffers circulares podem ser implementados utilizando uma lista linkada, onde o último elemento se conecta com o primeiro. O problema dessa abordagem é o gasto extra de memória para colocar os ponteiros da lista. O modo mais "econômico" é utilizar apenas um vetor com dois índices, indicando o início e o fim do buffer. Os elementos devem ser adicionados no "fim" do vetor e removidos no "início". A declaração destas variáveis é bastante simples:

```
1 #define BUFFSIZE 6
2 int circBuff[BUFFSIZE];
3 int ini;
4 int fim;
```

A cada inserção de elementos a variável **fim** é incrementada. A cada remoção de elementos a variável **ini** é incrementada. Quando as variáveis **fim** ou **ini** chegam no final do vetor, elas devem voltar a apontar para a posição zero. Apesar de ser chamado circular, na memória o buffer é armazenado como um vetor sequencial, conforme a figura a seguir.

O buffer vazio indica que não existe nenhum elemento de interesse armazenado no buffer. Isso não quer dizer que o vetor está vazio. Na verdade, existem valores armazenados em cada posição, só que esses valores não tem importância para a aplicação, sendo geralmente chamados de lixo. Esses valores são resultados de cálculos anteriores de variáveis temporárias ou qualquer outra função do processador.

O índice **fim** sempre aponta para uma posição "disponível" na memória, ou seja, que pode receber um valor. Deste modo, adicionando 2 elementos no buffer temos a seguinte representação

A remoção de um elemento significa incrementar o índice **ini**, "removendo" o item da lista de itens úteis. Podemos notar na figura abaixo que o elemento não é fisicamente removido do buffer, mas agora ele está na região de elementos que

consideramos como lixo. Todos os elementos válidos estão entre o índice **ini** e o índice **fim**.

Ao adicionar mais dois elementos o índice **fim** passa a apontar para a última posição linear do vetor: a posição 4. Nesta situação o buffer possui três elementos válidos ('B', 'C' e 'D') e duas posições com lixo (apesar de sabermos que a posição 0 possui o valor 'A', neste momento ele é um lixo na memória).

Como o sistema será desenvolvido para que o vetor se comporte como um buffer circular, quando adicionarmos o último elemento no buffer o índice **fim** passa a valer zero e, nesse momento, a posição 0 poderá ser reescrita.

Como pode ser visto na sequência acima, a próxima posição livre é sempre uma unidade à frente da posição atual, exceto quando o índice aponta para a última posição do vetor.

Para simplificar o processo de procura da próxima posição, podemos utilizar um incremento seguido do resto de divisão pelo tamanho do vetor. Esse processo pode ser definido com o seguinte código:

```
1 #define next(elem) ((elem+1)%BUFSIZE)
```

O resto da divisão faz com que o resultado retorne para zero sempre que o índice ultrapassar o final do vetor. Esta é uma estratégia simples, mas com um custo computacional maior pois requer uma operação de cálculo de resto da divisão. Se processamento mais baixo for um requisito para o seu projeto, faça a verificação por teste de limites, mesmo que isto gere um código maior, como no código abaixo:

```
1 int next(int elem){
2     if((elem+1) < BUFSIZE){
3         return (elem+1);
4     }else{
5         return 0;
6     }
7 }
```

Um problema que aparece no gerenciamento de buffers circulares é definir quando o vetor está cheio ou vazio, já que, em ambos os estados, o indicador de início e fim estão no mesmo local.

Existem pelo menos 3 alternativas de se resolver este problema: manter um slot sempre aberto, usar um indicador de cheio/vazio ou contar a quantidade de leituras/escritas. Procurando manter o código mais simples possível optamos por manter sempre um slot vazio.

O processo de adicionar elementos no buffer consiste em atribuir um valor ao elemento apontado pelo índice fim. No entanto devemos primeiro verificar se o buffer não está cheio.

Quando o buffer estiver vazio, os índices **ini** e **fim** possuem o mesmo valor. Quando o buffer estiver cheio, o índice **fim** está uma posição atrás de **ini**.

Para verificar se o buffer não está cheio podemos verificar se a próxima posição do índice **fim** não é igual ao índice **ini**, ou seja, se incrementando **fim** ele não

sobrepõe o valor de **ini**

```
1 void addBuff(int elem){
2     if(next(fim) != ini){
3         circBuff[fim] = elem;
4         fim = next(fim);
5     }
6 }
```

Para remover um elemento, devemos verificar se o buffer não está vazio, ou seja, se o índice **ini** é diferente de **fim**. Sendo diferente podemos remover o elemento do buffer, simplesmente incrementando o índice **ini**.

```
1 void removeBuff(void){
2     if(ini != fim){
3         ini = next(ini);
4     }
5 }
```

Note que nada é feito com a posição do buffer. Na verdade, o valor antigo continua armazenado no vetor. Com o uso do buffer esse valor será eventualmente sobrescrito.

Estamos prontos agora para implementar a primeira versão do kernel, utilizando um buffer circular como estrutura de dados de armazenamento dos processos.

Implementando um kernel com buffer circular

Nesta primeira versão os processos serão implementados como funções, e o gerenciamento dos processos será feito através de um buffer circular. O código do kernel pode ser dividido em três seções:

1. criação dos processos;
2. rotinas do kernel;
3. função principal.

1 - Processos

As funções que serão executadas pelo kernel (**tst1()**, **tst2()**, **tst3()**) passarão a ser denominadas processos, com ideia semelhante aos processos de um desktop. Além disso, todos os processos têm que ter a mesma assinatura do ponteiro ptrFunc, no caso **void F(void)** ;

```
1 #include "stdio.h"
2
3 //protótipos dos "processos"
4 void tst1(void);
5 void tst2(void);
6 void tst3(void);
7
8 //"processos"
9 void tst1(void) {
10     printf("1\n");
11 }
12 void tst2(void) {
13     printf("22\n");
14 }
15 void tst3(void) {
16     printf("333\n");
17 }
18
19 //declaracao do tipo ponteiro para funcao
20 typedef void(*ptrFunc)(void);
```

2 - Kernel

Este primeiro kernel possui três funções: uma para inicializar as variáveis internas, uma para adicionar processos no buffer circular e uma para executar o kernel. Em geral, a função que executa o kernel possui um loop infinito. Nesta primeira versão apenas executamos as funções na ordem em que foram passadas. Não existe nenhum outro tipo de controle sobre os processos, isto fica para o próximo artigo.

```
1 //variáveis do kernel
2 ptrFunc pool[4];
3 int ini;
4 int fim;
5
6 //protótipos das funções do kernel
7 void InicializaKernel(void);
8 void AddKernel(ptrFunc newFunc);
9 void ExecutaKernel(void);
10
11 //funções do kernel
12 void InicializaKernel(void){
13     ini = 0;
14     fim = 0;
15 }
16 void AddKernel(ptrFunc newFunc){
17     if(next(fim) != ini){
18         pool[fim] = newFunc;
19         fim = next(fim);
20     }
21 }
22 void ExecutaKernel(void){
23     int i;
24     //no microcontrolador real o loop abaixo
25     //deve ser trocado por um loop infinito
26     //na implementação em desktop utilizamos um loop
27     //com 100 iterações para simular o funcionamento do kernel
28     for(i=0;i<100;i++){
29         if(ini != fim){
30             (*pool[ini])();
31             ini = next(ini);
32         }
33     }
34 }
```

3 - Rotina Principal

Utilizar o kernel desenvolvido é bastante simples. Com os processos já implementados de acordo com a assinatura padrão, basta:

1. inicializar o sistema;
2. adicionar os processos que devem ser executados na ordem em que serão executados e, por fim;
3. executar o kernel.

```
1 int main(int argc, char **argv){
2     printf("Inicio\n\n");
3     InicializaKernel();
4     AddKernel(tst1);
5     AddKernel(tst2);
6     AddKernel(tst3);
7
8     ExecutaKernel();
9
10    printf("\nFim\n");
11    return 0;
12 }
```

Esta primeira versão do kernel é capaz de receber os processos (codificados como ponteiros de funções) e executá-los sequencialmente. O próximo passo é fazer com que a função **ExecutaKernel()** seja capaz de gerenciar os processos, reexecutando-os e tomando decisões sobre qual é o próximo a ser executado.

Outros artigos da série

<< Desenvolvendo um RTOS: processos e tarefas

 Facebook 63

 Twitter 0

 Google+ 2

 LinkedIn 86

Este post faz da série [Desenvolvendo um RTOS](#). Leia também os outros posts da série:

- [Desenvolvendo um RTOS: Introdução](#)
- [Desenvolvendo um RTOS: processos e tarefas](#)
- [Desenvolvendo um RTOS: Utilizando buffer circular como gestor de processos](#)



NEWSLETTER

Receba os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.

E-mail

CADASTRAR E-MAIL

Fique tranquilo, também não gostamos de spam.

Desenvolvendo um RTOS: Utilizando buffer circular como gestor de processos por **Rodrigo Almeida** . Esta obra está licenciado com uma Licença [Creative Commons Atribuição-NãoComercial-Compartilhalgual 4.0 Internacional](#) .

Rodrigo Almeida

<http://sites.google.com/site/rmaalmeida> 

Professor da Universidade Federal de Itajubá onde leciona sobre programação embarcada, sistemas operacionais e desenvolvimento de produtos eletrônicos. Pesquisa na área de sistemas críticos desenvolvendo técnicas para melhoria de confiabilidade e segurança.



Este site utiliza cookies. Ao usá-lo você concorda com nossos Termos de Uso.
[Saiba mais.](#)

Continuar

RTOS: Um ambiente multi-tarefas para Sistemas Embarcados

Por **José Morais** - 09/07/2018



ÍNDICE DE CONTEÚDO [MOSTRAR]

Este post faz parte da série [RTOS](#). Leia também os outros posts da série:

- [RTOS: Um ambiente multi-tarefas para Sistemas Embarcados](#)
- [RTOS: Scheduler e Tarefas](#)
- [RTOS: Semáforos para sincronização de tarefas](#)
- [RTOS: Uso de Queue para sincronização e comunicação de tarefas](#)
- [RTOS: Uso de grupo de eventos para sincronização de tarefas](#)
- [RTOS: Software Timer no FreeRTOS](#)

Atire a primeira pedra quem nunca tenha sofrido com problemas de multi-tarefas em embarcados, já que em qualquer sistema, desde pequeno ou grande, pode-se ter mais de um sensor ou atuador realizando tarefas específicas no sistema. Fazer o controle simultâneo de todos pode exigir um alto grau de programação e sincronização, deixando o projeto muito complexo. Por fim, acabamos usando outro embarcado para ajudar no sistema principal. Com esse mesmo pensamento, pode ser resolvido com um embarcado multi-core, mas os custos aumentam e não é interessante em produtos comerciais.

Então como podemos tirar essa pedra do caminho? Utilizando Sistemas Operacionais, mais especificamente Sistemas Operacionais de Tempo Real (Real Time Operating Systems - RTOS) que se difere em alguns aspectos de um Sistema Operacional Genérico (Generic Purpose Operating System - GPOS) como Windows, Mac e baseados em Linux...

O que é um Sistema Operacional?

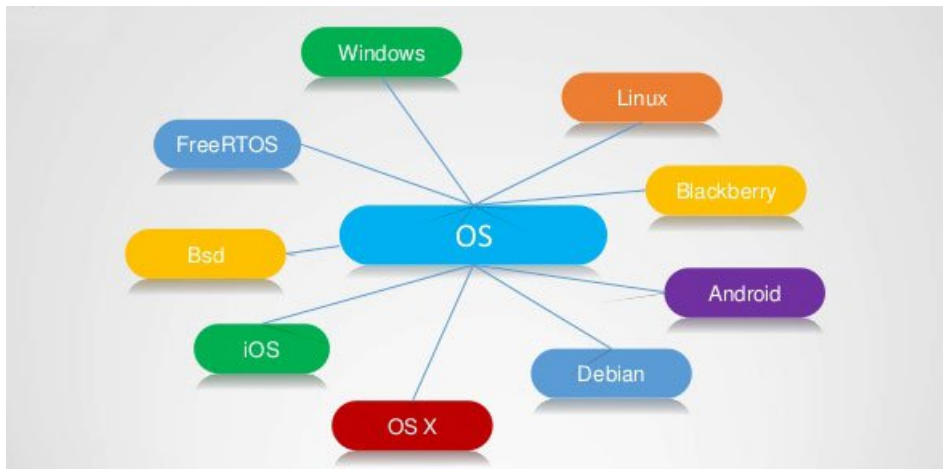


Figura 1 - Sistemas Operacionais.

Um Sistema Operacional é um conjunto de ferramentas (softwares) que criam um ambiente multi-tarefas e também é uma abstração entre software e hardware, incluindo gerenciamento de recursos internos.

Principais itens de um Sistema Operacional

Kernel: É o núcleo do sistema operacional, sendo uma camada de abstração entre o Software e Hardware para gerenciar diversos recursos, como por exemplo:

- Gerenciamento de memória RAM;
- Gerenciamento de processos.

Scheduler: Scheduler algorithm é o software do sistema operacional que decide quais tarefas serão escolhidas para serem executadas. Normalmente são selecionadas pela maior prioridade ou, em caso de prioridades iguais, o scheduler tenta dividir o tempo entre todas tarefas. Vamos entender melhor na segunda parte desta série.

Tarefas: São como “mini programas”. Cada tarefa pode desempenhar algo totalmente diferente das outras ou também iguais e compartilhar recursos, como periféricos. São nas tarefas que iremos dividir nosso código em partes, deixando que o scheduler execute todas com suas devidas importâncias.

Quais são as diferenças entre GPOS e RTOS?

GPOS: É amplamente utilizado em computadores e similares por ter um alto throughput (vazão) de dados, é focado na execução de muitas tarefas simultaneamente, onde atraso de uma não é crítico, irá apenas atrasá-la. As tarefas do GPOS não tem um tempo limite para serem executadas, então é comumente chamado de "Not Time Critical".

RTOS: É um sistema operacional mais especializado onde o tempo de resposta é mais importante do que executar centenas de tarefas simultaneamente. O tempo de resposta não precisa necessariamente ser o mais rápido possível, como o nome pode sugerir, mas deve ser previsível, logo, "Real-Time" pode ser uma resposta de vários minutos ou nanos segundos, dependendo da forma que seu sistema funciona. As tarefas do RTOS contam com tempo limite para serem executadas, então é comumente chamado de "Time Critical".

Pensando em aplicações práticas

As aplicações práticas para um RTOS são normalmente quando precisamos efetuar muitas tarefas ao mesmo tempo ou trabalhar com "Real-Time" onde falhas além do tempo definido é crítico. Vamos imaginar um cenário onde o sistema embarcado efetua 5 tarefas simultaneamente, listadas a seguir na figura 3:

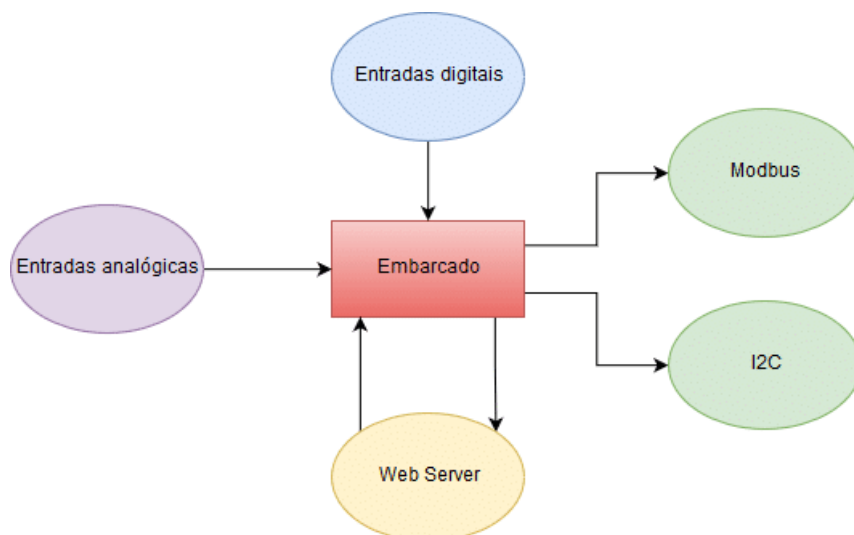


Figura 2 - Exemplo de ambiente multi-tarefa.

Nosso embarcado precisa fazer todas essas tarefas simultaneamente, sem que uma atrase a outra, já que implicaria em falha crítica. Nesse sistema, dados são colhidos através de entradas digitais e analógicas, e são enviados através de Modbus, I2C e um Web Server para o usuário, tudo ao mesmo tempo!

Programar esse pequeno sistema em Bare Metal (programação sem sistemas operacionais) poderia ser tão complexo, pelo fato de ser necessário efetuar toda sincronização com que uma tarefa não “trave” a outra, que acabaria forçando o desenvolvedor a utilizar mais microcontroladores ou similares para dividir as tarefas entre eles, causando um custo maior no projeto e podendo inviabilizá-lo por custos.

RTOS será nossa salvação para problemas em ambientes multi-tarefas. Nos próximos artigos da série vamos aprender e praticar os principais conceitos de RTOS com o FreeRTOS, como Tarefas, Queues (Filas de dados), Semáforos e assim por diante.

Saiba mais

[Desenvolvendo um RTOS: Introdução](#)

[Implementando elementos de RTOS no Arduino](#)

[Criando um projeto no IAR com o FreeRTOS](#)

Referências

https://freertos.org/Documentation/RTOS_book.html 

https://en.wikipedia.org/wiki/Operating_system 

https://en.wikipedia.org/wiki/Real-time_operating_system 

Outros artigos da série

[RTOS: Scheduler e Tarefas >>](#)

 Facebook 76 Twitter 0 Google+ 3 LinkedIn 2

Este post faz da série [RTOS](#). Leia também os outros posts da série:

- [RTOS: Um ambiente multi-tarefas para Sistemas Embarcados](#)
- [RTOS: Scheduler e Tarefas](#)
- [RTOS: Semáforos para sincronização de tarefas](#)
- [RTOS: Uso de Queue para sincronização e comunicação de tarefas](#)
- [RTOS: Uso de grupo de eventos para sincronização de tarefas](#)
- [RTOS: Software Timer no FreeRTOS](#)


NEWSLETTER

Receba os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.

E-mail

CADASTRAR E-MAIL

Fique tranquilo, também não gostamos de spam.

RTOS: Um ambiente multi-tarefas para Sistemas Embarcados por **José Morais**. Esta obra está licenciado com uma Licença [Creative Commons Atribuição-Compartilhual 4.0 Internacional](#) .

José Morais

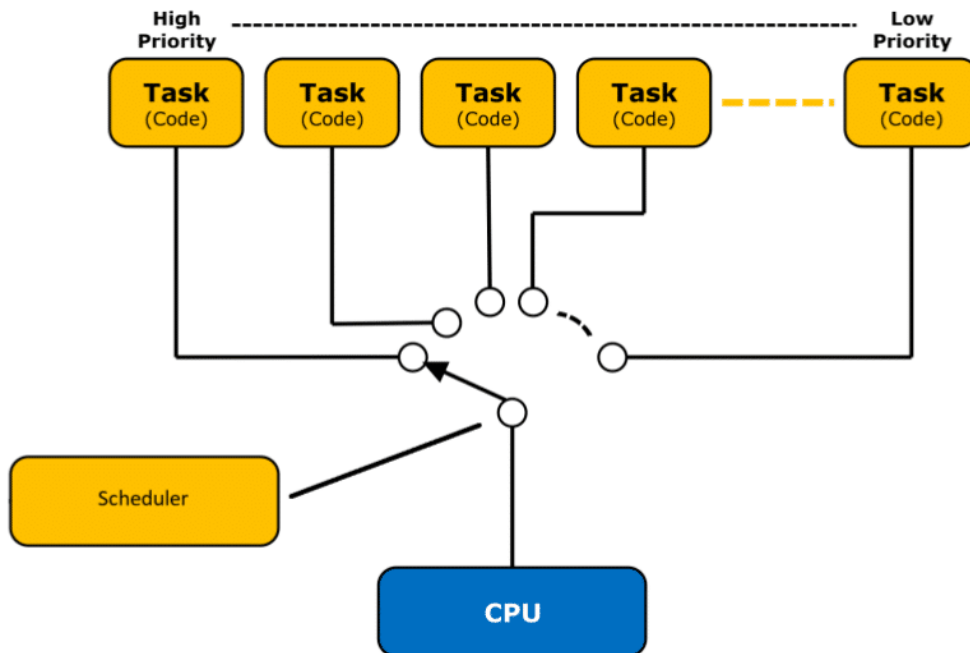
Estudante de Engenharia da Computação pela USC, pretende se aprimorar e fazer a diferença nesta imensa área da tecnologia. Apaixonado por IoT, sistemas embarcados, microcontroladores e integração da computação nos mais diversos fins práticos e didáticos.

Este site utiliza cookies. Ao usá-lo você concorda com nossos Termos de Uso.
[Saiba mais.](#)

Continuar

RTOS: Scheduler e Tarefas

Por **José Moraes** - 20/07/2018



ÍNDICE DE CONTEÚDO [MOSTRAR]

Este post faz parte da série [RTOS](#). Leia também os outros posts da série:

- [RTOS: Um ambiente multi-tarefas para Sistemas Embarcados](#)
- [RTOS: Scheduler e Tarefas](#)
- [RTOS: Semáforos para sincronização de tarefas](#)
- [RTOS: Uso de Queue para sincronização e comunicação de tarefas](#)
- [RTOS: Uso de grupo de eventos para sincronização de tarefas](#)
- [RTOS: Software Timer no FreeRTOS](#)

Este artigo é talvez o mais importante e a base de tudo o que veremos pela frente, leia com atenção. Vamos começar definindo alguns termos do scheduler e, logo depois, como uma tarefa se comporta dentro de um RTOS. Pode ser que o entendimento de ambas partes faça mais sentido para você em ordem inversa, então leia sobre as tarefas antes do scheduler.

Scheduler

Scheduler (agendador ou escalonador) é o grande responsável por administrar as tarefas que irão obter o uso da CPU. Há diversos algoritmos para que o scheduler decida a tarefa e você também pode escolher o mais apropriado ao seu embarcado,

como por exemplo: RR (Round Robin), SJF (Shortest Job First) e SRT (Shortest Remaining Time). Não entraremos em detalhes sobre eles.

Formas de trabalho do scheduler

Preemptivo: São algoritmos que permitem uma tarefa em execução ser interrompida antes do tempo total de sua execução, forçando a troca de contexto. Os motivos de interrupção são vários, desde uma tarefa com maior prioridade ou o Time Slicing (explicado logo abaixo):

Podemos observar na figura 1 como a preempção e Time Slicing funciona. As tarefas são interrompidas pois há tarefas com maiores prioridades prontas para serem executadas.

A tarefa "Idle" sempre estará presente, onde o RTOS executa alguns gerenciamentos do sistema, como gerenciamento de memória RAM. É a tarefa de menor prioridade (0), logo, todas suas tarefas restantes no sistema são aconselhadas a terem prioridade maior ou igual a 1.

Figura 1 - Scheduler preemptivo e Time Slicing alterando tarefas.

Cooperativo: São algoritmos que não permitem uma tarefa em execução ser interrompida, as tarefas precisam cooperar para que o sistema funcione. A tarefa em execução continuará em execução até que seu tempo total de execução termine e ela mesmo force a troca de contexto, assim permitindo que outras tarefas obtenham uso do CPU.

Podemos observar na figura 2, caso outras tarefas estejam prontas para serem executadas, não serão pelo simples fato de que a tarefa em atual execução deve exigir a troca de contexto ou terminar sua execução.

Figura 2 - Scheduler cooperativo alterando tarefas.

Troca de contexto: É o ato do S.O. salvar ou recuperar o estado do CPU, como registradores, principalmente o IP (Instruction Pointer). Isso permite que o S.O. retome o processamento da tarefa de onde foi interrompida.

Time Slicing: É o ato do S.O. dividir o tempo de uso do CPU entre as tarefas. Cada tarefa recebe uma fatia desse tempo, chamado quantum, e só é permitida ser executada por no máximo o tempo de um quantum. Se após o término do quantum a tarefa não liberou o uso do CPU, é forçada a troca de contexto e o scheduler será executado para decidir a próxima tarefa em execução. Caso não haja outra tarefa para ser escolhida, incluindo motivos de prioridade, o scheduler retornará para a mesma tarefa anterior à preempção. Você entenderá melhor ao decorrer deste post.

Ao término de todos quantum's, é efetuada a troca de contexto e o scheduler decidirá a próxima tarefa em execução, perdendo um pequeno tempo para si, já que irá interromper a atual tarefa em execução. O Time Slicing é indicado com tarefas de mesma prioridade, já que sem o uso, as tarefas de mesma prioridade terão tempos de

execução diferentes. No ESP32 em 240 MHz, o tempo para o scheduler decidir a tarefa e esta entrar em execução é ≤ 10 us.

No FreeRTOS, o período do Time Slicing pode ser configurado, sendo aconselhável valores entre 10 ms e 1 ms (100-1000 Hz), cabe a você escolher o que melhor atende ao seu projeto, podendo ultrapassar os limites aconselháveis.

Em todos artigos desta série, utilizaremos o FreeRTOS preemptivo com Round Robin e Time Slicing em 1000 Hz, que é o padrão do nosso microcontrolador ESP32.

Tarefas

As tarefas (Task) são como mini programas dentro do nosso embarcado, normalmente são loops infinitos que nunca retornarão um valor, onde cada tarefa efetua algo específico. Como já visto no scheduler, ele que fará todas nossas tarefas serem executadas de acordo com sua importância, e assim, conseguimos manter inúmeras tarefas em execução sem muitos problemas.

Em embarcados de apenas uma CPU, só existirá uma tarefa em execução por vez, porém, o scheduler fará a alternância de tarefas tão rapidamente, que nos dará a impressão de que todas estão ao mesmo tempo.

Prioridades

Toda tarefa tem sua prioridade, podendo ser igual a de outras. A prioridade de uma tarefa implica na ordem de escolha do scheduler, já que ele sempre irá escolher a tarefa de maior prioridade para ser executada, logo, se uma tarefa de alta prioridade sempre estiver em execução, isso pode gerar problemas no seu sistema, chamado Starvation (figura 4), já que as tarefas de menor prioridade nunca executarão até que sejam a de maior prioridade para o scheduler escolher. Por esse motivo, é sempre importante adicionar delay's no fim de cada tarefa, para que o sistema tenha um tempo para "respirar".

A menor prioridade do FreeRTOS é 0 e aumentará até o máximo explícito nos arquivos de configuração. Uma tarefa mais prioritária é a que têm o número maior que a outra, por exemplo na figura 3 veja as prioridades:

- Idle task: 0;
- Task1: 1;
- Task2: 5.

A tarefa de maior prioridade é a "Task2" e a menor é a "Idle task".

Figura 3 - Tarefa de prioridade alta gerando Starvation.

No caso de tarefas com mesma prioridade, nosso scheduler com Round Robin e Time Slicing tentará deixar todas tarefas com o mesmo tempo de uso da CPU, como mostrado na figura 4.

Figura 4 - Tarefas de prioridades iguais dividindo uso do CPU.

Estados

As tarefas sempre estarão em algum estado. O estado define o que a tarefa está fazendo dentro do RTOS no atual tempo de análise do sistema.

Bloqueada (Blocked): Uma tarefa bloqueada é quando está esperando que algum dos dois eventos abaixo ocorram. Em um sistema com muitas tarefas, a maior parte do tempo das tarefas será nesse estado, já que apenas uma pode estar em execução e todo o restante estará esperando por sua vez, no caso de sistemas single core.

- **Temporal (Timeout):** Um evento temporal é quando a tarefa está esperando certo tempo para sair do estado bloqueado, como um Delay. Todo Delay dentro de uma tarefa no RTOS não trava o microcontrolador como em Bare Metal, o Delay apenas bloqueia a tarefa em que foi solicitado, permitindo todas outras tarefas continuar funcionando normalmente.
- **Sincronização (Sync):** Um evento de sincronização é quando a tarefa está esperando (Timeout) a sincronização de outro lugar para sair do estado bloqueado, como Semáforos e Queues (serão explicados nos próximos artigos).

Suspensa (Suspended): Uma tarefa suspensa só pode existir quando for explicitamente solicitada pela função "vTaskSuspend()" e só pode sair desse estado também quando solicitado por "vTaskResume()". É uma forma de desligar uma tarefa até que seja necessária mais tarde, entretanto, é pouco usado na maioria dos sistemas simples.

Pronta (Ready): Uma tarefa está pronta para ser executada quando não está nem bloqueada, suspensa ou em execução. A tarefa pode estar pronta após o Timeout de um Delay acabar, por exemplo, entretanto, ela não estará em execução e sim esperando que o scheduler a escolha. Este tempo para a tarefa ser escolhida e executar pode variar principalmente pela sua prioridade.

Execução (Running): Uma tarefa em execução é a tarefa atualmente alocada na CPU, é ela que está em processamento. Se seu embarcado houver mais de uma CPU, haverá mais de uma tarefa em execução ao mesmo tempo. O ESP32 conta com 3 CPU's, entretanto, apenas 2 podem ser usadas pelo FreeRTOS na IDF, com isso, temos no máximo duas tarefas em execução ao mesmo tempo e o restante estará nos outros estados.

Observe na figura 5 o ciclo de vida de uma tarefa, atente-se que apenas tarefas prontas para execução podem entrar em execução diretamente. Uma tarefa bloqueada ou suspensa nunca irá para execução diretamente.

Figura 5 - Ciclo de vida de uma tarefa.

Agora que já sabemos como funciona a base do scheduler e suas tarefas, vamos finalmente botar a mão na massa e testar essa maravilha funcionando na prática.

Como já foi dito, vamos utilizar o ESP32 com a IDF e FreeRTOS para nossos testes, mas você pode testar em seu embarcado como ARM, talvez mudando apenas alguns detalhes!

Vamos fazer um teste simples com três tarefas para mostrar os principais itens acima. O princípio dessas três tarefas é apenas mostrar duas tarefas com mesma prioridade dividindo o uso da CPU enquanto a terceira tarefa é executada poucas vezes para mostrar a preempção.

Código do projeto:

```

1  #include <driver/gpio.h>
2  #include <freertos/FreeRTOS.h>
3  #include <freertos/task.h>
4  #include <esp_system.h>
5
6  void t1(void*z)
7  { //Tarefa que simula PWM para observar no analisador logico
8      while (1)
9      {
10         gpio_set_level(GPIO_NUM_2, 1);
11         ets_delay_us(25);
12         gpio_set_level(GPIO_NUM_2, 0);
13         ets_delay_us(25);
14     }
15 }
16
17 void t2(void*z)
18 { //Tarefa que simula PWM para observar no analisador logico
19     while (1)
20     {
21         gpio_set_level(GPIO_NUM_4, 1);
22         ets_delay_us(25);
23         gpio_set_level(GPIO_NUM_4, 0);
24         ets_delay_us(25);
25     }
26 }
27
28 void t3(void*z)
29 { //Tarefa que simula PWM para observar no analisador logico
30     while (1)
31     {
32         for (uint8_t i = 0; i < 200; i++)
33         {
34             gpio_set_level(GPIO_NUM_15, 1);
35             ets_delay_us(25);
36             gpio_set_level(GPIO_NUM_15, 0);
37             ets_delay_us(25);
38         }
39     }
40 }

```

Primeiramente, vamos analisar o que a teoria acima nos diz. Com duas tarefas compartilhando a mesma prioridade, nosso gráfico irá se comportar igual à figura 4, entretanto, nós temos uma terceira tarefa que tem prioridade maior que as outras duas. Essa última é executada menos frequentemente, porém por mais tempo que as outras. Nosso gráfico deve ficar parecido com a figura 6:

Figura 6 - Funcionamento teórico do código.

Observe que as duas tarefas de prioridades iguais (Task1 e Task2) compartilham o tempo de uso do CPU enquanto a outra tarefa (Task3) permanece bloqueada por um delay. Logo que a Task3 está pronta para ser executada, o scheduler irá executá-la até que encontre o delay novamente, que é quando a tarefa fica bloqueada, permitindo tarefas de prioridade menor serem executadas. Lembre-se que delay não trava todo o microcontrolador, apenas a tarefa em que foi solicitado.

Ok, vamos ver se a teoria bate com a prática? Cada tarefa faz um simples Toggle em um pino nos permitindo analisar com o Analisador Lógico nas figuras 7,8 e 9.

Figura 7 - Aplicando as tarefas na prática.

As tasks 1 e 2 compartilham o uso do CPU até que a task3 seja desbloqueada e, quando isso ocorre, a task3 que tem maior prioridade, será executada até o término do seu código (quando encontra o delay). Observe o tempo de 200 ms no canto superior direito da figura 7, é nosso delay de 200 ms da tarefa 3, ou seja, ela realmente permaneceu bloqueada por 200 ms.

Dando um zoom, podemos observar as duas tarefas “brigando” pela CPU, veja na figura 8. No canto superior direito da figura 8, a tarefa usa a CPU por 1 ms, que é o tempo do Time Slicing configurado em 1000 Hz.

Figura 8 - Tarefas de prioridade igual concorrendo pelo CPU.

Observando o gráfico numa base de tempo “humana” (figura 9), para nós parece que ambas estão executando simultaneamente, entretanto, podemos provar que as duas compartilham a CPU (figura 8), atente-se a isso em seus projetos de “Time Critical”.

Podemos ir mais longe já que nosso microcontrolador permite 2 das 3 CPU's serem usadas pelo FreeRTOS. Atribuindo a tarefa 3 no outro CPU (1), podemos ver na figura 9 que ela executará realmente em simultâneo enquanto as tarefas 1 e 2 “brigam” pelo CPU (0).

Figura 9 - FreeRTOS em embarcado Multi-Core.

No próximo artigo desta série vamos abordar os semáforos, que funcionam para sincronizar eventos e tarefas dentro do RTOS.

Saiba mais

[Desenvolvendo um RTOS: Introdução](#)

[Implementando elementos de RTOS no Arduino](#)

[Criando um projeto no IAR com o FreeRTOS](#)

Referências

https://freertos.org/Documentation/RTOS_book.html 

<http://esp-idf.readthedocs.io/en/latest/api-reference/system/freertos.html> 

Outros artigos da série

<< RTOS: Um ambiente multi-tarefas para Sistemas Embarcados RTOS: Semáforos para sincronização de tarefas >>

 Facebook 0

 Twitter 0

 Google+ 3

 LinkedIn 1

Este post faz da série **RTOS**. Leia também os outros posts da série:

- **RTOS: Um ambiente multi-tarefas para Sistemas Embarcados**
- **RTOS: Scheduler e Tarefas**
- **RTOS: Semáforos para sincronização de tarefas**
- **RTOS: Uso de Queue para sincronização e comunicação de tarefas**
- **RTOS: Uso de grupo de eventos para sincronização de tarefas**
- **RTOS: Software Timer no FreeRTOS**

NEWSLETTER

Receba os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.

E-mail

CADASTRAR E-MAIL

Fique tranquilo, também não gostamos de spam.

RTOS: Scheduler e Tarefas por **José Morais**. Esta obra está licenciado com uma Licença **Creative Commons Atribuição-Compartilhual 4.0 Internacional** .

José Morais

Estudante de Engenharia da Computação pela USC, pretende se aprimorar e fazer a diferença nesta imensa área da tecnologia. Apaixonado por IoT, sistemas embarcados, microcontroladores e integração da computação nos mais diversos fins práticos e didáticos.

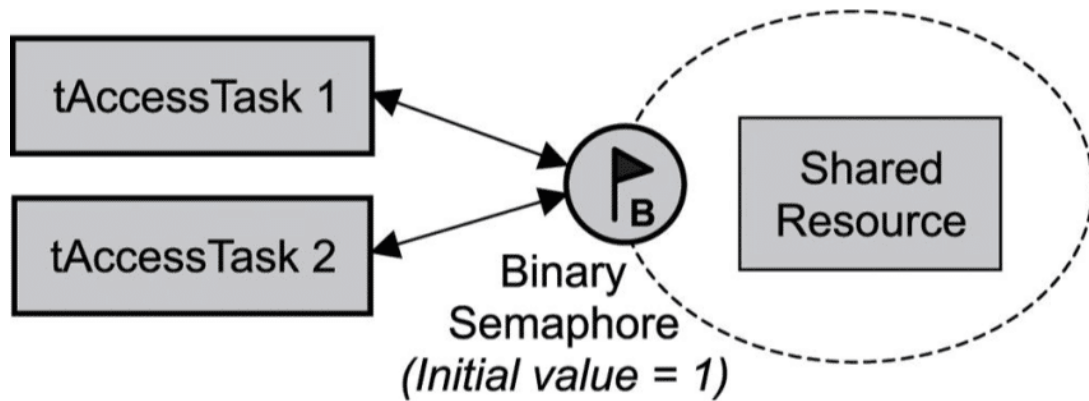
Este site utiliza cookies. Ao usá-lo você concorda com nossos Termos de Uso.

[Saiba mais.](#)

Continuar

RTOS: Semáforos para sincronização de tarefas

Por **José Moraes** - 02/08/2018



ÍNDICE DE CONTEÚDO [MOSTRAR]

Este post faz parte da série **RTOS**. Leia também os outros posts da série:

- [RTOS: Um ambiente multi-tarefas para Sistemas Embarcados](#)
- [RTOS: Scheduler e Tarefas](#)
- [RTOS: Semáforos para sincronização de tarefas](#)
- [RTOS: Uso de Queue para sincronização e comunicação de tarefas](#)
- [RTOS: Uso de grupo de eventos para sincronização de tarefas](#)
- [RTOS: Software Timer no FreeRTOS](#)

Sistemas operacionais multi-tarefas sofrem com um grande problema, a concorrência de recursos. Podemos ter várias tarefas usando um mesmo recurso, como um periférico, entretanto, ele só pode fazer uma coisa por vez. E isso gera problemas de concorrência, quando duas ou mais tarefas precisam do mesmo recurso ao mesmo tempo.

O método mais simples para resolver esses problemas é com semáforos, que são como variáveis 0 e 1. Entretanto, há mais recursos que permitem nosso código fluir incrivelmente melhor, como Timeout e Atomicidade.

O que são semáforos?

Semáforos, em programação paralela, são um tipo abstrato de dado que visa a sincronização de tarefas restringindo o acesso de um recurso ou comunicação entre tarefas e ISRs. Os semáforos trabalham de uma forma muito simples e parecida com os semáforos das vias urbanas, onde tentam sincronizar o fluxo de carros.

Os semáforos contam com duas operações básicas, que é quando a tarefa pega (take) o semáforo para uso e depois o libera (give) para outra tarefa ou ela mesmo usá-lo, que estão descritas abaixo e nas figuras 1 e 2.

Take (obter o semáforo): Se o semáforo tiver o valor ≥ 1 que é quando disponível, a tarefa poderá obtê-lo, já que o semáforo está disponível para uso e a função retornará TRUE. Caso o semáforo esteja indisponível, a função retornará FALSE. Ao obter o semáforo, o valor dele é decrementado em 1.

Give (liberar o semáforo): Normalmente após a tarefa obter o semáforo com o "take", no fim de seu processamento, deve liberar o semáforo para outras tarefas o utilizarem. Ao liberar o semáforo, o valor dele é incrementado em 1.

Atomicidade: Em sistemas multi-tarefas ou multi-cores, temos um problema comum que é quando duas tarefas ou processadores tentam usar o mesmo recurso e isso pode gerar problemas. Um processador pode obter um semáforo exatamente ao mesmo tempo que o outro processador também está obtendo, gerando conflitos. Um item ser atômico nos diz que ele é "indivisível", ou seja, dois processadores ou tarefas não podem entrar na mesma região do código ao mesmo tempo, conhecido como "regiões críticas".

Na figura 1, a "Task" está esperando a ISR liberar o semáforo para prosseguir seu processamento e, enquanto isso não ocorre, a "Task" permanece em espera (bloqueada) permitindo que outras tarefas sejam executadas.

Figura 1 - Tarefa esperando e utilizando o semáforo.

Na figura 2, a "Task2" está esperando o semáforo ser liberado para ela mesmo iniciar seu processamento, enquanto isso, ela permanece "dormindo (bloqueada)", permitindo que outras tarefas continuem sendo executadas.

Figura 2 - Tarefa em espera pelo semáforo.

Tipos de semáforos

Binário: É o mais simples dos semáforos, como uma única variável valendo "1" ou "0". Quando em "1", permite que tarefas o obtenham e quando "0", a tarefa não conseguirá obtê-lo.

Mutex: Similar ao binário, entretanto, implementa a herança de prioridade. A herança de prioridade é uma implementação do FreeRTOS que aumenta a prioridade da tarefa que está usando o semáforo quando é interrompida por outra tarefa que também está tentando usar o semáforo. Isso garante que a tarefa que está usando o semáforo tenha a maior prioridade entre todas as outras que tentarem obtê-lo. Este método tenta minimizar o problema de inversão de prioridades.

Counting: Similar ao binário mas conta com uma fila de valores, similar a um vetor (array). É muito utilizado para minimizar problemas entre ISR e os outros semáforos, já que se ocorrer mais de uma ISR antes que a tarefa o obtenha, perderemos essa ISR visto que os outros semáforos só têm um "espaço". Utilizando o semáforo counting, não perdemos a ISR já que temos vários "espaços" (figura 3), sendo similar a uma Queue que vamos aprender no próximo artigo.

Figura 3 - Semáforo Counting.

Na figura 4, a "Task1" e "Task2" estão esperando pelo uso do semáforo binário que é liberado pela ISR, porém, só uma conseguirá executar por vez, que será a de maior prioridade. Ou seja, podemos causar conflitos (Starvation) no sistema, já que apenas uma tarefa de várias irá obter o uso do semáforo, uma solução para isso é o semáforo counting ou criar semáforos separados para cada tarefa.

- Nos tempos t2 e t3 as tarefas começam a esperar o semáforo com um Timeout "infinito" (será explicado na prática).
- A ISR (interrupção por hardware) libera o semáforo. Qualquer prioridade de ISR é maior que qualquer prioridade do FreeRTOS.
- Logo que o semáforo é liberado, a tarefa de maior prioridade (Task1) será escolhida pelo scheduler, ocasionando Starvation na Task2.

Figura 4 - Concorrência de tarefas pelo semáforo binário.

Vamos finalmente botar a mão na massa e testar o semáforo binário para sincronizar tarefas a partir de uma ISR. Como no artigo anterior, vamos comparar a teoria com a prática. O método que faremos é conhecido como DIH (Deferred Interrupt Handling), onde buscamos remover o processamento da ISR e atribuí-lo a uma tarefa, visto que ISR deve ser o mais rápido possível.

Nosso código terá uma tarefa responsável por analisar o semáforo binário e vamos trabalhar com o Timeout do semáforo, ou seja, se o semáforo não estiver disponível dentro do tempo definido, efetuará uma ação (figura 5).

- Quando a tarefa obter o semáforo dentro do tempo definido, será feito um Toggle no GPIO23.
- Quando a tarefa não obter o semáforo dentro do tempo definido, será feito um Toggle no GPIO22.
- t1, t2, t3, t4, t9 e t10 são marcações quando o Timeout + Delay expirou (300 ms).

Figura 5 - Funcionamento teórico do código.

Código do projeto:

```
1 #include <driver/gpio.h>
2 #include <freertos/FreeRTOS.h>
3 #include <freertos/task.h>
4 #include <freertos/semphr.h>
5 #include <esp_system.h>
```

```
6
7 SemaphoreHandle_t SMF; //Objeto do semaforo
8
9 void isr(void*z)
10 {
11     BaseType_t aux = false; //Variavel de controle para a Troca de Contexto
12     xSemaphoreGiveFromISR(SMF, &aux); //Libera o semaforo
13
14     if (aux)
15     {
16         portYIELD_FROM_ISR(); //Se houver tarefas esperando pelo semaforo, deve ser forçado a Troc
17     }
18
19 }
20
21
22 void t1(void*z)
23 {
24     while(1)
25     {
26         if (xSemaphoreTake(SMF, pdMS_TO_TICKS(200)) == true) //Tenta obter o semaforo durante 200m
27         {
28             //Se obteve o semaforo entre os 200ms de espera, fara o toggle do pino 23
29
30             for (uint8_t i = 0; i < 10; i++)
31             {
32                 gpio_set_level(GPIO_NUM_23, 1);
33                 delay_us(150);
34             }
35         }
36     }
37 }
```

Agora, vamos ver o que o analisador lógico nos diz com a figura 6:

Figura 6 - Funcionamento prático do código.

Veja que funcionou como o esperado, o Toggle do GPIO22 está em 300 ms, quando ocorre o Timeout do semáforo mais o delay (canto superior direito) e quando ocorre a ISR, o semáforo é liberado e a tarefa efetua o Toggle do GPIO23.

Além da DIH, os semáforos são muito utilizados para sincronização entre tarefas que tentam obter o mesmo recurso, como um periférico, porém, ele só pode efetuar uma ação (ou menos que a quantidade de tarefas tentando usá-lo) por vez. Vamos entender melhor como isso funciona na figura 7.

Figura 7 - Tarefas concorrendo pelo periférico.

Podemos observar que o periférico é utilizado em todo o tempo de análise, entretanto, por tarefas diferentes que são sincronizadas por um semáforo binário:

- No tempo t1, a Task2 obtém o semáforo para utilizar o periférico.

- No tempo t2, a Task1 tenta obter o semáforo, porém, não está disponível e a tarefa entra em espera até que seja liberado.
- No tempo t4, a Task2 libera o semáforo e a Task1 acordará automaticamente para obtê-lo.
- No tempo t5, a Task1 libera o semáforo para outras tarefas utilizarem.

No próximo artigo desta série, vamos aprender sobre Filas de dados (Queues), que funcionam similarmente ao semáforo, entretanto, podemos passar valores! Isso permite, além de algo parecido com o semáforo, comunicação entre tarefas ou ISRs.

Saiba mais

[Desenvolvendo um RTOS: Introdução](#)

[Implementando elementos de RTOS no Arduino](#)

[Criando um projeto no IAR com o FreeRTOS](#)

Referências

https://freertos.org/Documentation/RTOS_book.html 

<http://esp-idf.readthedocs.io/en/latest/api-reference/system/freertos.html> 

Outros artigos da série

<< RTOS: Scheduler e Tarefas RTOS: Uso de Queue para sincronização e comunicação de tarefas >>

Este post faz da série [RTOS](#). Leia também os outros posts da série:

- [RTOS: Um ambiente multi-tarefas para Sistemas Embarcados](#)
- [RTOS: Scheduler e Tarefas](#)
- [RTOS: Semáforos para sincronização de tarefas](#)
- [RTOS: Uso de Queue para sincronização e comunicação de tarefas](#)
- [RTOS: Uso de grupo de eventos para sincronização de tarefas](#)
- [RTOS: Software Timer no FreeRTOS](#)


NEWSLETTER

Receba os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.

E-mail

CADASTRAR E-MAIL

Fique tranquilo, também não gostamos de spam.

RTOS: Semáforos para sincronização de tarefas por **José Morais**. Esta obra está licenciado com uma Licença [Creative Commons Atribuição-Compartilha Igual 4.0 Internacional](#) .

José Morais

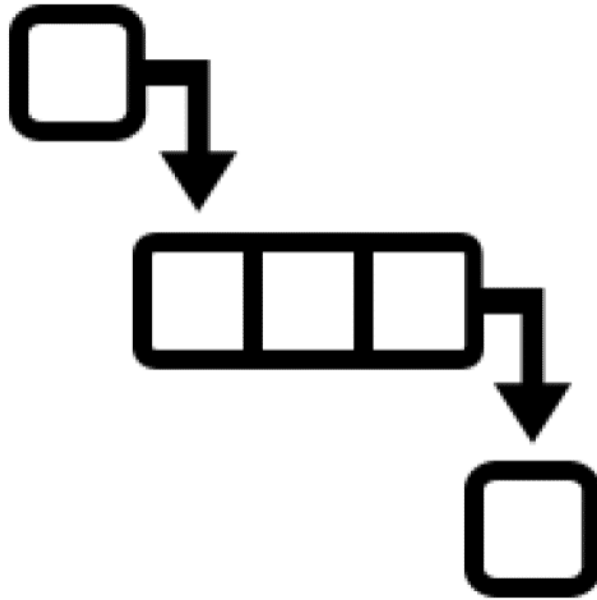
Estudante de Engenharia da Computação pela USC, pretende se aprimorar e fazer a diferença nesta imensa área da tecnologia. Apaixonado por IoT, sistemas embarcados, microcontroladores e integração da computação nos mais diversos fins práticos e didáticos.

Este site utiliza cookies. Ao usá-lo você concorda com nossos Termos de Uso.
[Saiba mais.](#)

Continuar

RTOS: Uso de Queue para sincronização e comunicação de tarefas

Por **José Moraes** - 07/08/2018



Este post faz parte da série [RTOS](#). Leia também os outros posts da série:

- [RTOS: Um ambiente multi-tarefas para Sistemas Embarcados](#)
- [RTOS: Scheduler e Tarefas](#)
- [RTOS: Semáforos para sincronização de tarefas](#)
- [RTOS: Uso de Queue para sincronização e comunicação de tarefas](#)
- [RTOS: Uso de grupo de eventos para sincronização de tarefas](#)
- [RTOS: Software Timer no FreeRTOS](#)

Os semáforos que vimos no artigo anterior permitem a sincronização de tarefas e uma simples comunicação entre elas com apenas valores binários. Mas se seu projeto precisa de algo mais complexo para sincronização ou comunicação entre tarefas e ISRs, pode ser a vez da Queue.

O que é uma Queue e como funciona?

Queue é um buffer, uma fila de dados no formato FIFO, que permite sincronização de tarefas, como vimos em semáforos, porém é mais utilizada para comunicação entre tarefas e ISRs onde precisamos enviar valores (variáveis) para outros lugares. Usar queues visa a diminuição de variáveis globais em seu código. Mesmo ele sendo um

objeto global, conta com métodos de atomicidade e timeout que podem agregar uma dinâmica muito interessante em seu projeto e produto.

Figura 1 - FIFO.

Figura 2 - Animação de uma Queue FIFO.

A queue funciona na forma de um buffer FIFO, mas há funções que permitem escrever no começo do buffer em vez de apenas no fim. É tão simples que com a figura 3 já podemos entender o funcionamento da queue e testar na prática!

Figura 3 - Escrita e leitura da queue.

As queues do FreeRTOS são objetos com capacidade e comprimento fixo. Como o exemplo da figura 3, a queue foi criada com 5 “espaços” (slots) e 2 bytes por slot. Isso implica na forma e frequência em que a queue será utilizada. Ela deve atender o tamanho máximo de suas variáveis e também ter comprimento considerável para evitar que fique lotada (a menos que seja proposital, como nas “MailBox”). Apesar da queue também funcionar para sincronização de tarefas como vimos em semáforos, o foco dessa prática será unicamente na transferência de dados entre duas tarefas. Então vamos começar!

Código do projeto:

```
1 #include <driver/gpio.h>
2 #include <freertos/FreeRTOS.h>
3 #include <freertos/task.h>
4 #include <freertos/semphr.h>
5 #include <esp_system.h>
6 #include <esp_log.h>
7
8 QueueHandle_t buffer; // Objeto da queue
9
10 void t1(void*z)
```

```
11 {  
12     uint32_t snd = 0;  
13     while(1)  
14     {  
15         if (snd < 15) //se menor que 15  
16         {  
17             xQueueSend(buffer, &snd, pdMS_TO_TICKS(0)); //Envia a variavel para queue  
18             snd++; //incrementa a variavel  
19         }  
20         else //se nao, espera 5seg para testar o timeout da outra tarefa  
21         {  
22             vTaskDelay(pdMS_TO_TICKS(5000));  
23             snd = 0;  
24         }  
25         vTaskDelay(pdMS_TO_TICKS(500));  
26     }  
27 }  
28 }  
29  
30 void t2(void*z)  
31 {  
32     uint32_t rcv = 0;  
33     while(1)  
34     {
```

Figura 4 - Tarefas se comunicando por Queues.

Testando o código, podemos observar que enquanto a tarefa responsável por enviar variáveis (**t1**) envia valores abaixo de 15, a outra tarefa (**t2**) mostra o valor recebido na tela. Porém, quando **t1** chega no número 15, entra em delay por 5 segundos, ocasionando no timeout da leitura na **t2**, mostrando na tela que o timeout de 1 segundo expirou.

Queues podem tanto funcionar como um tipo especial de semáforos, onde podemos analisar valores recebidos e sincronizar tarefas a partir disso, como também, e principalmente, efetuar a comunicação entre tarefas e ISRs.

Saiba mais

Desenvolvendo com o Zephyr RTOS: Controlando o Kernel

Primeiras impressões com o react.o

Fila circular para sistemas embarcados

Referências

https://freertos.org/Documentation/RTOS_book.html 

<http://esp-idf.readthedocs.io/en/latest/api-reference/system/freertos.html> 

Outros artigos da série

<< RTOS: Semáforos para sincronização de tarefas RTOS: Uso de grupo de eventos para sincronização de tarefas >>

 Facebook 20

 Twitter 0

 Google+ 0

 LinkedIn 1

Este post faz da série **RTOS**. Leia também os outros posts da série:

- RTOS: Um ambiente multi-tarefas para Sistemas Embarcados
- RTOS: Scheduler e Tarefas
- RTOS: Semáforos para sincronização de tarefas
- RTOS: Uso de Queue para sincronização e comunicação de tarefas
- RTOS: Uso de grupo de eventos para sincronização de tarefas
- RTOS: Software Timer no FreeRTOS

NEWSLETTER

Receba os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.

E-mail

CADASTRAR E-MAIL

Fique tranquilo, também não gostamos de spam.

RTOS: Uso de Queue para sincronização e comunicação de tarefas por *José Morais*.

Esta obra está licenciado com uma Licença [Creative Commons Atribuição-Compartilhual 4.0 Internacional](#) [↗](#).

José Morais

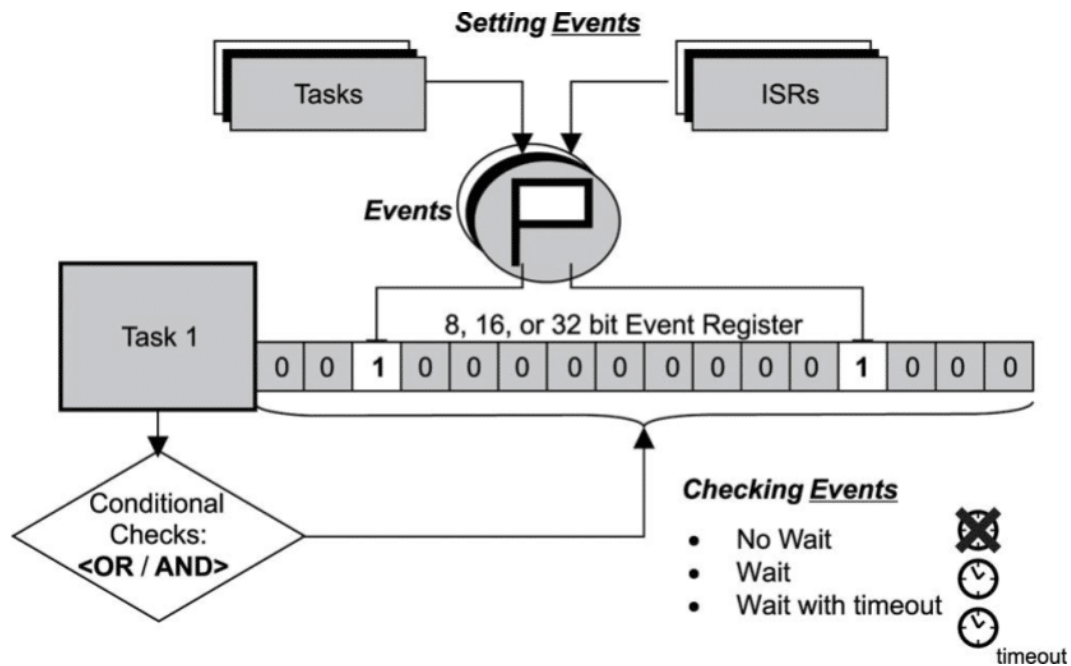
Estudante de Engenharia da Computação pela USC, pretende se aprimorar e fazer a diferença nesta imensa área da tecnologia. Apaixonado por IoT, sistemas embarcados, microcontroladores e integração da computação nos mais diversos fins práticos e didáticos.

Este site utiliza cookies. Ao usá-lo você concorda com nossos Termos de Uso.
[Saiba mais.](#)

Continuar

RTOS: Uso de grupo de eventos para sincronização de tarefas

Por **José Morais** - 14/09/2018



Este post faz parte da série [RTOS](#). Leia também os outros posts da série:

- [RTOS: Um ambiente multi-tarefas para Sistemas Embarcados](#)
- [RTOS: Scheduler e Tarefas](#)
- [RTOS: Semáforos para sincronização de tarefas](#)
- [RTOS: Uso de Queue para sincronização e comunicação de tarefas](#)
- [RTOS: Uso de grupo de eventos para sincronização de tarefas](#)
- [RTOS: Software Timer no FreeRTOS](#)

FreeRTOS oferece, além dos 2 métodos anteriores para comunicação e sincronização de tarefas, os grupos de eventos (abreviando para G.E.). Um G.E. pode armazenar diversos eventos (flags) e, com isso, podemos deixar uma tarefa em espera (estado bloqueado) até que um ou vários eventos específicos ocorram.

O que é e como funciona um grupo de eventos?

Antes de entender um grupo de eventos, vamos relembrar o que são eventos (flags ou bandeiras). Flags são variáveis (normalmente binárias) que indicam quando algo ocorreu ou não, como por exemplo:

1. Um sensor ativa uma flag ao término de sua leitura;
2. Uma interrupção atribuída a um botão ativa uma flag indicando que ele foi pressionado.

Sendo assim, nosso embarcado pode ter muitas flags indicando diferentes coisas para que possamos tomar atitudes de acordo com a ocorrência delas.

Os G.E são variáveis de 16 ou 32 bits (definido pelo usuário) onde cada bit é uma flag específica indicando que algo ocorreu ou não. Nos nossos exemplos, usaremos os grupos de eventos de 32 bits, sendo 24 bits de uso livre e 8 reservados. Já que cada objeto do grupo de eventos permite manipular até 24 eventos diferentes, dependendo do tamanho do seu projeto, pode-se economizar uma preciosa quantidade de memória RAM, comparado com o uso de semáforos binários ou queues para cada evento.

Veja um exemplo das flags em um grupo de eventos na figura 1 abaixo:

Figura 1 - Exemplo de flags em um grupo de eventos.

Diferentemente dos semáforos e queues, um grupo de eventos tem algumas características importantes:

1. Podemos acordar as tarefas com combinação de uma ou várias flags diferentes;
2. Reduz o consumo de memória RAM se comparado quando usado vários semáforos binários ou queues para sincronização;
3. **Todas** as tarefas que estão esperando (estado bloqueado) pelas flags, são desbloqueadas, funcionando como uma "mensagem broadcast" e não apenas a de maior prioridade, como acontece com os semáforos e queues. Vamos observar essa **importante** característica nas figuras 2 e 3.

Na figura 2, duas tarefas (T1 e T2) esperam pelo mesmo semáforo binário (ou queue) para executar certa ação. Entretanto, quando a tarefa (Main) libera o semáforo para uso, apenas a tarefa de maior prioridade (T2) obtém o semáforo, causando Starvation na tarefa (T1), já que ela nunca será executada.

Figura 2 - Tarefas esperando pelo mesmo semáforo.

Na figura 3 abaixo, há a mesma lógica porém com grupo de eventos, que desbloqueia todas tarefas (broadcast) e não apenas a de maior prioridade. Este detalhe é muito importante quando você pretende ter mais de uma tarefa esperando pelo mesmo item.

Figura 3 - Tarefas esperando pelo mesmo evento.

Como já foi dito, um grupo de eventos é um conjunto de bits onde cada um tem seu significado específico, logo, precisamos entender o básico sobre manipulação de bits na linguagem C e você pode aprender mais vendo este ótimo artigo do Fábio Souza [aqui](#). Vamos testar um G.E. na prática, onde uma tarefa espera por um flag para executar sua rotina.

Código do projeto:

```
1 #include <freertos/FreeRTOS.h>
2 #include <freertos/task.h>
3 #include <freertos/event_groups.h>
4 #include <esp_system.h>
5 #include <esp_log.h>
6
7 EventGroupHandle_t evt; // Cria o objeto do grupo de eventos
8
9 #define EV_1SEG (1<<0) // Define o BIT do evento
10
11
12 void t1(void*z)
13 {
14     EventBits_t x; // Cria a variável que recebe o valor dos eventos
15     while (1)
16     {
17         // Vamos esperar pelo evento (EV_1SEG) por no máximo 1000ms
18         x = xEventGroupWaitBits(evt, EV_1SEG, true, true, pdMS_TO_TICKS(1000));
19
20         if (x & EV_1SEG) // Se X & EV_1SEG (máscara binária), significa que o evento ocorreu
21         {
```

```

22     ESP_LOGI("T1", "OK");
23 }
24 else
25 {
26     ESP_LOGE("T1", "Event group TIMEOUT");
27 }
28 }
29 }
30
31 extern "C" void app_main()
32 {
33     evt = xEventGroupCreate(); //Cria o grupo de eventos

```

Testando esse simples código, podemos ver que é praticamente idêntico ao funcionamento dos semáforos e queues, porém, como foi dito anteriormente, podemos manter a tarefa bloqueada por um ou mais eventos e isso é feito através de uma pequena lógica, veja a seguir.

Vamos resumir e apenas mostrar a parte onde definimos os eventos e a função que os aguarda.

```

1 #define EV_1 (1<<0)
2 #define EV_2 (1<<1)
3 #define EV_3 (1<<2)
4
5 x = xEventGroupWaitBits(evt, EV_1 | EV_2 | EV_3, true, true, pdMS_TO_TICKS(1000));

```

Com o 4º parâmetro da função em **true**, a função irá esperar, durante 1 segundo, até que **todos** os três eventos fiquem valendo 1. Se qualquer um dos três não ocorrer **dentro do tempo limite**, a função retornará apenas após o tempo limite.

Além disso, temos mais uma possibilidade que é configurar a função para esperar por qualquer um dos três e não mais todos os três juntos, veja abaixo:

```

1 #define EV_1 (1<<0)
2 #define EV_2 (1<<1)
3 #define EV_3 (1<<2)
4
5 x = xEventGroupWaitBits(evt, EV_1 | EV_2 | EV_3, true, false, pdMS_TO_TICKS(1000));

```

Com o 4º parâmetro da função em **false**, a função irá esperar, durante 1 segundo, por **qualquer** um dos três eventos. Se qualquer um dos três eventos ocorrer, a função retornará, imediatamente.

A função sempre retorna o valor atual do grupo de eventos, logo, você precisa aplicar máscaras binárias para descobrir se o evento específico ocorreu. O artigo citado

acima do Fábio Souza, pode te ajudar a entender máscaras binárias!

Os grupos de eventos são muito importantes em diversos ecossistemas, já que permite a propagação global de eventos (broadcast) por todas tarefas. Muito similar aos semáforos e queues, é de importância para qualquer projetista que pretende seguir com RTOS.

Saiba mais

[Desenvolvendo com o Zephyr RTOS: Controlando o Kernel](#)

[Primeiras impressões com o react.o](#)

[Fila circular para sistemas embarcados](#)

Referências

<https://www.freertos.org/event-groups-API.html> 

https://freertos.org/Documentation/RTOS_book.html 

<http://esp-idf.readthedocs.io/en/latest/api-reference/system/freertos.html> 

Outros artigos da série

<< RTOS: Uso de Queue para sincronização e comunicação de tarefas
RTOS: Software Timer no FreeRTOS >>

 Facebook 14

 Twitter 0

 Google+ 0

 LinkedIn 0

Este post faz da série [RTOS](#). Leia também os outros posts da série:

- [RTOS: Um ambiente multi-tarefas para Sistemas Embarcados](#)
- [RTOS: Scheduler e Tarefas](#)
- [RTOS: Semáforos para sincronização de tarefas](#)
- [RTOS: Uso de Queue para sincronização e comunicação de tarefas](#)
- [RTOS: Uso de grupo de eventos para sincronização de tarefas](#)
- [RTOS: Software Timer no FreeRTOS](#)

NEWSLETTER




Receba os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.

E-mail

CADASTRAR E-MAIL

Fique tranquilo, também não gostamos de spam.

RTOS: Uso de grupo de eventos para sincronização de tarefas por **José Moraes**. Esta obra está licenciado com uma Licença [Creative Commons Atribuição-Compartilhagual 4.0 Internacional](#) .

José Moraes

Estudante de Engenharia da Computação pela USC, pretende se aprimorar e fazer a diferença nesta imensa área da tecnologia. Apaixonado por IoT, sistemas embarcados, microcontroladores e integração da computação nos mais diversos fins práticos e didáticos.

Este site utiliza cookies. Ao usá-lo você concorda com nossos Termos de Uso.
[Saiba mais.](#)

Continuar

RTOS: Software Timer no FreeRTOS

Por **José Moraes** - 17/09/2018



ÍNDICE DE CONTEÚDO [MOSTRAR]

Este post faz parte da série [RTOS](#). Leia também os outros posts da série:

- [RTOS: Um ambiente multi-tarefas para Sistemas Embarcados](#)
- [RTOS: Scheduler e Tarefas](#)
- [RTOS: Semáforos para sincronização de tarefas](#)
- [RTOS: Uso de Queue para sincronização e comunicação de tarefas](#)
- [RTOS: Uso de grupo de eventos para sincronização de tarefas](#)
- [RTOS: Software Timer no FreeRTOS](#)

Sistemas embarcados contam com diversos timers físicos (hardware) para uso interno, como em PWM, ou de uso geral com extrema precisão e imunidade a interferências do código. Porém, em muitos projetos, podemos precisar de muito mais timers do que o embarcado nos disponibiliza e aí entra o software timer. Uma das únicas limitações para a quantidade de software timers no seu embarcado é a RAM disponível e, no ESP32 com ~300 KB de RAM livre, podemos criar até ~5800 timers independentes se for necessário.

Quais as vantagens e desvantagens do software timer no FreeRTOS?

Os software timers funcionam da mesma forma que os hardware timers, entretanto, há alguns detalhes que vale ressaltar:

1. Assim como o hardware timer, o software timer não utiliza nenhum processamento da CPU enquanto ativo;
2. Pode ser comparado a uma tarefa do RTOS;
3. Todos os comandos do timer são enviados à tarefa "Timer Service ou Daemon task" por uma queue, o que vamos entender mais à frente.
4. One-shot e Auto-reload.

Vantagens

- Não precisa de suporte do hardware, sendo de total responsabilidade do FreeRTOS;
- Limite de timers é a memória disponível;
- Fácil implementação.

Desvantagens

- Latência varia de acordo com a prioridade da tarefa "Timer Service" e frequência do FreeRTOS, ambos configuráveis;
- Pode perder comandos se usado excessivamente durante um curto período de tempo, já que a comunicação com a tarefa "Timer Service" é feita por uma queue, a qual pode ficar lotada.

Timer Service ou Daemon task

Essa tarefa é iniciada automaticamente junto ao scheduler caso você habilite os software timer no FreeRTOS. Ela é responsável por receber e executar os comandos sobre timers e também executar a função de callback quando o timer expira. Essa tarefa funciona exatamente igual a qualquer outra tarefa do RTOS, então, se houver dúvidas, basta se aprofundar nos detalhes sobre tarefas do RTOS. Como foi dito anteriormente, a prioridade é configurável e altamente aconselhável deixá-la maior que todas suas outras tarefas.

Veja nas figuras 1 e 2 a comunicação com a tarefa "Timer Service" através da queue.

Figura 1 - Comunicação com a Timer Service por queue.

Figura 2 - Timer Service efetuando preempção da própria tarefa que está configurando o timer.

Ao usar excessivamente comandos de controle do timer, a queue responsável por essa comunicação pode ficar lotada e, caso você deixe o parâmetro de espera das funções em zero, o comando será perdido. Por esse motivo, é sempre aconselhado a colocar um tempo de espera para que as funções aguardem a queue esvaziar. O motivo mais comum para lotar a queue da tarefa "Timer Service" é utilizar os comandos dentro de ISRs, já que uma ISR tem prioridade mais alta e não deixará tempo para a "Timer Service" processar dados anteriores caso venha acontecer alguma oscilação ou bouncing, por exemplo.

Vamos botar a mão na massa e criar alguns timers para aprender a usá-los e também verificar a frequência para saber se é estável ou não.

Código do projeto

```
1 #include <freertos/FreeRTOS.h>
2 #include <freertos/task.h>
3 #include <freertos/semphr.h>
4 #include <freertos/timers.h>
```



```
5 #include <esp_system.h>
6 #include <esp_log.h>
7
8 SemaphoreHandle_t smf;//Cria o objeto do semaforo
9 TimerHandle_t tmr;//Cria o objeto do timer
10
11 void ISR(void*z)
12 {
13     int aux = 0;
14     xSemaphoreGiveFromISR(smf, &aux);//Libera o semaforo para uso
15
16     if (aux)
17     {
18         portYIELD_FROM_ISR();//Forca a troca de contexto se necessario
19     }
20 }
21
22 extern "C" void app_main()
23 {
24     smf = xSemaphoreCreateBinary();//Cria o semaforo
25
26     tmr = xTimerCreate("tmr_smf", pdMS_TO_TICKS(1000), true, 0, ISR);//Cria o timer com 1000 ms
27     xTimerStart(tmr, pdMS_TO_TICKS(100));//Inicia o timer
28
29     while (1)
30     {
31         if (xSemaphoreTake(smf, portMAX_DELAY))
32         {
33             ESP_LOGI("Timer", "expirou!");//Ao obter o semaforo, enviara a string para o terminal
```

Testando esse simples código, já podemos visualizar no terminal que o microcontrolador está enviando as mensagens perfeitamente em 1 segundo, mas vamos analisar mais a fundo e ver se ele se mantém estável mesmo com frequências mais altas, lembrando que a frequência máxima do software timer está limitada à frequência do RTOS. Vamos observar o desempenho do timer com o analisador lógico nas figuras 3 e 4. Em ambos testes, a ISR do timer efetuou um pulso de 50 us em um GPIO.

Veja na figura 3 (clique para ampliar se necessário), o software timer efetuando o pulso de 50 us no GPIO com frequência de 10 Hz (100 ms), observe no canto direito da imagem os detalhes da análise de ~100 pulsos.

Figura 3 - Analisador lógico para software timer de 10 Hz.

De acordo com o analisador lógico, nossa frequência média foi de 9,997 Hz (100,030009002701 ms), o que é aceitável para a maioria dos propósitos e deve-se lembrar do próprio erro do analisador lógico.

Agora vamos observar na figura 4, o software timer efetuando o pulso de 50 us no GPIO com frequência de 500 Hz (2 ms), observe no canto direito da imagem os detalhes da análise de ~100 pulsos.

Figura 4 - Analisador lógico para software timer de 500 Hz.

De acordo com o analisador lógico, nossa frequência média foi de 499,8 Hz (2,000800320128 ms), o que também é aceitável para a maioria dos projetos e não devemos esquecer novamente sobre o erro do analisador lógico.

O software timer do FreeRTOS se mostra bem estável e pode ser extremamente útil na maioria dos projetos que usam timers, já que podemos trocar os timers físicos (hardware) pelo software caso a frequência não seja tão alta. Devemos lembrar que a frequência máxima indicada para uso do FreeRTOS é de 1000 Hz, deixando o software timer atrelado a esta frequência máxima também.

Saiba mais

Biblioteca de Soft Timers

Biblioteca rápida de Timer, Delay e Timeout sem desperdícios

Sistemas Operacionais de Tempo Real - Timers

Referências

<https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html> 

https://freertos.org/Documentation/RTOS_book.html 

<http://esp-idf.readthedocs.io/en/latest/api-reference/system/freertos.html> 

Outros artigos da série

<< RTOS: Uso de grupo de eventos para sincronização de tarefas

 Facebook 13

 Twitter 0

 Google+ 0

 LinkedIn 0

Este post faz da série **RTOS**. Leia também os outros posts da série:

- [RTOS: Um ambiente multi-tarefas para Sistemas Embarcados](#)
- [RTOS: Scheduler e Tarefas](#)
- [RTOS: Semáforos para sincronização de tarefas](#)
- [RTOS: Uso de Queue para sincronização e comunicação de tarefas](#)
- [RTOS: Uso de grupo de eventos para sincronização de tarefas](#)
- [RTOS: Software Timer no FreeRTOS](#)


NEWSLETTER

Receba os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.

E-mail

CADASTRAR E-MAIL

Fique tranquilo, também não gostamos de spam.

RTOS: Software Timer no FreeRTOS por **José Morais**. Esta obra está licenciado com uma Licença [Creative Commons Atribuição-CompartilhaIgual 4.0 Internacional](#) .

José Morais

Estudante de Engenharia da Computação pela USC, pretende se aprimorar e fazer a diferença nesta imensa área da tecnologia. Apaixonado por IoT, sistemas embarcados, microcontroladores e integração da computação nos mais diversos fins práticos e didáticos.

Este site utiliza cookies. Ao usá-lo você concorda com nossos Termos de Uso.
[Saiba mais.](#)

Continuar