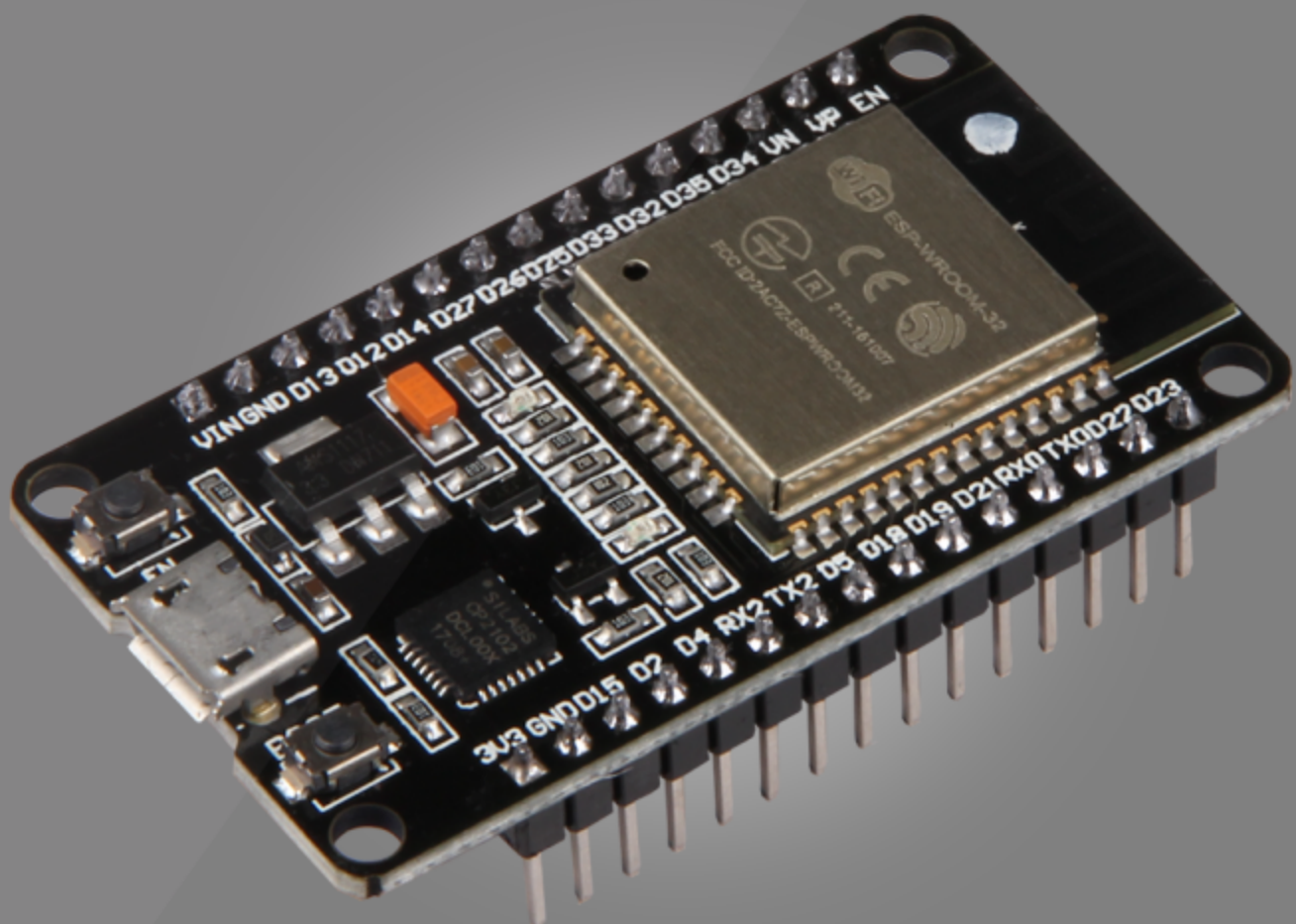


Coleção ESP32 do Embarcados

Explore o FreeRTOS com ESP32



EMBARCADOS

Olá,

Obrigado por baixar o nosso ebook: **Coleção ESP32 do Embarcados** - Parte 2. Esse ebook traz uma coleção de textos já publicados no Embarcados.

Fizemos um compilado de textos que consideramos importantes para os primeiros passos com a plataforma. Espero que você aproveite esse material e lhe ajude em sua jornada.

Continuamos com a missão de publicar textos novos diariamente no site.

Um grande abraço.

Equipe Embarcados.

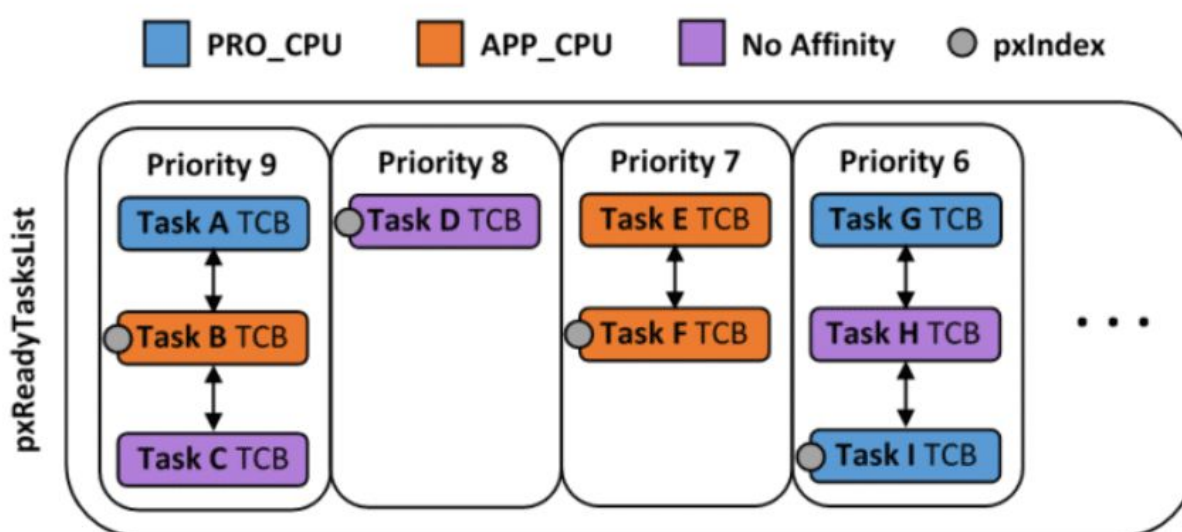
Sumário

ESP32 - Lidando com Multiprocessamento - Parte I	4
Multiprocessamento Simétrico ou Assimétrico?	5
O suporte ao multiprocessamento simétrico no ESP32	6
Delegando funções para a PRO_CPU com o IPC	8
Conclusão	10
Referências	11
ESP32 - Lidando com Multiprocessamento - Parte II	12
O ESP32 é multitarefa e dual-core, como ganhar com isso	13
Introduzindo o conceito de afinidade por núcleo	13
Exemplo de uso no Arduino	14
Conclusão - Multiprocessamento no ESP32	18
Referências	19
RTOS: Um ambiente multi-tarefas para Sistemas Embarcados	20
O que é um Sistema Operacional?	21
Principais itens de um Sistema Operacional	21
Quais são as diferenças entre GPOS e RTOS?	22
Pensando em aplicações práticas	22
Referências	24
RTOS: Scheduler e Tarefas	25
Scheduler	26
Formas de trabalho do scheduler	26
Tarefas	29
Prioridades	29
Estados	30
Referências	37
RTOS: Semáforos para sincronização de tarefas	38
O que são semáforos?	39
Tipos de semáforos	41
Referências	48
RTOS: Uso de Queue para sincronização e comunicação de tarefas	49
O que é uma Queue e como funciona?	49
Referências	55
RTOS: Uso de grupo de eventos para sincronização de tarefas	56
O que é e como funciona um grupo de eventos?	57
Referências	62
RTOS: Software Timer no FreeRTOS	63
Quais as vantagens e desvantagens do software timer no FreeRTOS?	64
Vantagens	64
Desvantagens	64
Timer Service ou Daemon task	65

Referências	69
TinyPICO - Pequena placa com o ESP32	70
Detalhes da TinyPICO	70
Play Shield	73
Audio Shield	73
RTC Shield	74
Grove I ² C Shield	74
Proto Shield	75
3 Up Shield	75
WiPhone - Telefone VoIP baseado em ESP32	77
Características do WiPhone	78
MAKERphone - um celular DIY baseado no ESP32	82
Considerações Finais	86

ESP32 - Lidando com Multiprocessamento - Parte I

Autor: [Felipe Neves](#)



Olá caro leitores. Já tem um tempo que tenho utilizado o conhecido ESP32 em uma gama de projetos em que ele não seja comumente empregado como: Controle de movimento e unidade de sensores, e a cada interação com esse simpático chip tenho encontrado algumas funcionalidades dentro do IDF, o framework principal de desenvolvimento da Espressif, que me chamaram muito a atenção, além de claro ter facilitado muito a minha vida.

Nesse meu primeiro texto de 2020, gostaria de apresentar para vocês um recurso que talvez poucos tenham ouvido falar e nem usem pois:

- O IDF é uma base de código bem extensa e um pouco complexa;
- O uso do ESP32 pelo Arduino é a forma mais popular de desenvolver ainda que o port do Arduino não ofereça tudo que está disponível no IDF.

Esse recurso é o multiprocessamento, para quem não se recorda, o ESP32 conta com dois núcleos físicos XTENSA LX6 (o popular dual-core), cada um deles rodando a modestos 240MHz. Isso pode não chamar a atenção inicialmente, porém o IDF oferece suporte a

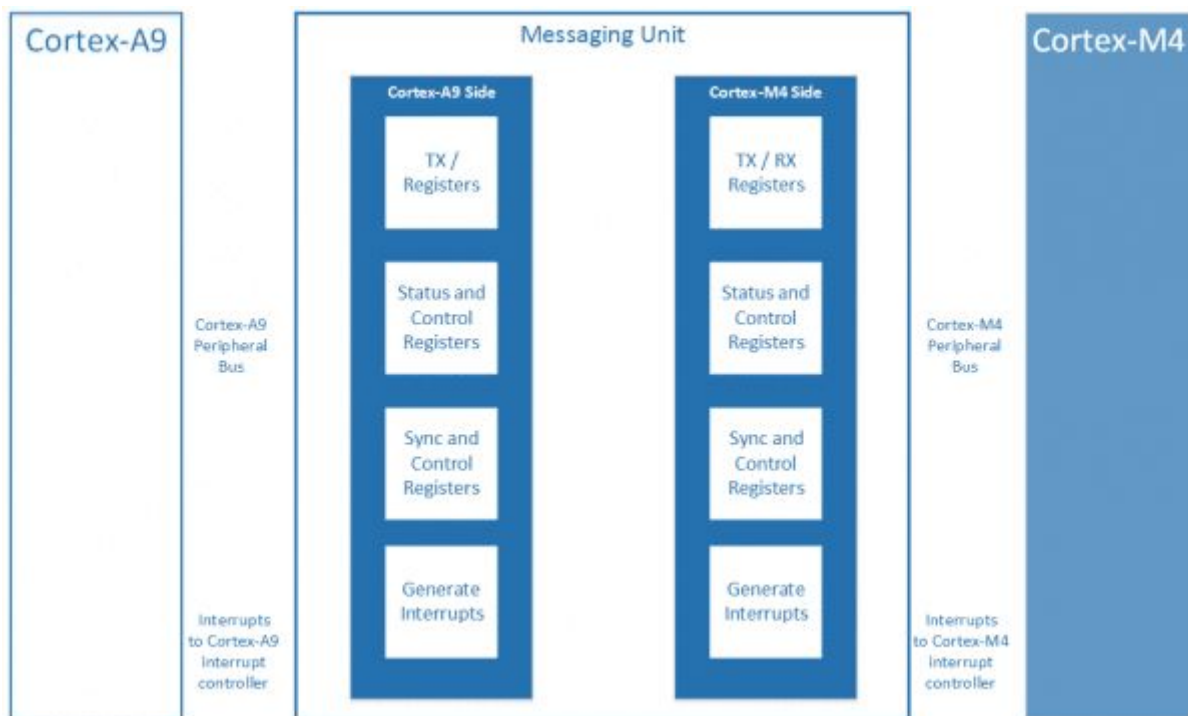
multiprocessamento de forma transparente ao usuário, estando ele desenvolvendo dentro do Arduino ou fora dele.

Para deixar as coisas mais simples iremos rodar os exemplos no Arduino IDE, mas antes vamos revisar um pouco sobre multiprocessamento, prometo que é rapidinho.

Multiprocessamento Simétrico ou Assimétrico?

O conceito de multiprocessamento, como o nome sugere, situa na capacidade de rodar um determinado programa em uma CPU que possui mais de 1 núcleo físico, ou seja imagine que, no caso do ESP32 que exista não um mas dois processadores, onde podemos balancear quais partes da firmware devem rodar entre os núcleos. De forma similar o sistema operacional do seu smartphone, composto de vários processos contidos em um aplicativo, pode delegar em qual núcleo físico um determinado processo vai rodar. Dessa mesma forma ESP32 pode criar tasks, que são parecidas com um processo, de forma que o agendamento das tasks não compartilhar apenas um núcleo entre elas, agora o sistema operacional do ESP32 pode fazer isso dividindo as tasks entre dois (ou mais) núcleos!

O multiprocessamento se divide ainda em dois grandes grupos, sendo o primeiro deles o assimétrico. Imagine que tenhamos um ESP32 com dois núcleos, porém cada um deles acessando apenas uma determinada área de memória fisicamente separadas, ou seja esses núcleos fictícios consomem instruções de localidades diferentes, e podem trocar informações através de uma área de memória compartilhada, nesse caso teríamos duas instancias de firmware cada uma compilada de uma forma diferente. A grande vantagem desse tipo de arquitetura é que nesse caso os ESP32 poderiam ter núcleos diferentes sem qualquer relação, já que um núcleo jamais vai executar uma instrução que pertence a área de código do outro. A figura abaixo mostra bem esse conceito, embora não seja um ESP32:



Na figura 1 temos um caso clássico, dois núcleos ARM, sendo um deles um Cortex-A e outro um Cortex-M, observe que eles são bem diferentes entre si, a começar por não serem binariamente compatíveis, embora isso evidencie o fato de se tratar de uma arquitetura assimétrica, arquiteturas com núcleos binariamente compatíveis também podem ser consideradas assimétricas desde que cada núcleo consuma isoladamente sua própria instância de firmware.

O ESP32 opera na segunda categoria, a arquitetura simétrica onde, como o nome sugere, temos dois núcleos idênticos (primeiro requisito para ser simétrica) e além disso os dois núcleos compartilham tudo, desde memórias, periféricos e consomem a mesma instância de firmware, basicamente isso significa que embora cada núcleo possa estar executando um pedaço diferente da área de código, esses pedaços pertencem ao mesmo binário da firmware, tendo, dessa forma dois processos físicos sendo compartilhados entre a firmware e não apenas um como estamos habituados nos microcontroladores mais comuns.

O suporte ao multiprocessamento simétrico no ESP32

Abaixo temos a arquitetura simplificada do ESP32:

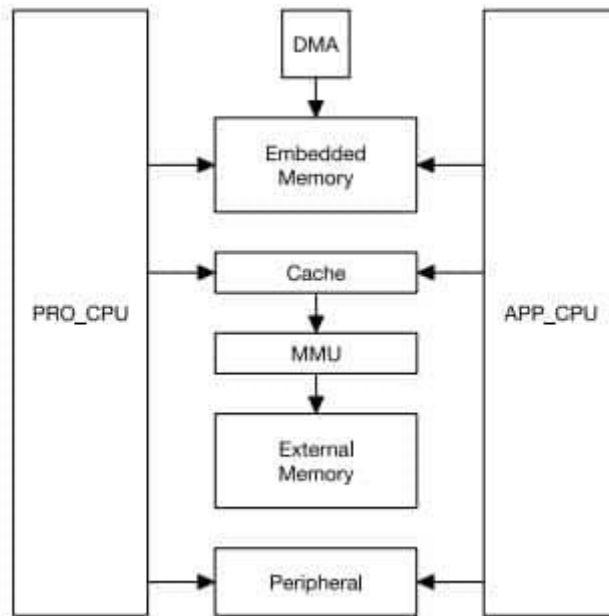


Figura 2: Arquitetura do ESP32

Percebam que os dois núcleos possuem codinomes, são eles o PRO_CPU e APP_CPU, usarei essa notação daqui em diante para facilitar a identificação, mas em princípio:

- PRO_CPU é o núcleo padrão, quando ESP32 é inicializado, apenas esse núcleo consome instruções da memória de programa;
- APP_CPU é o núcleo secundário, inicia desabilitado, mas uma vez ativo ele começa a consumir instruções a partir do valor inicial colocado no seu contador de programa, o conhecido PC.

No ESP32, ao utilizar o IDF ou o Arduino (pra quem não sabe a bibliotecas do Arduino para o ESP32 é construída em cima do IDF), ambos os núcleos são inicializados e colocados para rodar muito antes do programa chegar na parte da aplicação, o fato é que o sistema operacional interno do ESP32 no momento em que o programa alcança a função main(), setup() ou loop(), PRO_CPU e APP_CPU estão a disposição e prontas para rodar.

Por padrão a função loop() roda na APP_CPU, e para quem já está familiarizado com o FreeRTOS do ESP32 talvez já tenha criado alguma task, que por padrão tem afinidade inicial com a PRO_CPU (nos bastidores o FreeRTOS modificado do ESP32 pode executar um balanceamento de carga jogando algumas tarefas para a PRO_CPU ou APP_CPU sem controle do usuário).

Mas então imagina agora que você teve a ideia: "E se eu mover funções da minha aplicação para executarem somente na PRO_CPU terei mais processamento livre?". Sim você acertou e nessa primeira parte vamos explicar como delegar uma função para ser executada na PRO_CPU enquanto a APP_CPU cuida da função loop(), utilizando a IPC (Inter-Processor Call) API.

Delegando funções para a PRO_CPU com o IPC

Vamos direto pro código, como comentado antes, podemos usar componentes do IDF diretamente do Arduino, apenas incluindo os arquivos necessários, se esqueça de instalar o suporte para o Arduino do ESP32 adicionando o link 2 ao final do artigo no seu gerenciador de boards para busca e instalação de suporte.

E agora sim, vamos para o nosso primeiro exemplo:

```
#include <Arduino.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <esp_ipc.h>

void setup(){
    Serial.begin(115200);
}

void LoopOnProCpu(void *arg) {
    (void)arg;
    Serial.print("This loop runs on PRO_CPU which id is:");
    Serial.println(xPortGetCoreID());
    Serial.println();
    Serial.println();
}
```

```

void loop(){
    //Default loop runs on APP_CPU
    Serial.print("This loop runs on APP_CPU which id is:");
    Serial.println(xPortGetCoreID());
    Serial.println();
    Serial.println();

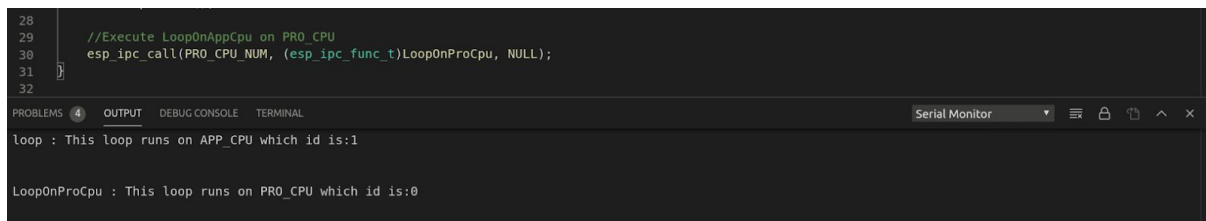
    //Execute LoopOnAppCpu on PRO_CPU
    esp_ipc_call(PRO_CPU_NUM, LoopOnProCpu, NULL);
}

```

Esse exemplo super simples mostra como executar uma função em qualquer que seja o núcleo desejado, o usuário que tiver algum ESP32 na mesa pode criar um sketch Arduino com esse código e já gravar nele. Vamos explicar o que ocorre, primeiro de tudo é importante reforçar que todo o IDF está acessível mesmo pelo Arduino, vejam que apenas inclui os arquivos do FreeRTOS e a API de IPC do ESP32 diretamente no sketch, isso vale para qualquer componente do IDF core. Com os arquivos devidamente incluídos temos as habituais funções `setup()` e `loop()` que como dissemos antes, elas rodam na APP_CPU, em `setup()`, nada de muito novo além de inicializar o monitor serial, é dentro de `loop` que vemos algo bem interessante.

A função `xPortGetCoreID()` retorna o número do núcleo onde aquela função está executando, então em `loop`, basicamente mostramos no console que estamos rodando `loop` da da APP_CPU ou seja no ID número 1, agora reparem que após essa mensagem temos uma função nova sendo chamada.

A função **`esp_ipc_call()`**, chamada ao final de `loop`, recebe três parâmetros, o primeiro é o ID do núcleo, podendo ser o PRO_CPU ou APP_CPU, o segundo parâmetro é a função que desejamos executar, o terceiro é um argumento que pode ter um formato definido pelo usuário (sendo do tipo **`void*`**) caso desejemos passar uma informação para essa função. É essa chamada que provoca que a execução da função **`LoopOnProCpu()`** que faz exatamente o que `loop` faz, ou seja imprime o ID do núcleo na CPU, carregue esse sketch no seu ESP32 e abra o monitor serial, você deve ver algo do tipo na tela:



```
28
29 //Execute LoopOnAppCpu on PRO_CPU
30 esp_ipc_call(PRO_CPU_NUM, (esp_ipc_func_t)LoopOnProCpu, NULL);
31
32
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL Serial Monitor

loop : This loop runs on APP_CPU which id is:1

LoopOnProCpu : This loop runs on PRO_CPU which id is:0

Figura 3: Monitor serial executando o sketch

Um ponto interessante a se destacar é que a execução da função em outro core é assíncrona ou seja, uma vez que o IPC seja chamado a função passada executa imediatamente no outro core, podendo terminar sua execução antes ou depois da função que a chamou, podendo ser entendido como uma aplicação rodando em paralelo. Adicionalmente a API IPC oferece a função **esp_ipc_call_blocking()** cuja a funcionalidade é idêntica, porém que chama essa função aguarda que a função do IPC finalize antes de prosseguir com sua execução, sendo interessante quando o usuário deseja sincronizar processos em dois cores diferentes. **Tenha em mente que as funções delegadas dessa forma devem ser do tipo Run-To-Completion , ou seja elas devem ter um ponto de retorno, diferentemente de uma task ou da função main() elas não devem conter loops infinitos como while(1).**

Conclusão

Esse artigo visou demonstrar que o ESP32 pode oferecer muito mais do que a aparência mostra, uma dessas ofertas está no multicore simétrico, perfeito para dividir o processamento entre os dois núcleos permitindo o desenvolvimento de aplicações mais complexas ou a delegação de responsabilidade, os nomes APP_CPU e PRO_CPU não tem esse nome a toa já que derivam de Application CPU e Protocol CPU respectivamente, onde um núcleo dedica-se a aplicação enquanto outro processa e encaminha o processamento de comunicações sem que um núcleo penalize o outro por sua natureza de processamento. Fique ligado pois na parte II iremos apresentar uma forma de contornar a limitação das funções serem Run-To-Completion permitindo que você usuário execute o que quiser do IDF no núcleo que desejar. Muito obrigado pela sua leitura e até a próxima.

Referências

1. [ESP Interprocessor Call API Reference](#)
2. [Suporte a placas com ESP32 para Arduino](#)



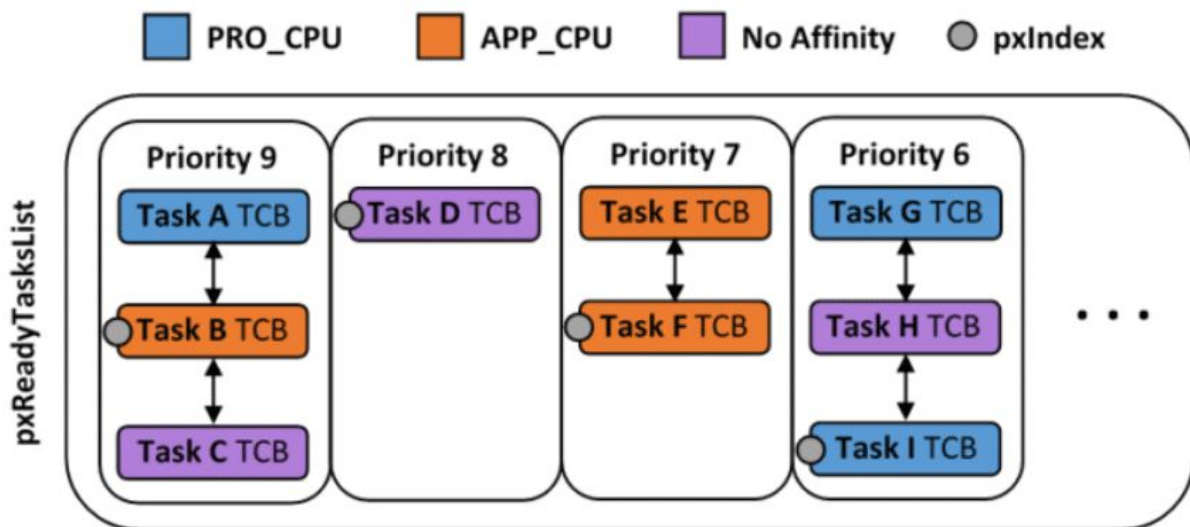
Publicado originalmente no Embarcados, no dia 13/01/2020: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhalgual 4.0 Internacional](#).



Assista os webinars gravados

ESP32 - Lidando com Multiprocessamento - Parte II

Autor: [Felipe Neves](#)



Olá caro leitor, seguindo com série sobre multiprocessamento com o [ESP32](#), hoje vou trazer a você mais uma abordagem de como executar suas aplicações em um dos núcleos desejados. No artigo anterior foi apresentada a política de multiprocessamento do ESP32, a qual definimos como simétrica, ou seja os dois núcleos do módulo consome o mesmo stream de instruções na memória de programa, sendo esse gerenciamento feito pelo RTOS interno do ESP32, totalmente transparente ao usuário. Apresentamos naquele texto então como delegar funções isoladas para determinado núcleo.

O ESP32 é multitarefa e dual-core, como ganhar com isso

Aqui no Embarcados temos diversos textos sobre uso e detalhes de multitarefa em sistemas embarcados com o uso de RTOS, assim sendo não pretendo entrar em detalhes deste tópico aqui (mas encorajo o leitor a fazer suas perguntas na seção de comentários abaixo), mas é bom lembrar que um dos grandes benefícios do uso de um RTOS em sistemas embarcados está na isolação entre os pedaços de uma aplicação, por exemplo com o uso de tasks (threads, ou tarefas) um grupo de desenvolvedores podem trabalhar separadamente de forma que uma parcela cuidaria de comunicações, outro grupo em sensores e outro lidando com atuadores sem que, diretamente, os domínios de suas tarefas impactem entre si.

Ao falar sobre RTOS quase sempre lembramos da expressão: "...de ter múltiplas mains na aplicação", e o ESP32 deixa isso significativamente melhor, já que agora o RTOS não entrega fatias de um núcleo por task, mas dois deles de forma que as políticas de execução ganham ainda mais flexibilidade.

Vamos pegar um exemplo de caso de drone. Sabemos que dois dos seus grandes domínios são as comunicações com a base e o processamento de algoritmos de controle. Utilizando um RTOS comum teríamos que escolher corretamente as prioridades das tasks para que as comunicações tenham uma resposta agradável aos comandos da base e ao mesmo tempo precisamos garantir que a execução periódica dos algoritmos de estabilização e controle do drone não sofram atrasos.

Obter um núcleo extra nos permite separar melhor as coisas, podendo controlar além da prioridade, também qual núcleo deve lidar mais intensamente com o que.

Introduzindo o conceito de afinidade por núcleo

Antes de entrarmos no ponto prático, precisamos entender primeiro um pequeno conceito que envolvem as tasks quando estamos lidando com um processador com 2 ou mais núcleos, como o ESP32.

A afinidade de tarefa por núcleo, em que basicamente podemos determinar que uma task em particular será executada apenas num determinado núcleo mesmo se houver outra CPU disponível. De forma similar uma task pode não ter nenhuma afinidade, dessa forma o RTOS terá que lidar com essa task para entregá-la para um núcleo disponível. Para esse segundo caso não existe um procedimento padrão mas sim protocolos para balancear a carga de processamento entre os núcleos.

Para tasks sem afinidade no ESP32, o RTOS utiliza um protocolo de dar preferência para execução de tarefas com afinidade primeiro, caso o núcleo disponível não encontre nenhuma task pronta com sua afinidade então ele escolhe uma outra da lista que também não tenha afinidade, mantendo ao mesmo tempo, o protocolo de preempção padrão, ou seja dentre as várias tasks sem afinidade prontas para rodar, será selecionada aquela com maior prioridade, caso duas ou mais dessas tasks compartilhem dessa prioridade, o núcleo escolherá a task por ordem de chegada nessa lista.

Parece bastante coisa para o RTOS se preocupar não? Mas não se preocupe esse algoritmo acontece nos bastidores e o código de usuário raramente precisa se preocupar com qualquer coisa além de escolher se determinada task deve ter afinidade ou não durante a sua criação.

Exemplo de uso no Arduino

Agora que já sabemos o mínimo necessário do multiprocessamento no ESP32, vamos ilustrar o que discutimos, com o sketch para Arduino abaixo. Você tem total liberdade para copiar e colar na sua IDE e gravar no ESP32 que tiver a mão:

```
#include <Arduino.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>

void TaskRunningOnAppCore(void *arg) {
    while(1) {

        Serial.print(__func__);
```

```

    Serial.print(" : ");
    Serial.print(xTaskGetTickCount());
    Serial.print(" : ");
    Serial.print("This loop runs on APP_CPU which id is:");
    Serial.println(xPortGetCoreID());
    Serial.println();

    vTaskDelay(100);
}
}

```

```

void TaskRunningOnProtocolCore(void *arg) {
    while(1) {

        Serial.print(__func__);
        Serial.print(" : ");
        Serial.print(xTaskGetTickCount());
        Serial.print(" : ");
        Serial.print("This loop runs on PRO_CPU which id is:");
        Serial.println(xPortGetCoreID());
        Serial.println();

        vTaskDelay(100);
    }
}

```

```

void setup(){
    Serial.begin(115200);

    xTaskCreatePinnedToCore(TaskRunningOnAppCore,
                            "TaskOnApp",
                            2048,
                            NULL,
                            4,
                            NULL,
                            APP_CPU_NUM);
}

```



```

    xTaskCreatePinnedToCore(TaskRunningOnProtocolCore,
                            "TaskOnPro",
                            2048,
                            NULL,
                            8,
                            NULL,
                            PRO_CPU_NUM);
}

void loop(){
    Serial.print(__func__);
    Serial.print(" : ");
    Serial.print(xTaskGetTickCount());
    Serial.print(" : ");
    Serial.print("Arduino loop is running on core:");
    Serial.println(xPortGetCoreID());
    Serial.println();

    delay(500);
}

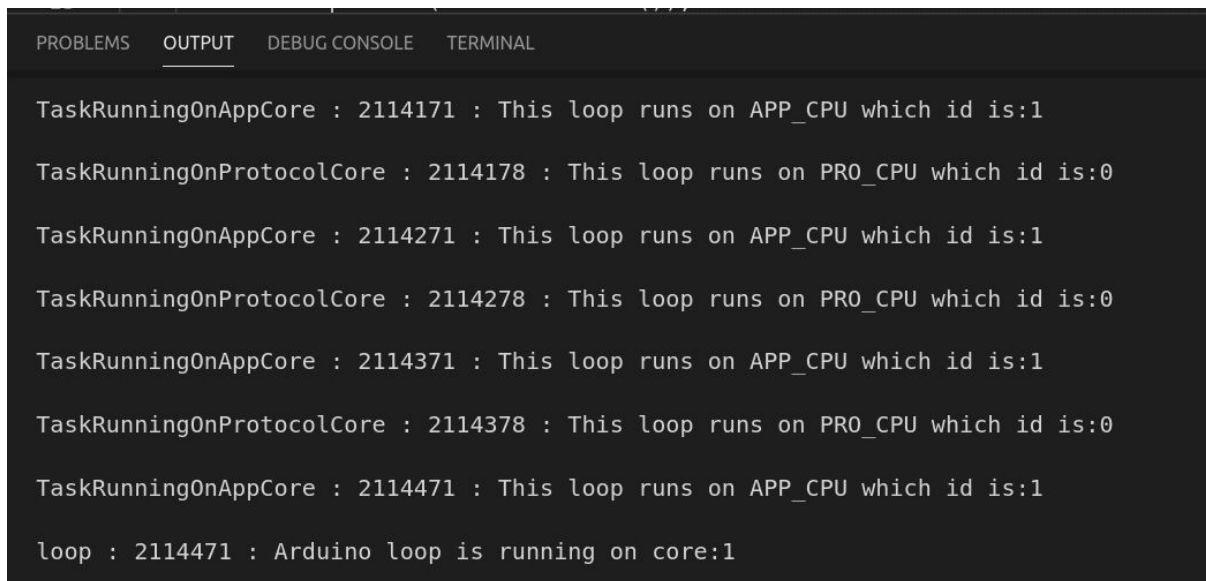
```

Vejam que o sketch é bem simples, como mencionamos no artigo anterior, `loop()` por si só já é uma task que roda no núcleo `APP_CPU`, que fica continuamente mostrando no console em qual núcleo ele está executando. Na função `setup()`, temos uma função da API do RTOS do ESP32, que é uma versão modificada do FreeRTOS, essa nova função para criação de tasks é igualzinha a do FreeRTOS conhecido por muitos, exceto pelo parâmetro extra.

`xTaskCreatePinnedToCore()` é utilizada para criar tasks (as "múltiplas mains" da sua aplicação) e adicionalmente assinalar ou não uma afinidade por núcleo, que é o último argumento que a função leva, percebam que foram criadas duas tasks, cada uma com afinidade em um dos núcleos, para quem não lembra vamos refrescar a memória sobre o significado dos demais parâmetros:

- O primeiro parâmetro é a função principal da sua task, uma função tipicamente em um loop infinito, embora ela possa retornar, esses casos não são tão comuns, perceba que o argumento nesse caso são basicamente o nome das funções que estão antes de `setup()` e `loop()`;
- O segundo parâmetro é um nome para essa task, sendo muito útil para fazer debug, além de ser usado pelo componente de trace do IDF (que podemos falar num próximo artigo);
- O terceiro parâmetro é destinado ao tamanho de pilha para essa task, toda aplicação seja ela uma task ou bare-metal (sem sistema operacional) possui um espaço de pilha usado para alocação de variáveis locais, portanto quanto mais variáveis locais você alocar dentro task, ou quanto mais níveis de função são chamados, maior esse valor, a unidade é em **bytes** para o caso do ESP32;
- O quarto parâmetro é o argumento para ser passado a task, toda task aceita um argumento do tipo `(void *)` sendo ele muito útil quando queremos usar uma mesma task para lidar com diferentes contextos;
- O parâmetro a seguir é a prioridade dessa task, tenha em mente sobre o funcionamento do algoritmo de preempção do FreeRTOS, quanto mais alto esse valor, maior a prioridade da task, o RTOS do ESP32 oferece até 25 níveis de prioridade;
- O sexto parâmetro é um local para guardar o Identificador único para essa task, raramente usado, pois o RTOS possui API para obter o ID de uma task depois que ela é criada;
- E o último parâmetro é a afinidade dessa task, podendo ela ter afinidade com a **PRO_CPU**, com a **APP_CPU**, ou passe **tskNO_AFFINITY** caso queira que o RTOS decida em quais núcleos essa task vai rodar, lembrando que outros valores são considerados inválidos e a API vai retornar erro.

As funções de task basicamente irão imprimir o uptime, ou seja, quantos ticks do RTOS já se passaram desde que ele começou a rodar juntamente com o ID do núcleo em que essa task está executando, sendo uma forma bem clara de ver o que está acontecendo, você, pode fazer o upload desse sketch no seu ESP32. Ligue o monitor serial em seguida e você deve ter algo assim na tela:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

TaskRunningOnAppCore : 2114171 : This loop runs on APP_CPU which id is:1
TaskRunningOnProtocolCore : 2114178 : This loop runs on PRO_CPU which id is:0
TaskRunningOnAppCore : 2114271 : This loop runs on APP_CPU which id is:1
TaskRunningOnProtocolCore : 2114278 : This loop runs on PRO_CPU which id is:0
TaskRunningOnAppCore : 2114371 : This loop runs on APP_CPU which id is:1
TaskRunningOnProtocolCore : 2114378 : This loop runs on PRO_CPU which id is:0
TaskRunningOnAppCore : 2114471 : This loop runs on APP_CPU which id is:1
loop : 2114471 : Arduino loop is running on core:1
```

Figura 1 : Sketch mostrando mensagens das tasks

Observe um detalhe, como as tasks criadas rodam a cada 100ms, e a loop() a cada 500ms iremos ver a loop executando com menos frequência porém a relação sempre se mantém, para cada 5 execuções das tasks, temos 1 da loop atestando o comportamento real-time do RTOS. O mais legal é que mesmo se travássemos o loop principal da task que roda na PRO_CPU, as demais funções continuariam executando normalmente já que estão fisicamente separadas em outro núcleo que apenas compartilha o mesmo binário. A partir desse simples exemplo você já poderá trazer suas aplicações antigas no Arduino e repensar em como dividir as tarefas entre tasks menores e qual dos núcleos poderia tomar conta deles, ou mesmo criando tasks sem afinidade para que o RTOS faça o balanceamento entre os núcleos permitindo assim o usuário extrair todo o potencial dos 240MHz em dois núcleos do ESP32!

Conclusão - Multiprocessamento no ESP32

Nessa série de dois artigos apresentamos os fundamentos de como lidar com multiprocessamento no ESP32 e porque o leitor deve aproveitar esse componente capaz de entregar muito mais processamento permitindo que o usuário possa desenvolver ou prototipar aplicações ricas nas mais variadas características.

O ESP32 oferece dois modelos para utilização de multiprocessamento, sendo o IPC para funções comuns, o método de afinidade criando tasks especializadas para cada núcleo e o

método de tasks sem afinidades para que o RTOS efetue o balanceamento de carga de processamento.

Espero que você, caro leitor, agora se sinta encorajado a aproveitar mais do seu ESP32 e aproveite que os componentes avançados do IDF podem ser acessados diretamente do Arduino. Teremos mais artigos legais a respeito de recursos do IDF que irei escrever sobre nos próximos texto, portanto fiquem ligados!

Referências

1. [Guia de referência SMP do IDF para o ESP32;](#)
2. [Zephyr, Symmetric multiprocessing;](#)



Publicado originalmente no Embarcados, no dia 20/01/2020: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhalgual 4.0 Internacional](#).



Baixe os nossos ebooks

RTOS: Um ambiente multi-tarefas para Sistemas Embarcados

Autor: José Morais



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?

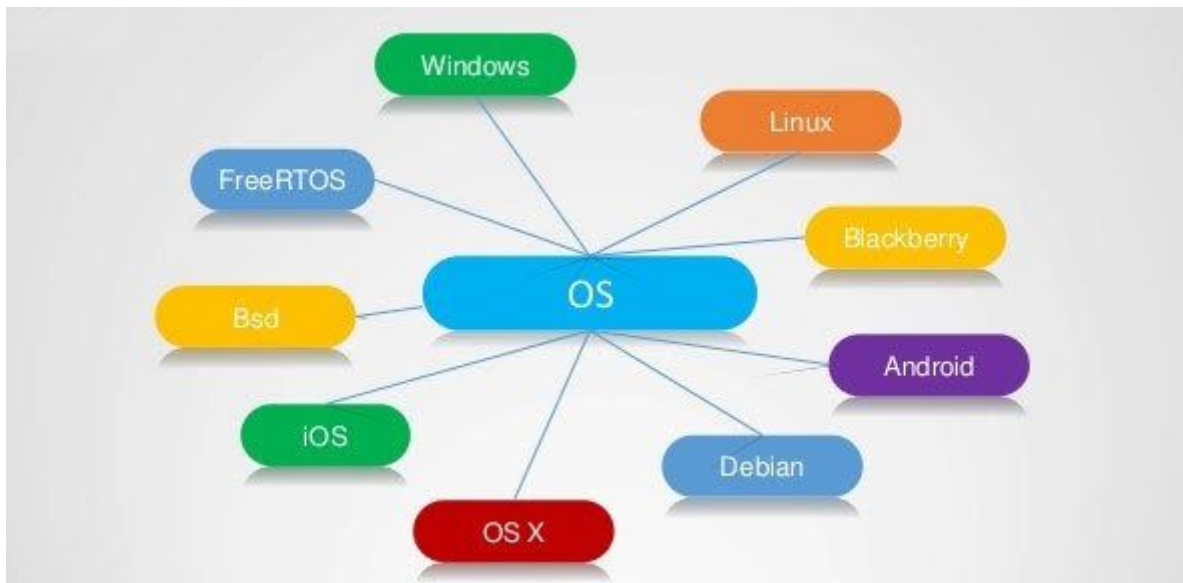


Figura 1 - Sistemas Operacionais.

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:

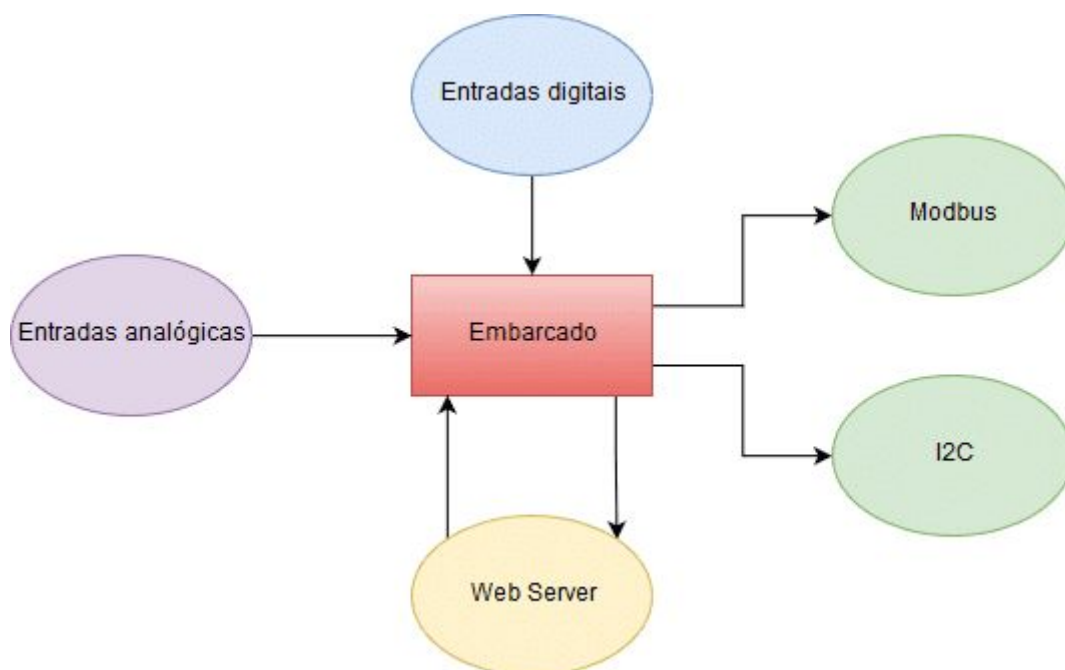


Figura 2 - Exemplo de ambiente multi-tarefa.

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.

Referências

1. https://freertos.org/Documentation/RTOS_book.html
2. https://en.wikipedia.org/wiki/Operating_system
3. https://en.wikipedia.org/wiki/Real-time_operating_system



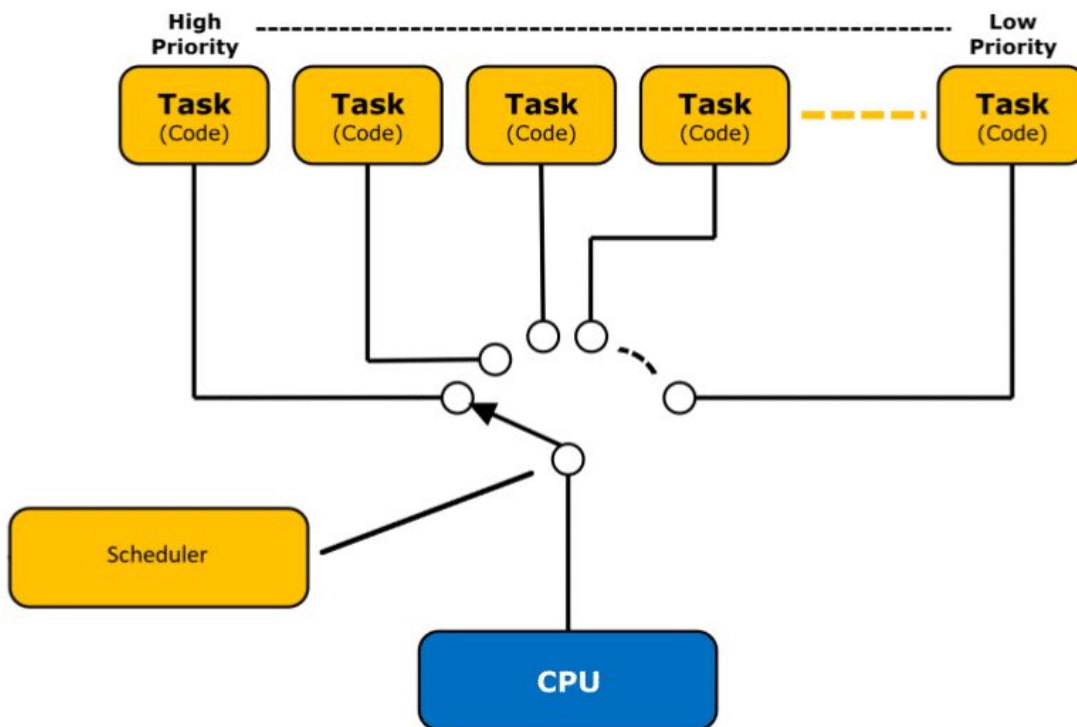
Publicado originalmente no Embarcados, no dia 09/07/2018: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhalgual 4.0 Internacional](#).



Acesse nossos artigos

RTOS: Scheduler e Tarefas

Autor: [José Morais](#)



Este artigo é talvez o mais importante é 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 as tarefas restantes no sistema são aconselhadas a terem prioridade maior ou igual a 1.

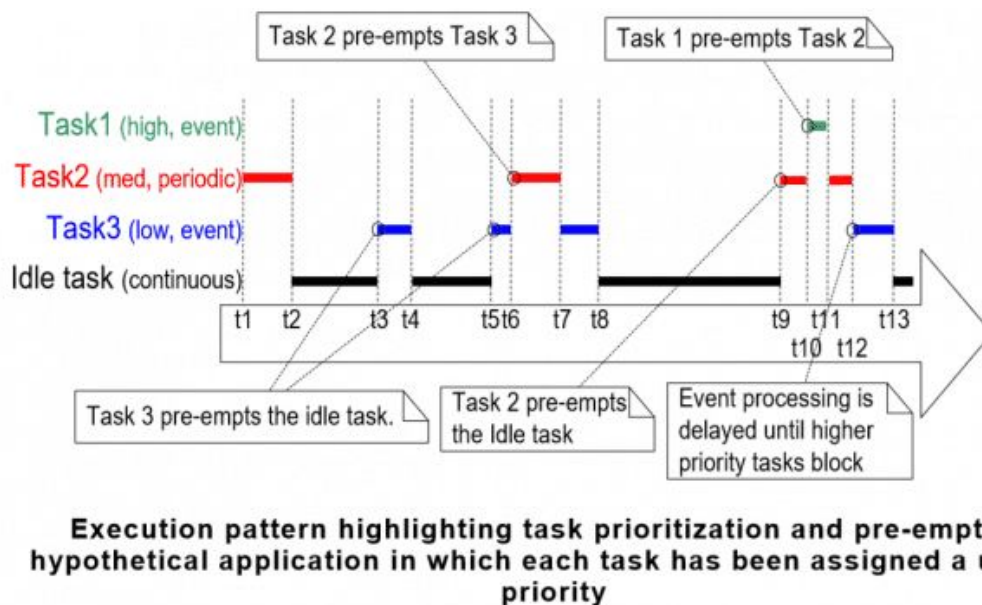
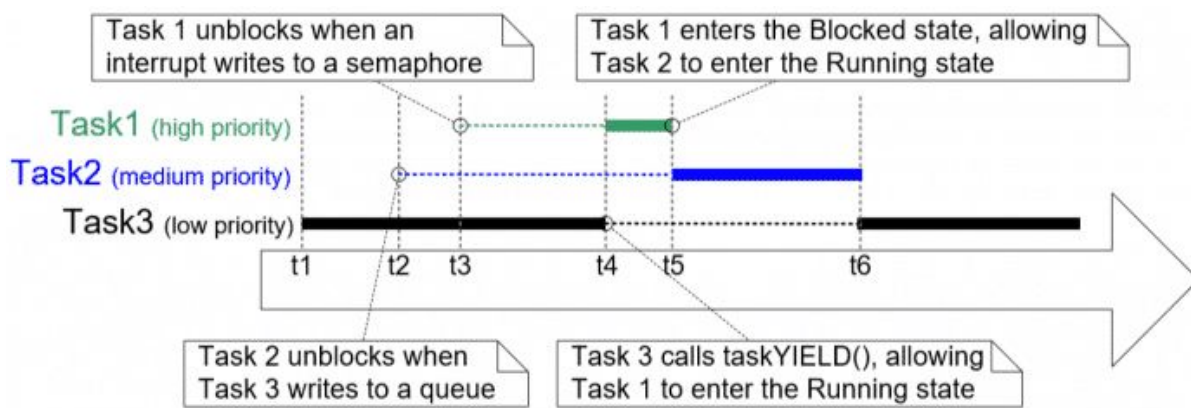


Figura 1 -

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.



Execution pattern demonstrating the behavior of the co-operative scheduler

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 $\leq 10 \mu s$.

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".

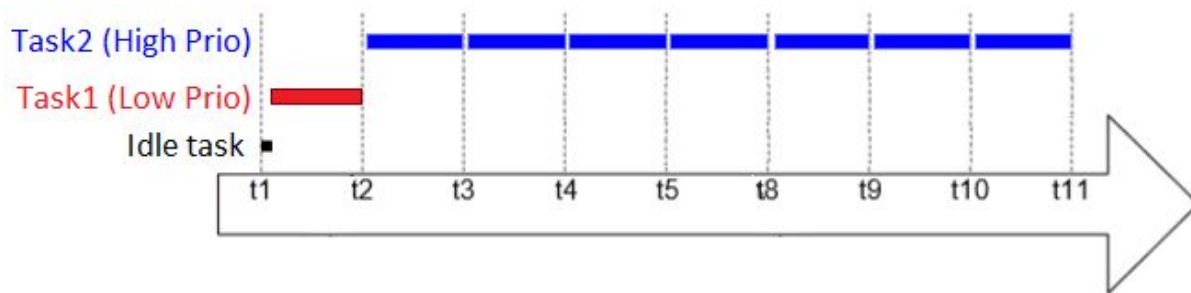


Figura 3 -

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.

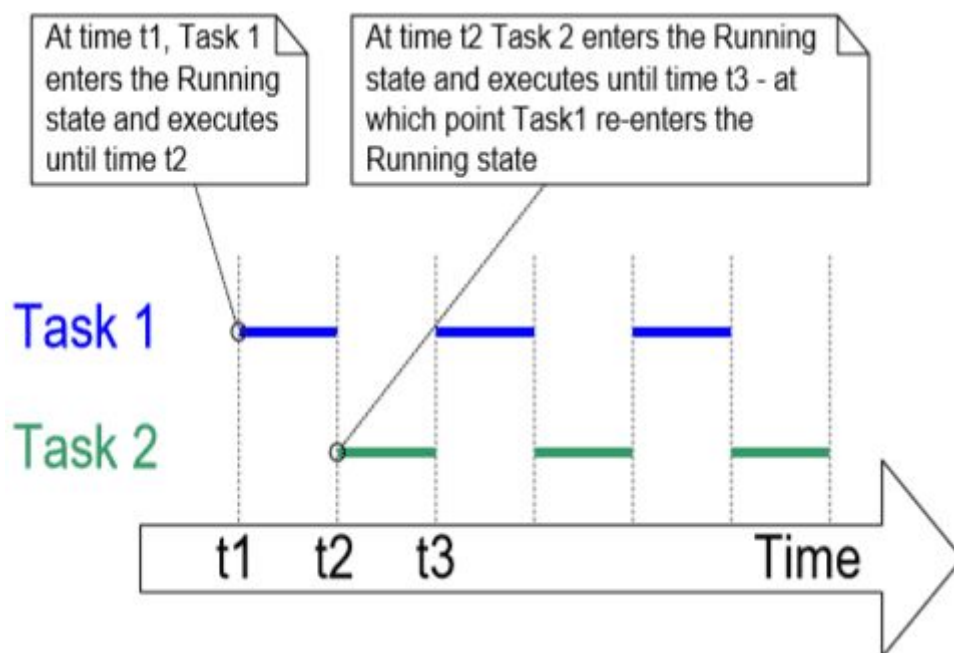


Figura 4 - Tarefas de prioridades iguais dividindo uso do CPU.

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.

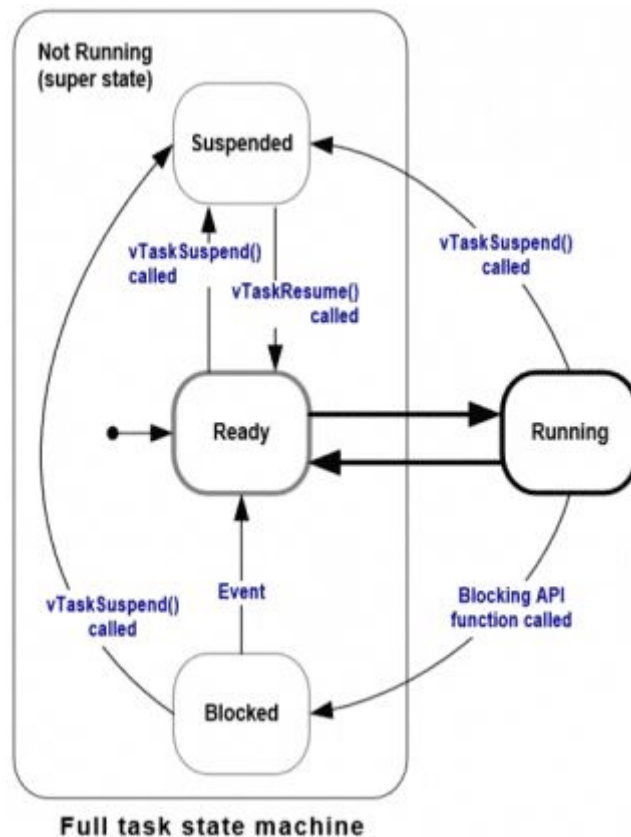


Figura 5 - Ciclo de vida de uma tarefa.

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:

```
#include <driver/gpio.h>
```

```

#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <esp_system.h>

void t1(void*z)
{
    //Tarefa que simula PWM para observar no analisador logico
    while (1)
    {
        gpio_set_level(GPIO_NUM_2, 1);
        ets_delay_us(25);
        gpio_set_level(GPIO_NUM_2, 0);
        ets_delay_us(25);
    }
}

void t2(void*z)
{
    //Tarefa que simula PWM para observar no analisador logico
    while (1)
    {
        gpio_set_level(GPIO_NUM_4, 1);
        ets_delay_us(25);
        gpio_set_level(GPIO_NUM_4, 0);
        ets_delay_us(25);
    }
}

void t3(void*z)
{
    //Tarefa que simula PWM para observar no analisador logico
    while (1)
    {
        for (uint8_t i = 0; i < 200; i++)
        {
            gpio_set_level(GPIO_NUM_15, 1);
            ets_delay_us(500);
            gpio_set_level(GPIO_NUM_15, 0);
            ets_delay_us(500);
        }
    }
}

```

```

    }

    vTaskDelay(pdMS_TO_TICKS(200));
}

}

void app_main()
{
    //Seleciona os pinos que serao usados
    gpio_pad_select_gpio(GPIO_NUM_2);
    gpio_pad_select_gpio(GPIO_NUM_4);
    gpio_pad_select_gpio(GPIO_NUM_15);

    //Configura os pinos para OUTPUT
    gpio_set_direction(GPIO_NUM_2, GPIO_MODE_OUTPUT);
    gpio_set_direction(GPIO_NUM_4, GPIO_MODE_OUTPUT);
    gpio_set_direction(GPIO_NUM_15, GPIO_MODE_OUTPUT);

    //Cria as tarefas
    xTaskCreatePinnedToCore(t1, "task1", 2048, NULL, 1, NULL,
0); //Tarefa 1 com prioridade UM (1) no core 0
    xTaskCreatePinnedToCore(t2, "task2", 2048, NULL, 1, NULL,
0); //Tarefa 2 com prioridade UM (1) no core 0
    xTaskCreatePinnedToCore(t3, "task3", 2048, NULL, 2, NULL,
0); //Tarefa 3 com prioridade DOIS (2) no core 0
}

```

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:

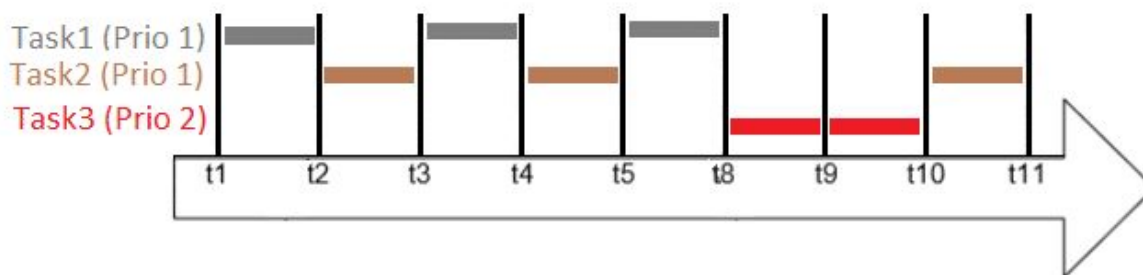


Figura 6 - Funcionamento teórico do código.

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.

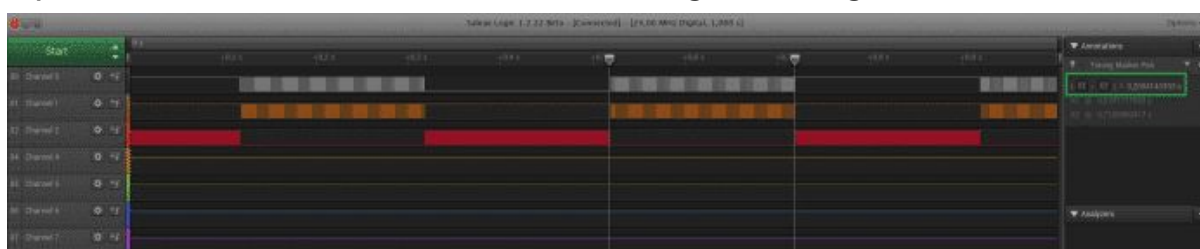


Figura 7 - Aplicando as tarefas na prática.

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.

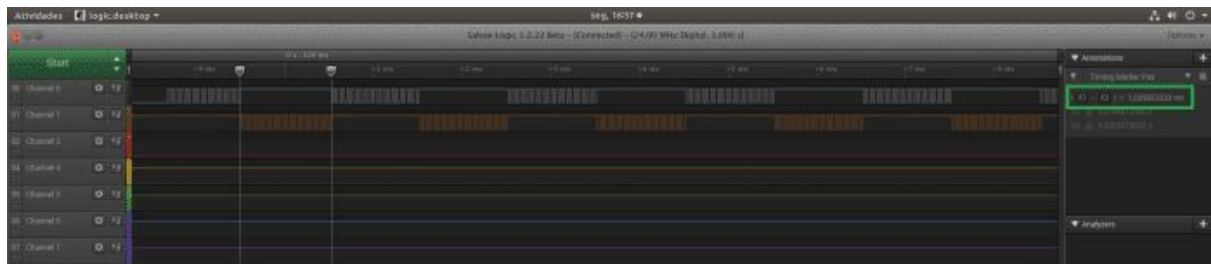


Figura 8 - Tarefas de prioridade igual concorrendo pelo CPU.

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).

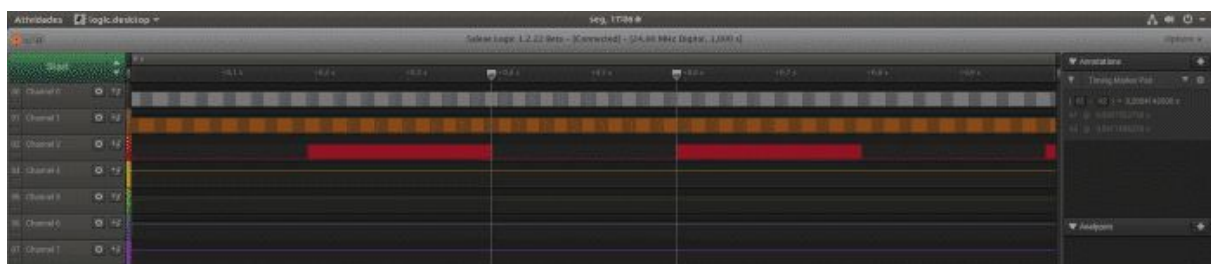


Figura 9 - FreeRTOS em embarcado Multi-Core.

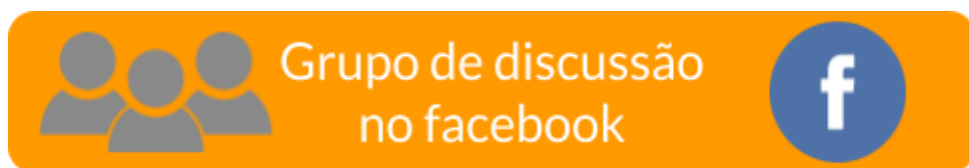
No próximo artigo desta série vamos abordar os semáforos, que funcionam para sincronizar eventos e tarefas dentro do RTOS.

Referências

1. https://freertos.org/Documentation/RTOS_book.html
2. <http://esp-idf.readthedocs.io/en/latest/api-reference/system/freertos.html>

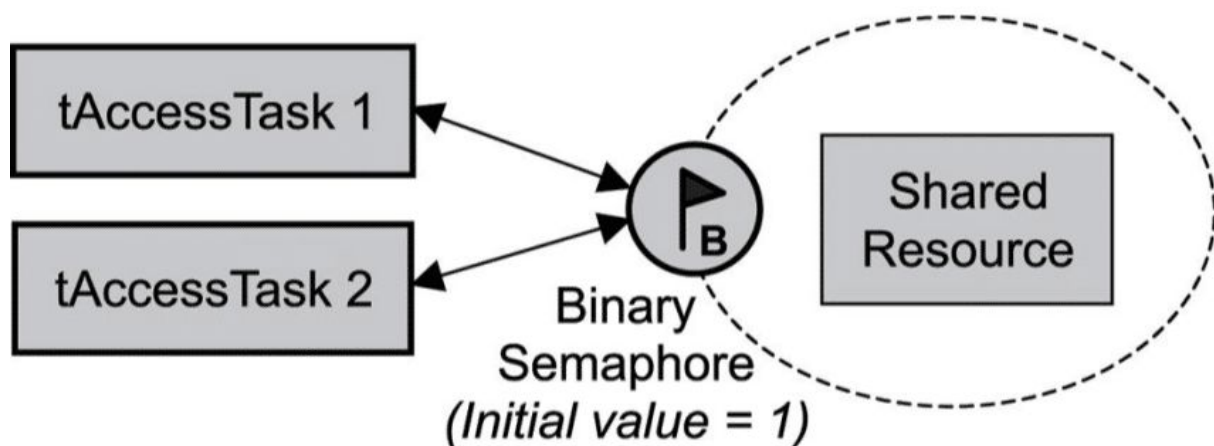


Publicado originalmente no Embarcados, no dia 20/10/2018: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhalgual 4.0 Internacional](#).



RTOS: Semáforos para sincronização de tarefas

Autor: [José Morais](#)



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.

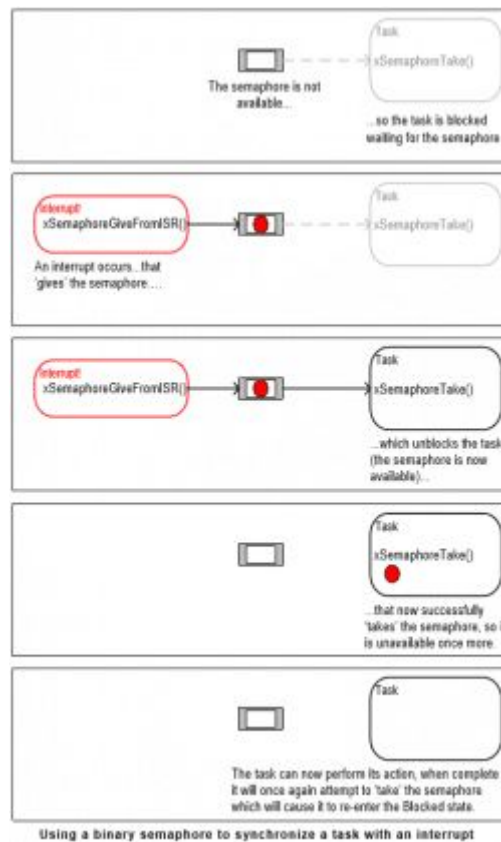


Figura 1 - Tarefa esperando e utilizando o semáforo.

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.

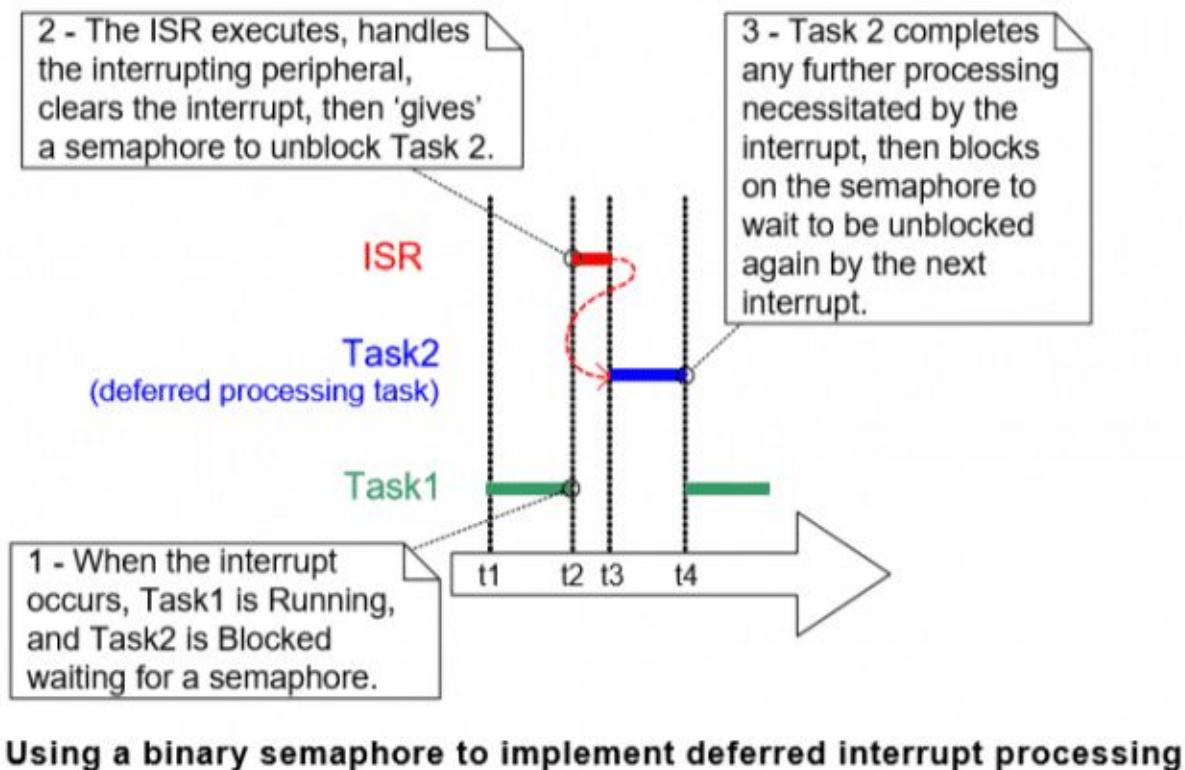


Figura 2 - Tarefa em espera pelo semáforo.

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.

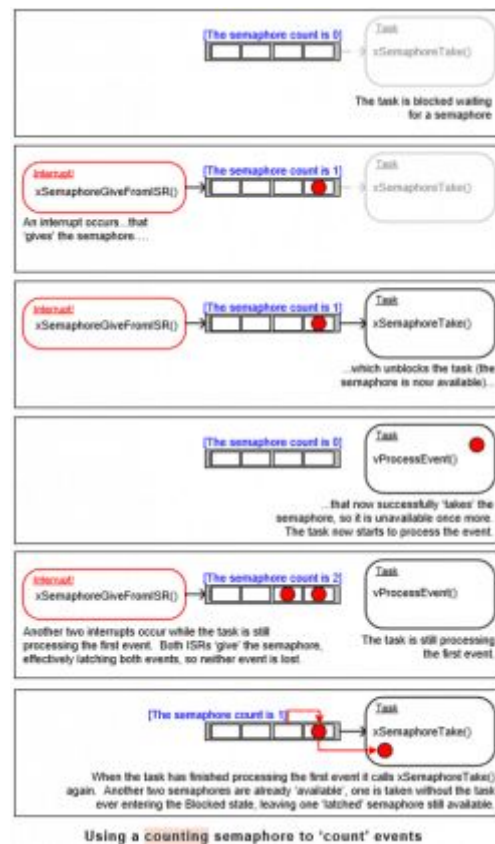


Figura 3 - Semáforo Counting.

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.

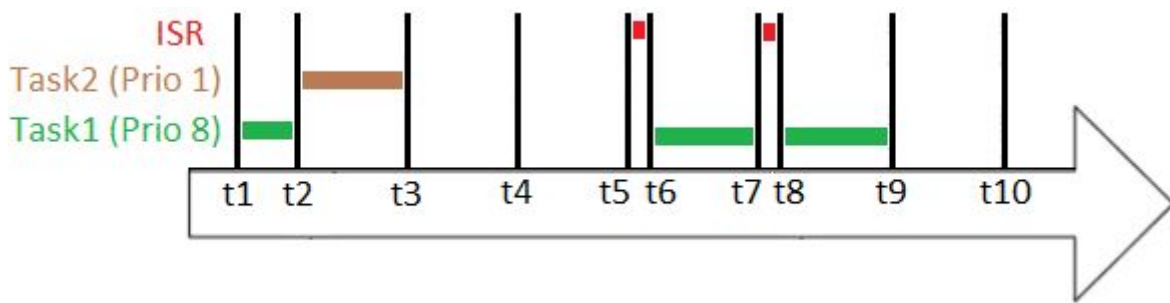


Figura 4 - Concorrência de tarefas pelo semáforo binário.

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).

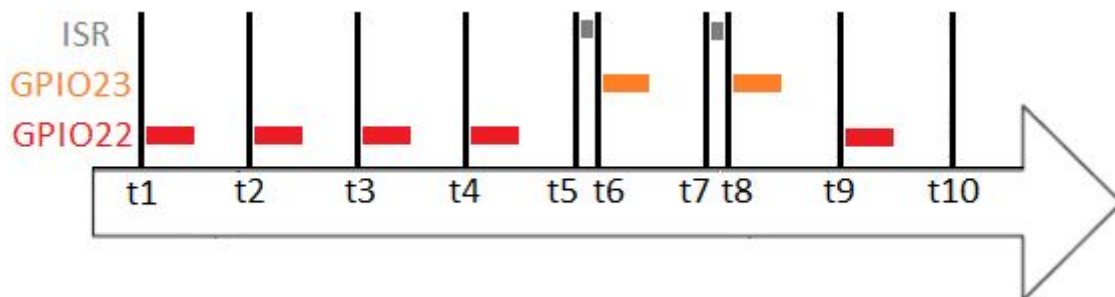


Figura 5 - Funcionamento teórico do código.

Código do projeto:

```
#include <driver/gpio.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <freertos/semphr.h>
#include <esp_system.h>
```

```
SemaphoreHandle_t SMF;//Objeto do semaforo
```

```
void isr(void*z)
{
    BaseType_t aux = false;//Variavel de controle para a Troca de Contexto
    xSemaphoreGiveFromISR(SMF, &aux);//Libera o semaforo

    if (aux)
    {
        portYIELD_FROM_ISR();//Se houver tarefas esperando pelo semaforo, deve ser forçado a Troca de Contexto (afim de minimizar latencia)
    }
}
```

```
void t1(void*z)
{
    while(1)
    {
        if (xSemaphoreTake(SMF, pdMS_TO_TICKS(200)) == true)//Tenta obter o semaforo durante 200ms (Timeout). Caso o semaforo nao fique disponivel em 200ms, retornara FALSE
        {
            //Se obteu o semaforo entre os 200ms de espera, fara o toggle do pino 23
        }
    }
}
```

```

        for (uint8_t i = 0; i < 10; i++)
        {
            gpio_set_level(GPIO_NUM_23, 1);
            ets_delay_us(150);
            gpio_set_level(GPIO_NUM_23, 0);
            ets_delay_us(150);
        }
    }
    else
    {
        //Se nao obter o semaforo entre os 200ms de espera, fara o
toggle do pino 22

        for (uint8_t i = 0; i < 10; i++)
        {
            gpio_set_level(GPIO_NUM_22, 1);
            ets_delay_us(150);
            gpio_set_level(GPIO_NUM_22, 0);
            ets_delay_us(150);
        }
    }

    vTaskDelay(pdMS_TO_TICKS(100));
}
}

void app_main()
{

    SMF = xSemaphoreCreateBinary();//Cria o semaforo binario e atribui
ao objeto que criamos


    //Configura o GPIO22 e GPIO32 como OUTPUT em LOW
    gpio_pad_select_gpio(GPIO_NUM_22);

```

```

gpio_set_direction(GPIO_NUM_22, GPIO_MODE_OUTPUT);
gpio_set_level(GPIO_NUM_22, 0);
gpio_pad_select_gpio(GPIO_NUM_23);
gpio_set_direction(GPIO_NUM_23, GPIO_MODE_OUTPUT);
gpio_set_level(GPIO_NUM_23, 0);

```

```

//Configura o GPIO4 como INPUT e PULL UP
gpio_pad_select_gpio(GPIO_NUM_4);
gpio_set_direction(GPIO_NUM_4, GPIO_MODE_INPUT);
gpio_pad_pullup(GPIO_NUM_4);

```

```

//Configura a interrupcao em rampas de decida para o GPIO4
gpio_install_isr_service(ESP_INTR_FLAG_LEVEL1);
gpio_intr_enable(GPIO_NUM_4);
gpio_set_intr_type(GPIO_NUM_4, GPIO_INTR_NEGEDGE);
gpio_isr_handler_add(GPIO_NUM_4, isr, NULL);

```

```

xTaskCreatePinnedToCore(t1, "t1", 4096, NULL, 1, NULL, 0); //Cria a
tarefa que analisa o semaforo
}

```

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

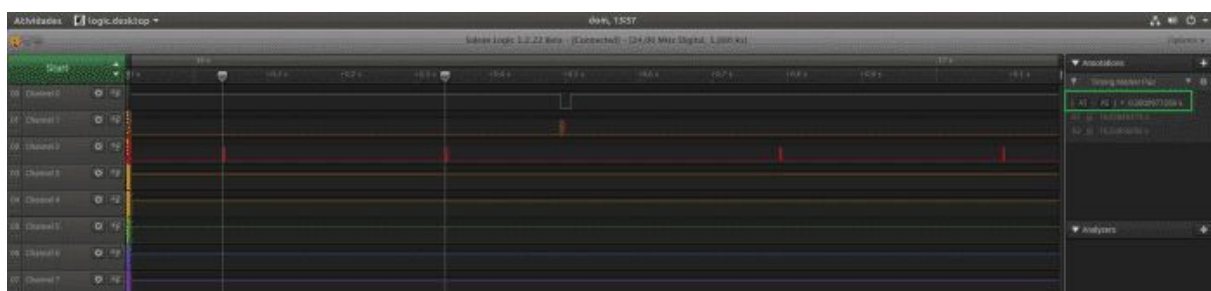


Figura 6 - Funcionamento prático do código.

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.

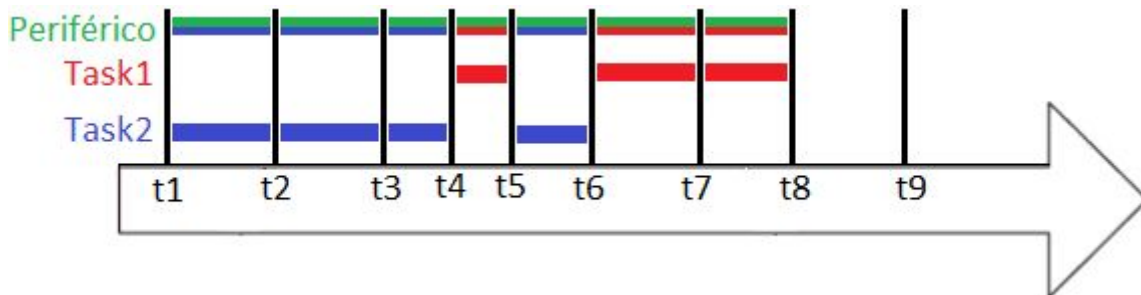


Figura 7 - Tarefas concorrendo pelo periférico.

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.

Referências

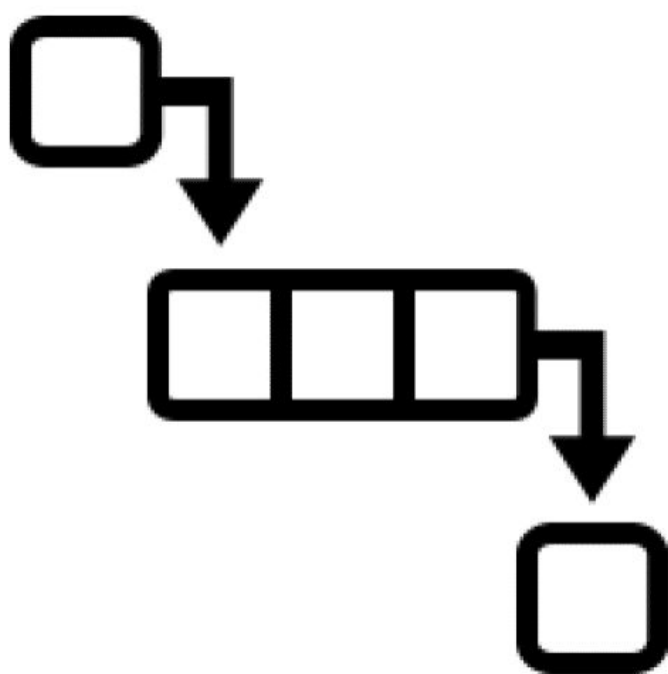
1. https://freertos.org/Documentation/RTOS_book.html
2. <http://esp-idf.readthedocs.io/en/latest/api-reference/system/freertos.html>



Publicado originalmente no Embarcados, no dia 20/10/2018: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhalgual 4.0 Internacional](#).

RTOS: Uso de Queue para sincronização e comunicação de tarefas

Autor: [José Moraes](#)



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.

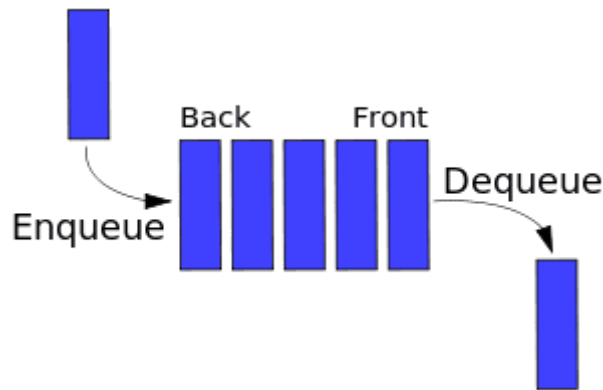


Figura 1 - FIFO.

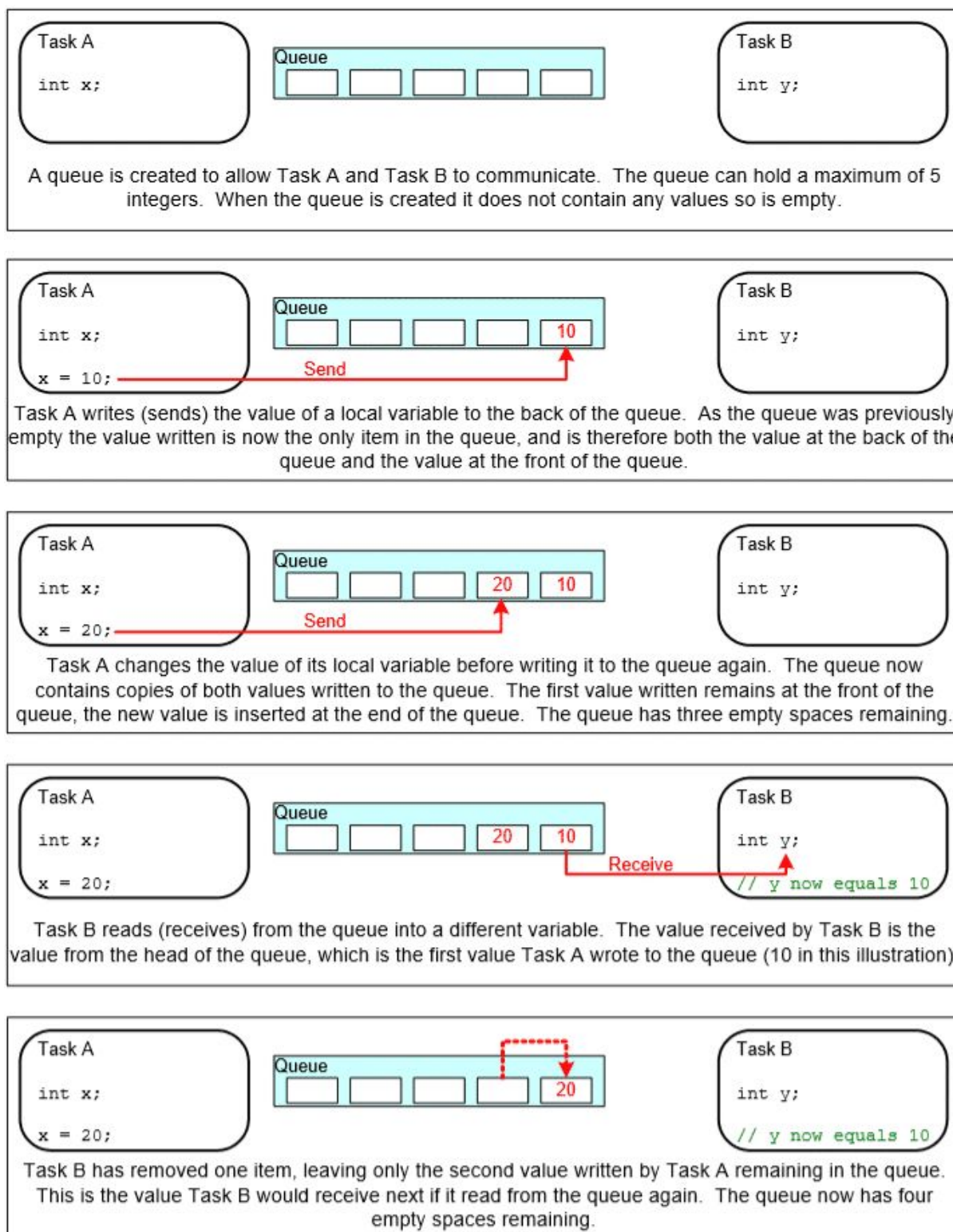
IN



OUT

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!



An example sequence of writes to, and reads from a queue

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:

```
#include <driver/gpio.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <freertos/semphr.h>
#include <esp_system.h>
#include <esp_log.h>
```

```
QueueHandle_t buffer;//Objeto da queue
```

```
void t1(void*z)
{
    uint32_t snd = 0;
    while(1)
    {
        if (snd < 15)//se menor que 15
        {
            xQueueSend(buffer, &snd, pdMS_TO_TICKS(0));//Envia a
variavel para queue
            snd++;//incrementa a variavel
        }
        else//se nao, espera 5seg para testar o timeout da outra tarefa
        {
            vTaskDelay(pdMS_TO_TICKS(5000));
            snd = 0;
        }

        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

```

    }
}

void t2(void*z)
{
    uint32_t rcv = 0;
    while(1)
    {
        if (xQueueReceive(buffer, &rcv, pdMS_TO_TICKS(1000)) ==
true)//Se recebeu o valor dentro de 1seg (timeout), mostrara na tela
        {
            ESP_LOGI("Queue", "Item recebido: %u", rcv);//Mostra o
valor recebido na tela
        }
        else
        {
            ESP_LOGE("Queue", "Item nao recebido, timeout
expirou!");//Se o timeout expirou, mostra erro
        }
    }
}

void app_main()
{
    buffer = xQueueCreate(10, sizeof(uint32_t));//Cria a queue *buffer*
com 10 slots de 4 Bytes

    xTaskCreatePinnedToCore(t1, "t1", 4096, NULL, 1, NULL, 0);//Cria a
tarefa que escreve valores na queue
    xTaskCreatePinnedToCore(t2, "t2", 4096, NULL, 1, NULL, 0);//Cria a
tarefa que le valores da queue
}

```

```
(141) Queue: Item recebido: 0
(641) Queue: Item recebido: 1
(1141) Queue: Item recebido: 2
(1641) Queue: Item recebido: 3
(2141) Queue: Item recebido: 4
(2641) Queue: Item recebido: 5
(3141) Queue: Item recebido: 6
(3641) Queue: Item recebido: 7
(4141) Queue: Item recebido: 8
(4641) Queue: Item recebido: 9
(5141) Queue: Item recebido: 10
(5641) Queue: Item recebido: 11
(6141) Queue: Item recebido: 12
(6641) Queue: Item recebido: 13
(7141) Queue: Item recebido: 14
(8141) Queue: Item nao recebido, timeout expirou!
(9141) Queue: Item nao recebido, timeout expirou!
(10141) Queue: Item nao recebido, timeout expirou!
(11141) Queue: Item nao recebido, timeout expirou!
(12141) Queue: Item nao recebido, timeout expirou!
(13141) Queue: Item recebido: 0
(13641) Queue: Item recebido: 1
```

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.

Referências

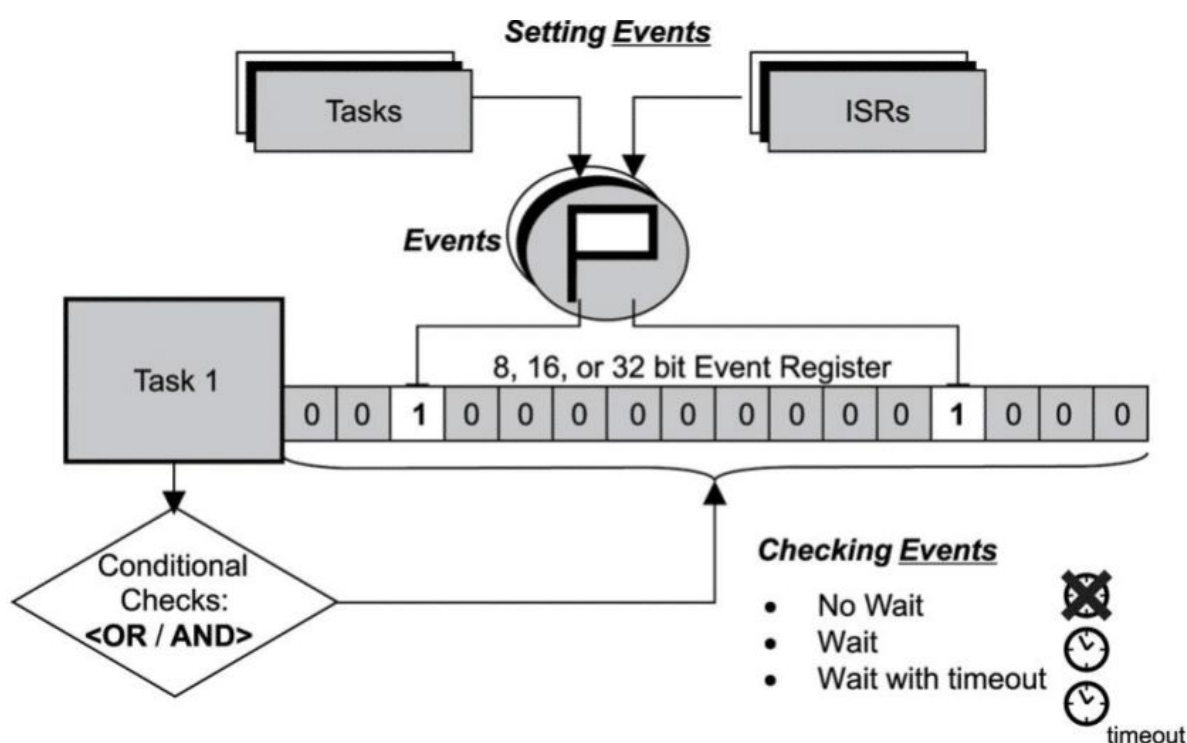
1. https://freertos.org/Documentation/RTOS_book.html
2. <http://esp-idf.readthedocs.io/en/latest/api-reference/system/freertos.html>



Publicado originalmente no Embarcados, no dia 04/03/2020: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhalgual 4.0 Internacional](#).

RTOS: Uso de grupo de eventos para sincronização de tarefas

Autor: [José Morais](#)



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.
- 3.

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:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	
1	BIT	31	30	29	28	27	26	25	24	23	...	6	5	4	3	2	1	0	
2	Descrição	Reservado	Reservado	Reservado	Reservado	Reservado	Reservado	Reservado	Reservado				PWM fade end	Botão de emergência 2 pressionado	Botão de emergência 1 pressionado	Timer: Desconectar cliente do WiFi	Modbus error	Cliente conectado no WIFI	
3																			

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.

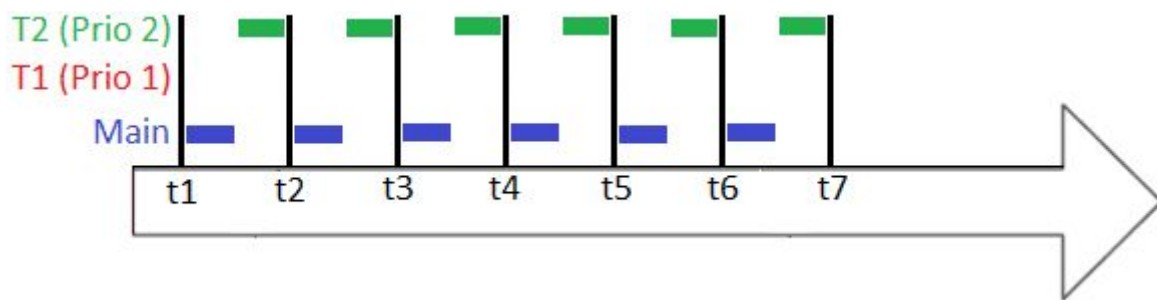


Figura 2 - Tarefas esperando pelo mesmo semáforo.

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.

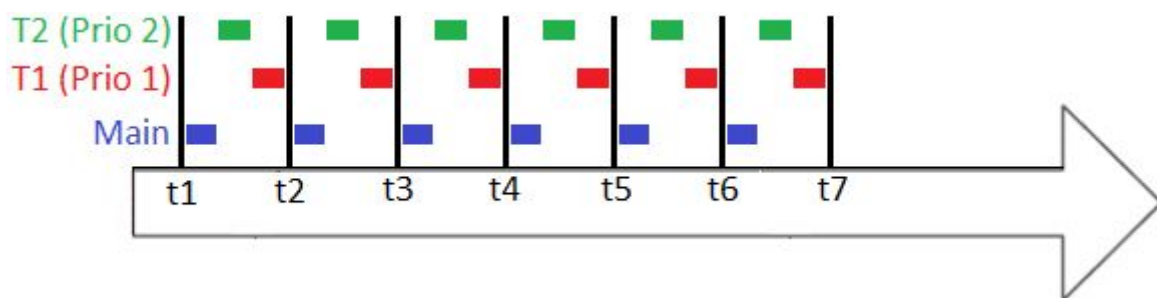


Figura 3 - Tarefas esperando pelo mesmo evento.

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:

```
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <freertos/event_groups.h>
#include <esp_system.h>
#include <esp_log.h>
```

```
EventGroupHandle_t evt;//Cria o objeto do grupo de eventos
```

```

#define EV_1SEG (1<<0)//Define o BIT do evento

void t1(void*z)
{
    EventBits_t x;//Cria a variavel que recebe o valor dos eventos
    while (1)
    {
        //Vamos esperar pelo evento (EV_1SEG) por no maximo 1000ms
        x = xEventGroupWaitBits(evt, EV_1SEG, true, true,
pdMS_TO_TICKS(1000));

        if (x & EV_1SEG)//Se X & EV_1SEG (mascara binaria), significa
que o evento ocorreu
        {
            ESP_LOGI("T1", "OK");
        }
        else
        {
            ESP_LOGE("T1", "Event group TIMEOUT");
        }
    }
}

extern "C" void app_main()
{
    evt = xEventGroupCreate();//Cria o grupo de eventos

    xTaskCreatePinnedToCore(t1, "t1", 2048, NULL, 1, NULL, 0);//Cria a
tarefa que espera pelos eventos

    while (1)
    {
        for (uint8_t i = 0; i < 3; i++)//Envia 3 eventos antes do
timeout
        {

```

```

        vTaskDelay(pdMS_TO_TICKS(333));
        xEventGroupSetBits(evt, EV_1SEG); //Configura o BIT
(EV_1SEG) em 1
    }

    vTaskDelay(pdMS_TO_TICKS(1000)); //Espera o timeout para mostrar
o erro
}
}

```

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.

```

#define EV_1 (1<<0)
#define EV_2 (1<<1)
#define EV_3 (1<<2)

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:

```

#define EV_1 (1<<0)
#define EV_2 (1<<1)
#define EV_3 (1<<2)

```

```
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.

Referências

1. <https://www.freertos.org/event-groups-API.html>
2. https://freertos.org/Documentation/RTOS_book.html
3. <http://esp-idf.readthedocs.io/en/latest/api-reference/system/freertos.html>



Publicado originalmente no Embarcados, no dia 19/09/2018: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhalgual 4.0 Internacional](#).

RTOS: Software Timer no FreeRTOS

Autor: [José Moraes](#)



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.

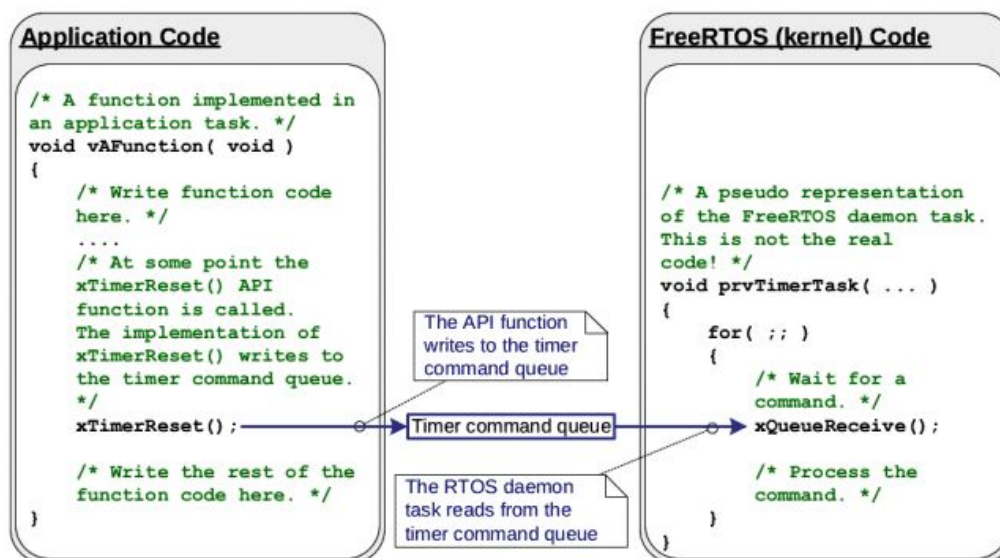
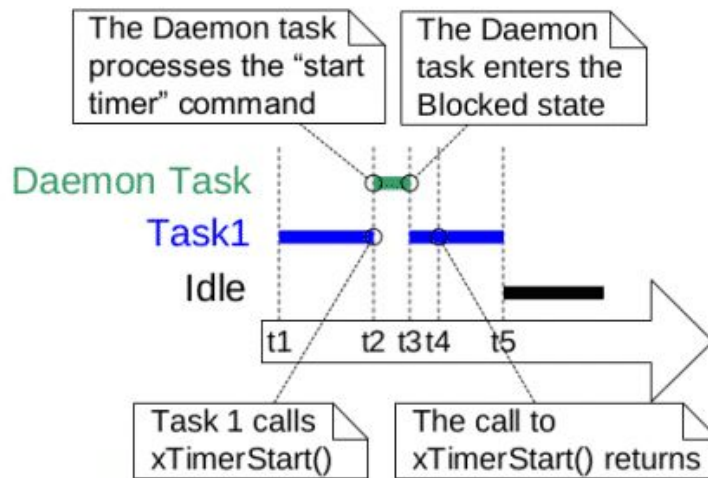


Figura 1 - Comunicação com a Timer Service por queue.



! The execution pattern when the priority of a task calling xTimerStart() is below the priority of the daemon task

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

```
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <freertos/semphr.h>
#include <freertos/timers.h>
#include <esp_system.h>
#include <esp_log.h>
```

```

SemaphoreHandle_t smf;//Cria o objeto do semaforo
TimerHandle_t tmr;//Cria o objeto do timer

void ISR(void*z)
{
    int aux = 0;
    xSemaphoreGiveFromISR(smf, &aux);//Libera o semaforo para uso

    if (aux)
    {
        portYIELD_FROM_ISR();//Forca a troca de contexto se
necessario
    }
}

extern "C" void app_main()
{
    smf = xSemaphoreCreateBinary();//Cria o semaforo

    tmr = xTimerCreate("tmr_smf", pdMS_TO_TICKS(1000), true, 0,
ISR);//Cria o timer com 1000ms de frequencia com auto-reload
    xTimerStart(tmr, pdMS_TO_TICKS(100));//Inicia o timer

    while (1)
    {
        if (xSemaphoreTake(smf, portMAX_DELAY))
        {
            ESP_LOGI("Timer", "expirou!");//Ao obter o semaforo,
enviara a string para o terminal
        }

        vTaskDelay(pdMS_TO_TICKS(10));
    }
}

```

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.

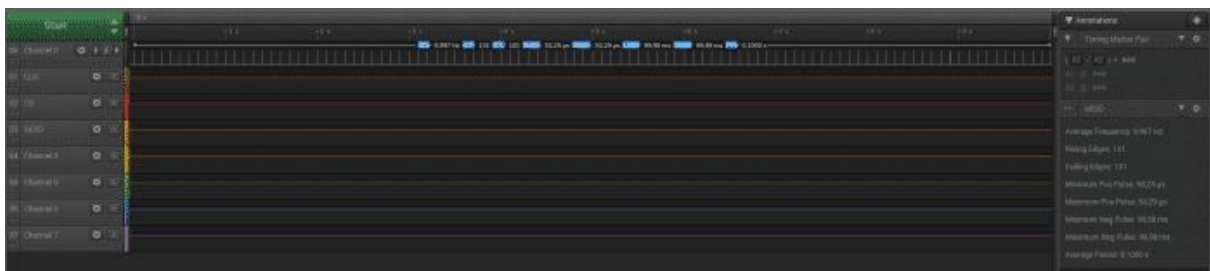


Figura 3 - Analisador lógico para software timer de 10 Hz.

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.

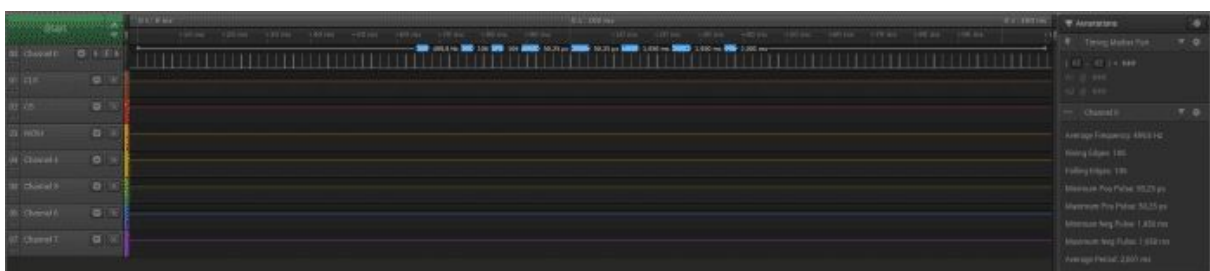


Figura 4 - Analisador lógico para software timer de 500 Hz.

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.

Referências

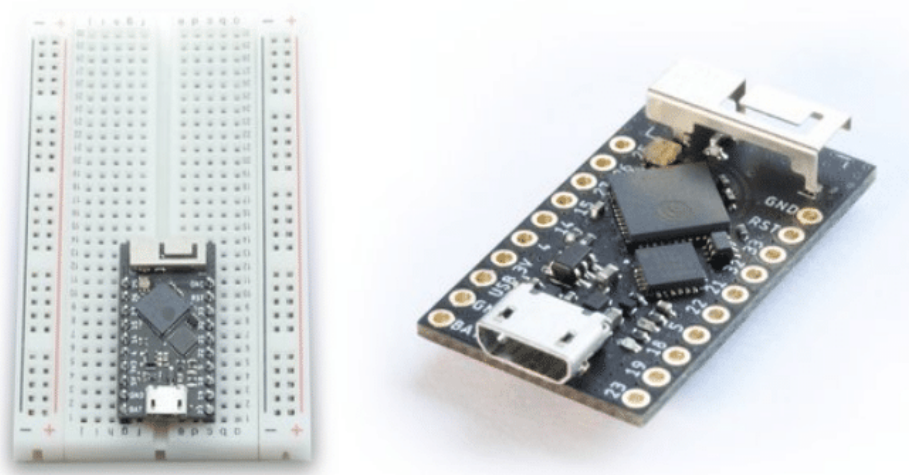
1. <https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html>
2. https://freertos.org/Documentation/RTOS_book.html
3. <http://esp-idf.readthedocs.io/en/latest/api-reference/system/freertos.html>



Publicado originalmente no Embarcados, no dia 17/09/2018: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhalgual 4.0 Internacional](#).

TinyPICO - Pequena placa com o ESP32

Autor: [Fábio Souza](#)

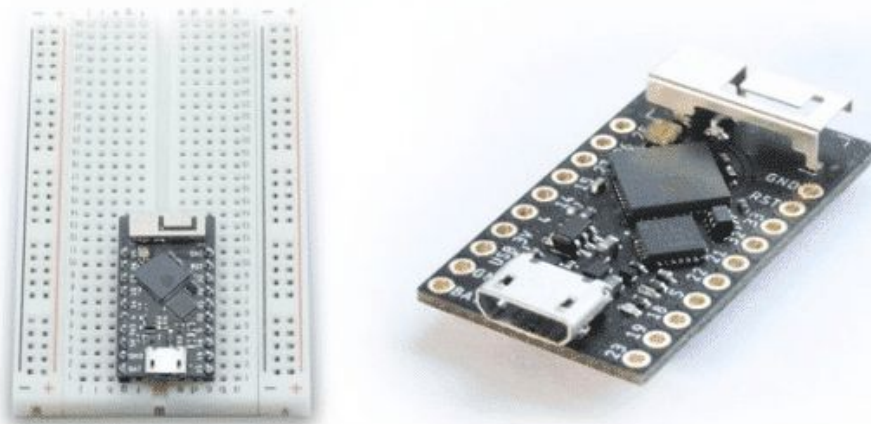


A empresa Unexpected Maker lançou um Crowdfunding para promover a TinyPICO, uma pequena e poderosa placa com o ESP32. Medindo apenas 32 x 18 mm, a placa possibilita a fácil prototipagem com o ESP32, já que é compatível com protoboard e já vem o MicroPython carregado.

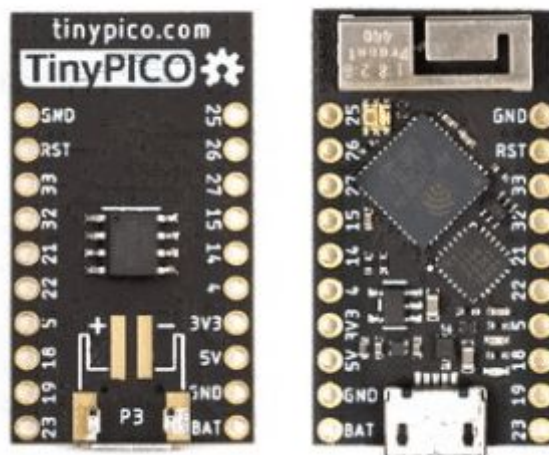
A seguir vamos apresentar os detalhes da TinyPICO e mais detalhes da campanha.

Detalhes da TinyPICO

A placa [TinyPICO](#) é uma das menores placas de desenvolvimento ESP32 do mundo. Foi desenvolvida para fornecer acesso ao processador dual core de 240 MHz e conectividade à Internet do ESP32, tudo em um pequeno form factor.



Além do SOC ESP32, a placa conta com 4 MB de RAM extra, LED RGB on board, regulador de tensão que suporta até 700 mA e circuito para bateria.

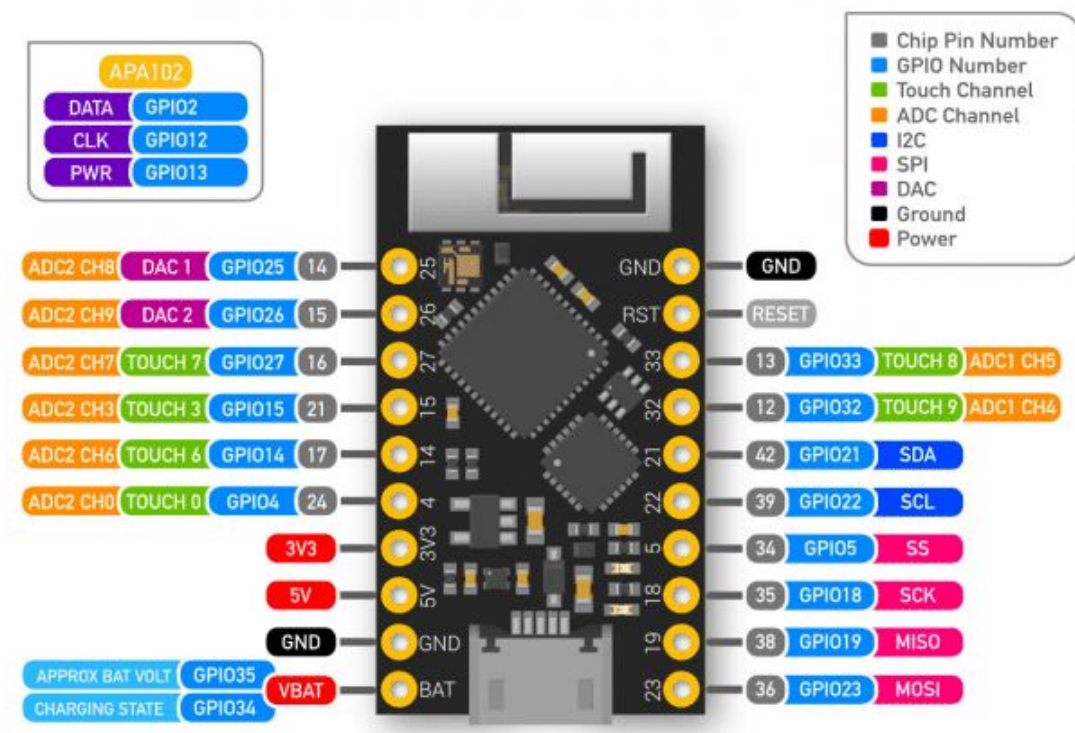


A seguir são apresentados os seus recursos:

- SiP ESP32-PICO-D4 com processador ESP32 dual-core operando a 240 MHz, e 4 MB de memória flash SPI;
- Memória RAM externa de 4 MB;
- Conectividade - 2.4 GHz WiFi 4 802.11b/g/n, Bluetooth 4.2 LE, 3D antena;
- 14 GPIO e compatível com protoboard;
- Conector USB;
- LED para Power (vermelho), carregamento (laranja) e LED RGB APA102;
- Alimentação:
 - 5 V via micro USB;

- Regulador LDO de 3,3 V - 700 mA;
- Otimizada para alimentação por bateria;
- Gerenciamento de bateria LiPo;
- Pads para conector de bateria na parte inferior;
- Consumo – Deep sleep: 18 uA.
- Dimensão – 32 x 18 mm.

A figura a seguir exibe o seu pinout:



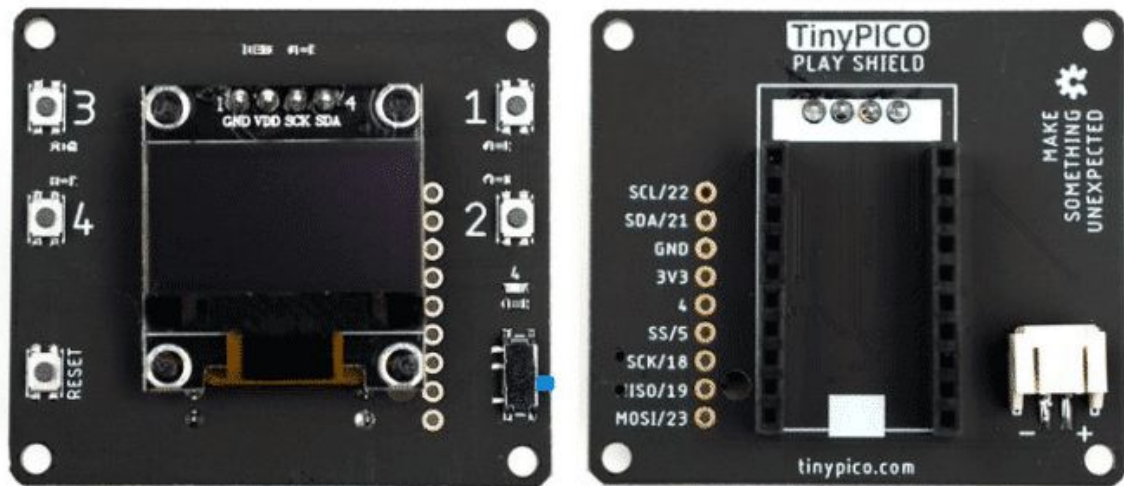
O TinyPICO vem com o MicroPython pré-instalado, suporta o Arduino IDE e também o Espressif IDF. Isso permite flexibilidade para codificar da maneira que desejar.

A equipe [Unexpected Maker](#) está trabalhando para fornecer bibliotecas auxiliares MicroPython e Arduino.

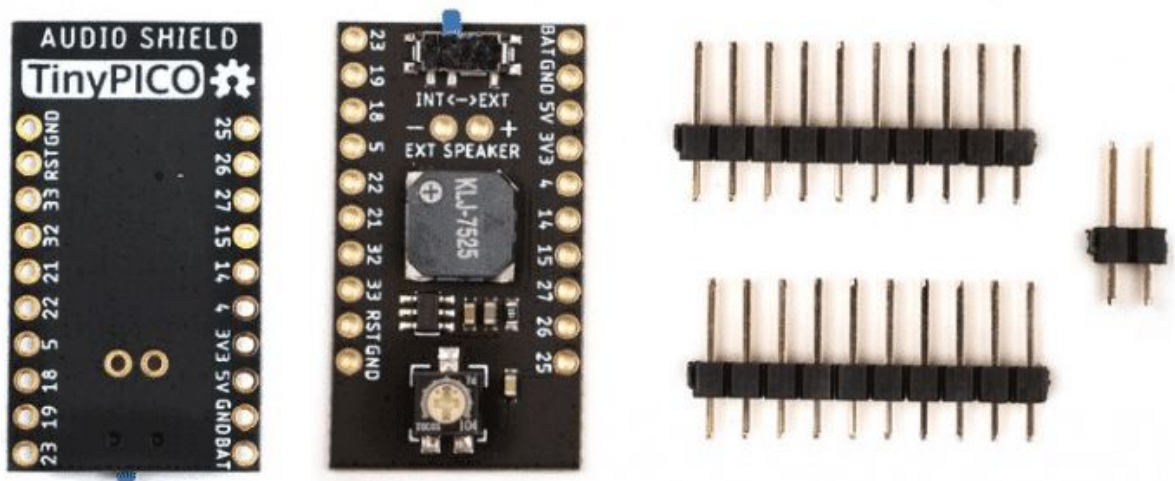
Você encontra exemplos no [site da TinyPico](#) e no [repositório do Github](#).

Além da placa TinyPico, eles estão trabalhando em um conjunto de placas extras, chamadas de TinyPICO Shields. Todas as placas são open hardware e 6 já estão disponíveis como recompensas durante a campanha:

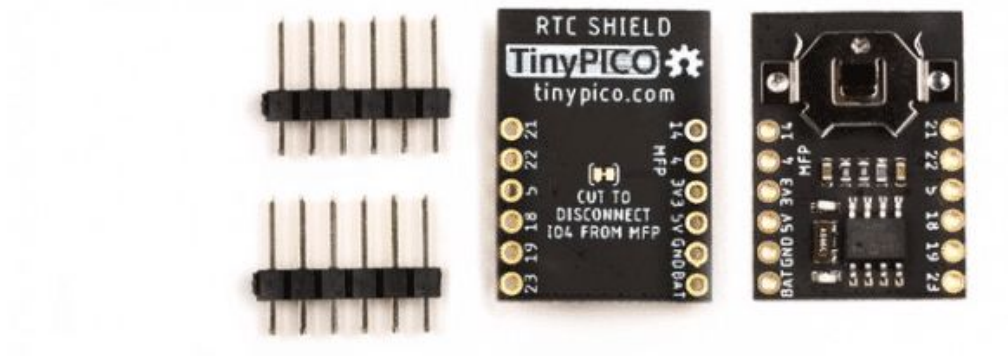
Play Shield



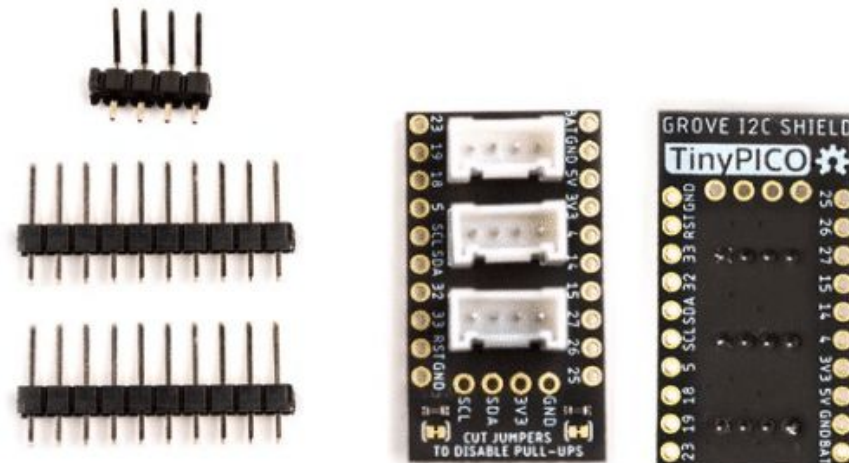
Audio Shield



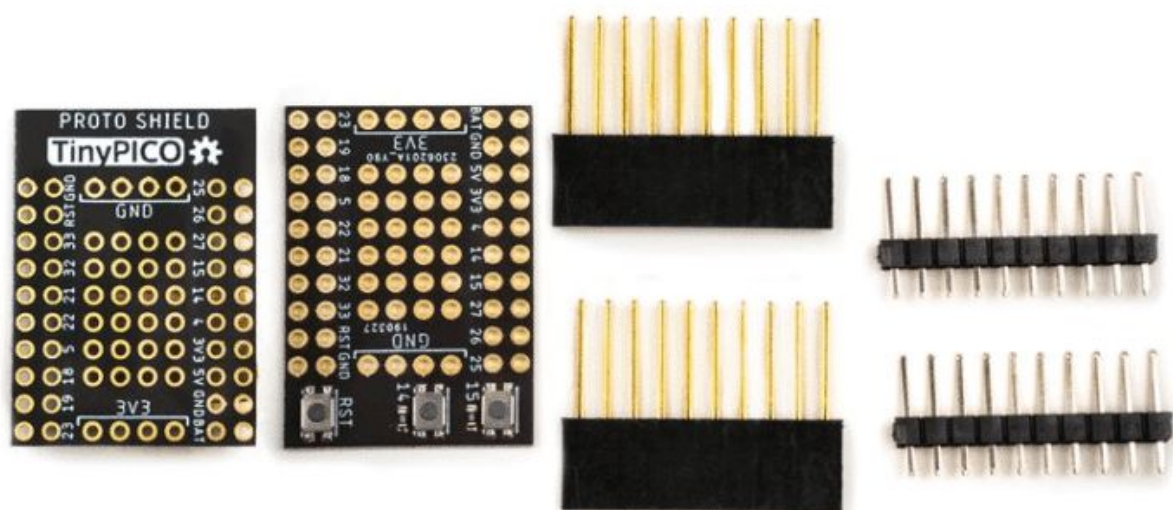
RTC Shield



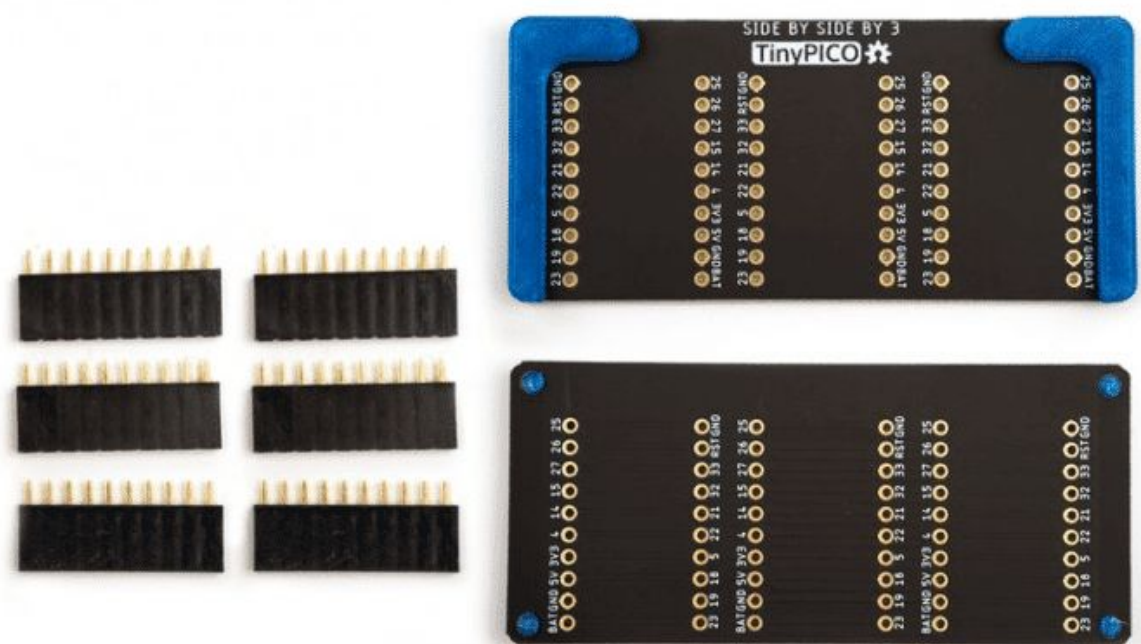
Grove I²C Shield



Proto Shield



3 Up Shield



A TinyPICO está sendo promovida através de [campanha no Crowd Supply](#), que ficará ativa até 20 de Junho. A expectativa de entrega está para o final de Julho de 2019.



Publicado originalmente no Embarcados, no dia 14/05/2019: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhalgual 4.0 Internacional](#).

WiPhone - Telefone VoIP baseado em ESP32

Autor: [Fábio Souza](#)



Foi lançada recentemente uma campanha de financiamento coletivo do WiPhone, um pequeno telefone VoIP baseado no ESP32. O projeto, que tem foco no público maker e hacker, pode ser facilmente desmontado, modificado, tanto em nível de software quanto de hardware.

Ele tem um design similar a um celular do início dos anos 2000, porém, não possui modem 3G/4G, permitindo apenas chamadas de voz via WiFi. Isso permite a comunicação sem a necessidade de uma operadora.

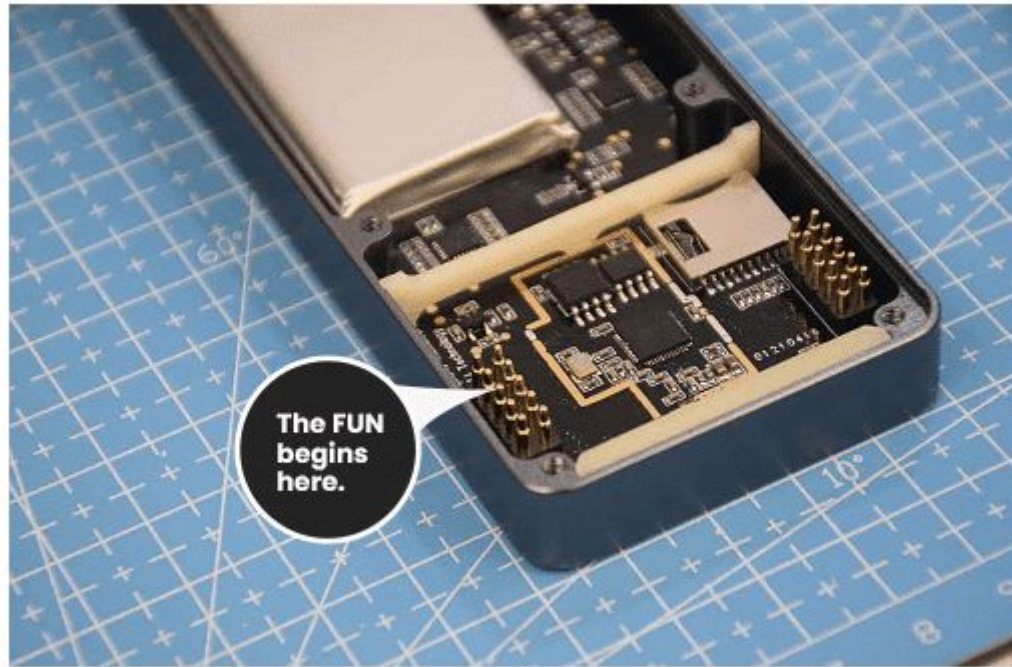


Vamos explorar suas características.

Características do WiPhone

O WiPhone está sendo promovido em duas versões, uma básica e uma pro. Basicamente a diferença está no case, onde a versão básica é feita de policarbonato e a pró em alumínio.

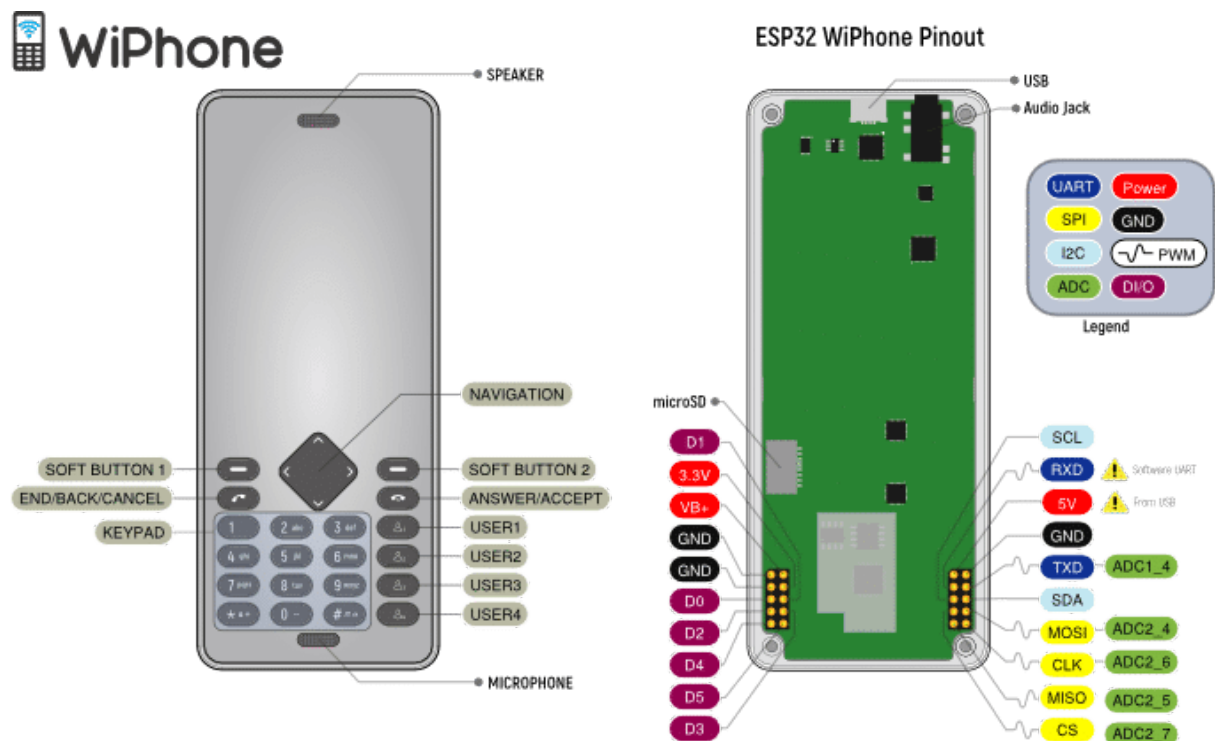
O hardware, baseado em ESP32, permite acesso aos pinos de I/O possibilitando o acoplamento de placas de expansão.



A interface com usuário se dá através de um display de 320 x 240 e um teclado. A seguir temos um resumo das suas características:

- SoC – ESP32 dual core Tensilica LX6 @ 240 MHz
- Memória do sistema – 4 MB PSRAM
- Armazenamento – 16 MB flash, microSD card slot
- Display – tela de 2.4” com 320×240
- Conectividade – 802.11b/g/n WiFi, Bluetooth 4.1 d
- Áudio – 3.5 mm audio jack
- 1 porta micro USB
- Expansão – 20 pinos de I/o para expansão com UART, SPI, I2C, PWM, I/O digital, ADC
- Teclado de 25 teclas
- Bateria de 700 mAh
- Dimensão – 120 x 50 x 12 mm

O WiPhone pode ser programado na IDE do Arduino ou com micropython, permitindo a criação de aplicações diversas. A figura a seguir exibe o seu pinout:



Uma série de placas de expansão já estão disponíveis na campanha, como, por exemplo: LoRa, NFC+RFID, Câmera, expansão para grove kit, entre outras.





A seguir é exibido o vídeo da campanha no Kickstarter:

A campanha está nas últimas horas e ultrapassou a meta de US\$ 40,000. Com um valor de \$89 é possível obter um WiPhone de recompensa, com previsão de entrega para agosto de 2019. Infelizmente não há envio para o Brasil. Confira todos os detalhes da campanha em: [WiPhone, A Phone for Hackers and Makers](#)



Publicado originalmente no Embarcados, no dia 16/04/2019: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhual 4.0 Internacional](#).

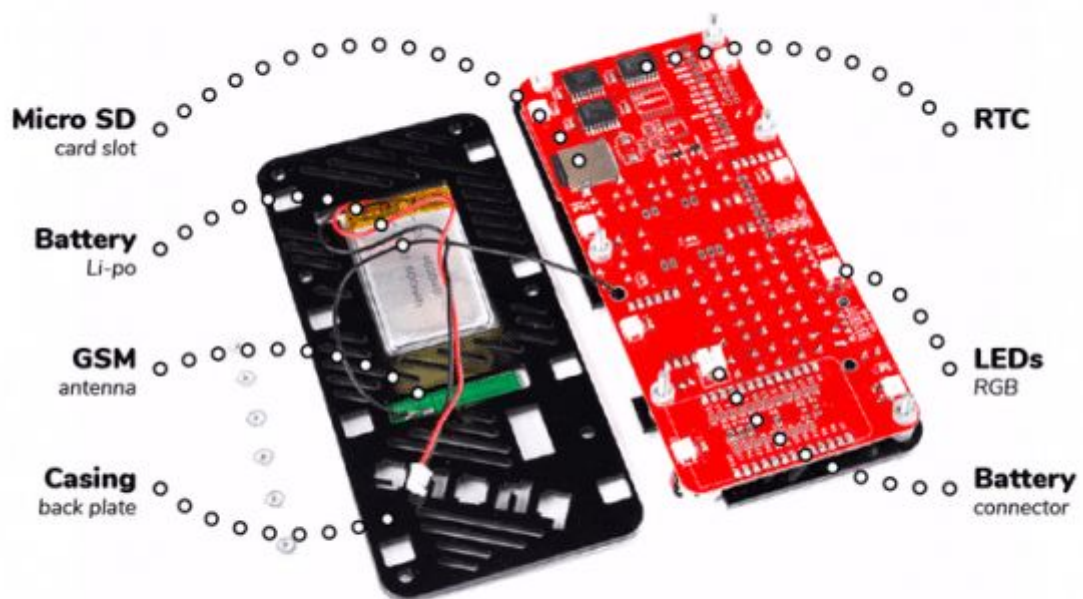
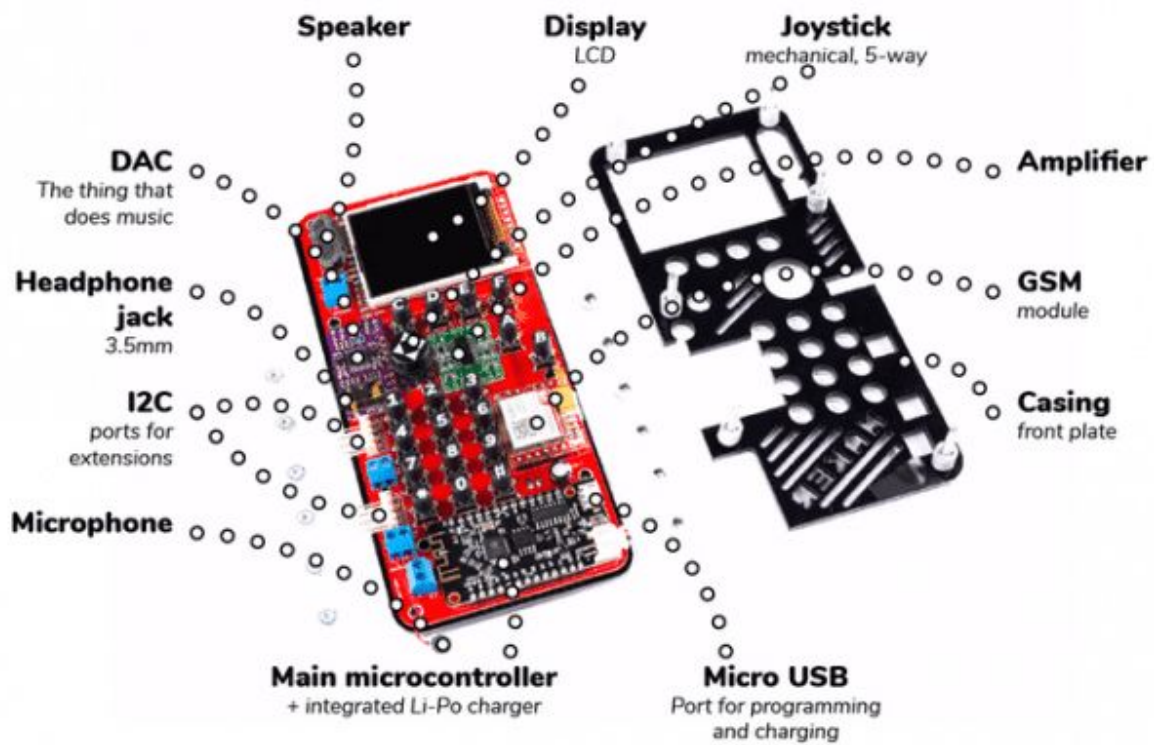
MAKERphone - um celular DIY baseado no ESP32

Autor: [Fábio Souza](#)



Está no ar a campanha no Kickstarter do MAKERphone, um celular de propósito educacional projetado para trazer eletrônica e programação para as pessoas de uma forma divertida e interessante.

O projeto lançado por Albert Gajšak (o mesmo desenvolvedor do [MAKERbuino](#)) é baseado no [poderoso ESP32](#), que possui dois núcleos de 32 bits. A placa possui módulo GSM é uma série de recursos para serem explorados após a montagem:



Conforme exibido no [site da campanha](#), após a montagem a placa poderá ser programada com MicroPython, Scratch e Arduino IDE.

O kit, estimado para entrega em março de 2019, virá desmontado e o tempo de montagem está estimado em 7 horas. É destinado para maiores de 11 anos:



Além do formato DIY para o hardware, onde é possível aprender sobre eletrônica, o kit também é uma excelente plataforma para aprender programação.



Confira os detalhes do projeto no [Kickstarter](#).



Publicado originalmente no Embarcados, no dia 21/10/2018: [link](#) para o artigo original, sob a Licença [Creative Commons Atribuição-Compartilhalgual 4.0 Internacional](#).

Considerações Finais

Os artigos são escritos pela comunidade, isso mesmo, **você pode contribuir com artigos para o Embarcados**. Veja como é fácil: [Seja Colaborador](#).

Somos uma grande comunidade que conversa diariamente sobre assuntos relacionados à área, compartilhando conhecimento de forma online e offline.

Participamos de diversos eventos e também [realizamos eventos](#) com o foco em desenvolvimento de Sistemas Embarcados

Para aproximar os integrantes da comunidade temos 3 canais de comunicação:

- [Comunidade Embarcados no Facebook](#)
- [Comunidade Embarcados no LinkedIn](#)
- [Comunidade Embarcados no Telegram](#)

Convidamos você a compartilhar conhecimento sobre eletrônica e sistemas embarcados

Participe da comunidade Embarcados, compartilhe e aprenda muito!

Se ainda não é registrado no site, aproveite e [faça seu cadastro](#) para receber os melhores conteúdos sobre sistemas eletrônicos embarcados, dicas, tutoriais e promoções.

Caso você tenha encontrado algum problema no material ou tenha alguma sugestão, por favor, entre em contato conosco. Sua opinião é muito importante para nós: contato@embarcados.com.br

Siga o Embarcados na Redes Sociais



<https://www.facebook.com/osembarcados/>



<https://www.instagram.com/portalembarcados/>



<https://www.youtube.com/embarcadostv/>



<https://www.linkedin.com/company/embarcados/>



<https://twitter.com/embarcados>