DISCIPLINA EA801
Laboratório de Projetos de Sistemas Embarcados

Sistemas Operacionais em Tempo Real (RTOS)

Profs. Fabiano Fruett, Antonio Augusto Fasolo Quevedo e Wu Shin-Ting.

FEEC / UNICAMP

Revisado por Wu Shin-Ting em março de 2026 (adequação do enfoque de Pico/VS Code para BlackPill/STM32Cube)
Editado em setembro de 2025 com suporte de Chatgpt, Gemini e NotebookLM.

INTRODUÇÃO

Até aqui, desenvolvemos projetos embarcados explorando dois níveis fundamentais de abstração: o uso de **CMSIS** (do inglês *Cortex Microcontroller Software Interface Standard)* a nível de registradores e a **HAL** *(Hardware Abstraction Layer*). No nível do CMSIS, trabalhamos muito próximos do *hardware*, **acessando diretamente registradores e periféricos** por meio de definições padronizadas, o que oferece alto controle e clareza sobre o funcionamento interno do microcontrolador. Já com a HAL, passamos a operar em um nível mais alto, utilizando **funções e bibliotecas que encapsulam detalhes de baixo nível**, tornando o desenvolvimento mais produtivo, portátil e menos susceptível a erros específicos de *hardware*. Ambas as abordagens são essenciais

para a formação do desenvolvedor de sistemas embarcados, pois permitem compreender desde a manipulação direta dos registradores até o uso estruturado dos periféricos. No entanto, à medida que os sistemas se tornam mais complexos, incorporando múltiplas funcionalidades simultâneas, requisitos de tempo real mais rigorosos e interações concorrentes com sensores, atuadores e interfaces de comunicação, surgem novos desafios.

Mesmo com o suporte da HAL, a organização do *software* costuma seguir um fluxo sequencial ou baseado em interrupções, o que pode dificultar a escalabilidade, os testes e a manutenção. Nesse cenário, a coordenação de múltiplas atividades, o compartilhamento seguro de recursos e a garantia de tempos de resposta previsíveis se tornam desafios centrais do projeto. É justamente nesse contexto que surge o uso de um **sistema operacional de tempo real** (em inglês *Real Time Operating System* - RTOS), introduzindo um novo nível de abstração. Em vez de focar apenas no controle direto do *hardware* ou na configuração de periféricos, o *software* passa a ser estruturado como **um conjunto de tarefas concorrentes**, com mecanismos formais de sincronização, comunicação e escalonamento. Essa abordagem permite lidar de maneira mais sistemática com a crescente complexidade dos sistemas embarcados modernos. Assim, um RTOS não apenas organiza o *software* em tarefas independentes, mas também garante que as tarefas críticas sejam executadas no momento adequado, ou seja, não se trata apenas de "fazer a coisa certa", mas de "fazer a coisa certa, na hora certa

Imagine os seguintes exemplos:

- Um *airbag* que demora milissegundos a mais para inflar já falhou na sua missão.
- Um robô industrial que precisa coordenar seus movimentos com exatidão pode causar acidentes se uma tarefa atrasar.
- Em um carro autônomo, decisões de frenagem, desvio e leitura de sensores precisam acontecer em momentos muito bem definidos.
- Em sistemas de IoT, sensores remotos e atuadores precisam se comunicar e reagir de forma coordenada e confiável.
- Em um dispositivo médico de suporte à vida (por exemplo, ventilador pulmonar), o sistema precisa ser capaz de mudar rapidamente o modo de operação de acordo com informações lidas pelos sensores.

Nesses casos, a precisão no tempo de resposta é tão importante quanto a lógica do programa. Um atraso, mesmo pequeno, pode trazer consequências graves. O RTOS provê mecanismos para os desenvolvedores garantirem que tarefas críticas sejam executadas no momento certo, sem surpresas.

Apesar das vantagens, nem todo projeto precisa de um RTOS. Sistemas simples, com poucas tarefas e sem exigência de precisão temporal, funcionam muito bem com a programação *bare-metal* ou apenas a HAL. Por exemplo, acionar um LED com base em uma leitura simples de sensor pode ser feito com um *superloop* (aquele laço infinito onde tudo acontece em sequência). Nesses casos, usar um RTOS só traria mais complexidade, consumo de memória e tempo de desenvolvimento. Por outro lado, se o sistema tiver várias tarefas concorrentes, por exemplo medir temperatura, comunicar por Wi-Fi e controlar um motor ao mesmo tempo, precisar gerar reações a eventos externos com prazos curtos (ex: interrupções que exigem resposta rápida), e tiver que satisfazer requisitos de tempo previsíveis e confiáveis (ex: controle em tempo real, tarefas periódicas com precisão de microssegundos), então vale a pena considerar o uso de um RTOS.

Neste roteiro, vamos entender o que é um RTOS, para que ele serve e em quais situações seu uso é realmente vantajoso. Para isso, faremos uma comparação com as abordagens que você já conhece, a programação bare-metal e o uso da HAL, a fim de desenvolver uma visão crítica sobre o tema. A proposta é compreender que um RTOS pode tornar o sistema mais robusto, organizado e confiável, mas também reconhecer quando sua adoção não se justifica.

PROJETOS DE REFERÊNCIA COM RTOS

Esta seção apresenta três projetos de referência desenvolvidos com o FreeRTOS, um sistema operacional de tempo real (RTOS) de código aberto, portátil e amplamente utilizado em sistemas embarcados. Esses projetos utilizam os periféricos da BitDogLab e têm como objetivo introduzir, de forma prática, os principais conceitos de **programação concorrente** em microcontroladores, além de demonstrar como um RTOS organiza o *software* em múltiplas tarefas concorrentes, em contraste com o modelo tradicional *bare-metal*, tipicamente estruturado como um laço principal com controle explícito do fluxo e uso de interrupções.

Originalmente, os três projetos foram desenvolvidos pelo Prof. Quevedo para a plataforma Pico, utilizando o SDK oficial do Raspberry Pi Pico (Pico SDK) em conjunto com o FreeRTOS, no ecossistema do Visual Studio por meio da extensão Raspberry Pi Pico. Os códigos estão totalmente comentados, permitindo compreender não apenas o que está sendo feito, mas também as razões por trás de cada decisão.

```
git clone https://gitlab.unicamp.br/rtos_pico/freertos_mutex.git
git clone https://gitlab.unicamp.br/rtos_pico/freertos_signal.git
git clone https://gitlab.unicamp.br/rtos_pico/freertos_queue.git
```

Para este roteiro, eles foram migrados para a plataforma BlackPill, passando a utilizar a biblioteca HAL e a interface CMSIS-RTOS v2 (sobre o FreeRTOS), tirando proveito do ambiente de desenvolvimento oferecido pelo STM32Cube (CubeMX e CubeIDE). A descrição apresentada aqui funciona como um guia introdutório para facilitar a navegação pelos projetos. Os três projetos são organizados como subpastas dentro do repositório:

https://gitlab.unicamp.br/teaching/bitdoglab/blackpill/rtos/

Para evitar baixar arquivos desnecessários, recomenda-se usar *partial clone* e *sparse checkout*, permitindo trabalhar apenas com os projetos desejados:

```
git clone --filter=blob:none --sparse https://gitlab.unicamp.br/teaching/bitdoglab/blackpill/rtos.git
cd rtos
git sparse-checkout init --cone

git sparse-checkout set rtos_freertos_mutex
git sparse-checkout set rtos_freertos_signal
git sparse-checkout set rtos_freertos_queue
```

Observe que o *plugin* EGit do STM32CubeIDE não oferece suporte completo a essas funcionalidades; portanto, recomenda-se realizar esses passos via terminal.

Todos os projetos foram desenvolvidos com o STM32CubeMX em conjunto com o STM32CubeIDE, como no Roteiro 2. A configuração do sistema é realizada no CubeMX e

armazenada no arquivo `.ioc`, a partir do qual o código do projeto é gerado automaticamente, incluindo a configuração de pinos, frequências de *clock* e a inicialização dos periféricos.

Além dessas configurações básicas, neste roteiro mostramos que o CubeMX também permite configurar o *middleware*[1] FreeRTOS por meio da interface CMSIS-RTOS v2, possibilitando a criação de tarefas (*threads*), a definição de prioridades e a utilização de mecanismos de comunicação e sincronização, como filas, semáforos e mutexes. Entre os arquivos gerados automaticamente, destacam-se:

- `main.c`: contém a função `main`, a inicialização geral do sistema e *callbacks* básicos;
- `freertos.c`: concentra a criação das tarefas e a configuração dos mecanismos de sincronização e comunicação do RTOS.

Após a clonagem, os projetos devem ser importados no STM32CubeIDE. A partir disso, será possível:

- compilar o código;
- gerar o executável;
- explorar a documentação das funções.

Cabe apenas a ressalva de que o projeto `rtos_freertos_queue` utiliza funções para controle de LEDs NeoPixel (WS2812), disponíveis na biblioteca estática `libhal_BDL_Library.a`. Para utilizá-la, é necessário

- clonar o projeto dessa biblioteca;
- adicionar os arquivos de cabeçalho com os protótipos das funções e incluir o caminho da biblioteca no projeto `rtos_freertos_queue`.

Esses passos são detalhados na seção “Utilização das funções de uma biblioteca estática” do Roteiro 2.

Freertos_mutex

O problema aqui consiste em criar duas tarefas que compartilham o mesmo recurso, a UART, e que devem enviar dados de forma concorrente. O objetivo é garantir que o **acesso à UART seja feito de maneira ordenada**, sem conflitos, e que ambas as **tarefas consigam realizar suas operações de forma independente**, sem interferir uma na outra.

Para este exercício, queremos enviar 20 caracteres idênticos à UART seguidos de uma quebra de linha (“\r\n”). A única diferença entre as tarefas será o caractere que cada uma envia. Deve-se conectar a ponte TX e RX (ST-LINK) nos pinos Rx e Tx da I2C0.

Em um **sistema *bare-metal***, sem o uso de um RTOS, o controle de concorrência entre as tarefas precisaria ser gerido diretamente pelo programador. Nesse caso, o controle de acesso à UART seria feito manualmente, geralmente por meio de interrupções. Uma ideia seria configurar uma interrupção para disparar a cada intervalo de tempo definido por `osDelay(ms)`. Cada vez que a interrupção ocorre, o processador alternaria entre as tarefas, enviando um caractere para a UART.

[1] *Middleware* é uma camada de *software* intermediária que conecta aplicações ao sistema operacional ou a outros serviços, facilitando a comunicação e o gerenciamento de recursos.

No entanto, como ambas as tarefas tentam acessar a UART, seria necessário implementar alguma forma de proteção contra o acesso simultâneo, algo como *flags* ou controle explícito de seções críticas, para garantir que apenas uma tarefa tenha acesso ao recurso UART de cada vez. Além disso, como o sistema *bare-metal* não possui um mecanismo para gerenciar a execução das tarefas, seria necessário implementar manualmente o escalonamento entre as tarefas. A programação seria mais direta, mas muito mais susceptível a erros e exigiria uma coordenação manual complexa.

Com o uso do FreeRTOS, o gerenciamento da concorrência entre tarefas que acessam a UART se torna mais simples e estruturado. O sistema operacional fornece abstrações como semáforos e mutexes, que permitem proteger recursos compartilhados de forma segura e previsível. Além disso, o escalonador do FreeRTOS é responsável por alternar a execução das tarefas de maneira eficiente, respeitando prioridades e estados de bloqueio. Este projeto demonstra tanto o uso de funções reentrantes quanto a aplicação de mecanismos de sincronização baseados em exclusão mútua (por meio de um semáforo binário). São criadas duas tarefas que compartilham o mesmo recurso, a UART, e executam a mesma função, diferenciando-se apenas pelo parâmetro recebido. Cada tarefa transmite uma sequência de 20 caracteres idênticos, com pequenos intervalos (`osDelay`) entre cada envio, seguidos de uma quebra de linha. A distinção entre elas está no conteúdo transmitido: uma envia o caractere `'1'`, enquanto a outra envia `'2'`. Dessa forma, evidencia-se o reuso de código por meio da passagem de parâmetros, bem como os efeitos da concorrência no acesso a um recurso compartilhado.

Para o exemplo, existem duas possíveis compilações condicionais, definidas pelo valor da macro `USE_MUTEX`. Se o valor for 0, as tarefas não usam `Mutex`; se o valor for 1, as tarefas usam `Mutex`. Compile com o valor 0 e veja o que acontece no monitor serial; depois mude o valor para 1 e veja a diferença.

Freertos_signal

O objetivo deste projeto é gerenciar duas tarefas concorrentes: uma que gera um "*bip*" em um *buzzer* periodicamente, e outra que altera a cor de um LED RGB em resposta a esses "*bips*". A questão principal é **garantir a comunicação entre as duas tarefas** para que o LED só mude de cor quando o *buzzer* emitir um "bip", e isso deve ser feito de maneira eficiente e **sem interferência entre as tarefas**.

Em um **sistema *bare-metal***, sem o auxílio de um RTOS, o controle de tarefas e a comunicação entre elas seria feito diretamente por interrupções e *flags*. A primeira tarefa, responsável por gerar o "*bip*" no *buzzer*, poderia ser acionada por uma interrupção periódica ou um temporizador configurado para gerar os "*bips*" com a frequência definida pelas macros `BUZZER_FREQ`, `BUZZER_ON_TIME` e `BUZZER_PERIOD`. Quando o *buzzer* desligasse, o *bit* de uma *flag* ou uma variável de estado seria alterado, indicando que o "*bip*" havia ocorrido. A segunda tarefa, que controla a cor do LED RGB, precisaria verificar periodicamente essa *flag* para saber quando o *buzzer* emitiu um "*bip*" e, com isso, alterar a cor do LED. Para garantir que a comunicação entre as duas tarefas fosse segura, o programador precisaria implementar manualmente algum mecanismo de sincronização, como interrupções ou *polling*, para garantir que a segunda tarefa só mudasse a cor do LED após a conclusão do "*bip*" no *buzzer*. Nesse modelo, a coordenação entre as tarefas é feita por *flags* ou variáveis compartilhadas, e o controle de tempo ou de espera entre as tarefas teria que ser

implementado manualmente. O programador teria que se preocupar com os detalhes de como as tarefas são escalonadas e de como garantir a exclusão mútua ou a sincronização entre as atividades de leitura e escrita nas variáveis compartilhadas.

Com o FreeRTOS, a solução para a comunicação entre as duas tarefas é significativamente simplificada, graças aos `Event Groups`. A primeira tarefa gera um "*bip*" no *buzzer* periodicamente. A frequência do sinal do *buzzer* é configurada por meio do CubeMX. Quando a tarefa desliga o *buzzer*, ela seta um *bit* do *Event Group*, `BUZZER_EVENT`, indicando o evento do "*bip*". A segunda tarefa inicia acendendo o LED RGB na cor vermelha. A cada vez que o *bit* do `Event Group` é setado, ela se torna ativa e muda a cor do LED. Note que a própria função que aguarda o *bit* faz a limpeza do mesmo. Note também que a variável que guarda o número do LED atualmente acesso não precisa ser *static*, apesar de ser local. **O contexto da tarefa é preservado quando a tarefa ativa muda, preservando o valor das variáveis locais**.

Freertos_queue

Neste projeto, o objetivo é implementar um sistema composto por duas tarefas cooperativas: uma produtora, que gera dados periodicamente, e uma consumidora, que recebe e processa esses dados. A tarefa produtora deve gerar dois números aleatórios, um para definir a posição de um LED em uma "matriz" 5x5 (0 a 24) e outro para definir uma entre sete cores possíveis (1 a 7). Esses dois valores são agrupados em uma estrutura e enviados para a tarefa consumidora, que os utiliza para acionar visualmente um LED da "matriz" RGB (NeoPixel) com a cor correspondente.

O desafio está em **sincronizar a troca de dados entre as tarefas**, de modo que a consumidora só atue quando houver novos dados disponíveis e processe corretamente os valores gerados pela produtora.

Em uma **implementação *bare-metal***, sem RTOS, não existem abstrações como filas ou gerenciamento automático de tarefas. O programador teria que lidar diretamente com a comunicação e sincronização entre os blocos de código responsáveis pela geração e o processamento dos dados. Uma abordagem possível seria um modelo Produtor-Consumidor regido pelo uso de uma área de memória compartilhada e um mecanismo de sinalização. Inicialmente, a tarefa produtora é configurada para ser executada periodicamente, utilizando um *timer* ou uma função de atraso (*delay*), por exemplo, a cada 1 segundo. Em cada ciclo, ela gera dois valores aleatórios, os organiza em uma estrutura e os armazena na área de memória compartilhada. Para sinalizar a disponibilidade de novos dados, uma *flag* ou uma variável de estado é definida. A tarefa consumidora, por sua vez, verifica continuamente (*polling*) o estado dessa *flag*. Ao detectar a disponibilidade de dados novos, ela copia os valores da estrutura, executa a ação de saída (apagando todos os LEDs da matriz RGB e acendendo o LED correspondente com a cor indicada) e, imediatamente após o processamento, a *flag* é resetado para aguardar o próximo ciclo.

Embora funcional, essa solução apresenta desvantagens importantes. O consumo de recursos é o problema mais imediato, pois a tarefa consumidora opera em *polling*, verificando continuamente a condição de novos dados e, consequentemente, desperdiçando ciclos de CPU. Além disso, há um risco inerente de condições de corrida se o acesso à estrutura de memória compartilhada não for protegido adequadamente, exigindo o uso de mecanismos como a desabilitação de interrupções

durante a leitura ou escrita. Por fim, a natureza dessa abordagem limita a escalabilidade do código, dificultando a adição e o gerenciamento eficiente de mais tarefas no futuro.

Com o FreeRTOS, a comunicação entre as tarefas produtora e consumidora é implementada de forma segura e eficiente utilizando uma **Queue (fila)**, que é uma estrutura de dados fornecida pelo sistema operacional para troca de mensagens entre tarefas. O projeto (freertos_queue) também tem duas tarefas, denominadas "Produtora" (*ProducerTask*, produz informação) e "Consumidora" (*ConsumerTask*, recebe e trata informação). No exemplo, a informação trabalhada é uma "struct" com dois valores. A tarefa produtora gera dois números aleatórios, sendo o primeiro entre 0 e 24 e o segundo entre 1 e 7. Essa geração ocorre a cada segundo. A seguir, a tarefa produtora coloca os dois valores na `LedMsg_t msg` e a coloca em uma *Queue*. A tarefa consumidora aguarda a chegada de valores pela *Queue* e, ao chegar um novo elemento na *Queue*, apaga todos os LEDs da matriz RGB e acende apenas um dos 25 LEDs (definido pelo primeiro valor da `msg`) com uma de 7 cores (definida pelo segundo valor da `msg`).

DESAFIO

Cada grupo de projeto deve migrar o projeto desenvolvido no Roteiro 2 (baseado em HAL e execução *bare-metal*) para uma arquitetura baseada em tarefas utilizando FreeRTOS por meio da interface CMSIS-RTOS v2, mantendo a mesma funcionalidade original. A migração deve ser guiada pela transformação do modelo sequencial (*bare-metal*) para o modelo concorrente (RTOS), conforme o seguinte mapeamento:

Modelo *bare-metal*	Modelo RTOS
fluxo de controle central em laço infinito (`while(1)`)	Execução distribuída em múltiplas tarefas
Controle manual do fluxo de execução	Escalonamento automático pelo RTOS
Uso de *delays* bloqueantes (`HAL_Delay`) a nível do sistema	Uso de *delays* bloqueantes (*osDelay*) apenas a nível da tarefa, sem bloqueio da CPU
Interrupções tratam diretamente a lógica da aplicação	Interrupções sinalizam tarefas (integração com o RTOS)
Acesso direto a periféricos	Acesso protegido por `Mutex`
Uso de *flags* e variáveis globais	Sincronização por semáforos
Polling de múltiplas condições	Sincronização por grupo de eventos
Compartilhamento direto de dados	Comunicação estruturada por filas

O projeto será dividido nas seguintes fases:

1. Fase de análise do projeto original: Identificar, no código *bare-metal*:
   a. funcionalidades principais
   b. periféricos utilizados (UART, GPIO, etc.)

c. estrutura do `while(1)`
d. uso de *delays* (`HAL_Delay`)
e. dependência de interrupções

2. Fase de decomposição em tarefas: Decompor o comportamento sequencial em tarefas independentes. Cada tarefa deve
   a. ter uma responsabilidade clara;
   b. ser, sempre que possível, isolada das demais;
   c. representar um fluxo de execução específico do sistema;
   d. nteragir com outras tarefas exclusivamente por meio de mecanismos do RTOS, evitando o uso direto de variáveis globais.
3. Fase de Implementação
   a. Configuração no STM32CubeMX
      i. Ativar FreeRTOS
      ii. Criar tarefas
      iii. Definir prioridades
      iv. Configurar mecanismos de sincronização e comunicação
   b. Implementação da lógica de cada tarefa no STM32CubeIDE
      i. Implementar a lógica de cada tarefa;
      ii. Substituir delays bloqueantes por `osDelay();`
      iii. Inserir mecanismos de sincronização (mutex, semáforos, filas, etc.)
   c. Testes**:** Realizar testes de unidade, integração e validação do sistema para garantir que todos os requisitos funcionais e não-funcionais sejam atendidos. Isso abrange testar a leitura de sensores, o acionamento de atuadores e a lógica de controle, quando aplicável.

4. Entregas do Projeto
   - Documento de Projeto (Relatório no arquivo README.md do repositório do projeto no GitLab seguindo o modelo, e o *link* do projeto no Moodle).
   - Código-Fonte:
     - Código completo do *firmware*, organizado e comentado, no repositório de controle de versão GitLab da Unicamp.
   - Demonstração do Protótipo:
     - Apresentação do sistema funcionando no *kit* BDL, com o circuito adicional montado no *protoboard*, durante a aula de entrega do projeto.

Avaliação

O projeto será avaliado com base na:

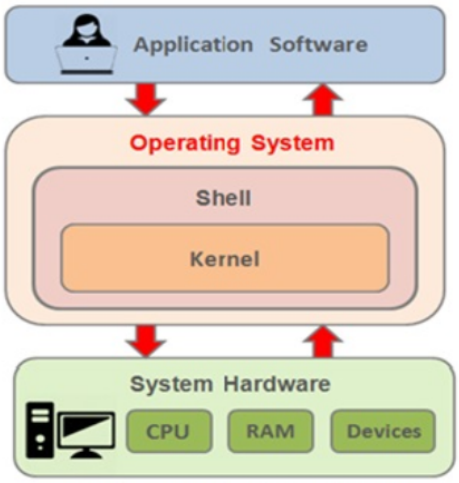
- funcionalidade da solução implementada.
- qualidade do código (estrutura, clareza, documentação).
- capacidade de aplicar os conceitos de sistemas embarcados.
- organização e clareza do relatório final e da apresentação.

FUNDAMENTOS TEÓRICOS

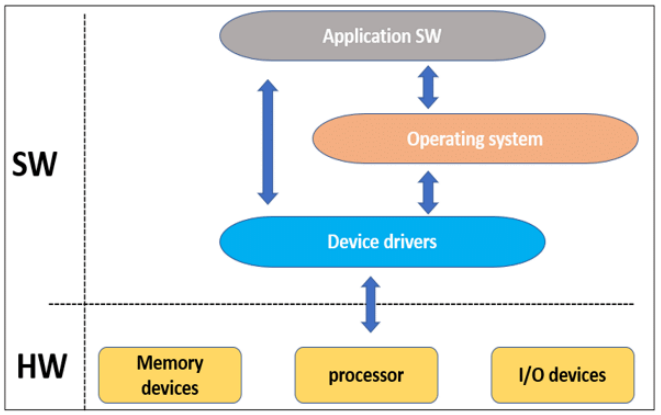
Um **sistema operacional (SO)** é a camada de *software* responsável por gerenciar os recursos de *hardware* de um sistema computacional, permitindo a execução de programas. Para isso, ele controla o uso da CPU, organiza a execução de tarefas ou processos, gerencia recursos como memória, tempo e periféricos, além de fornecer serviços e abstrações que facilitam o desenvolvimento e a execução das aplicações. Enquanto SOs de uso geral (em inglês, *General Purpose Operating System* ou GPOS), como Windows, Linux, macOS e Android, focam em

maximizar a eficiência e a experiência do usuário, sistemas embarcados em aplicações críticas têm outra prioridade: o tempo real. Neste contexto, a validade de uma tarefa depende não apenas de sua lógica, mas também de sua conclusão dentro de um prazo rigorosamente definido, conhecido como ***deadline***. Um exemplo clássico é o sistema de *airbag* automotivo, onde um atraso de milissegundos invalida completamente a função. Para atender a essas demandas temporais, foram criados Sistemas Operacionais de Tempo Real (RTOS), como o FreeRTOS, Zephyr, QNX e VxWorks, projetados para garantir **baixa latência** (tempo mínimo entre o estímulo e a resposta) e ***jitter* reduzido** (variação na latência de resposta), assegurando um comportamento previsível e confiável.

Os dois sistemas oferecem diferentes níveis de abstração em relação ao *hardware*. Um sistema operacional de propósito geral (GPOS) atua como uma camada intermediária que abstrai a complexidade do *hardware*, utilizando *drivers* e o *kernel* para gerenciar o acesso aos recursos de forma segura e controlada. Sua prioridade está na segurança, na estabilidade e na boa responsividade do sistema, o que também favorece a portabilidade das aplicações. Para isso, disponibiliza uma ampla gama de serviços genéricos, como sistemas de arquivos, gerenciamento avançado de memória e funções padronizadas de entrada e saída. Por outro lado, um sistema operacional de tempo real (RTOS) é projetado para garantir previsibilidade e baixa latência, características essenciais em sistemas embarcados de controle. Diferentemente de uma visão equivocada comum, que trata o RTOS como apenas um GPOS com restrições temporais mais rigorosas, sua principal distinção está na capacidade de oferecer comportamento determinístico, isto é, garantir que tarefas críticas sejam executadas dentro de prazos bem definidos.



Fonte: GPOS (LCS).



Fonte: RTOS (Researchgate).

Comparativo de Paradigmas: GPOS vs. RTOS

Característica	GPOS (Linux, Windows, macOs, Android)	RTOS (FreeRTOS, Zephyr RTOS, QNX, VxWorks)
Filosofia	Segurança, estabilidade e multitarefa	Previsibilidade, desempenho e baixa latência
Abstração	Alta (esconde o *hardware*)	Baixa (expõe o *hardware* para o desenvolvedor)
Interação	Sempre via API do *kernel* e *drivers*	Pode ser direta(via registradores) ou via APIs leves
Prioridade	Proteger o sistema e os processos	Garantir tempo de resposta crítico

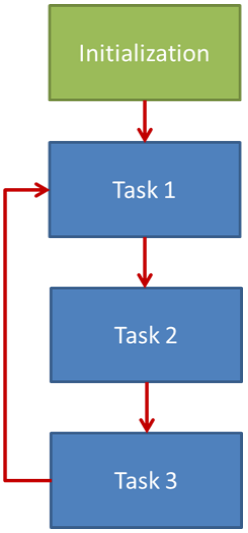
Limitações do *bare-metal*

Microcontroladores modernos, como os baseados na família ARM Cortex-M, oferecem recursos avançados de *hardware*, incluindo o controlador de interrupções vetorizadas com suporte a prioridades aninhadas (NVIC do inglês *Nested Vectored Interrupt Controller)*. Esse mecanismo permite que eventos críticos possam interromper tarefas de menor prioridade, aumentando a responsividade do sistema. No nível de *hardware*, esses eventos são geralmente representados por *flags* em registradores de estado. Cada *flag* indica a ocorrência ou ausência de uma determinada condição ou evento, como a chegada de um dado pela UART, a finalização de uma conversão do ADC ou a ativação de um pino de interrupção externa. Essas flags são monitoradas pelo NVIC e, quando setadas, desencadeiam automaticamente uma interrupção associada. Essa arquitetura orientada a eventos é essencial para sistemas embarcados reativos.

No entanto, mesmo com essas capacidades, um sistema *bare-metal* continua dependendo quase inteiramente do programador para definir o fluxo de execução. Nesse contexto, temporizadores (*timers*)[2] se tornam recursos de *hardware* essenciais, pois geram interrupções periódicas com uma frequência precisa, criando uma base de tempo para o sistema. É essa base de tempo que permite ao desenvolvedor garantir um controle preciso dos prazos atribuídos a cada tarefa, utilizando um modelo típico de desenvolvimento *bare-metal* que combina um laço principal (*superloop*) com rotinas de tratamento de interrupções. Cabe ao desenvolvedor decidir a ordem e o momento de execução de cada tarefa, além de como sincronizar o acesso a recursos compartilhados e como lidar com múltiplos eventos concorrentes sem perda de prazos.

Para tentar organizar a execução de múltiplas funcionalidades, alguns padrões de escalonamento[3] manual são aplicados em *superloops*:

- ***Round-robin* com *polling***: Cada tarefa é chamada em sequência fixa, verificando periodicamente se há algo a fazer. Nesta abordagem, tarefas que não precisam rodar ocupam tempo de CPU; eventos podem ser detectados com atraso.

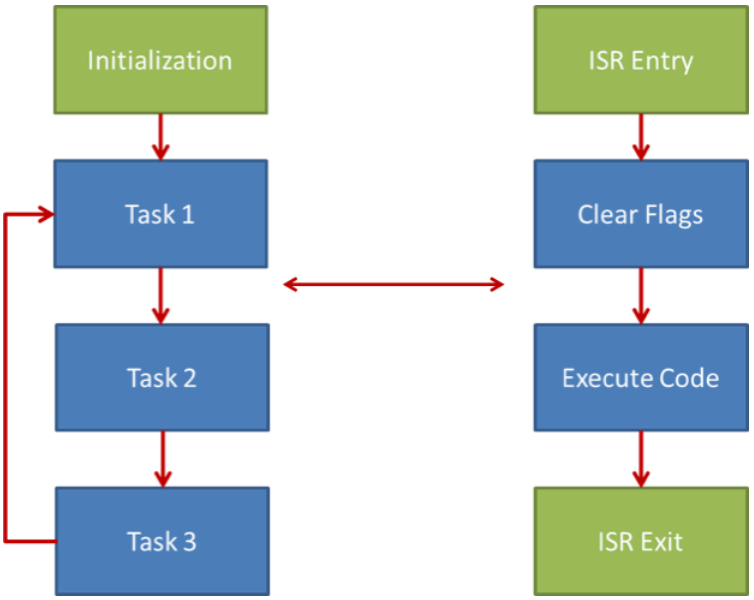


[2] Temporizadores em microcontroladores são periféricos de *hardware* que contam pulsos de *clock*. Eles são usados para medir intervalos de tempo, gerar atrasos ou criar eventos em intervalos regulares, como na modulação por largura de pulso (PWM) ou na execução de tarefas em um sistema operacional em tempo real.

[3] Escalonamento é a função do sistema operacional responsável por gerenciar a execução de múltiplas tarefas concorrentes. Ele determina qual processo ou *thread* terá acesso à CPU em um dado momento e por quanto tempo, com o objetivo de otimizar a utilização dos recursos do sistema e atingir os objetivos de desempenho predefinidos.

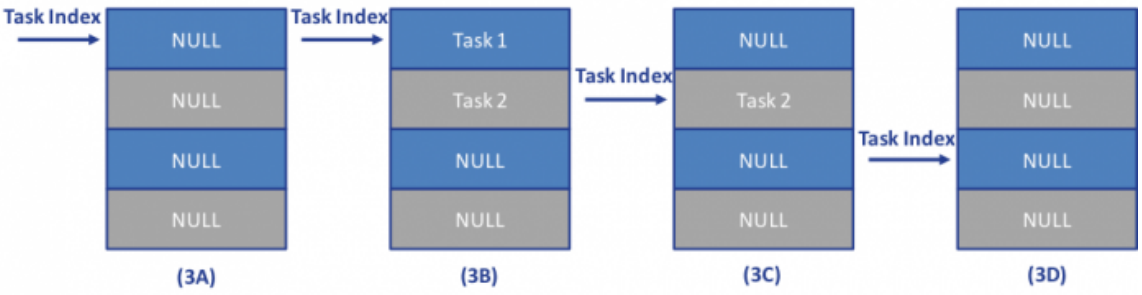
Fonte: Embedded Related.

- ***Round-robin* com interrupções**: O *superloop* (laço principal) percorre as tarefas em sequência, mas eventos urgentes são tratados por interrupções. Embora essa abordagem melhore a latência de resposta, não resolve conflitos de acesso nem garante ordem justa entre tarefas.



Fonte: Embedded Related.

- **Escalonamento por fila (*queue scheduling*)**: As tarefas são organizadas em uma estrutura do tipo FIFO (do inglês, *First In, First Out*) e processadas na ordem de chegada, geralmente em resposta a eventos como interrupções ou sinais do sistema. O escalonador[4] é responsável por gerenciar a fila de tarefas, decidindo qual tarefa deve ser executada a seguir. O desenvolvedor normalmente não precisa implementar manualmente o controle da fila. Bibliotecas do sistema oferecem primitivas para inserir, remover e despachar tarefas, automatizando esse fluxo e fornecendo um controle básico sobre a execução. Isso simplifica o desenvolvimento e dá ao sistema uma organização elementar, ainda que limitada.



Fonte: Embedded Related.

[4] O escalonador é o componente central de um RTOS, responsável por decidir qual tarefa será executada em um determinado momento. Sua função é garantir que as tarefas mais críticas tenham acesso à CPU de acordo com sua política de escalonamento (geralmente por prioridade), assegurando o cumprimento dos prazos e a previsibilidade do sistema.

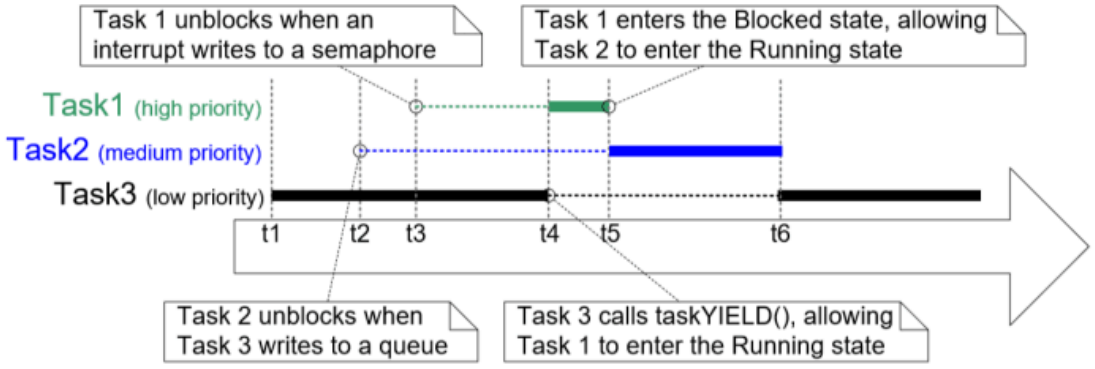
A principal característica desse modelo é a chamada justiça temporal: todas as tarefas são eventualmente atendidas. No entanto, não há diferenciação de criticidade entre elas, o que significa que uma tarefa importante pode acabar bloqueada por tarefas menos relevantes. Essa limitação compromete a previsibilidade temporal, algo essencial em sistemas com requisitos de tempo real. Por conta dessas limitações, sistemas baseados apenas em escalonamento por fila são geralmente considerados precursores dos RTOS modernos, ou ainda como estruturas mínimas de tempo de execução (em inglês, *runtime minimal*). Falta-lhes características fundamentais de um RTOS completo, como suporte a múltiplos níveis de prioridade, escalonamento preemptivo[5] e gerenciamento de *deadlines*.

Frequentemente, tais sistemas são estruturados sobre um *superloop*, que verifica continuamente se há eventos ou tarefas pendentes na fila. A cada iteração, o sistema consome a fila na ordem de chegada, despachando as rotinas correspondentes. Embora rudimentar, essa abordagem permite uma execução ordenada e razoavelmente previsível, desde que as tarefas sejam curtas e o tempo de resposta do sistema não seja crítico.

- **Escalonamento cooperativo**: É uma evolução do modelo baseado em fila, permitindo que o controle da CPU seja transferido de uma tarefa para outra de forma colaborativa. Nesse modelo, cada tarefa permanece no estado *Em Execução* enquanto utiliza a CPU, até voluntariamente "ceder" o processador, seja ao completar sua execução ou ao atingir um ponto lógico no código, onde invoca uma função como `yield()`. Essa ação move a tarefa do estado *Em Execução* para o estado *Pronto*, permitindo que o escalonador selecione a próxima tarefa a ser executada.

  Durante o período em que está no estado *Em Execução*, a tarefa pode realizar dois tipos de operação. Primeiramente, há as **operações não-bloqueantes**, nas quais a tarefa continua executando normalmente e pode, eventualmente, ceder o processador e retornar ao estado *Pronta*, sem necessidade de aguardar eventos externos. Por outro lado, existem as **operações bloqueantes**: nesse caso, a tarefa precisa aguardar algum evento, como a recepção de dados, a liberação de um recurso ou o término de um *delay*. Ao solicitar essa espera, ela sai do estado *Em Execução* e passa para o estado *Bloqueada*, ficando impedida de continuar sua própria execução. Ao mesmo tempo, esse bloqueio ocorre apenas no nível da tarefa. A CPU é liberada para o escalonador, que pode então executar outras tarefas, ou seja, permanece "desbloqueada" no nível do sistema. Assim, a tarefa permanece no estado *Bloqueada*, enquanto outras tarefas são executadas, até que o evento esperado ocorra. Quando isso acontece, a tarefa é reativada e retorna ao estado *Pronta*, aguardando novamente sua vez de ser escalonada para o estado *Em Execução*.

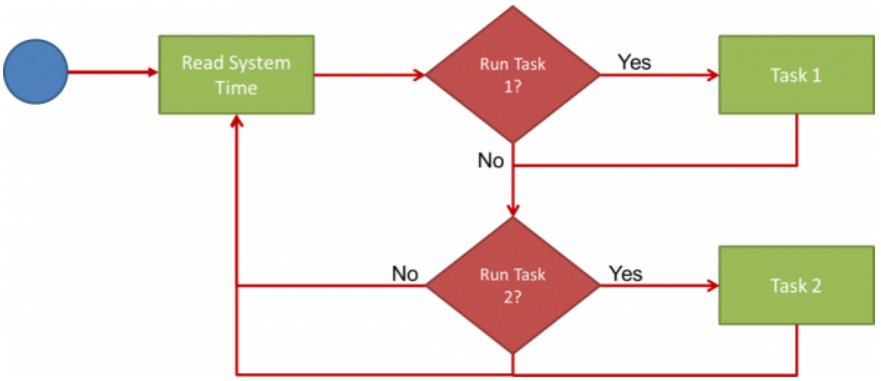
[5] Escalonamento preemptivo é um método onde o sistema operacional (ou RTOS) pode interromper a execução de uma tarefa que está rodando para dar a CPU a outra tarefa. Essa interrupção pode ocorrer por vários motivos, mas o mais comum é, de fato, a chegada de uma nova tarefa com prioridade mais alta.



Execution pattern demonstrating the behavior of the co-operative scheduler

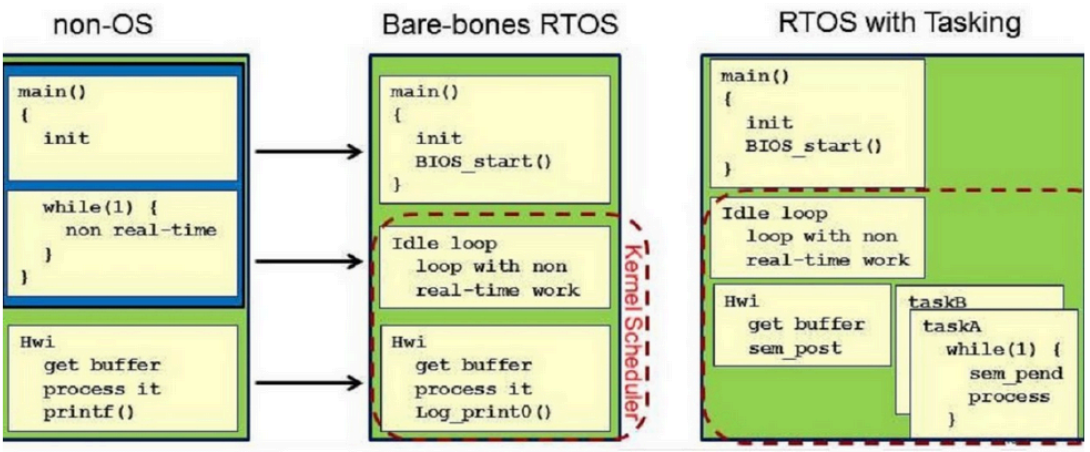
Fonte: Embarcados.

Nesse contexto, funções como `yield()` desempenham um papel fundamental, pois informam ao escalonador que a tarefa está pronta para liberar a CPU e dar lugar a outra, viabilizando o controle cooperativo da execução. Além disso, o escalonador é frequentemente acompanhado de primitivas de sincronização fornecidas por bibliotecas, como semáforos e mutexes, que evitam condições de corrida (*race conditions*) quando múltiplas tarefas acessam recursos compartilhados, como variáveis ou periféricos.



Fonte: Embedded Related.

Apesar dessas vantagens, a principal limitação do escalonamento cooperativo é a **ausência de preempção**. Se uma tarefa não ceder voluntariamente a CPU em tempo adequado, outras tarefas podem ser bloqueadas, comprometendo a responsividade do sistema. Embora isso dificulte o atendimento de requisitos mais rigorosos de tempo real, sistemas cooperativos ainda podem ser utilizados em aplicações com comportamento bem controlado. Nesse modelo, em vez de um *superloop* explícito, o escalonador é inicializado por uma função como `BIOS_start()`, que passa a organizar e executar as tarefas conforme uma política definida.



Fonte: Embedded Computing Design.

> Internamente, o comportamento do escalonador pode ser comparado ao de um *superloop* estruturado, porém encapsulado pela biblioteca do sistema. Assim, em vez de o desenvolvedor implementar manualmente esse laço (em inglês, *loop*), o próprio sistema gerencia a execução das tarefas cooperativas de forma organizada e previsível, ainda que sem o mecanismo de preempção automática característico de sistemas preemptivos.

Em muitos desses métodos, persistem limitações importantes, como a ausência de abstrações mais elaboradas de tarefas, mecanismos padronizados de sincronização de recursos, gerenciamento sistemático de deadlines e dificuldades de escalabilidade. Embora o *hardware* seja essencialmente reativo, ele não fornece, por si só, ferramentas suficientes para lidar com essas complexidades de forma estruturada. Nesse contexto, um RTOS preenche essa lacuna ao construir abstrações de tarefas sobre as capacidades do *hardware*, facilitando o desenvolvimento de sistemas embarcados com comportamento mais previsível. Ao oferecer mecanismos organizados para tratar eventos, concorrência e escalonamento, o RTOS contribui para transformar as capacidades reativas do *hardware* em um ambiente mais controlado e adequado ao desenvolvimento de sistemas multitarefas.

Além disso, um RTOS contribui significativamente para a **confiabilidade** do sistema, uma vez que seus mecanismos de escalonamento determinístico, gerenciamento de tempo real e isolamento de tarefas reduzem o risco de falhas imprevisíveis. O uso de *watchdogs*, gerenciamento robusto de memória e detecção de *deadlocks* contribui para a resiliência do sistema diante de falhas parciais ou condições inesperadas de execução.

A **portabilidade** também é aprimorada com a adoção de um RTOS, pois ele fornece uma camada de abstração entre o *software* da aplicação e o *hardware* subjacente. Isso permite que o mesmo código-fonte seja reutilizado em diferentes plataformas de *hardware*, com mínimas modificações, desde que o RTOS seja suportado nesses ambientes. Essa independência facilita migrações tecnológicas e amplia a longevidade do projeto.

No que diz respeito ao **reaproveitamento de código**, o RTOS promove uma arquitetura modular e bem estruturada, na qual *drivers*, tarefas, serviços de comunicação e bibliotecas podem ser reutilizados em diversos projetos. Isso acelera o desenvolvimento, reduz custos e promove a padronização de soluções dentro de uma organização.

Por fim, a **segurança** é fortalecida através de mecanismos nativos do RTOS, como proteção de memória, controle de acesso entre tarefas e suporte a políticas de execução segura. No entanto, é importante observar que, diferentemente dos GPOS, muitos RTOS adotam uma separação menos

rigorosa entre *user space* e *kernel space*, permitindo que o código da aplicação tenha acesso direto ao *hardware*. Essa abordagem reduz a sobrecarga e melhora o desempenho, que é um aspecto crítico em sistemas de tempo real. Mas, em contrapartida, diminui o grau de isolamento entre tarefas, o que pode comprometer a robustez frente a falhas ou vulnerabilidades.

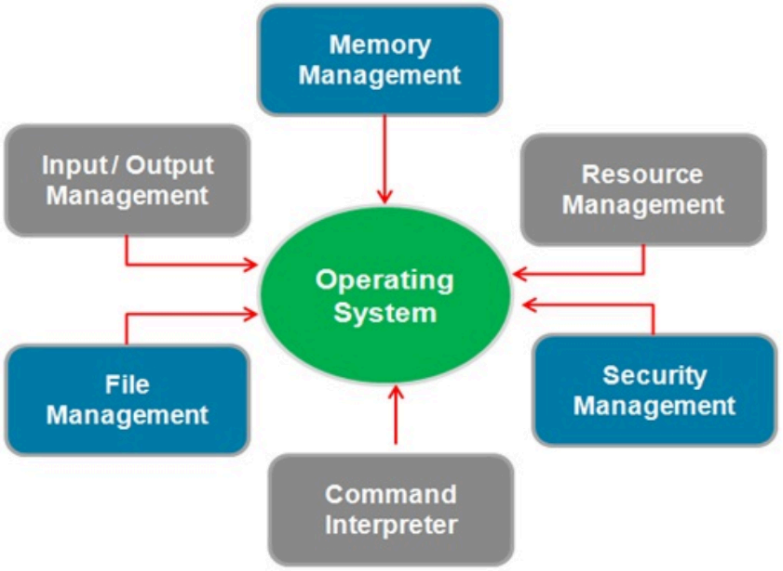
Trata-se, portanto, de um compromisso consciente entre **desempenho e segurança**, em que a transparência e o controle total sobre o sistema são priorizados, ainda que isso exija maior responsabilidade do desenvolvedor quanto à integridade do código e ao acesso seguro aos recursos do sistema. Essa arquitetura mais enxuta é típica de sistemas embarcados críticos, onde a previsibilidade e o tempo de resposta são mais relevantes do que a abstração total de camadas.

Sistema operacional de propósito geral (GPOS)

Antes de entender como funciona um RTOS, vale relembrar os princípios de um GPOS, pois muitos conceitos de ambos são semelhantes, embora utilizados de formas diferentes. Essa comparação é importante para compreender por que um sistema operacional de uso geral nem sempre é adequado para sistemas embarcados. Ao revisar como funciona um GPOS, torna-se mais claro em quais aspectos ele não atende plenamente aos requisitos desses sistemas e por que o RTOS se apresenta como uma solução mais apropriada.

Um GPOS é um tipo de sistema operacional projetado para atender a uma ampla variedade de aplicações e usuários, sem foco em requisitos determinísticos de tempo real. Ele gerencia de forma eficiente os recursos computacionais do sistema, como CPU, memória, dispositivos de entrada/saída e armazenamento, permitindo que múltiplas tarefas sejam executadas de forma concorrente, segura e organizada, com bom desempenho médio. Esse tipo de sistema operacional é amplamente utilizado em ambientes de uso geral, como computadores pessoais, servidores, laptops e dispositivos móveis. Exemplos populares incluem Windows, Linux, macOS e Android.

Um GPOS é composto por diversos módulos e subsistemas, cada um responsável por uma função específica. Entre os principais componentes, destacam-se: o núcleo (kernel), o gerenciamento de memória, o gerenciamento de processos, o sistema de arquivos, o subsistema de entrada/saída, os mecanismos de segurança e a interface com o usuário.



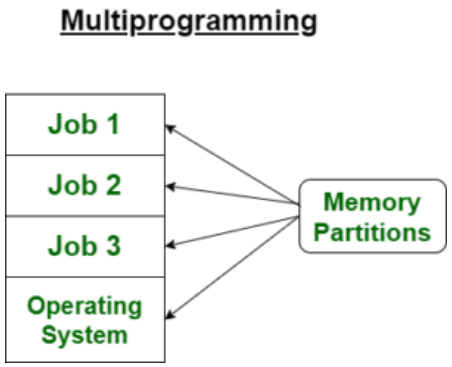
Fonte: Principais componentes de um GPOS (LCS).

O *kernel* (núcleo) é o componente central de um sistema operacional de uso geral, atuando como a interface entre o *hardware* e os programas por meio de uma API (do inglês *Application*

Programming Interface). Ele é responsável por gerenciar processos (criação, execução e escalonamento), memória (alocação e mapeamento), entrada e saída de dados (comunicação com dispositivos) e o sistema de arquivos (organização e acesso aos dados), além de garantir a segurança no uso desses recursos. Essas funcionalidades são acessadas por chamadas de sistema (em inglês, *system calls*), como `fork()`, `exec()`, `mmap()`, `read()` e `open()`. A interação do usuário com o sistema ocorre geralmente por meio de interpretadores de comando, como o `Bash` (comum em sistemas Linux e macOS) e o `PowerShell` (utilizado no Windows), que permitem acessar e controlar os serviços do sistema operacional.

GPOS são projetados para atender a uma vasta gama de aplicações e usuários, priorizando a flexibilidade, a interatividade e o uso eficiente dos recursos computacionais. Para alcançar esses objetivos, sua operação é baseada em conceitos-chave, como:

- **Multiprogramação**: A multiprogramação foi um marco histórico no desenvolvimento dos GPOS e permanece como base para a eficiência em ambientes multitarefa. Ela permite que diversos programas residam simultaneamente na memória. Essa técnica aumenta a utilização da CPU, pois, enquanto um processo[6] espera por um evento de entrada/saída (E/S), outro pode ocupar o processador.



Fonte: Geeksforgeeks.

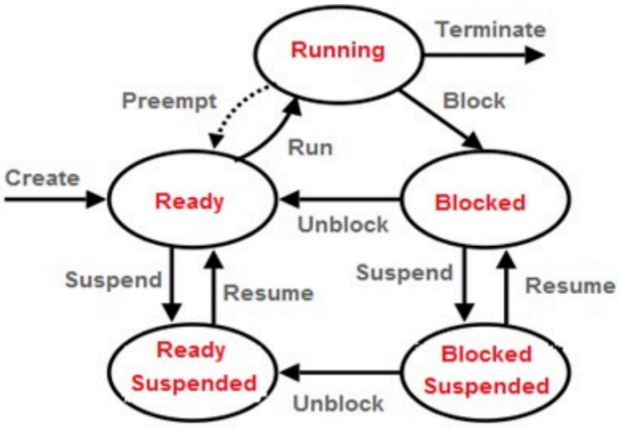
Para gerenciar a execução de processos e otimizar o uso da CPU, são definidos diferentes estados de processo, que organizam e controlam a alocação de recursos:

- Pronto (em inglês, *Ready*): O processo já está carregado na memória e aguarda apenas o escalonador selecionar sua vez de ser executado.
- Em Execução (em inglês, *Running*): O processo está utilizando a CPU naquele exato momento. Em sistemas de núcleo único, apenas um processo pode estar neste estado por vez.
- Bloqueado (em inglês, *Blocked* ou *Waiting*): O processo não pode continuar sua execução e libera a CPU, pois está esperando a conclusão de uma operação de Entrada/Saída (E/S) ou a ocorrência de algum evento externo.

Quando a memória principal fica sobrecarregada, o sistema pode mover processos para o disco (*memória secundária*), suspendendo-os para liberar espaço:

[6] Um processo é uma instância de um programa em execução. Ele é a unidade de trabalho de um sistema operacional, contendo não apenas o código do programa, mas também todos os recursos necessários para sua execução, como o espaço de memória alocado, os registradores da CPU e o estado atual de sua atividade (conhecido como contexto do processo).

- Bloqueado Suspenso (em inglês, *Blocked Suspended*): Um processo que já estava bloqueado é transferido para a memória secundária. Ele continua aguardando o evento de E/S, mas fora da memória principal.
- Pronto Suspenso (em inglês, *Ready Suspended*): Similarmente, um processo que estava pronto para executar é movido para o disco. Ele será trazido de volta à memória principal para o estado Pronto assim que houver espaço e oportunidade.



Fonte: Diferentes estados de um processo (LCS).

- **Escalonamento por Tempo Compartilhado (*Time-Sharing*)**: Em sistemas de propósito geral, o escalonador da CPU adota frequentemente políticas de tempo compartilhado. Cada processo recebe uma fatia de tempo (*time slice*), conhecida também como ***quantum***, para execução, e, ao final desse intervalo, o sistema realiza uma troca de contexto para outro processo. Essa abordagem garante equidade, interatividade e a sensação de responsividade, aspectos centrais em ambientes de uso geral.

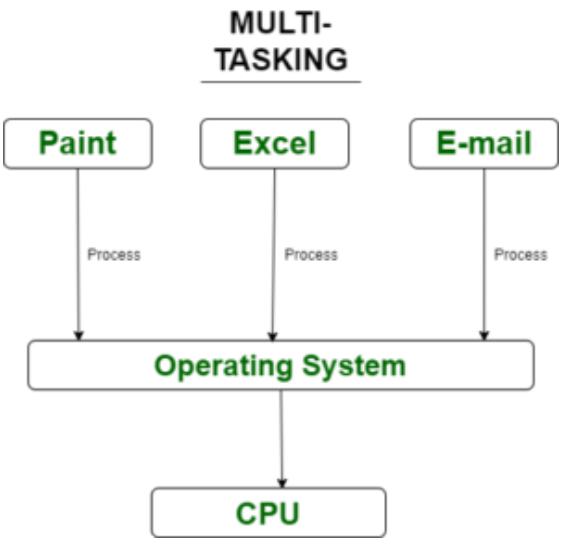
Figura: Execução de 3 processos em *quantuns* (Fonte: Geeksforgeeks).

- **Multitarefa (*Multitasking*)**: Com a adoção do escalonamento por tempo compartilhado, os GPOS passaram a alternar rapidamente entre múltiplos processos, mesmo quando estes não

estão bloqueados. Esse mecanismo, geralmente associado à preempção, permite interromper um processo em execução para dar lugar a outro, criando a impressão de execução simultânea e garantindo maior responsividade. Dessa forma, a multitarefa se caracteriza pela capacidade do sistema de executar e gerenciar múltiplos processos de forma interativa. Em contraste com a multiprogramação, o que a distingue é justamente essa alternância temporal controlada e preemptiva entre processos, e não apenas a sua coexistência na memória.

Nesse contexto, um **processo** é uma instância de um programa em execução, com seu próprio espaço de memória isolado. Já as **tarefas**, frequentemente implementadas como *threads*, são unidades de execução mais leves dentro de um processo, que compartilham memória e recursos. Isso torna a troca entre tarefas significativamente mais rápida do que entre processos distintos, sendo especialmente útil para melhorar a concorrência e a responsividade dentro de uma mesma aplicação.

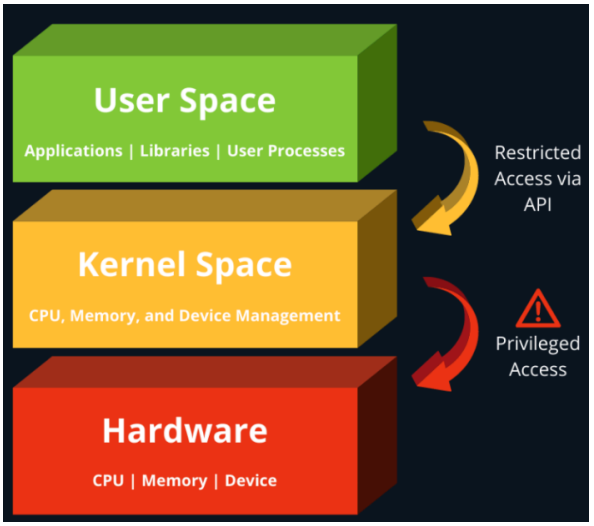
Em sistemas multitarefa modernos, como os baseados em Linux ou Windows, o escalonador gerencia a execução dos processos por meio de suas tarefas (*threads*), movendo-as entre estados como pronto, em execução e bloqueado, de forma a otimizar o uso da CPU e manter a responsividade do sistema. Na prática, isso permite que o usuário utilize múltiplos aplicativos simultaneamente, como editores de texto, planilhas, navegadores ou programas como Paint e Excel, enquanto o sistema alterna rapidamente entre eles. Mesmo quando apenas um aplicativo está em primeiro plano, os demais continuam executando em segundo plano. Essa fluidez é possível porque, em muitos sistemas, a interface gráfica é gerenciada por um sistema de janelas dedicado, que executa de forma concorrente com os demais aplicativos e permite ao usuário alternar entre eles de forma transparente. Dessa forma, a capacidade de executar e alternar rapidamente entre múltiplas tarefas cria a sensação de simultaneidade, garantindo uma experiência de uso contínua, interativa e eficiente.



Fonte: Geeksforgeeks.

- **Espaço de Usuário e Espaço do *Kernel***: A arquitetura dos GPOS costuma separar claramente o espaço do usuário, onde rodam as aplicações, e o espaço do *kernel*, reservado ao núcleo do sistema operacional. Essa divisão aumenta a segurança e a estabilidade: uma

falha em um programa de usuário não compromete, em princípio, o funcionamento do sistema como um todo.



Fonte: Lakeside.

- **Preempção**[7] **e Escalonamento por Prioridade**[8]: A preempção é o mecanismo que garante responsividade e equidade entre múltiplas tarefas. Isso significa que o sistema pode interromper uma tarefa em execução a qualquer momento, forçando a troca de contexto para que outras tarefas também possam acessar a CPU. Isso melhora a experiência do usuário e permite que tarefas interativas, como mover o *mouse* ou abrir uma janela, recebam atenção rapidamente. Costuma estar associada a escalonadores com prioridades dinâmicas, que ajustam automaticamente a prioridade de execução das tarefas com base em critérios como tempo de espera, tipo de tarefa (interativa ou em segundo plano) e uso recente da CPU. Esse ajuste dinâmico tem como objetivo evitar problemas como a inanição[9].

- **Semáforos e Sincronização**: Sistemas multitarefa precisam de mecanismos de sincronização para evitar condições de corrida[10] e inconsistências no acesso a recursos compartilhados, como veremos mais adiante. Os semáforos são um dos mecanismos mais clássicos, permitindo coordenação entre processos e garantindo exclusão mútua[11]. Esse conceito é igualmente relevante em RTOS, embora com adaptações específicas às exigências de tempo real.

---

[7] Preempção é a capacidade de um sistema operacional de interromper a execução de uma tarefa que está em andamento para permitir que outra tarefa, geralmente de maior prioridade, comece a ser executada imediatamente.
[8] Escalonamento por prioridade é uma técnica de gerenciamento de tarefas em um sistema operacional. Sua função é determinar qual tarefa deve ser executada pela CPU a seguir, sempre dando preferência àquela com a prioridade mais alta.
[9] Inanição (*starvation*) é a condição em que um processo ou *thread* é repetidamente impedido de obter acesso a um recurso necessário (como a CPU) porque outras tarefas de prioridade maior estão sempre prontas para rodar. Como resultado, a tarefa de baixa prioridade pode nunca ser executada, ficando "presa" indefinidamente.
[10] Condição de corrida (*race condition*) é uma situação em que o resultado de um sistema depende da sequência imprevisível de eventos, geralmente quando múltiplas tarefas acessam um recurso compartilhado simultaneamente, levando a um resultado incorreto ou não determinístico.
[11] Exclusão mútua (*mutual exclusion*) é uma propriedade de controle de concorrência que garante que apenas uma *thread* ou processo por vez tenha acesso a um recurso compartilhado, como uma variável ou um arquivo, prevenindo o surgimento de condições de corrida.

- **Multiprocessamento**: Com a popularização de processadores *multicore*, os GPOS transcendem a simples ilusão de multitarefa. Eles são capazes de distribuir processos e *threads* em diferentes núcleos de CPU, permitindo a multitarefa real (também conhecida como paralelismo). Nesse cenário, várias tarefas são executadas verdadeiramente ao mesmo tempo, cada uma em seu próprio núcleo. Isso amplia significativamente a capacidade de processamento e aumenta o desempenho em cargas de trabalho intensivas.

- **Memória Virtual**: Outro pilar dos GPOS é a memória virtual. Esse recurso permite que cada processo tenha a ilusão de dispor de um espaço de endereçamento contínuo e isolado, independentemente da memória física disponível. Além de ampliar a segurança, isso possibilita que aplicações utilizem mais memória do que a fisicamente instalada, através de técnicas como paginação e *swap*. No entanto, a abstração de memória virtual, embora poderosa, introduz latências que são incompatíveis com a previsibilidade exigida por sistemas de tempo real.

- **Portabilidade**: Um GPOS é normalmente desenvolvido com foco em portabilidade: a capacidade de ser executado em diferentes arquiteturas e plataformas de *hardware*, com alterações mínimas no código-fonte. Esse princípio contribui para sua ampla difusão em servidores, *desktops*, *notebooks* e dispositivos móveis.

Características distintivas de GPOS e RTOS

Nos **GPOS**, como Linux ou Windows, a multiprogramação e a multitarefa são utilizadas principalmente para permitir que múltiplos aplicativos independentes sejam executados concorrentemente, compartilhando os recursos do sistema de forma eficiente. O objetivo principal é proporcionar boa interatividade, alto desempenho médio e uma experiência satisfatória ao usuário, mesmo quando diversos programas competem simultaneamente pelo uso da CPU, da memória e dos dispositivos de entrada e saída. Já em um **RTOS**, a multitarefa desempenha um papel distinto. Em vez de suportar vários programas independentes, ela é empregada para organizar e coordenar as diferentes tarefas que compõem uma única aplicação embarcada. Cada tarefa é normalmente responsável por uma função específica, como a aquisição de dados de sensores, o processamento de sinais, o controle de atuadores, a comunicação com outros dispositivos ou a supervisão do sistema. Nesse contexto, o principal objetivo não é maximizar a utilização do processador nem oferecer interatividade ao usuário, mas garantir que cada tarefa seja executada de forma previsível, respeitando os requisitos temporais ((*deadlines*) previamente estabelecidos.

Devido às restrições típicas dos sistemas embarcados, como memória limitada, menor capacidade de processamento e consumo de energia reduzido, os RTOS são projetados como camadas de *software* enxutas. Em geral, fornecem apenas os mecanismos essenciais de escalonamento, sincronização, comunicação entre tarefas e gerenciamento de temporização. Diferentemente dos GPOS, não incluem, por padrão, subsistemas complexos, serviços genéricos de alto nível ou mecanismos elaborados de gerenciamento de recursos. Essa simplicidade reduz o *overhead* do sistema operacional, melhora a previsibilidade temporal e transfere ao desenvolvedor maior responsabilidade pelo gerenciamento dos recursos da aplicação.

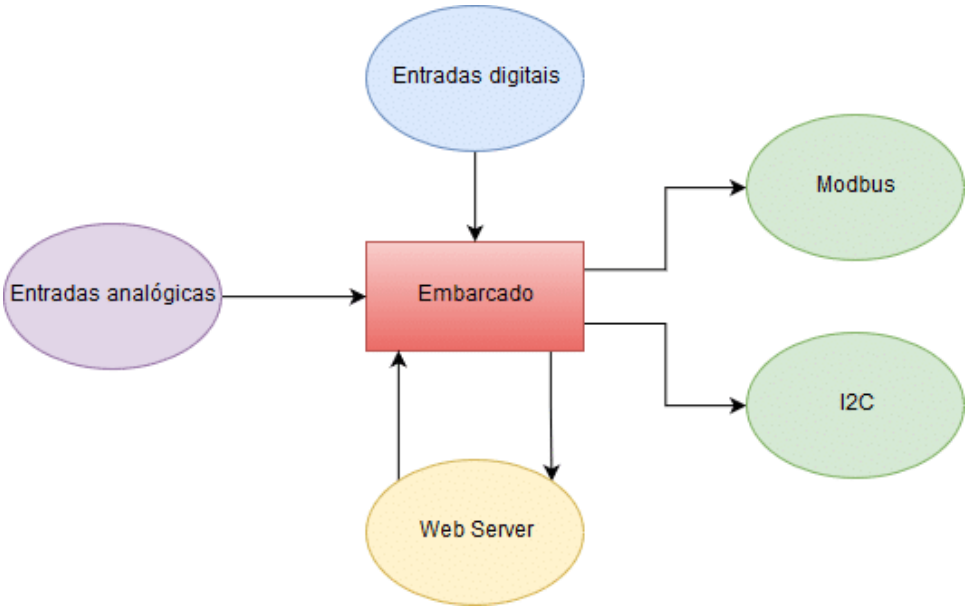


Figura: Embarcados.

Embora tanto os GPOS quanto os RTOS utilizem interrupções para tratar eventos assíncronos, seus objetivos são distintos. Nos GPOS, as interrupções fazem parte de um ambiente voltado ao compartilhamento eficiente dos recursos do sistema, visando maximizar o desempenho global e a responsividade percebida pelo usuário. Já nos RTOS, as interrupções constituem parte integrante do mecanismo de escalonamento, permitindo que eventos externos sejam tratados rapidamente e que tarefas críticas assumam imediatamente a execução sempre que necessário. Essa estreita cooperação entre o *hardware* de interrupções e o escalonador é um dos fatores que possibilitam o comportamento determinístico característico dos sistemas de tempo real.

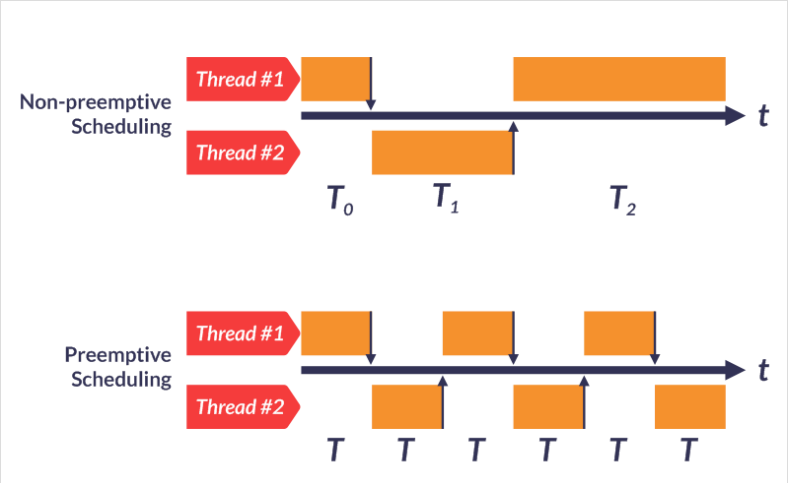
Enquanto um GPOS utiliza algoritmos de escalonamento com prioridades dinâmicas, ajustadas continuamente para favorecer a interatividade e evitar a inanição (*starvation*), um RTOS normalmente emprega **escalonamento preemptivo por prioridades fixas**, definidas pelo desenvolvedor durante o projeto da aplicação. Nesse modelo, cada tarefa possui uma prioridade associada à sua importância funcional e aos seus requisitos temporais. Sempre que uma tarefa de prioridade mais elevada se torna apta à execução, o escalonador interrompe a tarefa em execução e transfere imediatamente o processador para a nova tarefa.

Para que esse mecanismo funcione, o escalonador precisa ser informado sempre que ocorrer uma situação capaz de alterar a ordem de execução das tarefas. Essas situações são denominadas eventos de escalonamento (*scheduling events*). Um **evento de escalonamento** pode ocorrer, por exemplo, quando uma interrupção periódica de um temporizador de *hardware* é gerada, quando uma interrupção de um periférico desperta uma tarefa de maior prioridade, quando a tarefa em execução entra voluntariamente em espera ou quando a liberação de um semáforo, fila, mutex ou outro mecanismo de sincronização torna uma tarefa de maior prioridade apta à execução. A cada evento de escalonamento, o núcleo do RTOS reavalia quais tarefas estão prontas e seleciona a de maior prioridade para execução.

Em muitas arquiteturas de microcontroladores, esse mecanismo é implementado por meio de temporizadores integrados ao *hardware*, sincronizados com o relógio do processador e capazes de

gerar interrupções periódicas de alta precisão. Nas arquiteturas ARM Cortex-M, por exemplo, essa função é normalmente executada pelo temporizador SysTick. Outras arquiteturas, como RP2040, AVR e ESP32,, oferecem temporizadores equivalentes. Essas interrupções periódicas fornecem uma referência temporal para diversas funções do RTOS, como a atualização dos atrasos (*delays*), o gerenciamento de temporizadores de *software* e a verificação periódica das tarefas prontas para execução. Entretanto, o escalonador não depende exclusivamente dessas interrupções periódicas, podendo também ser acionado imediatamente por interrupções de periféricos ou por chamadas ao próprio núcleo do sistema operacional.

Como consequência desse modelo de escalonamento, a decisão sobre qual tarefa utilizará a CPU é sempre responsabilidade do escalonador. Assim, uma tarefa de alta prioridade pode interromper imediatamente a execução de uma de menor prioridade, tão logo esta se torne apta à execução. Quando há várias tarefas prontas com a mesma prioridade, o mecanismo de **fatiamento de tempo** (*time slicing*) permite que elas compartilhem o processador de forma justa, alternando a execução a cada quantum de tempo. Esse mecanismo evita a inanição entre tarefas de mesma prioridade, embora tarefas de menor prioridade ainda possam permanecer por longos períodos sem serem executadas, caso existam continuamente tarefas mais prioritárias prontas para execução.

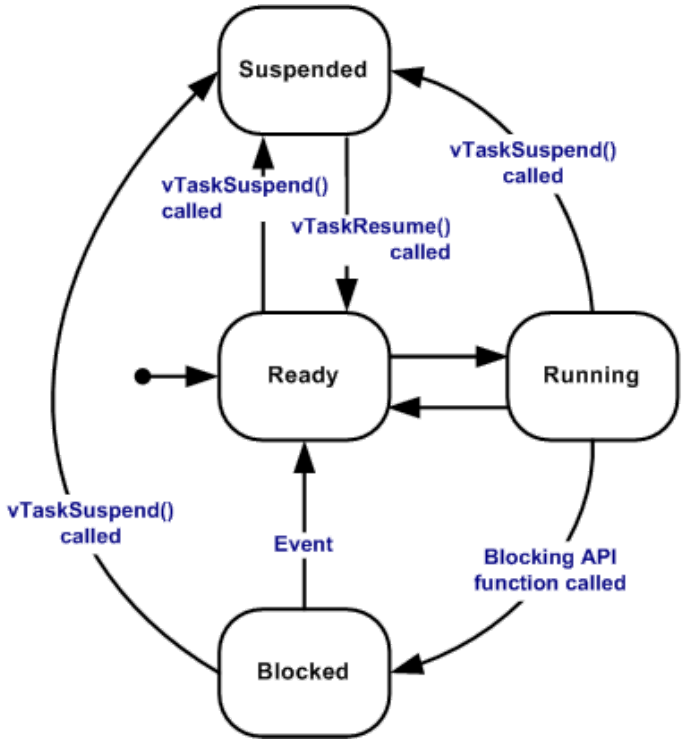


Fonte: Predictable Designs.

Para compreender o funcionamento do escalonador, é importante conhecer os **estados** em que uma tarefa pode se encontrar durante sua execução. O FreeRTOS, amplamente utilizado em sistemas embarcados, define quatro estados principais:

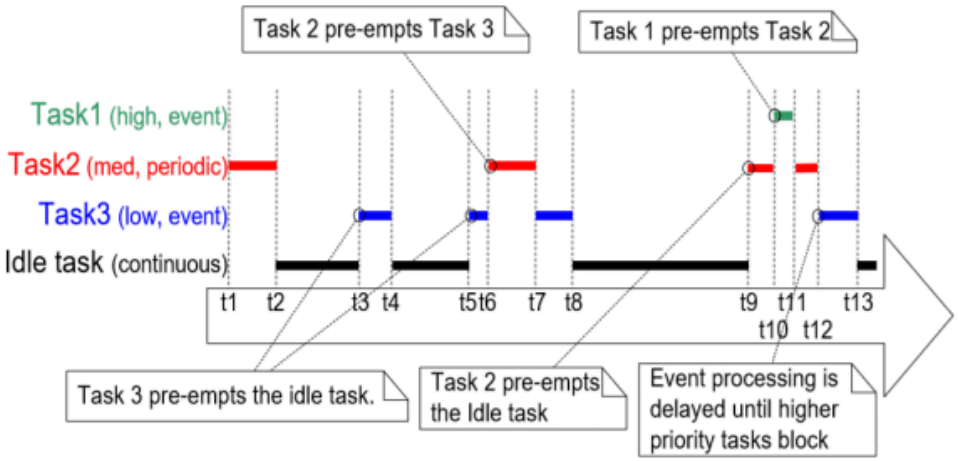
- **Em execução** (em inglês, ***Running***): a tarefa está utilizando a CPU. Em processadores de núcleo único, apenas uma tarefa pode estar neste estado por vez.
- **Pronta (em inglês, Ready):** a tarefa está apta a ser executada, mas aguarda porque outra de mesma prioridade (ou de prioridade mais alta) está ocupando o processador.
- **Bloqueada** (em inglês, ***Blocked***): a tarefa se encontra temporariamente inativa, aguardando a ocorrência de um evento, como o término de um atraso programado (delay) ou a chegada de dados em uma fila. Nesse estado, a tarefa não consome tempo de CPU e, em geral, possui um *timeout* que a retorna ao estado `Pronta` quando ele expira.

- **Suspensa** (em inglês, ***Suspended***): semelhante ao estado `Bloqueada`, mas sem *timeout*. A entrada ou saída deste estado ocorre apenas por meio de chamadas explícitas à API do FreeRTOS, o que confere maior controle ao programador.



Fonte: FreeRTOS.

Além dessas tarefas, o RTOS mantém permanentemente uma tarefa especial denominada *Idle Task*. Essa tarefa possui a menor prioridade do sistema e é executada sempre que nenhuma outra tarefa estiver pronta. Além de evitar que o processador permaneça ocioso, ela realiza diversas funções internas do sistema operacional, como a liberação de recursos de tarefas encerradas e outras atividades de manutenção do núcleo. Na figura a seguir, observa-se uma tarefa *idle*,



Execution pattern highlighting task prioritization and pre-emption in a hypothetical application in which each task has been assigned a unique priority

Fonte: Embarcados.

Em sistemas de tempo real, não basta que as tarefas produzam resultados corretos; esses resultados também precisam ser entregues dentro dos prazos estabelecidos (*deadlines*). Para garantir esse comportamento, é necessário conhecer o pior tempo de execução (*Worst-Case Execution Time* – WCET) de cada tarefa, assegurando que ele permaneça compatível com os requisitos temporais da aplicação. O não cumprimento desses prazos pode comprometer o funcionamento ou até mesmo a segurança do sistema, especialmente em aplicações críticas, como controle industrial, automação, dispositivos médicos e sistemas automotivos.

Um problema clássico associado ao escalonamento por prioridades é a *inversão de prioridade*. Esse fenômeno ocorre quando uma tarefa de baixa prioridade mantém um recurso compartilhado necessário para uma tarefa de alta prioridade. Se, nesse intervalo, uma tarefa de prioridade intermediária passar a ser executada, ela poderá impedir que a tarefa de baixa prioridade conclua sua execução e libere o recurso, prolongando indevidamente o bloqueio da tarefa mais importante. Embora esse problema possa ocorrer tanto em GPOS quanto em RTOS, ele acarreta consequências muito mais severas em sistemas de tempo real, devido à necessidade de garantir tempos máximos de resposta. Por esse motivo, muitos RTOS, como FreeRTOS, Zephyr RTOS, VxWorks e QNX, implementam protocolos de herança de prioridade (*priority inheritance*), nos quais a tarefa de menor prioridade recebe temporariamente a prioridade da tarefa bloqueada até que o recurso compartilhado seja liberado.

Outra diferença importante entre GPOS e RTOS está na arquitetura de gerenciamento de memória. GPOS utilizam amplamente mecanismos como memória virtual, paginação sob demanda e proteção entre processos, proporcionando maior segurança, isolamento e flexibilidade. Entretanto, essas funcionalidades introduzem latências variáveis, incompatíveis com os requisitos de previsibilidade temporal de muitas aplicações embarcadas. Em consequência, a maioria dos RTOS evita tais mecanismos ou os implementa de forma bastante simplificada, privilegiando tempos de resposta determinísticos. Alguns RTOS oferecem alocação dinâmica de memória por meio de regiões de *heap*; porém, seu uso exige cuidados adicionais, pois a fragmentação e os tempos de alocação variáveis podem comprometer o determinismo da aplicação.

Os dois tipos de sistemas operacionais também diferem quanto aos modos de execução. Nos GPOS, há uma separação rigorosa entre o modo de usuário, no qual as aplicações são executadas, e o modo de *kernel*, reservado às funções internas do sistema operacional. Essa separação aumenta a segurança e a robustez, porém introduz overhead durante as mudanças de modo. Em muitos RTOS, especialmente os destinados a microcontroladores, essa separação é reduzida ou inexistente, **permitindo que as tarefas executem diretamente com privilégios elevados**. Embora isso diminua a proteção contra falhas de software, reduz significativamente a latência e favoreça a previsibilidade temporal.

É importante destacar que o determinismo temporal não é um conceito absoluto, mas sim graduado. Dessa forma, costuma-se classificar os sistemas de tempo real em diferentes categorias:

- ***Soft Real-Time* (SRTOS):** garante tempos de resposta previsíveis na maioria dos casos, mas tolera eventuais atrasos. É adequado para aplicações em que o não cumprimento de um prazo causa apenas degradação da qualidade do serviço, e não falha crítica. Exemplos incluem *streaming* de áudio e vídeo, ou sistemas de telefonia.

- ***Firm Real-Time*:** admite que alguns prazos possam ser perdidos, mas, uma vez ultrapassado o *deadline*, o resultado associado perde totalmente a utilidade. A confiabilidade temporal é mais rígida do que no SRTOS, mas ainda não crítica. Um exemplo típico é o sistema de processamento de dados financeiros ou de transações.
- ***Hard Real-Time* (HRTOS)**: representa o nível de determinismo mais rigoroso. Aqui, nenhum prazo pode ser violado sem risco de falha catastrófica no sistema. Aplicações de controle de voo, equipamentos médicos críticos e sistemas de segurança automotiva dependem desse nível de previsibilidade.

Essa classificação evidencia que "**tempo real**" não significa necessariamente executar rapidamente, mas sim executar de forma **previsível dentro de prazos definidos**. Assim, o nível de rigor temporal deve ser escolhido de acordo com as exigências da aplicação, equilibrando desempenho, custo e complexidade.

Por fim, embora os RTOS sejam amplamente empregados em sistemas embarcados, nem toda aplicação de tempo real necessita de um sistema operacional. Em aplicações simples, é possível adotar uma abordagem *bare-metal*, na qual o *firmware* é executado diretamente sobre o *hardware*, utilizando um *superloop* ou um escalonador simples desenvolvido pelo próprio projetista. Essa solução oferece baixo *overhead* e pode apresentar excelente previsibilidade em aplicações pouco complexas. Entretanto, à medida que aumenta o número de funcionalidades concorrentes, cresce também a dificuldade de organizar o código, sincronizar atividades e manter a escalabilidade do sistema. Nesses casos, o uso de um RTOS torna-se vantajoso ao fornecer mecanismos padronizados para escalonamento, sincronização, temporização e gerenciamento estruturado das tarefas da aplicação.

GPOS vs. RTOS: DIFERENÇAS DE IMPLEMENTAÇÃO

Multitarefa
- GPOS: Execução concorrente de múltiplos processos; foco em responsividade para o usuário.
- RTOS: Execução concorrente de múltiplas tarefas; foco em previsibilidade temporal.

Multiprogramação
- GPOS: Vários programas permanecem na memória para melhor uso da CPU.
- RTOS: Pouco relevante; geralmente há poucos processos ativos, focados em controle direto do *hardware*.

Espaço de Usuário vs. *Kernel*
- GPOS: Separação rígida para segurança e isolamento.
- RTOS: Muitas vezes inexistente: tarefas de usuários e *kernel* compartilham o mesmo espaço de memória para reduzir latência.

Escalonamento
- GPOS: Baseado em tempo compartilhado (*time-sharing*), busca justiça e interatividade.
- RTOS: Baseado em prioridades fixas ou dinâmicas, garantindo que a tarefa mais crítica seja executada sem atrasos.

Preempção
- GPOS: Interrupções podem suspender tarefas, mas não garantem prazos rígidos.
- RTOS: Preempção determinística e controlada, essencial para atender *deadlines* de tempo real.

Prioridade de Tarefas
- GPOS: Usada, mas nem sempre seguida estritamente (processos de baixa prioridade ainda rodam em algum momento).
- RTOS: Estritamente respeitada: tarefas críticas têm prioridade absoluta.

Semáforos e Sincronização
- GPOS: Usados para coordenação e exclusão mútua entre processos.
- RTOS: Idem, mas com implementações otimizadas para evitar bloqueios longos e garantir tempos previsíveis.

Multiprocessamento
- GPOS: Processos distribuídos entre múltiplos núcleos; estados de processo bem definidos (pronto, executando, bloqueado, etc.).
- RTOS: Pode suportar múltiplos núcleos, mas muitas vezes simplificado; não há a mesma complexidade de estados de processo (tarefas estão prontas ou em execução).

Memória Virtual
- GPOS: Presente, oferece isolamento e abstração de memória, mas com latência imprevisível (*swap*/paginação).
- RTOS: Normalmente ausente, pois a latência inviabiliza *deadlines* rígidos. Tarefas acessam memória física diretamente.

Portabilidade
- GPOS: Projetado para rodar em diversas arquiteturas e plataformas.
- RTOS: Focado em plataformas específicas de sistemas embarcados, priorizando leveza e determinismo.

Concorrência em sistemas multitarefa (ênfase em RTOS)

Em sistemas multitarefa, a concorrência ocorre quando duas ou mais entidades de execução podem acessar simultaneamente os mesmos recursos do sistema, como memória, periféricos, filas de comunicação ou variáveis globais. Quando um recurso só pode ser utilizado por uma entidade de cada vez, ele caracteriza uma **região crítica** (*critical section*), isto é, um trecho de código ou um conjunto de dados cujo acesso deve ser exclusivo para preservar a consistência do sistema.

Em sistemas operacionais de propósito geral (GPOS), os problemas de concorrência surgem principalmente entre as tarefas ou processos gerenciados pelo escalonador. Nos RTOS, entretanto, esse cenário é mais abrangente. Como o escalonador é implementado sobre os mecanismos de interrupção e exceção do microcontrolador, a execução do sistema alterna continuamente entre dois contextos distintos: o das tarefas, executadas sob controle do escalonador, e o das rotinas de serviço de interrupção (*Interrupt Service Routines* – ISRs), executadas em resposta a eventos de *hardware*.

Embora as ISRs não sejam tarefas, elas podem acessar os mesmos recursos compartilhados utilizados pelas tarefas da aplicação. Consequentemente, **o desenvolvedor deve tratar não apenas a concorrência entre tarefas, mas também a concorrência entre tarefas e interrupções**. Em microcontroladores como os da família STM32, por exemplo, isso significa que tanto as tarefas gerenciadas pelo RTOS quanto as ISRs associadas ao controlador de interrupções NVIC constituem entidades concorrentes cujo acesso aos recursos compartilhados deve ser cuidadosamente

coordenado. Além disso, a atribuição das prioridades das interrupções passa a fazer parte do projeto do sistema. Em arquiteturas como ARM Cortex-M, **as interrupções de *hardware* possuem níveis de prioridade independentes das prioridades das tarefas do RTOS**. Enquanto o escalonador decide qual tarefa pronta deve executar, o controlador de interrupções (NVIC) decide qual ISR pode interromper a execução do processador.

Uma prática bastante comum consiste em dividir as interrupções em dois grupos. O primeiro grupo reúne interrupções extremamente críticas, que exigem a menor latência possível e executam apenas o tratamento mínimo necessário. Essas ISRs normalmente **não utilizam a API do RTOS**, justamente para evitar qualquer dependência do *kernel*. O segundo grupo reúne interrupções que precisam comunicar eventos às tarefas da aplicação, utilizando filas, notificações ou semáforos. Essas ISRs executam em prioridades compatíveis com o RTOS e utilizam exclusivamente as funções da API com o sufixo `FromISR`. No FreeRTOS, essa separação é implementada por meio da constante `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY` (ou, internamente, `configMAX_SYSCALL_INTERRUPT_PRIORITY`, dependendo da forma de configuração adotada). Essa constante define a prioridade máxima de interrupção autorizada a chamar funções do *kernel.* Interrupções configuradas com prioridade mais alta (maior urgência) não podem utilizar a API do FreeRTOS, enquanto interrupções de prioridade igual ou inferior podem utilizar as funções `FromISR` com segurança. Essa restrição permite que o *kernel* proteja suas estruturas internas mascarando apenas as interrupções que podem acessar seus serviços, preservando a baixa latência das interrupções mais críticas.

Como consequência, uma prática comum é reservar as prioridades mais altas do NVIC para eventos críticos de *hardware* — por exemplo, mecanismos de proteção, controle de potência ou aquisição de sinais de alta velocidade — e utilizar prioridades um pouco menores para periféricos como UART, SPI, I²C, USB ou *Ethernet* quando suas ISRs precisam acordar tarefas do RTOS. Dessa forma, obtém-se um equilíbrio entre baixa latência para eventos críticos e integração segura entre interrupções e o escalonador.

Assim, em um RTOS, garantir o comportamento determinístico do sistema não depende apenas do escalonamento das tarefas, mas também da coordenação correta entre tarefas e interrupções, pois todos esses eventos, embora desempenhem funções distintas, compartilham o mesmo mecanismo de arbitragem do processador. Em outras palavras, **é necessário executar a ação correta, no momento adequado e com os recursos apropriados,** assegurando que todos os acessos às regiões críticas ocorram de forma segura, previsível e consistente. Para isso, o sistema deve empregar mecanismos que controlem o acesso aos recursos compartilhados e sincronizem as diferentes entidades de execução.

Na ausência desses mecanismos, múltiplas tarefas e rotinas de serviço podem interferir entre si ao acessar regiões críticas, resultando em comportamento imprevisível e comprometendo a estabilidade do sistema. Entre as principais consequências, destacam-se:

- **Condição de Corrida** (em inglês, ***Race Condition***): Ocorre quando o resultado depende da ordem de execução das tarefas. Se duas tarefas acessam e modificam o mesmo recurso sem proteção, o valor final pode variar a cada execução. Exemplo: Duas tarefas incrementam um contador global. Ambas leem o valor inicial 10 e escrevem 11 no contador, incrementando-o

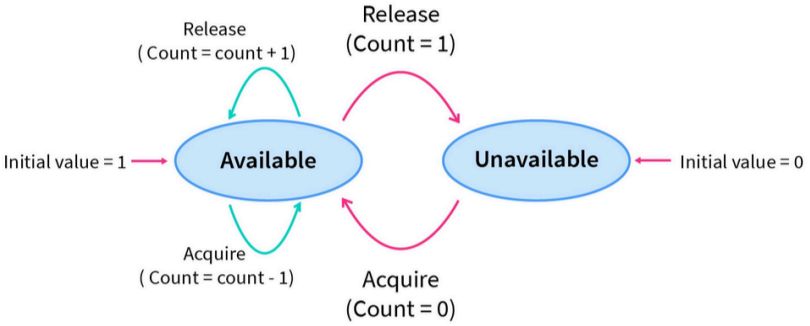
em 1 unidade. O resultado final seria 11, mas o correto seria 12. Esse tipo de falha leva à corrupção de dados.

- **Bloqueio Mútuo** (em inglês, ***Deadlock***): Acontece quando duas ou mais tarefas aguardam indefinidamente por recursos que estão sendo mantidos pela outra, criando um ciclo de espera sem saída. Exemplo: a Tarefa A utiliza o Recurso 1 e a Tarefa B utiliza o Recurso 2. Tarefa A tenta obter o Recurso 2 (bloqueado) e tarefa B tenta obter o Recurso 1 (bloqueado). Nenhuma das duas progredirá.
- **Inanição** (em inglês, ***Starvation***): Surge quando uma tarefa é repetidamente preterida e nunca recebe os recursos necessários para ser executada. Exemplo: Em um escalonamento por prioridade fixa, uma tarefa de baixa prioridade pode nunca rodar se tarefas de alta prioridade monopolizarem os recursos de que ela precisa.
- **Desincronização**: É a inconsistência entre tarefas ou dados quando não há sincronização adequada. Exemplo: Uma tarefa escreve os valores de um sensor. Outra tenta exibir esses valores simultaneamente. O resultado pode ser uma leitura parcial ou dados incorretos.

Embora a concorrência seja um desafio em qualquer sistema, sua relevância se torna especialmente crítica em um RTOS, devido às diferenças na filosofia de projeto. Em GPOS, como em Linux ou Windows, falhas na sincronização podem levar a travamentos, corrupção de dados ou comportamento inesperado, geralmente limitadas ao domínio da aplicação. Os desenvolvedores utilizam mecanismos de controle para preservar a integridade das informações, mas nem sempre há garantias temporais rigorosas. Em contraste, em um RTOS, falhas na sincronização de recursos podem comprometer diretamente o comportamento temporal do sistema. Por exemplo, uma tarefa de alta prioridade pode ser bloqueada por uma de menor prioridade (como em situações de inversão de prioridade), o que pode levar à perda de *deadlines*. Em HRTOS, isso não é apenas uma falha funcional, mas uma falha sistêmica que pode comprometer a segurança ou o controle do equipamento, como no não acionamento de um *airbag* ou em falhas em sistemas de controle de voo.

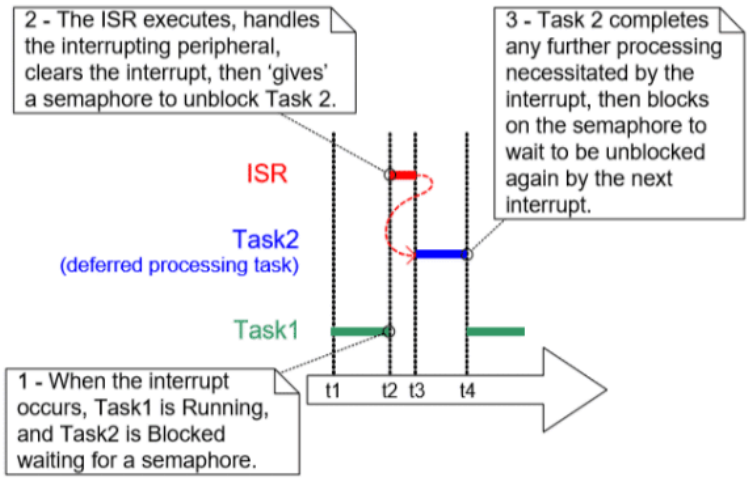
Para evitar esses problemas, os sistemas operacionais, incluindo RTOS, oferecem uma variedade de mecanismos de sincronização, incluindo

- **Semáforos** (em inglês, ***Semaphores)***: Um **semáforo de contagem** é um mecanismo de sincronização usado para controlar o acesso concorrente a N recursos idênticos. Seu valor representa o número de recursos disponíveis; quando for maior que zero, uma tarefa pode prosseguir e decrementar o contador. Quando chega a zero, as novas tarefas devem aguardar até que algum recurso seja liberado. O **semáforo binário** é um caso particular em que $N = 1$, funcionando como um bloqueio simples: 0 indica indisponível, 1 indica disponível. Nesse caso, apenas uma tarefa por vez pode acessar o recurso protegido.



Fonte: Scaler.

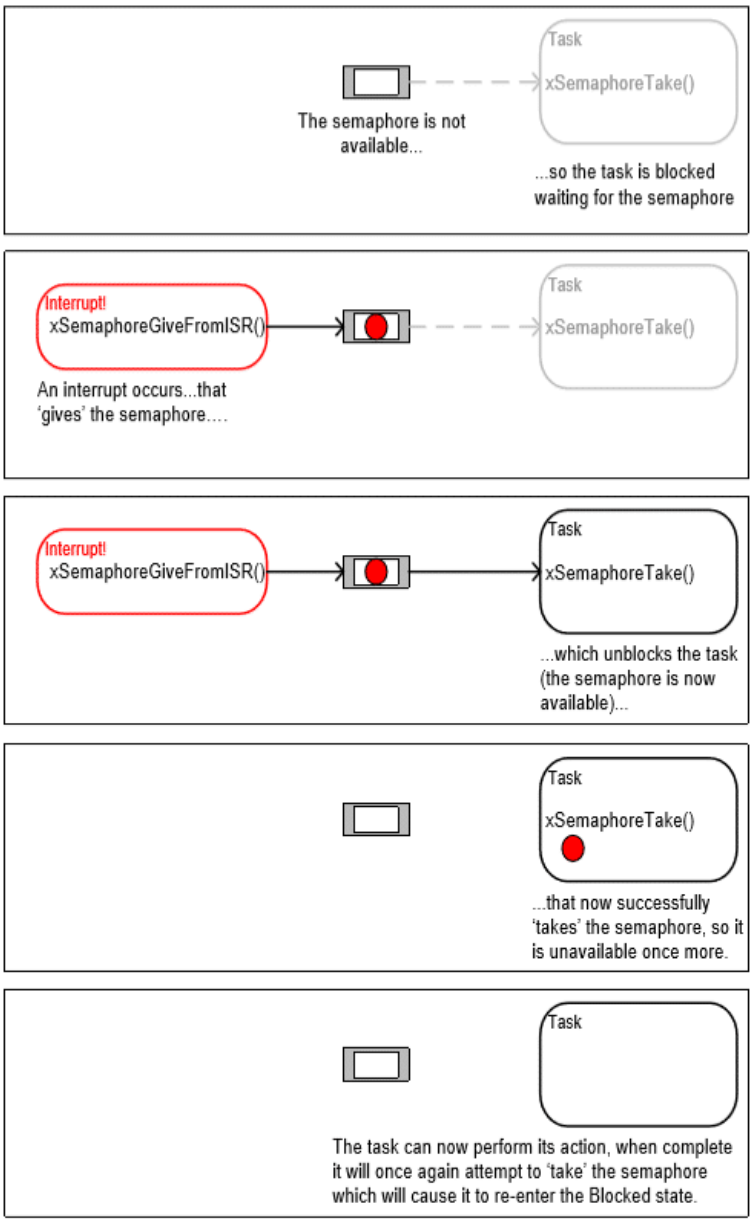
A figura a seguir ilustra como um **semáforo binário** pode atuar como um mecanismo de sincronização entre a ISR (no domínio do *hardware*), responsável por **sinalizar** a ocorrência de uma interrupção, e a tarefa *Task2* (no domínio do *software*), encarregada do processamento adiado (*deferred processing*), que é executado somente após a liberação do semáforo pela ISR.



Using a binary semaphore to implement deferred interrupt processing

Fonte: Embarcados.

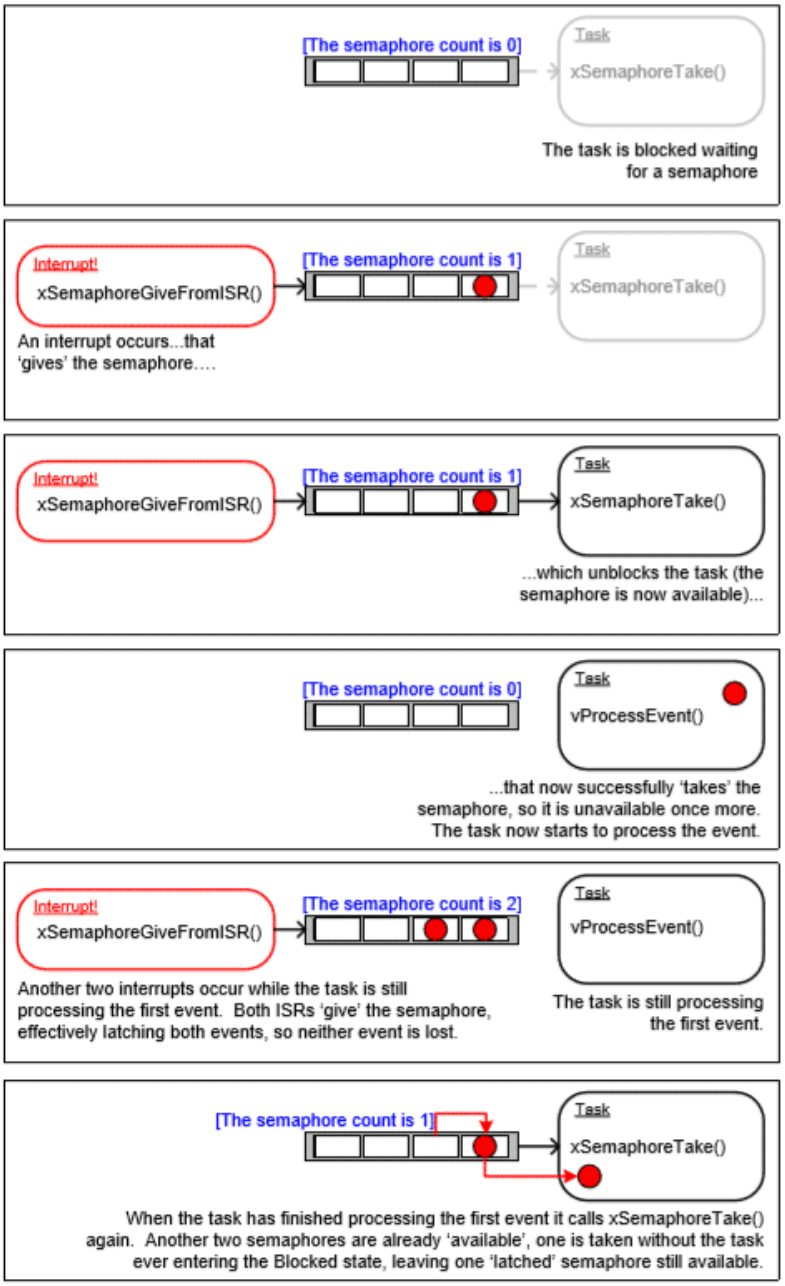
A figura a seguir apresenta uma implementação desse elo de sincronização usando as funções do FreeRTOS. Inicialmente, a tarefa `Task` fica no estado `Bloqueada`, aguardando que o semáforo associado seja liberado. Quando ocorre um evento de interrupção, o fluxo de execução desvia-se para a rotina de serviço de interrupção (ISR). Essa rotina deve ser minimalista, executando apenas a ação essencial: "liberar" (em inglês, *give*) o semáforo da `Task`. O escalonador detecta que o semáforo foi liberado e coloca a `Task` no estado `Pronta`. Se não existir nenhuma outra tarefa de prioridade mais alta também `Pronta`, o escalonador imediatamente passa a `Task` para o estado `Em Execução`, permitindo que ela processe as suas instruções. Após concluir a execução, a `Task` volta a tentar "obter" (em inglês, *take*) o semáforo. Caso nenhum novo evento de interrupção tenha ocorrido, o semáforo não estará disponível, e a `Task` retorna ao estado `Bloqueada`, aguardando novamente por um semáforo..



Using a binary semaphore to synchronize a task with an interrupt

Fonte: Embarcados.

Esse mecanismo garante que a tarefa utilize a CPU apenas quando houver trabalho a ser executado, contribuindo para a eficiência do sistema. Quando a tarefa aguarda um semáforo, ela permanece `bloqueada` e não consome processamento; ao ser liberado o semáforo, ela transita para o estado `Pronta`. A partir desse momento, o escalonador decide quando essa tarefa será executada, organizando a ordem de execução conforme as prioridades e o estado das demais tarefas.



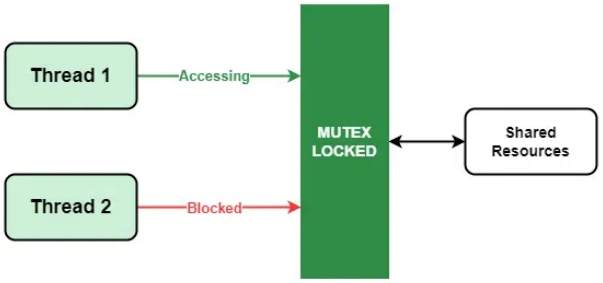
Using a counting semaphore to 'count' events

Fonte: Embarcados.

Quando uma aplicação precisa lidar com múltiplos eventos de interrupção de uma mesma fonte, um **semáforo de contagem** é a solução ideal. Nesse cenário, a rotina de serviço de interrupção (ISR) atua como um contador, liberando o semáforo a cada novo evento. Isso permite que a mesma tarefa receba a notificação de múltiplos eventos pendentes, já que o valor do semáforo é incrementado a cada "liberação" (*give*) da ISR. Dessa forma, a tarefa responsável pelo processamento, `Task` na figura anterior, pode obter o semáforo múltiplas vezes em seu ciclo de execução (uma vez para cada evento acumulado). Cabe ao escalonador gerenciar a execução da tarefa conforme seu estado e prioridade, garantindo que ela continue ativa até que todos os eventos pendentes sejam tratados.

É importante observar que, embora o semáforo contabilize os eventos, a tarefa deve ser projetada para lidar corretamente com os recursos associados a cada ocorrência. Se diferentes eventos demandarem o uso de recursos distintos (como *buffers*, dispositivos ou estruturas de dados), a tarefa precisa gerenciar explicitamente esses recursos, garantindo consistência e evitando conflitos. Essa abordagem aumenta significativamente a probabilidade de que nenhum evento de interrupção seja perdido.

- ***Mutexes*** (do inglês ***Mutual Exclusion***): Um *mutex* é um mecanismo de sincronização utilizado para garantir exclusão mútua no acesso a recursos compartilhados, como periféricos, variáveis globais ou estruturas de dados. Embora, à primeira vista, possa parecer semelhante a um semáforo binário, o *mutex* possui características adicionais fundamentais. A principal delas é o conceito de **propriedade** (em inglês, *ownership*): somente a tarefa que adquiriu o *mutex* pode liberá-lo, o que garante consistência no uso do recurso protegido.



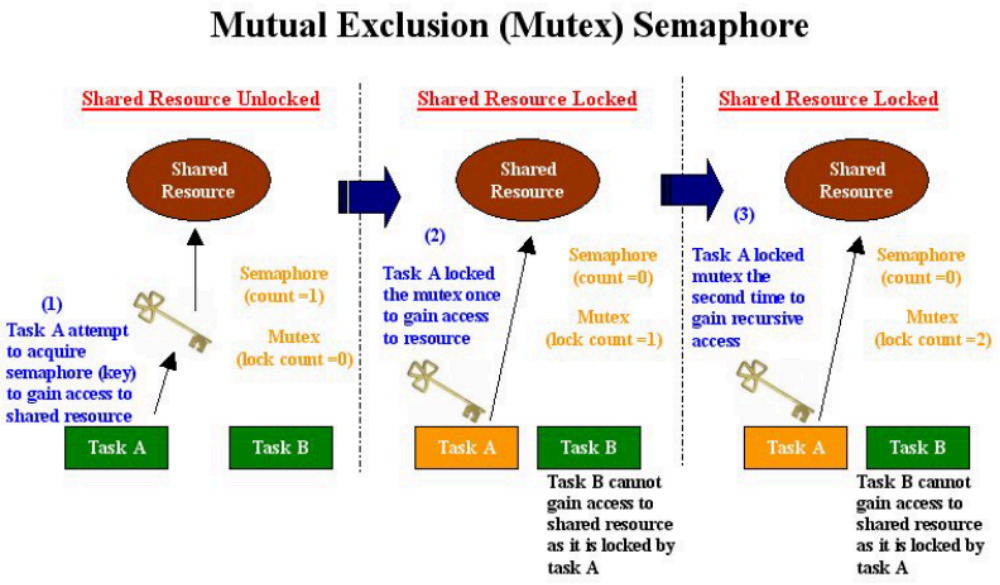
Fonte: Geeksforgeeks.

Um semáforo binário, por outro lado, não possui dono e pode ser liberado por qualquer contexto de execução, sendo mais adequado para **sinalização de eventos** do que para proteção de recursos. Essa limitação pode levar a problemas em cenários de exclusão mútua. Por exemplo, considere duas tarefas que acessam uma interface UART compartilhada. Se um semáforo binário for utilizado para controlar o acesso, nada impede que uma tarefa diferente daquela que iniciou a transmissão libere o semáforo, potencialmente interrompendo uma operação em andamento e corrompendo os dados transmitidos. Com um *mutex*, isso não ocorre, pois apenas a tarefa que adquiriu o recurso pode liberá-lo, garantindo integridade na operação. Outro cenário típico envolve o acesso a uma estrutura de dados compartilhada, como um *buffer* circular. Se uma tarefa for interrompida enquanto manipula o *buffer* e outra tarefa assumir o controle indevidamente (por uso incorreto de semáforo), pode haver inconsistência ou corrupção de dados. O uso de *mutex* evita esse tipo de problema, pois mantém a exclusividade até que a tarefa finalize completamente sua região crítica.

Dessa forma, enquanto semáforos binários são ideais para **notificar eventos** (por exemplo, uma ISR sinalizando que um dado está pronto), os *mutexes* são a escolha correta para **proteger recursos compartilhados**, garantindo acesso controlado, previsível e seguro em sistemas concorrentes.

Em sistemas com RTOS, o uso de *mutexes* também introduz a possibilidade de acesso recursivo. Isso significa que, se uma tarefa já possui o *mutex*, ela pode obter o mesmo *mutex*

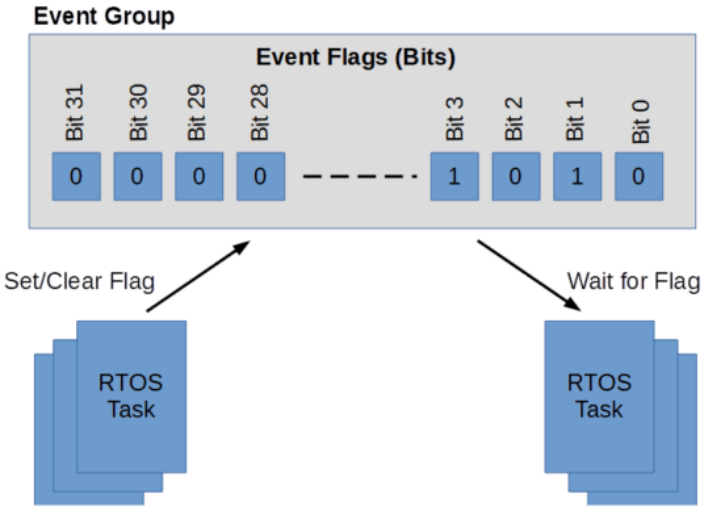
novamente (por exemplo, ao chamar uma função que também tenta acessar a mesma região crítica protegida). Nesse caso, o valor interno do *mutex* é incrementado a cada nova obtenção pela mesma tarefa e a tarefa continua com acesso exclusivo à região crítica. O *mutex* só será efetivamente liberado quando a mesma tarefa fizer o mesmo número de liberações que o número de vezes que o obteve. Enquanto isso, outras tarefas continuam bloqueadas, mesmo que a região crítica pareça "livre". Esse comportamento evita *deadlocks* acidentais em chamadas encadeadas dentro da mesma tarefa, mas exige cuidado: se a tarefa esquecer de liberar todas as vezes que obteve o mutex, o recurso permanecerá bloqueado.



Fonte: Renesas.

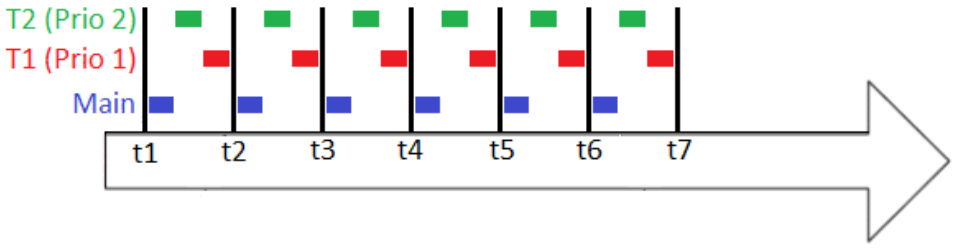
Além disso, o uso de *mutexes* pode levar ao problema de inversão de prioridade, em que uma tarefa de alta prioridade fica bloqueada esperando por um *mutex* retido por uma tarefa de baixa prioridade. Para lidar com isso, os RTOS geralmente implementam **protocolos de herança de prioridade**, onde a tarefa de baixa prioridade tem sua prioridade temporariamente elevada enquanto estiver segurando o mutex, evitando bloqueios indevidos.

- **Grupo de Eventos** (em inglês, ***Event Group***): Também conhecido como **Sinais** (em inglês, ***Signals***), é um mecanismo de sincronização leve em sistemas com RTOS, projetado para lidar com múltiplas fontes de eventos que podem desbloquear tarefas. Em vez de usar vários semáforos, ele concentra diferentes condições em um único registrador de estado (comumente de 8, 16 ou 32 *bits*). Cada *bit* desse registrador atua como uma *flag* de evento, representando, por exemplo, a chegada de dados pela UART, a leitura de um sensor ou o acionamento de um botão. Tanto tarefas `Task` quanto rotinas de interrupção (`ISRs`) podem setar (ativar) ou resetar (limpar) essas *flags*, indicando que determinado evento ocorreu ou foi tratado. Já uma ou mais tarefas podem ficar bloqueadas aguardando a liberação quando qualquer *flag* de interesse é ativada (no modo OR) ou quando todas as *flags* de interesse estão ativadas simultaneamente (no modo AND).



Fonte: Open4Tech.

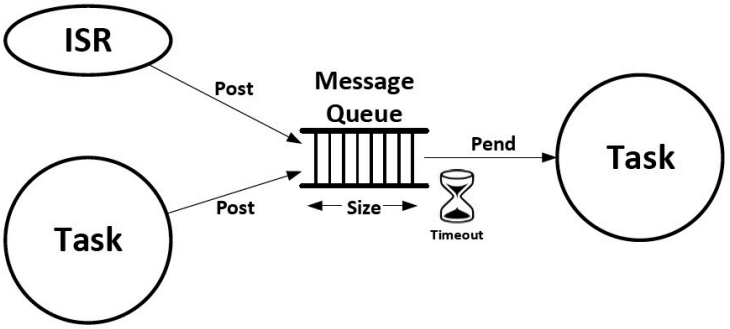
Essa flexibilidade torna os grupos de eventos mais poderosos do que os semáforos tradicionais, que normalmente representam apenas uma condição de cada vez. A figura a seguir ilustra um cenário em que duas tarefas, T1 e T2, podem ser desbloqueadas simultaneamente ao aguardarem as mesmas condições, passando ambas para o estado `PRONTA`. Nesse caso, o escalonador define qual tarefa será executada primeiro: se possuírem prioridades diferentes, a de maior prioridade será escolhida imediatamente; se tiverem a mesma prioridade, o processador alternará entre elas em cada quantum de tempo, aplicando o escalonamento *round-robin*. Além disso, as *flags* podem ser limpas automaticamente após o desbloqueio da tarefa, evitando que o mesmo evento seja processado repetidamente.



Fonte: Embarcados.

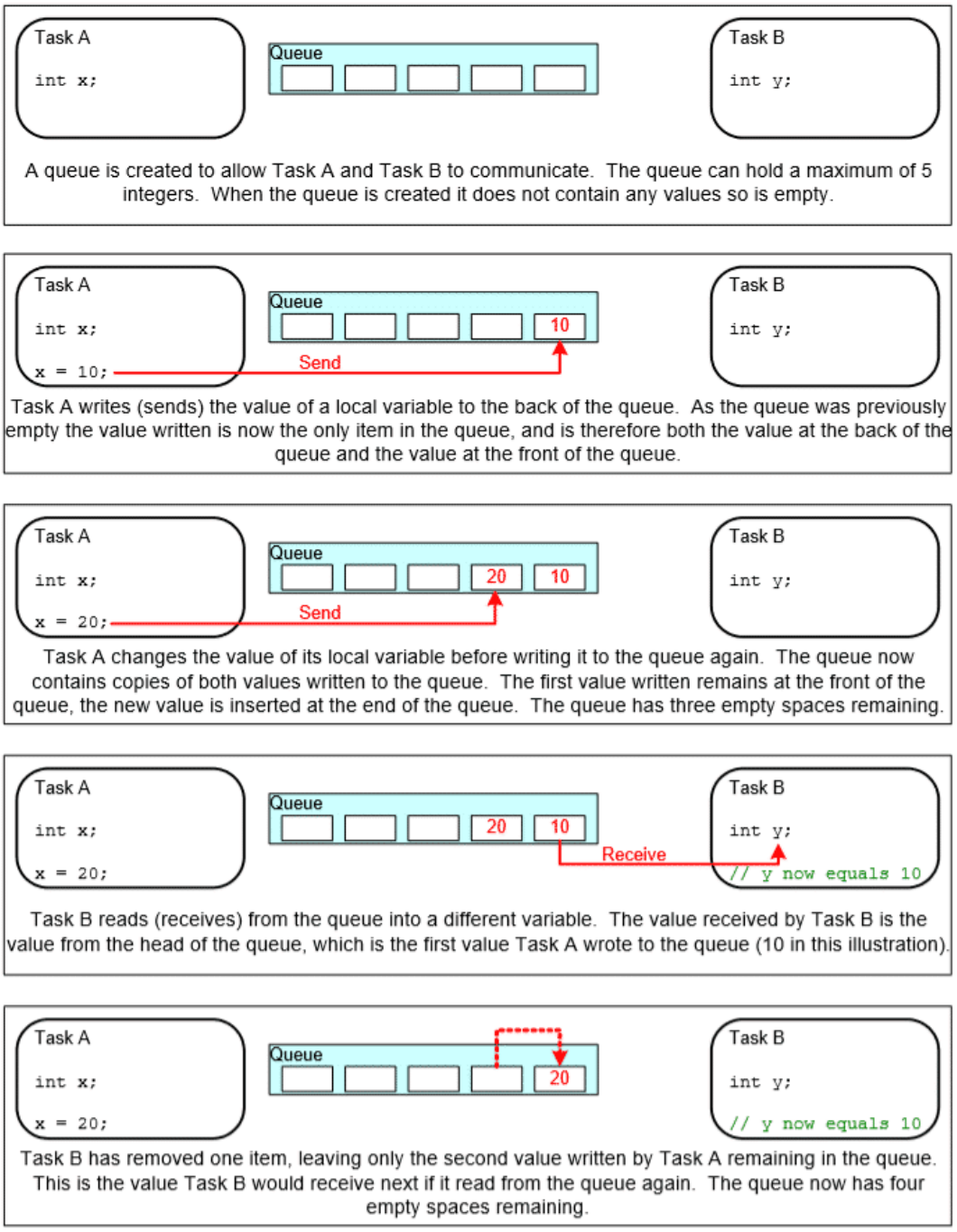
É importante observar que esse mecanismo não acumula múltiplas ocorrências do mesmo evento: se um *bit* for setado várias vezes antes da tarefa ser desbloqueada, ele será tratado apenas uma vez. Outro cuidado necessário é com a concorrência, pois várias tarefas podem acessar e modificar o mesmo grupo de eventos simultaneamente.

- **Filas de Mensagem** (em inglês, ***Message Queues***): As filas de mensagem são um mecanismo de comunicação assíncrona que permite a troca de dados entre tarefas. Em vez de acessarem um recurso compartilhado diretamente, as tarefas enviam e recebem mensagens através da fila. Isso ajuda a desacoplar as tarefas, reduzindo a necessidade de sincronização e o risco de *deadlock* e inversão de prioridade.



Fonte: Embedded Computing Design.

A fila no FreeRTOS é um objeto que funciona, na forma mais comum, como um *buffer* FIFO (do inglês *First In, First Out*), ou seja, o primeiro dado inserido é o primeiro a ser retirado. No entanto, existem funções que permitem inserir dados no início da fila, caso haja necessidade de prioridade inversa. As filas são criadas com tamanho fixo, tanto em número de *slots* (quantas mensagens elas podem armazenar) quanto em tamanho de cada *slot* (quantos *bytes* cada mensagem pode ocupar). Por exemplo, uma fila criada com 5 *slots* de 2 *bytes* cada pode armazenar até 5 mensagens de até 2 *bytes*. Esses parâmetros impactam diretamente como e com que frequência a fila pode ser usada, sendo importante garantir que cada mensagem não ultrapasse o tamanho definido por *slot* e que o comprimento da fila seja suficiente para evitar **sobrecarga** (fila cheia). O seguinte exemplo ilustra a transferência de dados da tarefa `Task A` à tarefa `Task B`.



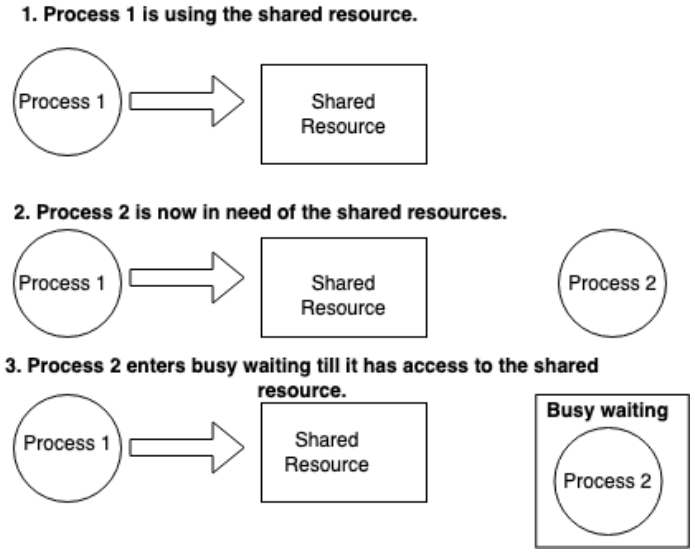
An example sequence of writes to, and reads from a queue

Fonte: Embarcados.

Embora as filas sejam frequentemente utilizadas para transferência de dados entre tarefas, como no exemplo em que a `Task A` envia dados à `Task B`, elas também podem ser usadas como mecanismo de sincronização, de forma similar aos semáforos. Quando uma tarefa tenta ler uma fila vazia ou escrever em uma fila cheia, ela pode ser automaticamente bloqueada (dependendo dos parâmetros passados para a função de envio ou recebimento). Nesse caso, a tarefa bloqueada é removida da execução e colocada em uma fila de tarefas bloqueadas e o escalonador do FreeRTOS é então ativado para escolher outra tarefa pronta para executar, respeitando as prioridades. Da mesma forma, quando uma nova mensagem é enviada para a fila (liberando um *slot* para leitura), ou um *slot* vazio é liberado para escrita (por uma tarefa que consumiu a mensagem), então o escalonador pode desbloquear a tarefa

que estava esperando e colocá-la de volta no estado *Pronta*, para que seja executada assim que possível (com base nas prioridades).

- **Trava de giro** (em inglês, ***Spinlocks***): Um *spinlock* é um tipo de bloqueio que faz a tarefa "girar" em um *loop* (consumindo CPU) enquanto espera por um recurso. Em sistemas com múltiplos núcleos, isso pode ser mais rápido do que a alternância de contexto do escalonador para bloqueios de curtíssima duração. No entanto, em sistemas de núcleo único, é um método ineficiente, pois desperdiça ciclos de processamento.



Fonte: Geeksforgeeks.

Temporizador de *software*

Embora os microcontroladores modernos disponham de diversos temporizadores de *hardware*, esses periféricos são recursos limitados e frequentemente já estão dedicados a funções críticas da aplicação, como geração de sinais PWM para controle de motores, temporização de protocolos de comunicação, captura de pulsos ou aquisição de sinais. Para oferecer maior flexibilidade e permitir o gerenciamento de um grande número de eventos temporizados, os RTOS disponibilizam **temporizadores de *software*** (*software timers*), que permitem agendar a execução de funções após um intervalo de tempo ou de forma periódica, sem a necessidade de reservar um temporizador de *hardware* para cada evento.

Os temporizadores de *software* são implementados sobre um **temporizador-base de *hardware***, configurado para gerar interrupções periódicas. No RTOS, esse temporizador-base constitui o ***tick* do sistema** (*system tick*) e, nos núcleos ARM Cortex-M, é normalmente implementado pela exceção **SysTick**. Por convenção, sua frequência é derivada diretamente do *clock* do núcleo (*core clock*), proporcionando sincronismo com a execução das tarefas, simplicidade no cálculo de atrasos (*delays*) e previsibilidade temporal no processo de escalonamento.

Cada interrupção do *system tick* representa um evento de escalonamento que atualiza a base de tempo do sistema e permite ao RTOS verificar quais temporizadores de *software* expiraram e quais tarefas devem ser despertadas. Dessa forma, o mecanismo de temporização reutiliza a mesma infraestrutura de interrupções apresentada nas seções anteriores: as interrupções periódicas do *system tick* coexistem, no mesmo sistema de interrupções e exceções do processador, com as

interrupções dos periféricos da aplicação e com as demais exceções empregadas pelo escalonador, sendo todas arbitradas de acordo com as prioridades configuradas pelo desenvolvedor.

Sobre essa base de tempo, o RTOS implementa o serviço de temporizadores de *software*, cujo papel é programar a execução futura de uma função (*callback*, ou função de retorno) após um determinado intervalo de tempo. Essa execução pode ser

- única (em inglês, *one-shot*), disparando apenas uma vez, ou
- periódica, repetindo-se a cada intervalo definido.

Assim, o programador pode "agendar" operações de forma simples, sem precisar manipular diretamente registradores de *hardware* nem escrever rotinas de serviço de interrupção complexas. Além dessa praticidade, os temporizadores de *software* trazem uma série de benefícios adicionais.

Eles oferecem escalabilidade, já que é possível criar tantos temporizadores quantos forem necessários, ficando o limite apenas na quantidade de memória disponível, e não no número físico de periféricos do microcontrolador. Fornecem também um nível importante de abstração, pois liberam o desenvolvedor do controle direto de *hardware* ao disponibilizar uma API clara e padronizada para iniciar, parar ou reiniciar contagens de tempo.

Outro ponto forte é a integração com o ambiente de tarefas do RTOS. Como o disparo dos temporizadores ocorre no contexto do próprio sistema operacional, suas funções associadas (*callbacks*) são executadas sem os conflitos típicos de prioridades que podem surgir em rotinas de interrupção. Isso torna o uso de temporizadores mais seguro e previsível em aplicações complexas.

Por fim, destaca-se a flexibilidade desse recurso, que facilita a implementação de eventos periódicos ou atrasados sem sobrecarregar a lógica principal do programa. Dessa forma, o desenvolvedor pode se concentrar no comportamento da aplicação, deixando ao RTOS a responsabilidade pela temporização de eventos.

Contextos de execução em RTOS

Em um sistema baseado em RTOS, o código pode ser executado em diferentes **contextos de execução**, cada um com regras próprias sobre como acessar os serviços do sistema operacional. Compreender esses contextos é essencial para utilizar corretamente a API do RTOS e evitar falhas que normalmente só aparecem sob carga elevada ou durante longos períodos de operação.

Embora existam pequenas diferenças entre RTOSs, três contextos são encontrados na maioria deles:

- contexto de tarefa;
- contexto de rotina de tratamento de interrupção (ISR, *Interrupt Service Routine*);
- contexto de callback de temporizador de software.

Cada um desses contextos possui características próprias e exige o uso adequado das funções disponibilizadas pelo RTOS.

O primeiro contexto é o **contexto de tarefa**. Nesse caso, o código é executado como parte de uma *thread* gerenciada pelo escalonador do RTOS. Todas as tarefas competem pela CPU de acordo com suas prioridades e estados (pronta, bloqueada, suspensa etc.), cabendo ao escalonador, componente do *kernel* do RTOS, decidir qual delas será executada a cada instante. É importante observar que,

embora a execução da tarefa seja controlada pelo RTOS, esse controle é implementado utilizando os próprios mecanismos de interrupção do processador. Em sistemas preemptivos, por exemplo, interrupções periódicas do temporizador do sistema (*tick interrupt*) e interrupções de *software,* como `PendSV` em processadores ARM Cortex-M, são utilizadas pelo escalonador para realizar a troca de contexto entre tarefas. Portanto, tanto tarefas quanto ISRs utilizam a infraestrutura de interrupções do *hardware*, mas representam contextos de execução distintos. Neste contexto é permitido utilizar praticamente toda a API do RTOS, incluindo funções que podem bloquear a tarefa enquanto ela aguarda algum evento, como `vTaskDelay()`, `xQueueReceive()` e `xSemaphoreTake()`. Quando uma dessas funções bloqueia a tarefa, o escalonador simplesmente seleciona outra tarefa pronta para executar.

O segundo contexto é o **contexto de interrupção (ISR)**. Nesse caso, o código é executado diretamente em resposta a um evento de *hardware*, como o recebimento de um caractere pela UART, o término de uma conversão ADC ou o disparo de um temporizador periférico. Durante a execução da ISR, o processador interrompe temporariamente a tarefa que estava executando, salva automaticamente seu contexto e transfere imediatamente o controle para a rotina de interrupção. Embora o RTOS utilize interrupções para implementar seu funcionamento interno, uma ISR **não é executada como uma tarefa do RTOS**. Ela possui prioridade definida pelo controlador de interrupções do processador e executa de forma assíncrona em relação ao escalonador. Por essa razão, uma ISR não pode realizar operações que impliquem bloqueio, espera ou mudança imediata de estado de execução como fazem as tarefas. Essa diferença explica por que o FreeRTOS, assim como diversos outros RTOSs, fornece duas versões de muitas funções de sua API: uma destinada ao contexto de tarefa e outra ao contexto de interrupção, identificada pelo sufixo `FromISR`.

As versões convencionais (`xQueueSend()`, `xSemaphoreGive()`, `xTaskNotifyGive()`, etc.) assumem que estão sendo executadas por uma tarefa controlada pelo escalonador. Assim, elas podem bloquear, alterar listas internas de tarefas, provocar reescalonamento imediato e utilizar mecanismos de exclusão mútua apropriados ao contexto de tarefas. Já as versões `FromISR` (`xQueueSendFromISR()`, `xSemaphoreGiveFromISR()`, `vTaskNotifyGiveFromISR()`, etc.) utilizam um protocolo diferente de sincronização. Elas nunca bloqueiam, executam apenas operações seguras durante uma interrupção e apenas registram que uma tarefa de maior prioridade ficou pronta para executar. A efetiva troca de contexto somente ocorrerá após o término da ISR, normalmente por meio da macro `portYIELD_FROM_ISR()`.

Essa separação existe porque os mecanismos usados para proteger as estruturas internas do *kernel* são diferentes em cada contexto. No contexto de tarefa, o RTOS pode suspender o escalonador ou mascarar temporariamente determinadas interrupções enquanto atualiza suas estruturas internas. Já no contexto de interrupção, parte dessas garantias não existe, pois o código está sendo executado em resposta a um evento assíncrono de *hardware*. Assim, as funções `FromISR` utilizam algoritmos específicos que preservam a consistência das estruturas do *kernel* sem depender de operações incompatíveis com uma ISR.

Outra consequência dessa diferença é que **mutexes não possuem variantes `FromISR`**. Um mutex representa a propriedade de um recurso por uma tarefa específica e pode envolver herança de prioridade (*priority inheritance*), mecanismo que depende diretamente do escalonador. Como uma

ISR não é uma tarefa, ela não pode se tornar proprietária de um mutex nem participar desse mecanismo. Por isso, ISRs devem se limitar a sinalizar eventos para tarefas utilizando filas, semáforos binários, notificações de tarefas ou outros mecanismos apropriados.

O terceiro contexto corresponde aos ***callbacks* de temporizadores de *softwar*e**. Apesar do nome, esses *callbacks* não são executados em interrupções. Eles são executados por uma tarefa interna criada pelo próprio RTOS para gerenciar os temporizadores de *software*. Como o *callback* é executado em contexto de tarefa, ele utiliza a API convencional do RTOS, sem o sufixo `FromISR`. Entretanto, existe uma restrição prática importante: todos os temporizadores compartilham essa mesma tarefa. Assim, *callbacks* demorados ou que realizem operações bloqueantes podem atrasar a execução dos demais temporizadores e aumentar significativamente a latência do sistema. Por esse motivo, **recomenda-se que o *callback* execute apenas operações rápidas e delegue o processamento mais pesado para outras tarefas**.

É importante não confundir callbacks de temporizadores com ISRs. Embora ambos possam ser acionados após um intervalo de tempo, a ISR é executada imediatamente em resposta ao evento de hardware, enquanto o callback do temporizador somente será executado quando a tarefa responsável pelos temporizadores for escalonada. Isso introduz uma latência dependente do estado do sistema, normalmente maior e menos previsível do que a obtida por uma interrupção de hardware.

Um exemplo clássico é a recepção de dados por uma UART. Utilizar uma ISR permite que cada *byte* recebido seja tratado imediatamente após sua chegada, minimizando perdas de dados, reduzindo o consumo de CPU e mantendo a previsibilidade temporal. Já uma solução baseada em temporizadores exigiria consultar periodicamente o periférico (*polling*), desperdiçando processamento e introduzindo latência variável. Nessa situação, **a ISR normalmente realiza apenas o processamento mínimo necessário e sinaliza uma tarefa responsável pelo processamento completo dos dados**:

```
void UART_IRQHandler(void)
{
   char c = UART_Read();

   BaseType_t xHigherPriorityTaskWoken = pdFALSE;

   xQueueSendFromISR(uartRxQueue,
                     &c,
                     &xHigherPriorityTaskWoken);

   portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Caso fosse utilizada a função `xQueueSend()` em vez de `xQueueSendFromISR()`, o comportamento seria indefinido. A função convencional pressupõe que está sendo executada em contexto de tarefa e pode executar operações incompatíveis com o contexto de interrupção, comprometendo a integridade das estruturas internas do *kernel* e podendo provocar travamentos, corrupção de memória ou falhas temporais difíceis de reproduzir.

Em resumo, a diferença entre tarefas e ISRs não está no fato de apenas uma delas utilizar interrupções de *hardware* — o próprio RTOS depende intensamente delas para implementar o escalonamento. A diferença fundamental é que **tarefas são entidades executadas e controladas pelo escalonador do RTOS, enquanto ISRs são executadas diretamente pelo *hardware* em resposta a eventos assíncronos**. Como esses dois contextos possuem mecanismos de sincronização e restrições operacionais distintos, o RTOS fornece APIs específicas para cada um. Utilizar a versão correta dessas APIs não é apenas uma questão de estilo ou convenção, mas um requisito para preservar a consistência do *kernel*, garantir o determinismo temporal e assegurar o funcionamento correto do sistema embarcado.

Reentrância em funções

A reentrância é um conceito crítico em sistemas embarcados com RTOS. Com múltiplas tarefas, interrupções e temporizadores operando simultaneamente, escrever funções seguras e previsíveis exige atenção rigorosa à forma como os dados são manipulados. Entender o que é reentrância e como implementá-la corretamente é essencial para garantir robustez, integridade e tempo de resposta confiável em sistemas tempo real.

Uma função é considerada **reentrante** quando pode ser interrompida no meio de sua execução e chamada novamente (por outra tarefa, interrupção ou contexto) sem corromper seus dados internos ou causar comportamento indefinido. Isso é possível quando a função:

- não utiliza variáveis globais ou estáticas modificáveis sem proteção,
- não depende de recursos compartilhados sem sincronização e
- armazena seus dados exclusivamente em variáveis locais na pilha.

Por exemplo, funções matemáticas puras como `sin()`, `sqrt()`, ou processamento de memória `memcpy()` são tipicamente reentrantes, desde que não usem *buffers* estáticos internos.

Em sistemas baseados em **RTOS**, múltiplas tarefas (em inglês, *threads*) podem tentar acessar a mesma função simultaneamente. Além disso, ISRs podem também invocar funções a qualquer momento. Se essas funções não forem reentrantes, o sistema pode apresentar comportamentos imprevisíveis, como:

- corrupção de dados
- condições de corrida
- travamentos intermitentes e difíceis de depurar

Portanto, a **reentrância é essencial para garantir a integridade e a previsibilidade do sistema**, dois pilares fundamentais de sistemas em tempo real.

Considere a seguinte função não reentrante:

```
int buffer_index = 0;
void add_to_buffer(char c) {
    buffer[buffer_index++] = c;
}
```

Se duas tarefas chamarem essa função simultaneamente, a variável `buffer_index` pode ser corrompida devido a uma condição de corrida, já que múltiplos contextos de execução acessam e modificam o mesmo recurso sem sincronização. Para tornar a função `add_to_buffer` reentrante, é necessário eliminar o uso de estado global, fazendo com que ela dependa apenas de dados fornecidos por cada chamada.

Nesses casos, mecanismos de sincronização, como mutexes, continuam sendo necessários.

```
void add_to_buffer(char* buffer, int* index, char c) {
    buffer[(*index)++] = c;
}
```

No entanto, embora isso torne a função reentrante, não garante automaticamente segurança em ambientes concorrentes, caso múltiplas tarefas ainda compartilhem o mesmo *buffer* ou índice.

O RTOS, por si só, não impede que funções não reentrantes sejam chamadas simultaneamente. Cabe ao desenvolvedor garantir que funções críticas sejam reentrantes ou protegidas por mecanismos de sincronização, como semáforos, mutexes ou seções críticas. Alguns RTOS oferecem API específicas para proteger seções de código em contextos concorrentes, como `taskENTER_CRITICAL()` e `taskEXIT_CRITICAL()` no FreeRTOS.

O compilador pode auxiliar na geração de código reentrante ao:

- alocar variáveis locais na pilha da tarefa (*stack*),
- evitar uso implícito de variáveis estáticas ou globais e
- emitir avisos sobre uso inseguro de dados estáticos em funções compartilhadas.

No entanto, **o compilador não transforma automaticamente funções não reentrantes em reentrantes**. O desenvolvedor deve escrever o código corretamente ou usar diretivas específicas (como `__reentrant` em compiladores para sistemas embarcados como o Keil C51).

Em ambientes multitarefa proporcionados pelo RTOS, temporizadores de *software* frequentemente executam *callbacks* em contexto de tarefa ou ISR. Se uma função chamada por um temporizador não for reentrante, e outra tarefa estiver utilizando essa mesma função, isso pode levar a comportamentos não determinísticos. Assim, ao projetar *callbacks* de temporizadores ou funções acessadas por múltiplas tarefas, é fundamental garantir que:

- as funções sejam reentrantes ou
- acessem recursos com a devida proteção contra concorrência.

Controle de tempo, sincronismo e uso de *delays*

Em **RTOS**, várias tarefas executam de forma concorrente, compartilhando o mesmo processador. Para garantir que todas tenham tempo de execução adequado e que o sistema opere de forma previsível, é necessário um mecanismo de **gerenciamento temporal,** definindo **quando** e **por quanto tempo** cada uma tarefa deve rodar. Um dos recursos mais fundamentais para isso é o **uso de**

atrasos temporizados (*delays*), que permitem suspender uma tarefa por um intervalo de tempo determinado, liberando o processador para outras tarefas nesse período.

Esse comportamento é essencial para o **sincronismo** entre tarefas e para a **otimização do uso da CPU**. Em vez de deixar uma tarefa presa em um laço de espera ativa (*busy-wait*), o RTOS oferece funções que bloqueiam a tarefa de forma cooperativa. Assim, enquanto uma tarefa aguarda seu próximo ciclo de execução, o processador pode executar outras tarefas, reduzir consumo de energia ou até entrar em modo de economia. Esse princípio está diretamente relacionado à eficiência e previsibilidade de sistemas embarcados multitarefa.

Todo RTOS utiliza um temporizador de *hardware* para gerar interrupções periódicas chamadas *ticks*. Cada *tick* representa a menor unidade de tempo do sistema e é usado para controlar atrasos, *timeouts* e a alternância entre tarefas. Esse temporizador é configurado para operar com base em uma fonte de *clock* estável, geralmente relacionada ao *clock* do processador. Em microcontroladores baseados em ARM Cortex-M, esse papel geralmente é desempenhado pelo SysTick *Timer*, um temporizador padrão integrado ao núcleo do processador. O SysTick é alimentado por um sinal de *clock* derivado do *clock* principal da CPU e pode ser configurado para gerar interrupções em intervalos regulares, por exemplo, a cada 1 milissegundo. A cada interrupção do SysTick, o RTOS incrementa o contador de *ticks* e atualiza internamente o tempo do sistema.

Essa estratégia é amplamente utilizada porque o SysTick é simples, confiável e independente de periféricos externos. Ele permite que o RTOS mantenha uma contagem de tempo precisa e uniforme, servindo de base para funções de temporização, como atrasos (`vTaskDelay()` no FreeRTOS), *timeouts* e tarefas periódicas. Outros RTOS ou microcontroladores podem usar temporizadores diferentes, mas o princípio é sempre o mesmo: um temporizador gera interrupções periódicas que fornecem ao sistema operacional sua referência temporal.

A precisão e o comportamento dos atrasos dependem da **frequência de *tick* do sistema**, que define a resolução mínima de tempo que o RTOS pode controlar. Os *ticks* representam **valores inteiros e discretos** da contagem de tempo do sistema. Um valor típico é 1000 Hz (1 ms por *tick*), que oferece boa precisão para a maioria das aplicações embarcadas. Ajustar essa taxa permite equilibrar desempenho e consumo de recursos: quanto maior a frequência de *tick*, maior a precisão, mas também maior o número de interrupções geradas por segundo. No FreeRTOS, por exemplo, o tempo é definido pelo parâmetro `configTICK_RATE_HZ`, que especifica quantos *ticks* ocorrem por segundo. Funções como `vTaskDelay()` permitem que uma tarefa seja suspensa por uma quantidade de *ticks*, e a macro `pdMS_TO_TICKS()` converte milissegundos em unidades compatíveis com o relógio interno do sistema.

É importante que o valor de *ticks* passado ao `pdMS_TO_TICKS()` não seja muito pequeno, pois atrasos muito curtos podem resultar em tempos efetivos de espera inferiores à resolução do sistema, sendo arredondado para zero ou para um *tick*. Isso pode fazer com que a tarefa não seja realmente atrasada, comprometendo o comportamento temporal esperado. Assim, recomenda-se utilizar valores de atraso que correspondam a pelo menos um ou mais *ticks* inteiros, garantindo que o agendamento e a temporização ocorram conforme previsto.

Quando uma tarefa insere um atraso, ela entra em estado de **auto-bloqueio**, sendo retirada temporariamente da lista de tarefas prontas. Durante esse período, o processador é liberado para executar outras tarefas. A tarefa bloqueada será **automaticamente acordada** quando o tempo especificado em *ticks* expirar. Esse comportamento é gerenciado internamente pelo escalonador e garante o sincronismo e a eficiência do sistema.

O **uso de atrasos é particularmente útil em tarefas periódicas**, que precisam executar em intervalos regulares, como a leitura de sensores, o envio de dados, ou a atualização de uma interface. Por exemplo, uma tarefa de leitura de temperatura pode ser configurada para executar a cada 100 milissegundos, garantindo que o sistema opere de maneira previsível. Para casos em que a periodicidade precisa ser exata, os RTOS geralmente oferecem uma função de atraso relativo ao tempo anterior (como `vTaskDelayUntil()` no FreeRTOS), que compensa automaticamente o tempo gasto na execução da tarefa, mantendo o ciclo constante.

Em contraste, **há situações em que o uso de *delays* não é adequado**. Em tarefas **orientadas a eventos**, o bloqueio ocorre naturalmente até que uma condição externa seja satisfeita, como o recebimento de uma mensagem, a liberação de um semáforo ou a ocorrência de uma interrupção. Nesses casos, inserir um *delay* não melhora o comportamento do sistema e pode até aumentar a latência de resposta. Assim, o uso de atrasos deve ser reservado para tarefas que realmente precisam de controle temporal periódico, enquanto as demais devem depender de mecanismos de sincronização baseados em eventos.

PROGRAMAÇÃO EM RTOS

Programação RTOS é a prática de desenvolver *software* para sistemas que precisam executar tarefas com prazos rigorosos e previsíveis. Ao contrário de sistemas como Windows ou Linux, onde a velocidade e a responsividade geral são o foco, a programação RTOS exige que o desenvolvedor garanta que certas operações ocorram em momentos exatos e dentro de um tempo limite definido.

A principal meta da programação RTOS é o determinismo: a certeza de que uma ação sempre levará o mesmo tempo para ser concluída, independentemente de outras atividades no sistema. O desenvolvedor de um RTOS não apenas escreve o código da aplicação, mas também gerencia e configura as tarefas que a compõem. Isso envolve:

- definir e priorizar tarefas: Dividir a aplicação em tarefas independentes, cada uma com uma prioridade e um prazo específicos. Por exemplo, a tarefa de ler um sensor pode ter prioridade mais alta que a de atualizar um *display*.
- sincronizar tarefas: Usar mecanismos de sincronização, como semáforos e *mutexes,* para controlar o acesso a recursos compartilhados, evitando que uma tarefa interfira nos dados de outra.
- gerenciar o tempo: Utilizar temporizadores e eventos de interrupção para garantir que as tarefas sejam executadas no momento certo para cumprir seus prazos.

Em essência, a programação RTOS não se trata apenas de “fazer funcionar”, mas de fazer funcionar com garantias de tempo. É a base de sistemas em que a falha em cumprir um prazo pode ter

consequências sérias, como em equipamentos médicos, sistemas de controle de voo ou robótica industrial.

RTOS populares

Além de compreender os princípios teóricos de um RTOS, é igualmente importante conhecer implementações amplamente utilizadas na prática. Essa familiaridade permite não apenas observar como conceitos como escalonamento, gerenciamento de tarefas e sincronização são aplicados em sistemas reais, mas também auxilia na escolha do RTOS mais adequado para cada aplicação. Em sistemas embarcados críticos, essa decisão é especialmente relevante, pois envolve aspectos como determinismo temporal, consumo de recursos, suporte a *hardware* e confiabilidade. A seguir, são apresentados alguns dos RTOS mais utilizados na indústria e na academia:

- FreeRTOS: Um dos sistemas operacionais de tempo real (RTOS) mais populares no mundo embarcado, o FreeRTOS é amplamente adotado em microcontroladores de baixo custo. É especialmente utilizado em aplicações de *Internet* das Coisas (IoT), dispositivos portáteis e sistemas embarcados voltados à educação. Seus principais diferenciais incluem ser *open source*, possuir extensa documentação, contar com uma comunidade global ativa e dispor de suporte oficial pela *Amazon Web Services* (AWS), o que facilita sua integração com soluções em nuvem. Além disso, oferece pacotes adicionais, como bibliotecas de conectividade e criptografia, disponíveis conforme o modelo de uso.
  Do ponto de vista técnico, o FreeRTOS apresenta um *kernel* leve e modular, com suporte a escalonamento preemptivo baseado em prioridades, gerenciamento de tarefas, temporizadores de *software* e mecanismos de sincronização, como filas, semáforos e *mutexes*. Seu baixo consumo de memória e simplicidade de configuração o tornam especialmente adequado para sistemas com recursos limitados, embora exija maior responsabilidade do desenvolvedor na definição da arquitetura e no gerenciamento de recursos do sistema.
- Zephyr: Projeto *open source* mantido pela Linux *Foundation*, o Zephyr é focado em dispositivos conectados e aplicações de IoT. É comumente empregado em *wearables*, sensores inteligentes e sistemas embarcados em rede. Entre seus pontos fortes destacam-se o suporte a múltiplas arquiteturas de *hardware*, integração com ferramentas modernas de desenvolvimento (como *Devicetree* e CMake) e uma comunidade ativa que contribui para sua constante evolução.
  Do ponto de vista técnico, o Zephyr apresenta uma arquitetura modular e altamente configurável, com suporte a escalonamento preemptivo e cooperativo, gerenciamento de múltiplas *threads* e diversos mecanismos de comunicação e sincronização, como filas, semáforos e *mutexes*. O sistema também oferece suporte nativo a conectividade (Bluetooth, Wi-Fi, entre outros) e recursos de segurança, como isolamento de memória e execução em modos privilegiados. Embora possua maior complexidade de configuração em comparação a RTOS mais enxutos, essa flexibilidade o torna uma solução robusta e escalável para aplicações embarcadas modernas.
- RTEMS (em inglês, *Real-Time Executive for Multiprocessor Systems*): RTOS de código aberto voltado a aplicações de tempo real com requisitos rigorosos, o RTEMS é amplamente utilizado em projetos aeroespaciais, acadêmicos e de pesquisa. Tem sido adotado em missões espaciais da NASA e da ESA, devido ao seu suporte a sistemas multiprocessados, à alta previsibilidade temporal e à estabilidade. Sua arquitetura modular e foco em sistemas

embarcados críticos o tornam uma escolha consolidada em ambientes que exigem elevada confiabilidade.

Do ponto de vista técnico, o RTEMS oferece um *kernel* com escalonamento preemptivo baseado em prioridades, suporte a multiprocessamento (SMP), gerenciamento eficiente de tarefas e diversos mecanismos de sincronização e comunicação. O sistema também inclui serviços típicos de sistemas operacionais, como sistema de arquivos, pilha de rede e *drivers* de dispositivos, mantendo ao mesmo tempo um comportamento determinístico. Embora apresente maior complexidade de configuração em comparação a soluções mais simples, sua robustez e aderência a padrões abertos o tornam especialmente adequado para aplicações críticas e de longa duração.

- VxWorks: RTOS comercial desenvolvido pela *Wind River Systems*, o VxWorks é amplamente reconhecido como padrão de mercado em sistemas embarcados críticos. Sua utilização abrange desde satélites e aeronaves até sistemas militares e médicos. Suas principais características incluem alto grau de confiabilidade, suporte a arquiteturas avançadas e certificações rigorosas para setores como aviação, automotivo e defesa, tornando-o ideal para aplicações que exigem segurança e desempenho em tempo real.

  Do ponto de vista técnico, o VxWorks oferece um *kernel* robusto com suporte a escalonamento preemptivo baseado em prioridades, baixa latência de interrupção e mecanismos avançados de sincronização e comunicação entre tarefas. Além disso, disponibiliza recursos como proteção de memória, suporte a multiprocessamento (SMP), sistema de arquivos e pilhas de rede completas. Embora seja uma solução proprietária e de maior custo, seu amplo suporte comercial, ferramentas de desenvolvimento integradas e certificações tornam-no uma escolha consolidada para sistemas de missão crítica que exigem alto nível de determinismo e confiabilidade.
- Micrium (µC/OS-II e µC/OS-III): RTOS comercial originalmente desenvolvido pela Micrium, é amplamente reconhecido por sua clareza de implementação e forte uso em contextos educacionais e industriais. É aplicado em dispositivos médicos, sistemas automotivos e diversos projetos embarcados que exigem estabilidade e previsibilidade. Entre seus principais diferenciais destacam-se o acesso ao código-fonte (valioso para fins didáticos), a documentação abrangente e um histórico consolidado de robustez em aplicações críticas.

  Do ponto de vista técnico, o µC/OS-II e o µC/OS-III oferecem um *kernel* com escalonamento preemptivo baseado em prioridades, gerenciamento eficiente de tarefas e mecanismos de sincronização como semáforos, *mutexes* e filas. O µC/OS-III amplia as capacidades do µC/OS-II ao permitir múltiplas tarefas por nível de prioridade e maior escalabilidade. Além disso, o sistema apresenta baixo consumo de recursos e comportamento determinístico, sendo adequado para microcontroladores com limitações de memória. Embora seja uma solução proprietária, sua organização interna clara e previsível o torna especialmente útil tanto para ensino quanto para aplicações que demandam alta confiabilidade.
- **QNX:** RTOS comercial desenvolvido pela BlackBerry Limited, o QNX é amplamente utilizado em sistemas embarcados críticos, com forte presença nos setores automotivo, industrial e de dispositivos médicos. É conhecido por sua elevada confiabilidade, sendo empregado, por exemplo, em sistemas de *infotainment* automotivo, controle industrial e aplicações que exigem alta disponibilidade. Entre seus principais diferenciais destacam-se a

arquitetura baseada em microkernel, a robustez e o suporte a certificações de segurança, o que o torna adequado para aplicações críticas.
Do ponto de vista técnico, o QNX adota uma arquitetura de microkernel na qual apenas funcionalidades essenciais — como escalonamento, comunicação entre processos (IPC) e gerenciamento básico — são executadas no núcleo, enquanto serviços como drivers, sistemas de arquivos e pilhas de rede operam em espaço de usuário. Essa abordagem aumenta a modularidade, a segurança e a tolerância a falhas. O sistema oferece suporte a escalonamento preemptivo por prioridade, comunicação eficiente entre processos e suporte a multiprocessamento (SMP). Embora seja uma solução proprietária e de maior complexidade, sua arquitetura avançada e confiabilidade o tornam uma escolha consolidada para sistemas de missão crítica que exigem alto nível de isolamento e determinismo.

- **QuRT (do inglês *Qualcomm Real-Time Operating System*):** RTOS proprietário desenvolvido pela Qualcomm, o QuRT é voltado principalmente para execução em processadores *DSP* (do inglês *Digital Signal Processor*) da família Hexagon, sendo amplamente utilizado em dispositivos móveis e sistemas embarcados avançados. É empregado em aplicações como processamento de áudio, visão computacional, inteligência artificial embarcada e modems, onde são exigidos alto desempenho e baixa latência. Entre seus principais diferenciais destacam-se a forte integração com o ecossistema de *hardware* da Qualcomm e a otimização para processamento intensivo de sinais.
  Do ponto de vista técnico, o QuRT oferece um *kernel* leve com escalonamento preemptivo baseado em prioridades, suporte a múltiplas *threads* e mecanismos eficientes de comunicação e sincronização. O sistema é otimizado para execução em DSPs, explorando paralelismo e aceleração por *hardware*, além de oferecer suporte a isolamento de memória e controle de acesso a recursos. Embora seja uma solução proprietária e fortemente dependente da plataforma Qualcomm, sua eficiência e desempenho o tornam uma escolha adequada para aplicações embarcadas que demandam processamento intensivo e resposta em tempo real.

Na prática, a escolha de um RTOS deve considerar uma série de fatores que vão além da sua filosofia ou popularidade. Entre os principais critérios estão: tipo de licença, suporte à arquitetura e facilidade de uso.

A licença é um dos primeiros aspectos a serem avaliados. RTOS de código aberto, como FreeRTOS e Zephyr, são especialmente atrativos para projetos acadêmicos, de prototipagem ou de baixo custo, por oferecerem flexibilidade, ausência de custos de licença e uma base comunitária ativa. Já opções comerciais, como VxWorks e QNX, são preferidas em aplicações industriais críticas, onde se exige suporte especializado, atualizações garantidas e certificações formais (como as aplicáveis a setores aeroespaciais, automotivos ou médicos).

Outro fator determinante é o suporte à arquitetura do processador ou microcontrolador escolhido. A compatibilidade com a plataforma de *hardware* e com os ambientes de desenvolvimento influencia diretamente o tempo de desenvolvimento, a estabilidade do sistema e a escalabilidade do projeto. Veja abaixo um panorama resumido:

- FreeRTOS se destaca por seu amplo suporte a microcontroladores de diversas famílias. Possui implementações maduras para STM32 (incluindo a série de alto desempenho STM32H7), ESP32 e Raspberry Pi Pico (RP2040). Graças à sua leveza e modularidade, também pode ser adaptado para plataformas mais simples, como o Arduino ATmega, embora esta não seja uma aplicação típica.
- RTEMS suporta diversas arquiteturas de processador, incluindo famílias como ARM, RISC‑V, PowerPC, SPARC e x86, e variantes específicas dentro desses grupos. Por meio de *Board Support Packages* (BSPs) ele pode ser configurado para rodar em plataformas baseadas em ARM, o que viabiliza sua execução em microcontroladores como os usados por STM32 e também em processadores compatíveis com ARM (incluindo Cortex‑M). Essa flexibilidade torna o RTEMS adequado tanto para sistemas embarcados de médio porte quanto para aplicações de missão crítica com requisitos rígidos de determinismo e confiabilidade.
- RTEMS suporta múltiplas arquiteturas de processadores, incluindo ARM, PowerPC, x86 e MIPS. É especialmente adequado para sistemas multiprocessados e embarcados críticos, onde previsibilidade e confiabilidade são essenciais. Diferente de RTOS voltados a microcontroladores de baixo custo, o RTEMS é mais utilizado em sistemas de médio e alto desempenho que exigem comportamento determinístico.
- Zephyr, voltado para soluções de IoT e dispositivos conectados, tem um ecossistema moderno e em constante expansão. Oferece excelente suporte para STM32, ESP32 e RP2040, com integração com ferramentas como Devicetree, CMake e subsistemas de segurança e conectividade.
- VxWorks e QNX, por serem soluções comerciais e altamente robustas, são direcionados a *hardware* mais sofisticado, com maior poder de processamento e requisitos críticos. Oferecem suporte completo para a série de alto desempenho da família STM32, incluindo o STM32H7, mas raramente são utilizados em plataformas como ESP32, RP2040 ou Arduino ATmega, tanto pelo custo da licença quanto pela complexidade do ecossistema, mais adequado a projetos industriais ou regulados.
- QuRT (do inglês *Qualcomm Real-Time Operating System*) é um RTOS proprietário otimizado para rodar sobre a arquitetura Qualcomm Hexagon DSP, amplamente usado em SoCs móveis da empresa. Ele é desenhado para fornecer baixa latência, eficiência energética e integração estreita com os subsistemas de áudio, vídeo e conectividade em dispositivos como smartphones e wearables. Por ser específico de *hardware* Qualcomm, não é encontrado em microcontroladores genéricos (como STM32, ESP32 ou RP2040), mas se destaca em cenários de computação embarcada de alto desempenho em plataformas móveis.

RTOS	RP2040 / RP2350	ESP32	STM32 (Cortex-M)	Notas
FreeRTOS	✅ Sim	✅ Nativo no ESP-IDF	✅ Integrado ao STM32CubeMX	Leve, amplamente utilizado
Zephyr	✅ Sim (em expansão para RP2350)	✅ Compatível	✅ Suporte oficial extenso	Ideal para projetos IoT complexos
VxWorks	❌ Não	❌ Não	⚠ Apenas STM32MP1 (com Cortex-A)	Comercial e robusto, exige MMU
QNX	❌ Não	❌ Não	⚠ Apenas STM32MP1 (com Cortex-A)	Sistema operacional seguro e caro
QuRT	❌ Não	❌ Não	❌ Não	Exclusivo da Qualcomm (DSP Hexagon)

Legenda

- ✅ **Compatível**: Suporte completo e recomendado.
- ⚠ **Parcial**: Compatível apenas em modelos mais robustos com MMU.
- ❌ **Incompatível**: Não aplicável à arquitetura ou não suportado.

Por fim, a facilidade de uso, que inclui qualidade da documentação, disponibilidade de exemplos, integração com ferramentas de desenvolvimento e suporte da comunidade, desempenha um papel crucial no sucesso de um projeto. RTOS com boa documentação e uma comunidade ativa, como FreeRTOS e Zephyr, são mais acessíveis para iniciantes e reduzem significativamente a curva de aprendizado. Já sistemas comerciais oferecem suporte técnico direto, ideal para empresas que necessitam de estabilidade e respostas rápidas em ambientes produtivos.

FreeRTOS

O FreeRTOS é um sistema operacional de tempo real projetado para ser leve, modular e portátil, permitindo que desenvolvedores usem recursos típicos de sistemas multitarefa em microcontroladores de baixo custo. Sua filosofia é a de oferecer apenas o essencial: um escalonador eficiente, mecanismos de sincronização e comunicação entre tarefas, e temporizadores de *software*, tudo em uma base compacta e fácil de portar para diferentes arquiteturas.

A sua origem remonta a 2003, quando foi criado por Richard Barry. A motivação inicial era preencher uma lacuna no mercado de sistemas embarcados, fornecendo um RTOS de nível profissional com código-fonte aberto, sem custos de licenciamento. Isso o diferenciava das opções

comerciais caras e de código fechado da época, tornando-o acessível a desenvolvedores e projetos acadêmicos. Em 2017, o projeto foi adquirido pela *Amazon Web Services* (AWS), que hoje atua como o principal desenvolvedor e mantenedor. A Amazon continua aprimorando o FreeRTOS, principalmente para integrá-lo ao seu ecossistema de serviços de Internet das Coisas (IoT), ao mesmo tempo que mantém a sua essência *open-source* e sua ampla compatibilidade com diversas plataformas.

O FreeRTOS não impõe restrições rígidas sobre como organizar o código: ele fornece a infraestrutura para que o programador defina tarefas, priorize sua execução e coordene o uso dos recursos do sistema. Para usar o FreeRTOS, é necessário clonar o repositório oficial, GitHub do FreeRTOS:

git clone https://github.com/FreeRTOS/FreeRTOS.git --recurse-submodules

As seguintes pastas são essenciais para desenvolvimento de *firmwares* em RTOS::

- `FreeRTOS/Source`: Contém os códigos-fonte do *kernel* do FreeRTOS (`tasks.c`, `list.c`, `queue.c`, etc.).
- `FreeRTOS/Source/include`: Contém os arquivos-cabeçalho das funções do *kernel* do FreeRTOS.
- `FreeRTOS/Source/portable`**:** Esta é a pasta mais importante para a portabilidade. Nela, há o código específico para diferentes arquiteturas de processador e compiladores (como o GCC para ARM, GCC para x86, etc.). É necessário incluir o arquivo de portabilidade correto para o microcontrolador usado (por exemplo, ARM Cortex-M4F).

O repositório também inclui projetos de exemplo completos (`FreeRTOS/Demo`) para várias placas e plataformas. Esses exemplos são uma excelente referência para entender a estrutura de um projeto com FreeRTOS.

Além disso, cada projeto requer um arquivo de configuração chamado `FreeRTOSConfig.h`. Esse arquivo é incluído no arquivo-cabeçalho `FreeRTOS.h`. Ele é fundamental, pois define parâmetros específicos da aplicação, como o tamanho da pilha, a frequência do *tick* do sistema e as políticas de escalonamento. A melhor maneira de criar esse arquivo é copiar um modelo (`FreeRTOSConfig.h`) de um projeto-exemplo do FreeRTOS que use uma arquitetura similar. Em seguida, o desenvolvedor deve abrir o arquivo e ajustar os valores das macros para corresponderem às especificações do microcontrolador e às necessidades da aplicação. Os modelos de exemplo geralmente contêm comentários detalhados que explicam o propósito de cada macro, orientando o processo de configuração.

Ao usar microcontroladores como RP2040 (Raspberry Pi Pico) ou STM32, normalmente os fabricantes oferecem pacotes de exemplo com o FreeRTOS já integrado ao SDK, simplificando a instalação.

Um arquivo `main.c` típico em FreeRTOS segue a seguinte organização:

1. Inclusão de bibliotecas do FreeRTOS e de cabeçalhos do microcontrolador (CMSIS ou SDK específico).
2. Criação das tarefas usando `xTaskCreate()` ou `xTaskCreateStatic()`.
3. (Opcional) Criação de filas, semáforos, grupos de eventos ou temporizadores para sincronização e comunicação.
4. Inicialização de periféricos do microcontrolador.
5. Início do escalonador com `vTaskStartScheduler()`, a partir do qual o controle passa ao RTOS.

A partir desse ponto, o programa deixa de ter uma função `main()` sequencial tradicional e passa a ser conduzido pelas tarefas que o programador definiu. A documentação do FreeRTOS oferece uma explicação sobre convenção de notações e um guia de estilo de código detalhado, que podem ser encontrados nas seções Naming Conventions e `Style Guide`, respectivamente.

Arquitetura do FreeRTOS

No FreeRTOS, o fluxo de execução gira em torno de **tarefas** que operam enquanto estão *Em Execução*, podendo ceder a CPU voluntariamente (*yield*) ou aguardar eventos de forma bloqueante. **ISRs** respondem a eventos externos e podem acordar tarefas ou transmitir dados. ***Callbacks*** permitem reagir a temporizadores e eventos sem interromper o fluxo das tarefas. Para comunicação e sincronização, **filas, semáforos, mutexes e grupos de eventos** garantem acesso seguro a recursos compartilhados. Essa estrutura modular e determinística permite organizar sistemas complexos de maneira eficiente, mantendo previsibilidade e aproveitando ao máximo os recursos limitados do microcontrolador. Do ponto de vista do desenvolvedor, esses elementos da arquitetura são apresentados como:

- **Tarefas** são unidades de execução concorrente, semelhantes a *threads* em GPOS. Cada tarefa possui uma função de execução (`TaskFunction_t`), uma prioridade, uma pilha própria e um **handle** (`TaskHandle_t`) que a identifica dentro do sistema. Entre as funções principais usadas para gerenciar tarefas estão `xTaskCreate()`, que cria uma nova tarefa; `vTaskDelete()`, que remove uma tarefa do escalonador; `vTaskDelay()`, que suspende a tarefa por um número determinado de ticks; e `vTaskStartScheduler()`, que inicia a execução do RTOS. Vale observar a convenção de nomes adotada pelo FreeRTOS: funções iniciadas com **x** retornam um valor (normalmente do tipo `BaseType_t`) indicando sucesso ou falha, enquanto funções iniciadas com **v** não retornam valor (void).
- **ISRs** lidam com eventos externos ao microcontrolador, como botões, sensores ou interfaces de comunicação. Elas são executadas fora do contexto das tarefas, mas podem interagir com elas utilizando APIs especiais terminadas em `FromISR`, que garantem operações seguras dentro de interrupções. Por exemplo, `xQueueSendFromISR()` envia dados a uma fila a partir de uma ISR, enquanto `xSemaphoreGiveFromISR()` libera um semáforo. Essas

funções são específicas para ISRs e asseguram que o RTOS não corrompa estruturas internas ao manipular tarefas durante a execução de interrupções.

- ***Callbacks*** são funções de retorno chamadas pelo sistema em resposta a eventos específicos, como a expiração de um temporizador ou a conclusão de uma operação. Um exemplo é o *callback* de temporizador de *software* `vTimerCallback(TimerHandle_t xTimer)`. Esses *callbacks* permitem integrar lógica de alto nível ao sistema sem bloquear a CPU, garantindo que tarefas críticas continuem executando de forma determinística.
- **Mecanismos de Comunicação e Sincronização** são estruturas que coordenam a execução das tarefas. No FreeRTOS, **filas** permitem a passagem de dados entre tarefas de forma segura, sendo manipuladas por funções como `xQueueCreate()`, `xQueueSend()` e `xQueueReceive()`. Durante o fluxo de execução, uma tarefa pode enviar ou receber dados via filas, bloqueando-se até que a operação seja concluída, garantindo sincronização entre produtor e consumidor. **Semáforos e *mutexes*** controlam o acesso a recursos compartilhados e evitam *race conditions*, com funções como `xSemaphoreCreateBinary()`, `xSemaphoreTake()` e `xSemaphoreGive()`. No fluxo típico, uma tarefa adquire o semáforo ou *mutex* antes de acessar o recurso e o libera ao finalizar, permitindo que outras tarefas possam prosseguir de forma segura.

  Para situações em que tarefas precisam aguardar múltiplos eventos simultaneamente, os **grupos de eventos** fornecem uma solução eficiente, usando `xEventGroupSetBits()` e `xEventGroupWaitBits()`. No fluxo de execução, uma tarefa pode bloquear aguardando que um ou mais *bits* de evento sejam definidos, retomando sua execução apenas quando todas as condições necessárias forem atendidas. Já os **temporizadores de *software*** permitem a execução de *callbacks* após um tempo determinado sem bloquear tarefas, por meio de `xTimerCreate()` e `xTimerStart()`. Eles são integrados ao fluxo como eventos assíncronos que despertam tarefas ou executam funções específicas de forma reativa.

A API do FreeRTOS segue uma convenção de nomes que facilita a compreensão: funções que retornam um valor de status indicam sucesso ou falha e começam com `x`; funções que não retornam valor usam `v`; e funções que podem ser chamadas com segurança dentro de uma ISR contêm o sufixo `FromISR`. Esses mecanismos se conectam de forma coordenada, permitindo que tarefas, interrupções e temporizadores interajam de maneira determinística, eficiente e modular, mantendo a previsibilidade essencial em sistemas de tempo real.

Para portabilidade entre arquiteturas diferentes, o FreeRTOS define tipos específicos:

- `TickType_t` → contagem de *ticks* do sistema (normalmente `uint32_t`)
- `BaseType_t` → tipo genérico para retornos e variáveis internas (`int` ou `long`)
- `UBaseType_t` → versão sem sinal de `BaseType_t`
- `TaskHandle_t`, `QueueHandle_t`, `SemaphoreHandle_t`, `TimerHandle_t` → identificadores de recursos

A documentação online da API do FreeRTOS, disponível também em PDF, fornece uma descrição funcional detalhada de cada função, incluindo os argumentos que devem ser fornecidos, os valores de retorno e o comportamento esperado durante a execução. Ela também mostra como utilizá-las corretamente dentro do contexto de um sistema embarcado, permitindo que o desenvolvedor compreenda o efeito de cada chamada, os requisitos de pré-condições e como integrar as funções de maneira segura e eficiente em aplicações multitarefa.

CMSIS-RTOS v2

O CMSIS-RTOS v2[12] é uma camada de abstração padronizada para sistemas embarcados, baseada na API **CMSIS** (do inglês *Cortex Microcontroller Software Interface Standard*), desenvolvida pela ARM para facilitar a portabilidade de aplicações em diferentes microcontroladores Cortex-M. Ele define um conjunto consistente de funções e tipos de dados para tarefas, temporizadores, filas, semáforos, mutexes e grupos de eventos, permitindo que o desenvolvedor utilize um RTOS sem se prender à implementação específica de cada fornecedor.

O CMSIS-RTOS v2 possui uma relação direta com o FreeRTOS, sendo frequentemente implementado sobre ele. Nesse cenário, o CMSIS-RTOS atua como uma camada de compatibilidade, oferecendo uma API padronizada enquanto delega a execução das tarefas e o gerenciamento de recursos para o núcleo do FreeRTOS. Essa abordagem combina a maturidade e robustez do FreeRTOS com a portabilidade e consistência da API CMSIS, permitindo que o mesmo código de aplicação funcione em diferentes microcontroladores sem alterações significativas.

A arquitetura do CMSIS-RTOS v2 é organizada em torno de conceitos semelhantes aos do FreeRTOS, incluindo tarefas, ISRs, *callbacks*, filas, semáforos, *mutexes*, temporizadores de *software* e grupos de eventos, mas com tipos de dados e convenções de nomenclatura padronizados. Por exemplo, os identificadores de recursos usam tipos como `osThreadId_t`, `osMutexId_t`, `osSemaphoreId_t`, `osTimerId_t` e `osEventFlagsId_t`, enquanto os tipos básicos de retorno incluem `osStatus_t` e `osPriority_t`. Já no FreeRTOS, os recursos são identificados por tipos como `TaskHandle_t`, `SemaphoreHandle_t`, `QueueHandle_t` e `TimerHandle_t`, e os retornos de *status* geralmente usam `BaseType_t` ou `UBaseType_t`. Essa padronização do CMSIS-RTOS facilita a leitura do código e a portabilidade entre diferentes núcleos RTOS compatíveis com CMSIS.

A sintaxe dos comandos do CMSIS-RTOS v2 segue a convenção `os<Operação>`, enquanto o FreeRTOS utiliza prefixos que indicam o tipo de retorno: funções iniciadas com `x` retornam valor (normalmente `BaseType_t`) e funções iniciadas com `v` não retornam valor (`void`). Por exemplo, para criar uma tarefa no CMSIS-RTOS v2 usa-se `osThreadNew()`, enquanto no FreeRTOS é `xTaskCreate()`. Para temporizadores de *software*, o CMSIS-RTOS v2 oferece `osTimerNew()`, `osTimerStart()` e `osTimerStop()`, enquanto o FreeRTOS disponibiliza `xTimerCreate()`, `xTimerStart()` e `xTimerStop()`. Para comunicação e sincronização, as funções do CMSIS-RTOS incluem `osSemaphoreAcquire()`, `osSemaphoreRelease()`, `osMutexAcquire()`, `osMutexRelease()`, `osMessageQueuePut()` e `osMessageQueueGet()`, enquanto no FreeRTOS utiliza-se `xSemaphoreTake()`, `xSemaphoreGive()`, `xQueueSend()` e `xQueueReceive()`.

No que se refere à execução, é fundamental distinguir os diferentes contextos do sistema: tarefas, ISRs e *callbacks*. Os *callbacks* não se limitam aos temporizadores do RTOS; eles também incluem *callbacks* da HAL associados a eventos de *hardware*, como `HAL_GPIO_EXTI_Callback()` ou `HAL_UART_RxCpltCallback()`. Enquanto *callbacks* de temporizadores do RTOS são executados em um contexto gerenciado pelo *kernel*, *callbacks* da HAL são geralmente executados no contexto de interrupção (ISR). Em ambos os casos, esses contextos exigem execução rápida e

[12] A API CMSIS contempla dois níveis principais de abstração: um voltado ao **núcleo do processador (Cortex-M)**, por meio de definições padronizadas de registradores e funções básicas de baixo nível, e outro voltado ao **sistema operacional**, por meio da CMSIS-RTOS, que fornece uma interface genérica para gerenciamento de tarefas, temporização e sincronização.

determinística, o que implica uma restrição fundamental: **não devem conter operações bloqueantes a nível do sistema**.

Essa restrição está diretamente relacionada ao uso dos mecanismos de comunicação e sincronização. Em tarefas, é apropriado utilizar operações bloqueantes, como `osMessageQueueGet()` com espera, `osSemaphoreAcquire()` com *timeout* ou `osEventFlagsWait()`, pois isso permite que a tarefa libere a CPU enquanto aguarda um evento, aumentando a eficiência do sistema. No entanto, em ISRs e *callbacks*, tais operações são inadequadas, pois implicariam em suspensão da execução em um contexto onde o escalonador não pode atuar da mesma forma. Nesses casos, devem ser utilizadas operações não-bloqueantes, como `osMessageQueuePut()` sem espera ou `osSemaphoreRelease()`, tipicamente para sinalizar eventos a tarefas. Assim, os mecanismos de sincronização e comunicação assumem papéis distintos conforme o contexto de execução: em tarefas, eles estruturam o fluxo de execução e permitem coordenação eficiente por meio de bloqueio; em ISRs e *callbacks*, são utilizados principalmente para **sinalização rápida e segura**, delegando o processamento mais complexo para tarefas. Essa separação é essencial para garantir previsibilidade, baixo tempo de resposta e uso eficiente da CPU em sistemas de tempo real.

Um ponto importante a ser destacado é que, diferentemente do FreeRTOS, o CMSIS-RTOS v2 **não explicita na nomenclatura das funções a distinção entre chamadas feitas no contexto de tarefas e aquelas realizadas em ISRs ou *callbacks*** (como ocorre com o sufixo `FromISR` no FreeRTOS). Em vez disso, essa distinção é tratada por meio da documentação e das convenções de uso da API. Em geral, funções que podem ser chamadas em ISRs são projetadas para não bloquear o fluxo de execução e devem ser utilizadas com parâmetros apropriados (por exemplo, tempo de espera igual a zero). Já funções que envolvem espera por eventos (como aquisição de semáforos com *timeout* ou recepção de mensagens com bloqueio) são destinadas ao contexto de tarefas. Dessa forma, cabe ao desenvolvedor compreender, com base na documentação, quais funções são seguras em cada contexto de execução. Essa escolha de projeto reforça a portabilidade da API, mas exige maior atenção do programador para garantir o uso correto e seguro dos mecanismos de sincronização em sistemas de tempo real.

A documentação oficial do CMSIS-RTOS v2, disponível *online*, fornece uma descrição funcional detalhada de cada função, incluindo argumentos, valores de retorno e comportamento esperado, além de templates e exemplos de uso de tarefas, filas, semáforos, mutexes e temporizadores. Essa documentação é essencial para o desenvolvimento seguro e eficiente de sistemas multitarefa, permitindo que o desenvolvedor compreenda as pré-condições de cada função e integre os recursos do RTOS de maneira consistente.

Dessa forma, o CMSIS-RTOS v2 integra a **portabilidade e padronização** da API CMSIS com a **robustez e maturidade** do FreeRTOS, oferecendo aos engenheiros uma forma consistente e eficiente de desenvolver aplicações embarcadas complexas, garantindo desempenho em tempo real, previsibilidade do sistema e compatibilidade entre diferentes microcontroladores.

Suporte do STM32Cube ao Desenvolvimento com CMSIS-RTOS v2

O ecossistema STM32Cube, composto principalmente pelo STM32CubeMX e pelo STM32CubeIDE, oferece suporte direto ao desenvolvimento de aplicações baseadas em CMSIS-RTOS v2, facilitando significativamente a criação de sistemas embarcados multitarefa.

Ao configurar um projeto no STM32CubeMX, o desenvolvedor pode habilitar o *middleware* do RTOS (tipicamente o FreeRTOS) com suporte à interface CMSIS-RTOS v2. Nesse processo, além da configuração tradicional de pinos e periféricos, torna-se possível definir graficamente elementos

fundamentais do sistema concorrente, como tarefas (*threads*), temporizadores de *software*, semáforos, *mutexes*, filas (em inglês, *message queues*) e grupos de eventos (em inglês, *event flags*). Essa abordagem permite que o desenvolvedor foque inicialmente na estrutura lógica do sistema, sem precisar escrever manualmente toda a infraestrutura de inicialização do RTOS.

Uma vez configurado, o STM32CubeMX gera automaticamente o código base do projeto, incluindo a inicialização do kernel, a criação dos objetos do RTOS e a estrutura das tarefas. No caso do uso de CMSIS-RTOS v2, as funções geradas seguem a convenção padronizada `os<Recurso><Operação>`, como `osThreadNew()`, `osDelay()` e `osSemaphoreAcquire()`. Isso permite que o código da aplicação seja escrito de forma independente da implementação específica do RTOS. Na prática, no ambiente STM32Cube, o *kernel* subjacente utilizado é o FreeRTOS; no entanto, o uso da API CMSIS-RTOS v2 torna possível, do ponto de vista conceitual, a substituição por outro RTOS compatível, com impacto mínimo no código da aplicação.

No STM32CubeIDE, o desenvolvedor pode então complementar esse código, implementando a lógica das tarefas, *callbacks* e mecanismos de comunicação. A organização típica separa a inicialização do sistema (em `main.c`) da definição das tarefas e objetos do RTOS (frequentemente em arquivos como `app_freertos.c` ou `freertos.c`), favorecendo a modularidade e a clareza do projeto.

Outro aspecto importante é que o STM32CubeMX também permite configurar parâmetros internos do FreeRTOS, como tamanho de pilha das tarefas, prioridades, política de escalonamento e uso de recursos (*heap*, *timers*, *hooks*), mesmo quando se utiliza a API CMSIS-RTOS v2. Isso evidencia que o CMSIS-RTOS atua como uma camada de abstração, enquanto o comportamento real do sistema ainda depende da configuração do *kernel* subjacente FreeRTOS.

Por fim, esse suporte integrado torna o STM32Cube uma ferramenta essencial no aprendizado de sistemas embarcados com RTOS, pois permite uma transição natural entre o desenvolvimento baseado em HAL (sequencial) e o desenvolvimento concorrente com RTOS. Ao automatizar a configuração inicial e padronizar a interface de programação, o ambiente reduz a complexidade inicial e permite que o desenvolvedor concentre seus esforços na compreensão dos conceitos fundamentais de concorrência, sincronização e tempo real.

Guia completo de desenvolvimento – projeto P3

Este guia descreve o processo de desenvolvimento da versão atualizada do projeto P3, baseada em um RTOS utilizando a interface CMSIS-RTOS v2. O projeto é uma evolução de uma implementação anterior em *superloop* com HAL (projeto P2), mantendo o *hardware* já montado em *protoboard* (extensão da placa BitDogLab). O objetivo principal é transformar um sistema sequencial em um sistema concorrente, mais modular, escalável e responsivo. Para isso, são utilizadas as seguintes ferramentas:

- STM32CubeMX para geração de códigos de configuração dos sinais de relógio e dos periféricos.
- STM32CubeIDE para implementação da lógica da aplicação, na construção do executável e na análise do comportamento do sistema em tempo de execução.
    - Eclipse EGit para serviço de hospedagem de repositórios remotos baseado em Git (https://gitlab.unicamp.br/)
    - Eclipse Terminal CDT para integrar um Terminal Serial no IDE.
    - Eclox para documentação de *firmwares*.

- ST-Link v2 para depuração de *firmware* por meio do protocolo SWD.
- (Opcional) analisador lógico, multímetro e osciloscópio.

Modelagem do Sistema (Nova Etapa – Fundamental)

Antes de qualquer implementação, é necessário converter o modelo ***superloop*** em um modelo **concorrente baseado em tarefas**. Para isso, **identifique as funcionalidades** listando todas as funcionalidades do sistema original, por exemplo:

- Leitura de sensores
- Processamento de dados
- Comunicação serial
- Controle de atuadores

Cada funcionalidade deve ser **mapeada para uma ou mais tarefas**, como

Funcionalidade	Tarefa RTOS
Leitura de sensor	SensorTask
Controle	ControlTask
Comunicação	CommTask
Processamento e produtor de dados	ProducerTask

Em seguida, atribui-se **o nível de prioridade de execução para cada tarefa**, de modo que as tarefas críticas recebam a maior prioridade, as tarefas periódicas fiquem com prioridade média e as tarefas de *logging* ou interface sejam designadas com a menor prioridade. Além disso, é importante analisar a possibilidade de **reutilizar uma mesma função** entre múltiplas tarefas, por meio da passagem de parâmetros (por exemplo, via ponteiro `(void *)`). Essa abordagem promove modularidade, reduz duplicação de código e facilita a manutenção, permitindo que diferentes tarefas executem comportamentos semelhantes com variações definidas pelos argumentos recebidos.

A comunicação e a sincronização entre tarefas são fundamentais para garantir que elas operem de forma cooperativa e coordenada, evitando conflitos e inconsistências no sistema. Para isso, é necessário conectar as tarefas por meio de mecanismos adequados que permitam tanto a troca eficiente de informações quanto o controle seguro do acesso aos recursos compartilhados. Nesse contexto, podem ser utilizadas **filas** (`osMessageQueue`) para viabilizar a troca de dados entre tarefas, **semáforos** (`osSemaphore`) para controlar o acesso a determinados recursos ou limitar a execução concorrente, **mutex** (`osMutex`) para assegurar a proteção de recursos compartilhados contra acessos simultâneos indevidos e **eventos** (`osEventFlags`) para realizar a sinalização entre tarefas, permitindo que uma tarefa informe a outra sobre a ocorrência de determinadas condições. Esses mecanismos, quando bem aplicados, garantem um funcionamento sincronizado e cooperativo do sistema.

A temporização em sistemas baseados em RTOS deve abandonar o uso de *delays* bloqueantes tradicionais, que paralisam completamente a execução do fluxo e comprometem a concorrência entre tarefas, em favor de mecanismos próprios do sistema operacional, como o `osDelay()`, que suspende apenas a tarefa corrente permitindo que o escalonador execute outras tarefas prontas, e os

timers do RTOS (`osTimer`), que possibilitam a execução de ações periódicas ou disparadas por tempo de forma assíncrona e eficiente

Mapa de Conexões de Periféricos *Off-board*

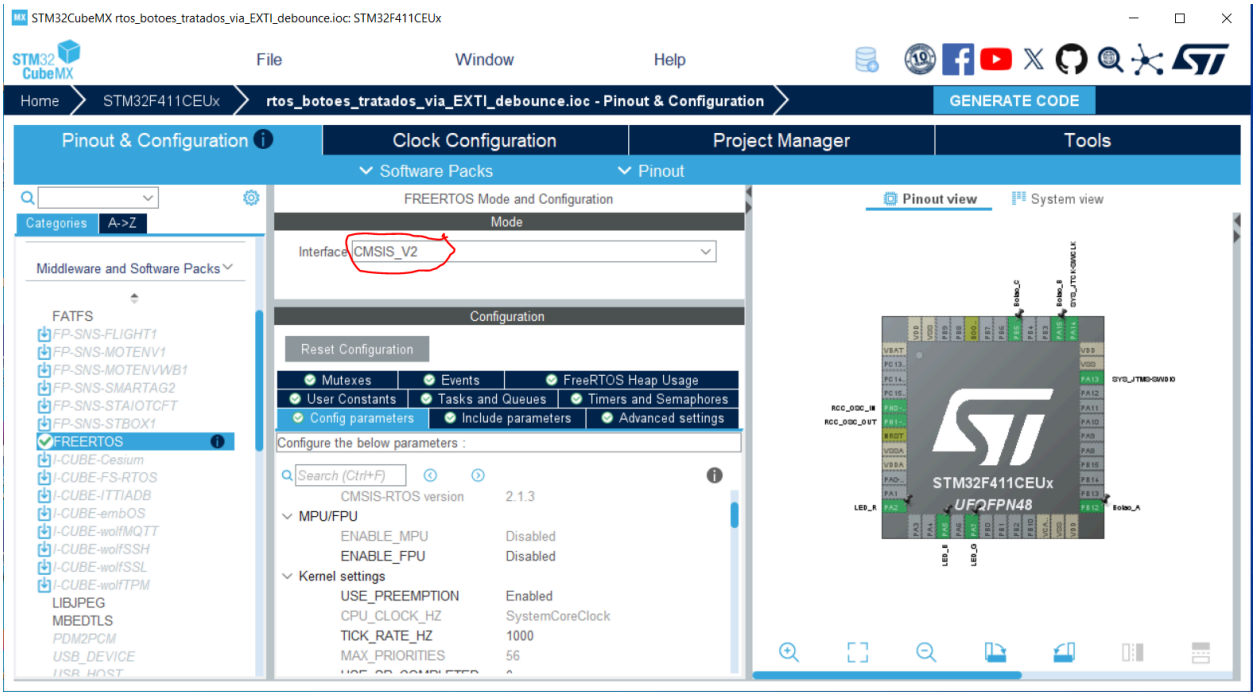
O procedimento para execução deste item é idêntico ao apresentado no Roteiro 2, podendo ser reutilizado integralmente.

Clonagem da pasta de projeto pelo STM32CubeIDE

O procedimento para execução deste item é idêntico ao apresentado no Roteiro 2, podendo ser reutilizado integralmente.

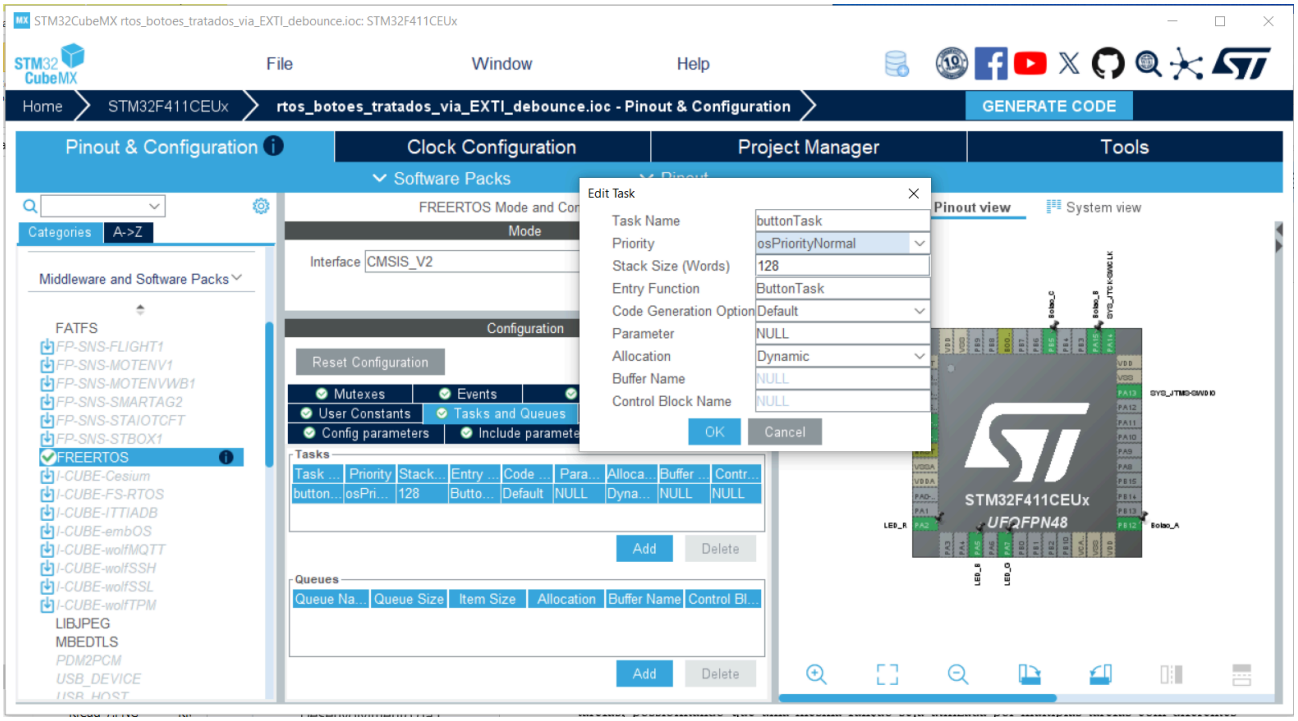
Ativação do RTOS e configuração dos elementos de RTOS no STM32CubeMX

A configuração de um sistema operacional de tempo real (RTOS) começa pela sua ativação e integração adequada ao ambiente de desenvolvimento. Nesse contexto, habilita-se o *middleware* correspondente, como o FreeRTOS, utilizando uma interface padronizada, por exemplo a CMSIS-RTOS v2, que abstrai detalhes específicos do *kernel* e facilita a portabilidade e organização do código. Essa camada de abstração permite ao desenvolvedor trabalhar com APIs consistentes para gerenciamento de tarefas, temporização e sincronização, independentemente da implementação subjacente do RTOS.

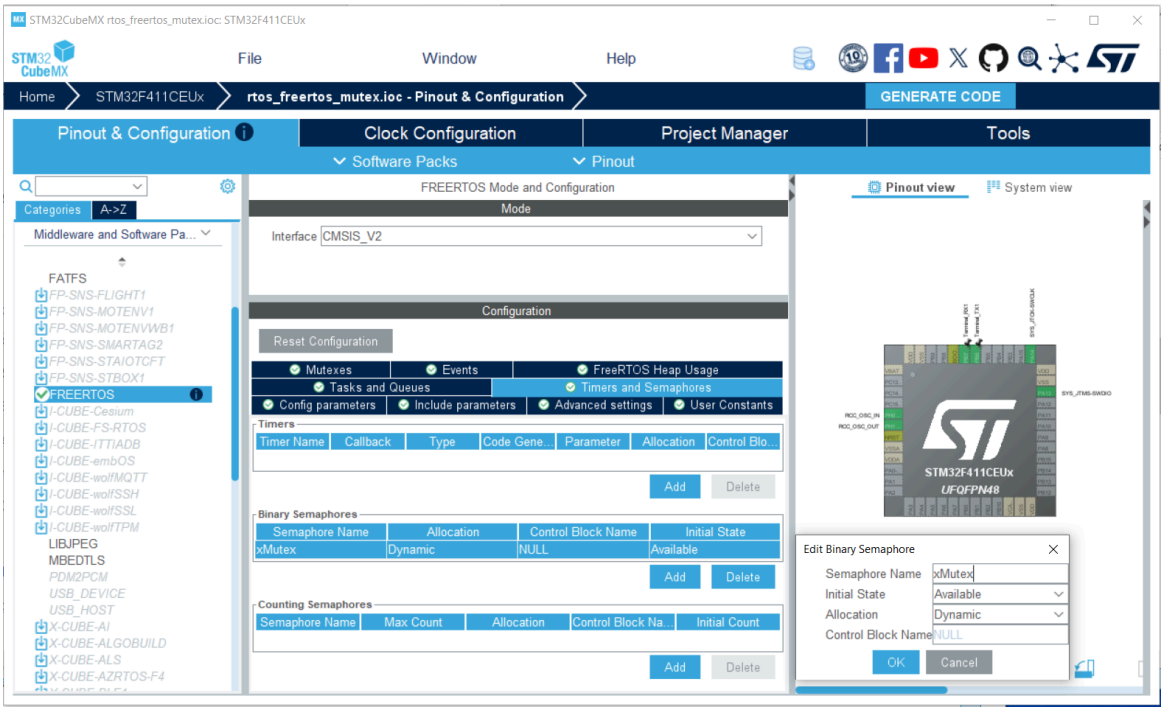


Uma vez ativado o RTOS, a próxima etapa consiste na **definição das tarefas** que compõem a aplicação. Acesse a aba "`Tasks and Queues`", que já inclui uma tarefa padrão com prioridade normal. Essa tarefa não pode ser removida. Caso seja necessário criar novas tarefas, utilize o botão "`Add`". Cada tarefa deve ser configurada com um **nome identificador**, o que facilita a leitura e a manutenção do sistema, especialmente em projetos mais complexos. A **prioridade** atribuída a cada tarefa é um dos aspectos mais críticos, pois determina a ordem de execução em um sistema preemptivo: tarefas de maior prioridade podem interromper a execução de tarefas de menor prioridade, garantindo o atendimento aos requisitos de tempo real. Além disso, o **tamanho da pilha** (*stack*) deve ser dimensionado de acordo com a complexidade da tarefa, considerando o uso de variáveis locais, chamadas de função e possíveis interrupções. Um dimensionamento adequado

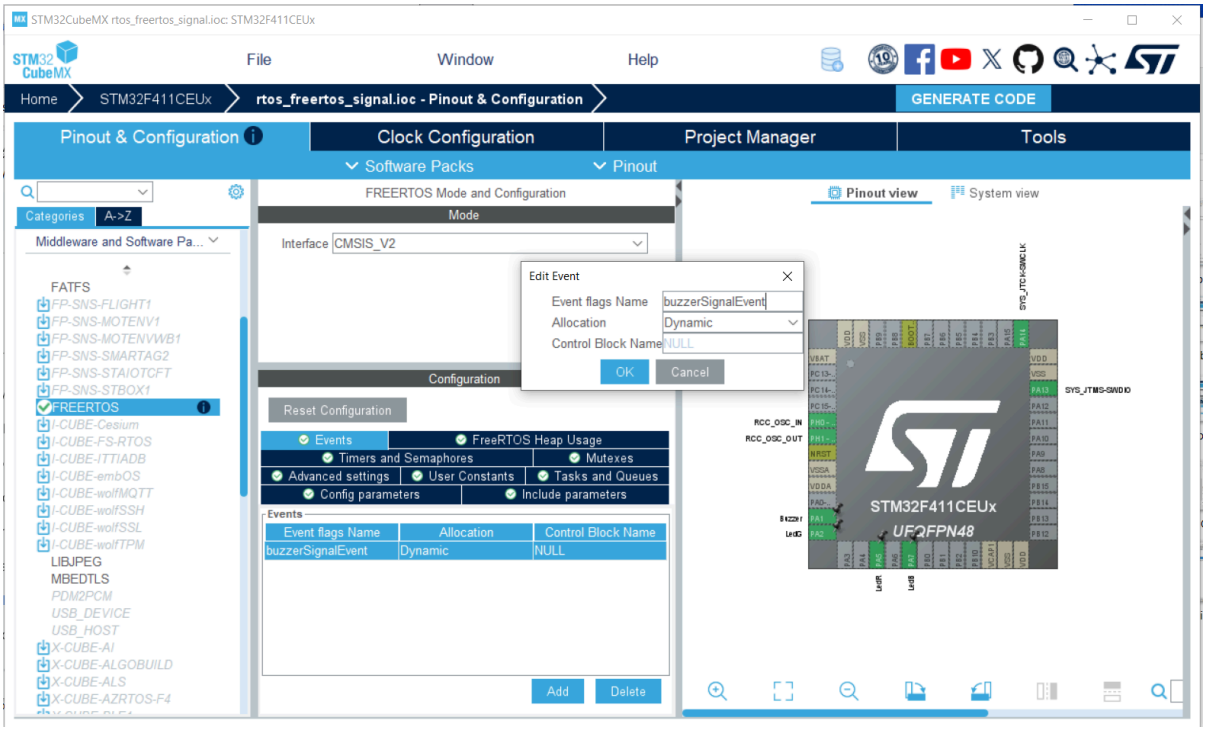
evita tanto o desperdício de memória quanto problemas como estouro de pilha. Outro aspecto relevante é que o FreeRTOS permite a utilização de **funções reentrantes** na criação de tarefas, possibilitando que uma mesma função seja utilizada por múltiplas tarefas com diferentes comportamentos, a partir da passagem de **parâmetros** distintos por meio do argumento de entrada da função de tarefa, como ilustra o projeto rtos_freertos_mutex.



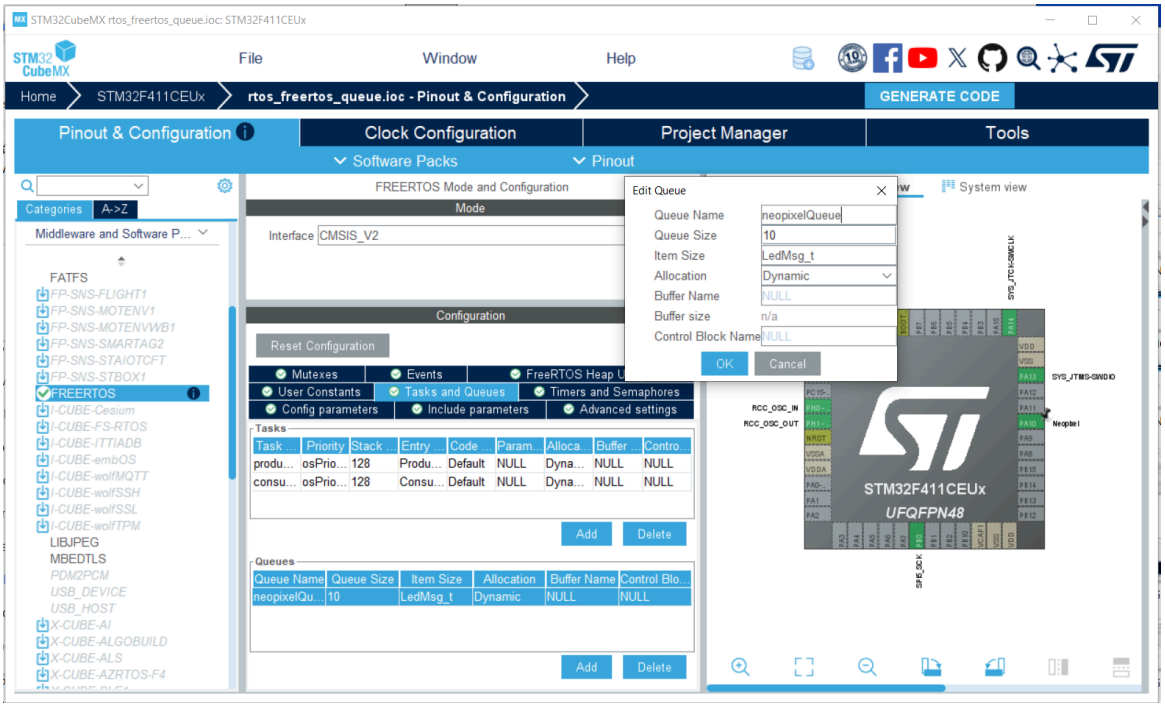
Para que múltiplas tarefas operem de forma coordenada e segura, é fundamental empregar **mecanismos de sincronização**. Entre os principais recursos estão os semáforos, *mutexes* e eventos. **Semáforos** podem ser utilizados para sinalizar a ocorrência de eventos entre tarefas ou entre interrupções e tarefas, como o projeto rtos_freertos_mutex, enquanto ***mutexes*** são especialmente importantes para proteger regiões críticas e evitar condições de corrida no acesso a recursos compartilhados.



Já os **grupos de eventos** permitem sincronizar múltiplas condições de forma eficiente, sendo úteis quando uma tarefa precisa aguardar por diferentes sinais simultaneamente, como no projeto rtos_freertos_signal.



Além da sincronização, a comunicação entre tarefas constitui outro pilar fundamental em sistemas baseados em RTOS. As **filas** são amplamente utilizadas para a troca de dados de forma segura e ordenada, assegurando que as informações sejam transmitidas entre tarefas sem risco de corrupção, como exemplificado no projeto rtos_freertos_queue. Vale destacar que o CubeMX permite a definição de tipos de dados personalizados para os itens das filas. Dependendo dos requisitos da aplicação, como volume de dados e latência, também podem ser empregados *buffers* circulares e *mailboxes*. Em sistemas mais complexos, a escolha do mecanismo de comunicação influencia diretamente o desempenho e a previsibilidade do sistema.



Assim, a configuração dos elementos de um RTOS envolve não apenas a ativação do sistema e definição das tarefas, mas também a escolha criteriosa dos mecanismos de sincronização e comunicação, assegurando que o sistema opere de maneira determinística, eficiente e confiável dentro das restrições de tempo real.

Configuração dos periféricos e geração do código no STM32CubeMX

O procedimento para execução deste item é idêntico ao apresentado no Roteiro 2, podendo ser reutilizado integralmente.

Desenvolvimento da lógica do projeto no STM32CubeIDE 2.0.0

Ao migrar uma aplicação tradicional (baseada em laço principal *superloop*) para uma arquitetura baseada em RTOS, não se trata apenas de reorganizar o código, mas de adotar um novo modelo mental de execução concorrente. Em vez de um fluxo único e sequencial, o sistema passa a ser composto por múltiplas tarefas independentes, que são escalonadas pelo *kernel* conforme prioridades e eventos.

Nesse contexto, é fundamental estruturar corretamente:

- a divisão da aplicação em tarefas,
- os mecanismos de temporização,
- a comunicação e sincronização entre tarefas e interrupções,
- e o acesso seguro a recursos compartilhados.

Além disso, um ponto frequentemente negligenciado é o tratamento de interrupções. Embora o STM32CubeMX gere automaticamente o esqueleto básico do projeto, os *callbacks* associados às ISRs **não são completamente implementados por padrão**. Esses *callbacks* devem ser escritos manualmente, tipicamente na seção USER CODE 4 do arquivo `main.c`, como mostram os projetos rtos_botoes_tratados_via_EXTI_debounce e rtos_hjoystick_interruptADCUSART. É nesses pontos que se integra o código de interrupção com os mecanismos do RTOS (como semáforos e filas), garantindo uma comunicação segura entre contexto de interrupção e tarefas.

A seguir, são apresentados alguns padrões comuns de mapeamento de código ao migrar para um RTOS.

a. Laço principal:

```
while(1) {
   tarefa1();
   tarefa2();
}
```

torna-se

```
void Task1(void *arg) {
   for(;;) {
      tarefa1();
      osDelay(100);
   }
}
void Task2(void *arg) {
   for(;;) {
      tarefa2();
      osDelay(100);
   }
```

```
}
```

b. *Delays* bloqueantes a nível do sistema

```
HAL_Delay(100);
```

substituídos por *delays* não-bloqueantes a nível do sistema

```
osDelay(100);
```

c. Variáveis globais compartilhadas por mecanismos de sincronização: Por exemnplo, o seguinte trecho de código com uma variável global *flag*

```
// Variável global usada para sinalizar evento
int flag = 0;

void loop() {
    // Verifica se o evento ocorreu
    if (flag == 1) {
        // Processa o evento
        process_event();
        // Limpa a flag
        flag = 0;
    }
}

// Interrupção de hardware
void EXTI0_IRQHandler(void) {
    flag = 1; // sinaliza que botão foi pressionado
}
```

torna-se uma tarefa que se comunica com uma rotina de interrupção por um semáforo.

```
#include "cmsis_os2.h"

osSemaphoreId_t eventSemaphore;

void Task_Event(void *arg) {
    for(;;) {
        // Espera pelo sinal da interrupção
        osSemaphoreAcquire(eventSemaphore, osWaitForever);
        // Processa o evento
        process_event();
    }
}

// Interrupção de hardware
void EXTI0_IRQHandler(void) {
    // Sinaliza a tarefa
    osSemaphoreRelease(eventSemaphore);
}
```

d. Acesso a periféricos, protegidos com `Mutex`: Por exemplo, uma transmissão bloqueante

```
HAL_UART_Transmit(...);
```

é substituída por

```
osMutexAcquire(uartMutex, osWaitForever);
HAL_UART_Transmit(...);
osMutexRelease(uartMutex);
```

Geração da documentação do código

O procedimento para execução deste item é idêntico ao apresentado no Roteiro 2, podendo ser reutilizado integralmente.

Versionamento do projeto

O procedimento para execução deste item é idêntico ao apresentado no Roteiro 2, podendo ser reutilizado integralmente.