



Modelos de Multitarefa em sistemas embarcados – Parte 3

Aula síncrona (20/05/2025)

Prof. Wilton Lacerda Silva

Executores:



Coordenação:



Iniciativa:



Sumário

- Objetivos
- Mutex
- Semáforos
- Exemplos de aplicação utilizando o FreeRTOS.

Objetivos

- Sincronização e controle de concorrência.
- Mutex (Exclusão Mútua - Mutual Exclusion).
- Semáforo
- Desenvolver alguns exemplos para fixação dos conceitos de sistemas operacionais em tempo real.



Tarefas em aplicações embarcadas

No contexto de **computação embarcada**, o conceito de tarefa assume características ainda mais específicas, pois esses sistemas geralmente operam com **recursos limitados** (como CPU, memória e energia) e precisam atender a **restrições temporais rígidas** para garantir a funcionalidade correta e em tempo hábil.

Tarefa em Computação Embarcada

Uma tarefa em sistemas embarcados é uma unidade de trabalho responsável por realizar uma operação específica ou atender a um objetivo relacionado ao funcionamento do dispositivo. Exemplos incluem:

- Leitura de sensores.
- Controle de atuadores (ex.: motores, válvulas).
- Comunicação com outros dispositivos.
- Processamento de sinais ou dados capturados.

Revisão: Conceito de tarefas

Características das Tarefas em Sistemas Embarcados

1. Restrições de Tempo (Tempo Real):

Muitas aplicações em sistemas embarcados implementam tarefas de tempo real, ou seja, devem ser concluídas dentro de prazos definidos. Elas podem ser classificadas como:

- Críticas (Hard Real-Time): A falha em atender ao prazo pode levar a danos ou falhas catastróficas (ex.: sistemas de frenagem em carros).
- Não-críticas (Soft Real-Time): A falha em atender ao prazo pode degradar o desempenho, mas sem consequências graves (ex.: atualização de uma interface de usuário).

2. Prioridade:

A prioridade das tarefas permite garantir que tarefas mais importantes sejam concluídas a tempo. Exemplo:

- Alta prioridade: Controle de temperatura em um forno industrial.
- Baixa prioridade: Registro de logs em memória.

3. Uso Eficiente de Recursos:

Como os recursos (CPU, memória, energia) são limitados, o sistema precisa gerenciar as tarefas de maneira eficiente, evitando desperdício e conflitos.

Laço Único (Loop Único ou Laço Principal)

- Nesse modelo, todas as tarefas são executadas **sequencialmente dentro de um único laço**.
- Não há preempção automática: uma tarefa só cede o controle para a próxima quando termina sua execução ou ativamente retorna o fluxo ao laço principal.
- Isso caracteriza **multitarefa cooperativa**, pois **as tarefas precisam ser bem projetadas para não bloquear o sistema** e permitir a execução das demais.

Laço com Interrupções

- Nesse modelo, o sistema conta com **interrupções de hardware**, que podem interromper a execução do laço principal para atender eventos externos (como pressionamento de botões, recebimento de dados, etc.).
- No entanto, o laço principal **ainda executa tarefas de forma cooperativa**, pois não há **um escalonador preemptivo**.
- A execução de uma tarefa só ocorre se o fluxo do programa permitir, ou seja, a tarefa não é forçada a parar para dar lugar a outra.

Multitarefa Cooperativa vs. Preemptiva

- **Multitarefa Cooperativa:** As tarefas **precisam liberar voluntariamente** a CPU.
- **Multitarefa Preemptiva:** Um escalonador ou sistema operacional **interrompe e alterna entre tarefas automaticamente**.

Conceitos Básicos de RTOS

- O que é um **RTOS (Real-Time Operating System)**?
- Funções principais de um RTOS:
 - 1-**Escalonador** (Scheduler).
 - 2-**Troca de contexto** (Context Switching).
 - 3-**Gerenciamento de tarefas** (Task Management).

Gerenciamento de Tarefas em RTOS

- Criando e executando tarefas.
- Definição de prioridades.

Filas (QUEUE) organizam o fluxo de dados entre tarefas ou periféricos. Em sistemas embarcados, são frequentemente usadas para comunicação entre tarefas do RTOS ou entre microcontroladores e sensores/atuadores.

Em um exemplo, uma fila pode armazenar **leituras de sensores** e garantir que uma tarefa de processamento pegue os dados na ordem correta. Filas são muito comuns em protocolos de comunicação, como **SPI**, **I2C** e **CAN**, garantindo que as mensagens sejam processadas sequencialmente.

Quando há poucos recursos, usar **filas circulares** (buffers circulares) para evitar realocação de memória. Também é bom reduzir o tamanho das filas para economizar RAM, garantindo que apenas os dados essenciais sejam armazenados.

Revisão: Comunicação entre tarefas ou Processos (IPC - Interprocess Communication)

Em sistemas embarcados, tarefas podem precisar trocar informações, como um processo de leitura de sensores enviando dados para um processo de controle de motores. Como esses sistemas têm **memória e processamento limitados**, os métodos de IPC precisam ser leves e eficientes.

Principais métodos em sistemas embarcados:

- **Filas de Mensagens:** Os processos trocam dados através de mensagens enviadas e recebidas.
- **Memória Compartilhada:** Os processos acessam um espaço de memória em comum. Pode ser mais rápida, mas exige mecanismos de sincronização, como mutexes ou semáforos.

Revisão: Convenções no FreeRTOS

Nomes de Variáveis

Os nomes de variáveis usados nos exemplos e argumentos do **FreeRTOS** seguem os seguintes prefixos:

- 'c' – Tipo char
- 's' – Tipo short
- 'l' – Tipo long
- 'x' – BaseType_t e outros tipos não cobertos acima

Além disso, uma variável recebe o prefixo 'u' para indicar um tipo sem sinal (unsigned). Se o valor for um ponteiro, o nome recebe o prefixo 'p'. Por exemplo, uma variável do tipo unsigned char usaria o prefixo 'uc', enquanto um ponteiro para um tipo char seria 'pc'.

Nomes de Funções

Os nomes das funções são prefixados com dois componentes:

- 1.O tipo de dado retornado pela função
- 2.O arquivo onde a função está definida

Alguns exemplos fornecidos:

- vTaskPrioritySet()** retorna void e está definida no arquivo task.c do FreeRTOS.
- xQueueReceive()** retorna uma variável do tipo BaseType_t e está definida no arquivo queue.c do FreeRTOS.
- vSemaphoreCreateBinary()** retorna void e está definida no arquivo semphr.h do FreeRTOS.

Revisão: Convenções no FreeRTOS

Nomes de Macros

Os nomes das macros são escritos em **letras maiúsculas**, exceto pelo prefixo que indica o arquivo onde são definidos.

Prefix	File	Example
port	portable.h	portMAX_DELAY
task	task.h	taskENTER_CRITICAL()
pd	projdefs.h	pdTRUE
config	FreeRTOSConfig.h	configUSE_PREEMPTION
err	projdefs.h	errQUEUE_FULL

Header File	Category	Description
FreeRTOS.h	First	Should be the first file included.
FreeRTOSConfig.h	Included by FreeRTOS.h	Not required when FreeRTOS.h has been included.
task.h	Tasks	Task support
queue.h	Queues	Queue support
semphr.h	Semaphores	Semaphore and mutex support
timers.h	Timers	Timer support

Tema principal desta apresentação: Sincronização e controle de concorrência

Aqui será discutido temas relacionados ao uso de **mutexes** e **semáforos** no **FreeRTOS**.

Estes dois recursos são fundamentais para garantir a **integridade** de dados compartilhados e **sincronizar** o acesso a recursos críticos em sistemas embarcados multitarefa.

Deveremos compreender:

- Conceito de seção crítica em sistemas multitarefa;
- Identificar situações que exigem controle de concorrência;
- Uso das funções do FreeRTOS: **xSemaphoreCreateMutex()**, **xSemaphoreTake()**, **xSemaphoreGive()**;
- Diferenciar mutexes de semáforos binários e contadores;
- Evitar problemas como **race condition**, **deadlock** e **prioridade invertida**.

Em sistemas multitarefa, duas ou mais tarefas podem ser executadas concurrentemente, seja por preempção, múltiplos núcleos ou alternância de contexto. Quando essas tarefas compartilham recursos — como **variáveis, periféricos ou barramentos** —, surgem riscos de acesso simultâneo, que podem levar a **condições de corrida** (race conditions), corrupção de dados e travamentos.

Problemas Comuns sem Proteção

- **Race condition**: resultado imprevisível ao acessar recurso compartilhado;
- **Deadlock** (impasse): duas tarefas esperam indefinidamente uma pela outra;
- **Starvation** (inanição): uma tarefa de menor prioridade nunca acessa o recurso;
- **Corrupção de dados**: inconsistência ou sobrescrita simultânea.

Concorrência em sistemas embarcados

Observação: As **race conditions** ocorrem quando duas ou mais tarefas acessam e manipulam uma mesma variável ou recurso compartilhado simultaneamente, e a ordem de execução dessas tarefas afeta o resultado final.

Exemplo clássico de race condition:

Suponha que duas tarefas (**Tarefa A** e **Tarefa B**) acessam uma variável global contador: “**contador = contador + 1**”;

Esse comando, embora pareça **atômico**, não é. Ele pode ser dividido em três etapas na prática:

Ler o valor de contador da memória. Somar 1 ao valor lido. Escrever o novo valor de volta na memória.

Se a **Tarefa A** e a **Tarefa B** executarem esse trecho **ao mesmo tempo**, pode acontecer o seguinte:

Tarefa A lê **contador = 5**. **Tarefa B** também lê **contador = 5**. Ambas somam 1, obtendo 6.
Ambas escrevem 6 de volta.

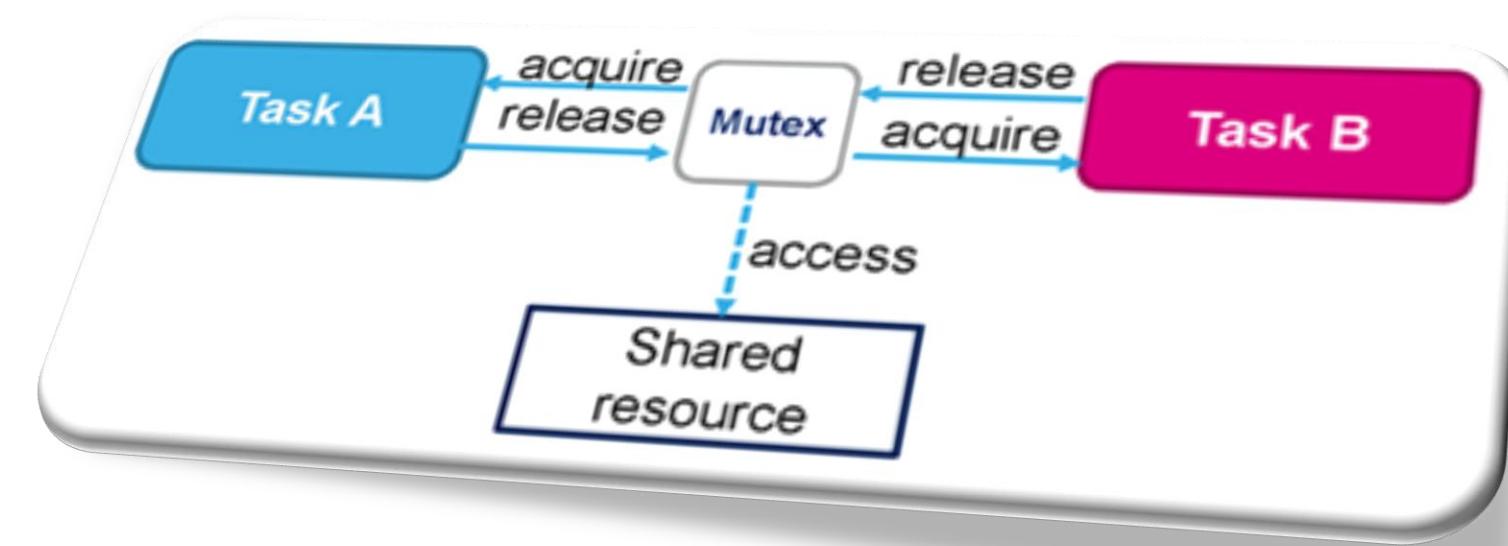
O valor correto deveria ser 7 ($5 + 1 + 1$), mas por causa da condição de corrida, o resultado final é incorreto: 6

Mutex (Exclusão Mútua - Mutual Exclusion)

Um mutex (Mutual Exclusion) é um objeto especial utilizado em sistemas operacionais de tempo real, como o **FreeRTOS**, para garantir que apenas uma tarefa por vez possa acessar um recurso compartilhado. Mutexes são fundamentais para proteger regiões críticas do código e evitar problemas como race conditions e corrupção de dados.

O mutex impede que duas tarefas accedam ao mesmo recurso ao mesmo tempo, evitando a corrupção de dados.

Por exemplo, se um processador embarcado accessa um buffer de comunicação UART enquanto outra tarefa ainda está escrevendo nele, pode ocorrer erro de transmissão.



Mutex: Criando um Mutex

A criação de um **mutex** é feita com a função:

```
SemaphoreHandle_t xMutex;  
xMutex = xSemaphoreCreateMutex();
```

Obs: É importante criar o **mutex** **antes** de utilizá-lo nas tarefas, geralmente dentro da função main() ou antes de iniciar o scheduler.

SemaphoreHandle_t é um tipo de ponteiro usado no FreeRTOS para referenciar semáforos e mutexes.

Observação: Antes do FreeRTOS v8.2.0, esse tipo era declarado como **xSemaphoreHandle** **xMutex;**, por isso ainda é comum encontrar códigos utilizando essa forma. Ambos são equivalentes (são **typedefs**), mas **SemaphoreHandle_t** é a nomenclatura moderna, mais clara e recomendada pelas versões atuais da documentação.

Mutex: Usando um Mutex “xSemaphoreTake()”

A função **xSemaphoreTake()** é utilizada para solicitar o acesso ao recurso protegido. Se o **mutex** estiver disponível, ele será adquirido e a função retornará pdTRUE. Se estiver ocupado por outra tarefa, a tarefa atual será bloqueada até que o **mutex** seja liberado ou o tempo de espera expire.

Sintaxe:

```
if (xSemaphoreTake(xMutex, tempo)==pdTRUE) {  
    // Seção crítica  
    xSemaphoreGive(xMutex);  
}
```

Parâmetro **tempo**:

- 0: não espera, retorna imediatamente se não conseguir o mutex
- **portMAX_DELAY**: espera indefinidamente até conseguir
- **pdMS_TO_TICKS(ms)**: espera pelo tempo especificado em milissegundos

Sempre verificar se a função retornou pdTRUE antes de acessar a seção crítica.

Mutex: Usando um Mutex “xSemaphoreGive()”

Após concluir a operação na região crítica, a tarefa deve liberar o mutex com **xSemaphoreGive()**. Isso permite que outras tarefas possam adquirir o recurso.

Apenas a tarefa que realizou o **xSemaphoreTake()** deve chamar **xSemaphoreGive()**. Caso contrário, o sistema poderá apresentar falhas.

Sintaxe:

```
if (xSemaphoreTake(xMutex, tempo)==pdTRUE) {  
    // Seção crítica  
    xSemaphoreGive(xMutex);  
}
```

Semáforo: Definição

Semáforos também são usados para sincronizar tarefas e controlar o acesso a múltiplos recursos simultaneamente. Diferente do **mutex**, que permite apenas um acesso por vez, o semáforo pode permitir múltiplos acessos conforme sua contagem.
Resumidamente: No **FreeRTOS**, podem ser usados para:

- Sincronizar tarefas
- Controlar o acesso a múltiplas instâncias de um recurso
- Sinalizar eventos a partir de interrupções

Por exemplo, um **semáforo binário** pode ser usado para avisar que dados estão prontos no ADC (Conversor Analógico-Digital), evitando que uma tarefa fique constantemente verificando (polling).

Em outro exemplo, um **semáforo de contagem** pode controlar o acesso a um banco de registros de sensores que pode ser lido por várias tarefas, mas tem um limite de acessos simultâneos. Uma dica é usar semáforos binários sempre que possível, pois são mais leves que os de contagem.

Semáforo: Tipos

Semáforo Binário - só assume os estados 0 e 1. Útil para sinalização.

Exemplo: Ele pode ser usado para avisar que dados estão prontos no ADC (Conversor Analógico-Digital), evitando que uma tarefa fique constantemente verificando (polling).

Semáforo de Contagem - permite múltiplos recursos simultâneos

Exemplo: Um **semáforo de contagem** pode controlar o acesso a um banco de registros de sensores que pode ser lido por várias tarefas, mas tem um limite de acessos simultâneos.

Semáforo: Criando um semáforo

A criação de semáforos é feita assim:

```
xSemaphoreHandle_t xSem;  
xSem = xSemaphoreCreateBinary(); // cria um semáforo binário  
                                (podendo assumir 1 = disponível e 0 = não disponível)  
xSem = xSemaphoreCreateCounting(5, 0); // Cria um semáforo de contagem  
(inicia com valor 0, e conta quantas vezes xSemaphoreGive() foi chamado sem que  
xSemaphoreTake tenha pego ainda.)
```

Tabela comparativa

Critério	Mutex	Semáforo
Propósito principal	Exclusão Mútua	Sinalização e sincronização
Controle de prioridade	Sim (herança de prioridade)	Não
Contagem de recursos	Não (1 recurso por vez)	Sim (contagem > 1)
Usando entre interrupções	Não recomendado	Sim (binário)
Funções típicas	xSemaphoreCreateMutex()	xSemaphoreCreateBinary() xSemaphoreCreateConting()
Função de espera	xSemaphoreTake()	xSemaphoreTeke()
Indicador de liberação	xSemaphoreGive()	xSemaphoreGive()

No Mutex: (herança de prioridade)

Isso significa que quando uma tarefa de baixa prioridade está usando um recurso protegido por mutex, e uma tarefa de alta prioridade tenta acessar esse mesmo recurso, ocorre o seguinte:

- O sistema aumenta temporariamente a prioridade da tarefa de baixa prioridade (**herança de prioridade**) para que ela termine logo e libere o mutex.
- Isso evita um problema chamado **priority inversion** (inversão de prioridade), onde uma tarefa importante ficaria bloqueada por uma tarefa menos importante.

Tabela comparativa: Observações

No Mutex: (herança de prioridade)

Isso significa que **quando uma tarefa de baixa prioridade está usando um recurso protegido por mutex, e uma tarefa de alta prioridade tenta acessar esse mesmo recurso**, ocorre o seguinte:

- O sistema **aumenta temporariamente** a prioridade da tarefa de baixa prioridade (**herança de prioridade**) para que ela **termine logo e libere o mutex**.
- Isso **evita um problema chamado *priority inversion* (inversão de prioridade)**, onde uma tarefa importante ficaria bloqueada por uma tarefa menos importante.

No Mutex: Usado entre interrupções

Não recomendado porque o mutex usa **mecanismos de prioridade e de bloqueio, que não funcionam corretamente em ISRs** (Interrupt Service Routines).

As ISR **não devem bloquear**, e não há troca de contexto nelas — logo, usar **xSemaphoreTake()** de um mutex **dentro de uma ISR é errado e perigoso**.

**Exemplos de programas desenvolvidos no FreeRTOS
para aprendizagem de Mutex.**



Este programa demonstra o uso de **mutex** (exclusão mútua) no sistema operacional FreeRTOS, rodando no microcontrolador Raspberry Pi Pico com a placa BitDogLab.

Objetivo: Proteger o acesso ao display OLED SSD1306 quando duas tarefas concorrentes tentam escrever ao mesmo tempo, evitando conflitos ou falhas na exibição.

Tarefa 1: Escreve “**Tarefa 1 ativa**” no display a cada 1 segundo.

Tarefa 2: Escreve “**Tarefa 2 ativa**” no display a cada 1,5 segundos.

As duas tarefas competem pelo uso do display. O acesso é sincronizado com um **mutex**, garantindo que apenas uma tarefa por vez possa modificar o conteúdo da tela.

Um **mutex** chamado **xDisplayMutex** é criado com a função: **xDisplayMutex = xSemaphoreCreateMutex();**

Cada tarefa tenta obter o **mutex** com: **if (xSemaphoreTake(xDisplayMutex, portMAX_DELAY) == pdTRUE)**

Se conseguir, escreve no display e depois libera com: **xSemaphoreGive(xDisplayMutex);**

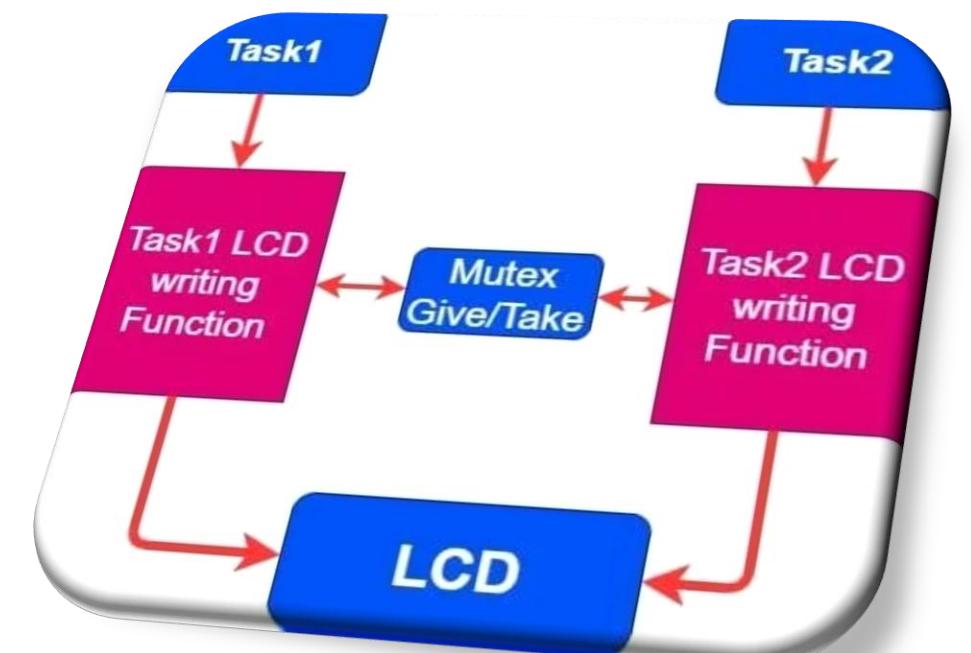
Simulação de duas tarefas que escrevem no display OLED SSD1306 utilizando um **mutex** para evitar conflitos de escrita.

```
C MutexDisplay.c X
C MutexDisplay.c > I2C_PORT
14 #include "pico/stlolib.h"
15 #include "hardware/i2c.h"
16 #include "hardware/gpio.h"
17 #include "lib/ssd1306.h"
18 #include "FreeRTOS.h"
19 #include "task.h"
20 #include "semphr.h"
21
22 #define I2C_PORT i2c1
23 #define I2C_SDA 14
24 #define I2C_SCL 15
25 #define ENDERECO 0x3C
26
27 ssd1306_t ssd;
28 SemaphoreHandle_t xDisplayMutex;
```

```
32 // Tarefa 1: escreve uma mensagem no display
33 void vTask1(void *params) {
34     while (true) {
35         if (xSemaphoreTake(xDisplayMutex, portMAX_DELAY)==pdTRUE) {
36             ssd1306_fill(&ssd, 0);
37             ssd1306_draw_string(&ssd, "Tarefa 1 ativa", 5, 20);
38             ssd1306_send_data(&ssd);
39             xSemaphoreGive(xDisplayMutex);
40         }
41         vTaskDelay(pdMS_TO_TICKS(1000));
42     }
43 }
```



```
45 // Tarefa 2: escreve outra mensagem
46 void vTask2(void *params) {
47     while (true) {
48         if (xSemaphoreTake(xDisplayMutex, portMAX_DELAY==pdTRUE)) {
49             ssd1306_fill(&ssd, 1);
50             ssd1306_draw_string(&ssd, "Tarefa 2 ativa", 5, 40);
51             ssd1306_send_data(&ssd);
52             xSemaphoreGive(xDisplayMutex);
53         }
54         vTaskDelay(pdMS_TO_TICKS(1500));
55     }
56 }
```

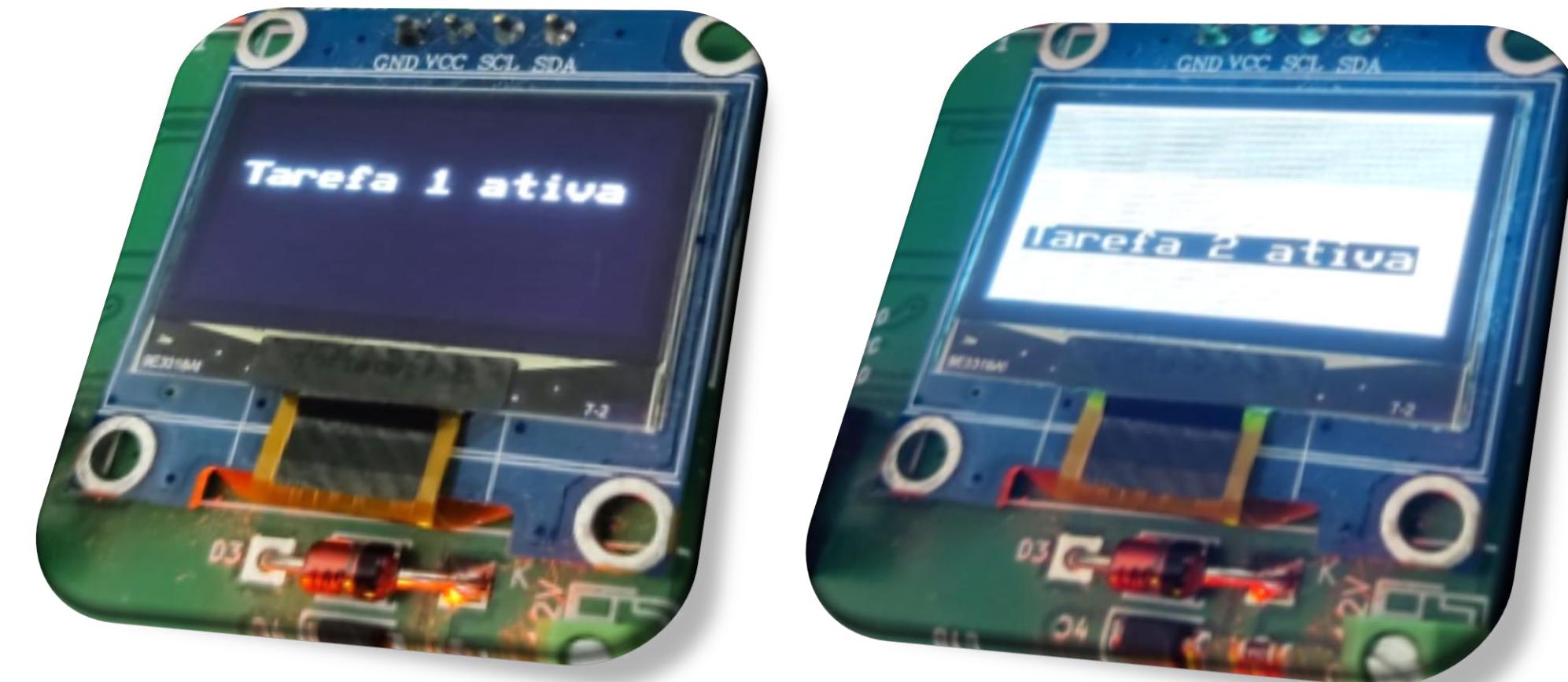


FreeRTOS no RP2040

Exemplo 1: Mutex

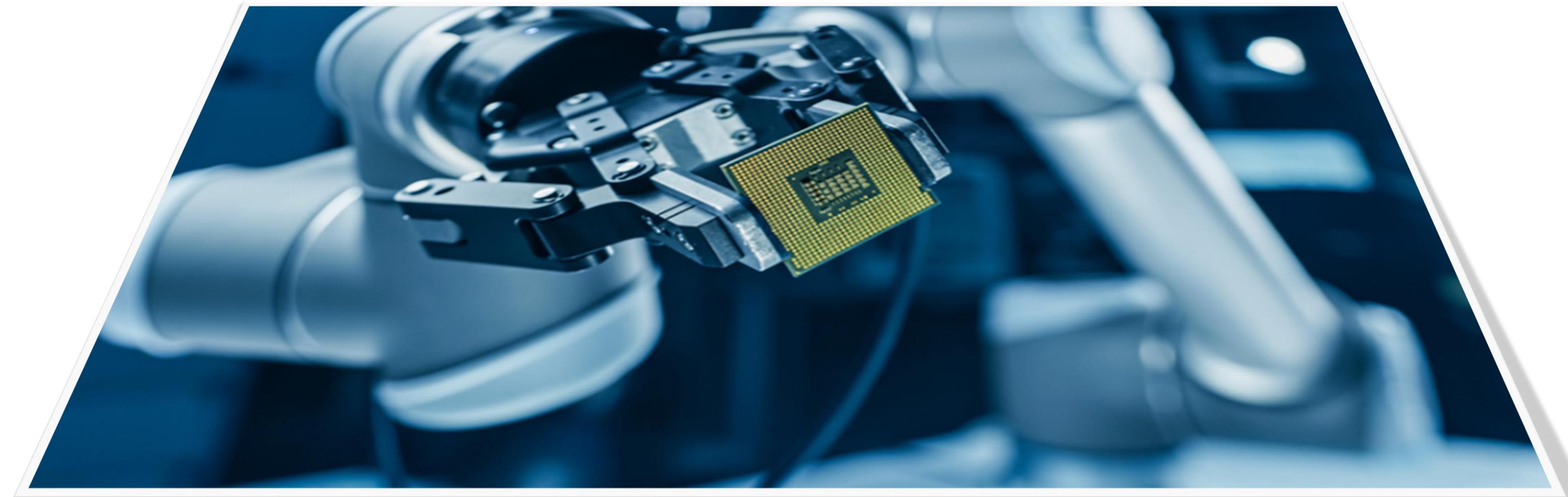
```
57 // Modo BOOTSEL com botão B
58 #include "pico/bootrom.h"
59 #define botaoB 6
60 // Interrupção: coloca o Pico em modo BOOTSEL via botão B
61 void gpio_irq_handler(uint gpio, uint32_t events) {
62     reset_usb_boot(0, 0); // Entra no modo bootloader
63 }
64
65 int main() {
66     stdio_init_all();
67
68     // Configura botão BOOTSEL
69     gpio_init(botaoB);
70     gpio_set_dir(botaoB, GPIO_IN);
71     gpio_pull_up(botaoB);
72     gpio_set_irq_enabled_with_callback(botaoB, GPIO_IRQ_EDGE_RISING, gpio_irq_handler);
73
74     // Inicializa I2C e display OLED
75     i2c_init(I2C_PORT, 400 * 1000);
76     gpio_set_function(I2C_SDA, GPIO_FUNC_I2C);
77     gpio_set_function(I2C_SCL, GPIO_FUNC_I2C);
78     gpio_pull_up(I2C_SDA);
79     gpio_pull_up(I2C_SCL);
80
81     ssd1306_init(&ssd, WIDTH, HEIGHT, false, ENDERECO);
82     ssd1306_config(&ssd);
83     ssd1306_send_data(&ssd);
```

```
86     // Cria o mutex do display
87     xDisplayMutex = xSemaphoreCreateMutex();
88
89     // Cria as tarefas
90     xTaskCreate(vTask1, "Task1", configMINIMAL_STACK_SIZE + 128, NULL, 1, NULL);
91     xTaskCreate(vTask2, "Task2", configMINIMAL_STACK_SIZE + 128, NULL, 1, NULL);
92
93     // Inicia o agendador
94     vTaskStartScheduler();
95     panic_unsupported();
96 }
97
```



FreeRTOS no RP2040

**Exemplo de programa desenvolvido no FreeRTOS para
aprendizagem de Semáforo Binário.**



Exemplo com Semáforo Binário e Display OLED

Este programa demonstra o uso de semáforo binário no FreeRTOS para sincronização entre uma interrupção de botão e uma tarefa, utilizando a BitDogLab.

Objetivo: Permitir que uma interrupção gerada por um botão A (GPIO 5) sinalize uma **tarefa dedicada**, a qual atualiza o display OLED SSD1306 com uma mensagem sempre que o botão é pressionado.

Funcionamento: O sistema inicializa e o display mostra: “**Aguardando evento...**”

Quando o botão A da GPIO 5 é pressionado, ocorre uma interrupção de hardware.

A função de interrupção (ISR) libera um semáforo binário com: **xSemaphoreGiveFromISR()** (Semáforo em 1)

A tarefa **vButtonTask**, que está bloqueada esperando o semáforo com:

xSemaphoreTake(xButtonSem, portMAX_DELAY) é despertada e executa a ação de mostrar no display “**Botao Pressionado!**”. Neste momento ele volta a ficar indisponível (Semáforo em 0)

Aguarda 4 segundos (**vTaskDelay**) e retorna à mensagem padrão: “**Aguardando evento...**”

Simulação de um **semáforo** com um botão. O **botão A** aciona o semáforo.
O **semáforo** é representado por um display OLED SSD1306.

```
C Semaforo01Display.c x
C Semaforo01Display.c > ...
16
17 #include "pico/stdc.h"
18 #include "hardware/i2c.h"
19 #include "hardware/gpio.h"
20 #include "lib/ssd1306.h"
21 #include "FreeRTOS.h"
22 #include "task.h"
23 #include "semphr.h"
24 #include "pico/bootrom.h"
25
26 #define I2C_PORT i2c1
27 #define I2C_SDA 14
28 #define I2C_SCL 15
29 #define ENDERECO 0x3C
30
31 #define BOTAO_A 5
32 #define BOTAO_B 6
33
34 ssd1306_t ssd;
35 SemaphoreHandle_t xButtonSem;
```

```
// ISR do botão
void gpio_callback(uint gpio, uint32_t events) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE; // Nenhum contexto de tarefa foi despertado
    xSemaphoreGiveFromISR(xButtonSem, &xHigherPriorityTaskWoken); // Libera o semáforo
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken); // Troca o contexto da tarefa
}

// Tarefa principal: espera semáforo e atualiza o display
void vButtonTask(void *params) {
    // Tela inicial
    ssd1306_fill(&ssd, 0);
    ssd1306_draw_string(&ssd, "Aguardando evento...", 5, 25);
    ssd1306_send_data(&ssd);

    while (true) {
        // Tarefa está bloqueada até que o semáforo seja liberado
        // O semáforo é liberado na ISR do botão
        if (xSemaphoreTake(xButtonSem, portMAX_DELAY) == pdTRUE) {
            // Exibe mensagem por 1 segundo
            ssd1306_fill(&ssd, 1);
            ssd1306_draw_string(&ssd, "Botao Pressionado!", 5, 25);
            ssd1306_send_data(&ssd);
            vTaskDelay(pdMS_TO_TICKS(4000));
        }

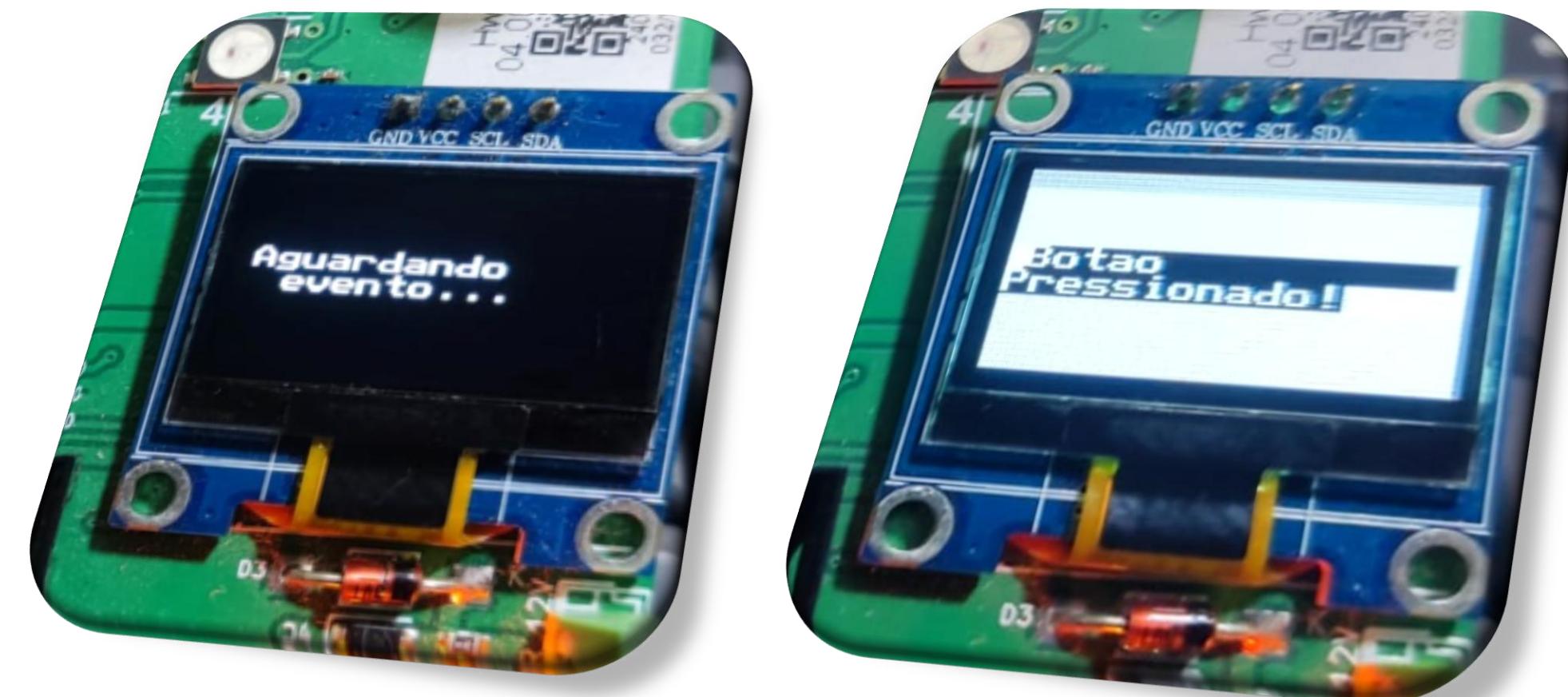
        // Retorna à mensagem de espera
        ssd1306_fill(&ssd, 0);
        ssd1306_draw_string(&ssd, "Aguardando evento...", 5, 25);
        ssd1306_send_data(&ssd);
    }
}
```

FreeRTOS no RP2040

Exemplo 2: Semáforo

```
67 // ISR para BOOTSEL e botão de evento
68 void gpio_irq_handler(uint gpio, uint32_t events) {
69     if (gpio == BOTAO_B) {
70         reset_usb_boot(0, 0);
71     } else if (gpio == BOTAO_A) {
72         gpio_callback(gpio, events);
73     }
74 }
75
76 int main() {
77     stdio_init_all();
78
79     // Inicialização do display
80     i2c_init(I2C_PORT, 400 * 1000);
81     gpio_set_function(I2C_SDA, GPIO_FUNC_I2C);
82     gpio_set_function(I2C_SCL, GPIO_FUNC_I2C);
83     gpio_pull_up(I2C_SDA);
84     gpio_pull_up(I2C_SCL);
85     ssd1306_init(&ssd, WIDTH, HEIGHT, false, ENDERECO, I2C_PORT);
86     ssd1306_config(&ssd);
87     ssd1306_send_data(&ssd);
88
89     // Botão do evento
90     gpio_init(BOTAO_A);
91     gpio_set_dir(BOTAO_A, GPIO_IN);
92     gpio_pull_up(BOTAO_A);
93
94     // Botão BOOTSEL
95     gpio_init(BOTAO_B);
96     gpio_set_dir(BOTAO_B, GPIO_IN);
97     gpio_pull_up(BOTAO_B);
```

```
98
99
100
101
102 // Cria semáforo
103 xButtonSem = xSemaphoreCreateBinary();
104
105
106
107
108 vTaskStartScheduler();
109 panic_unsupported();
110 }
```



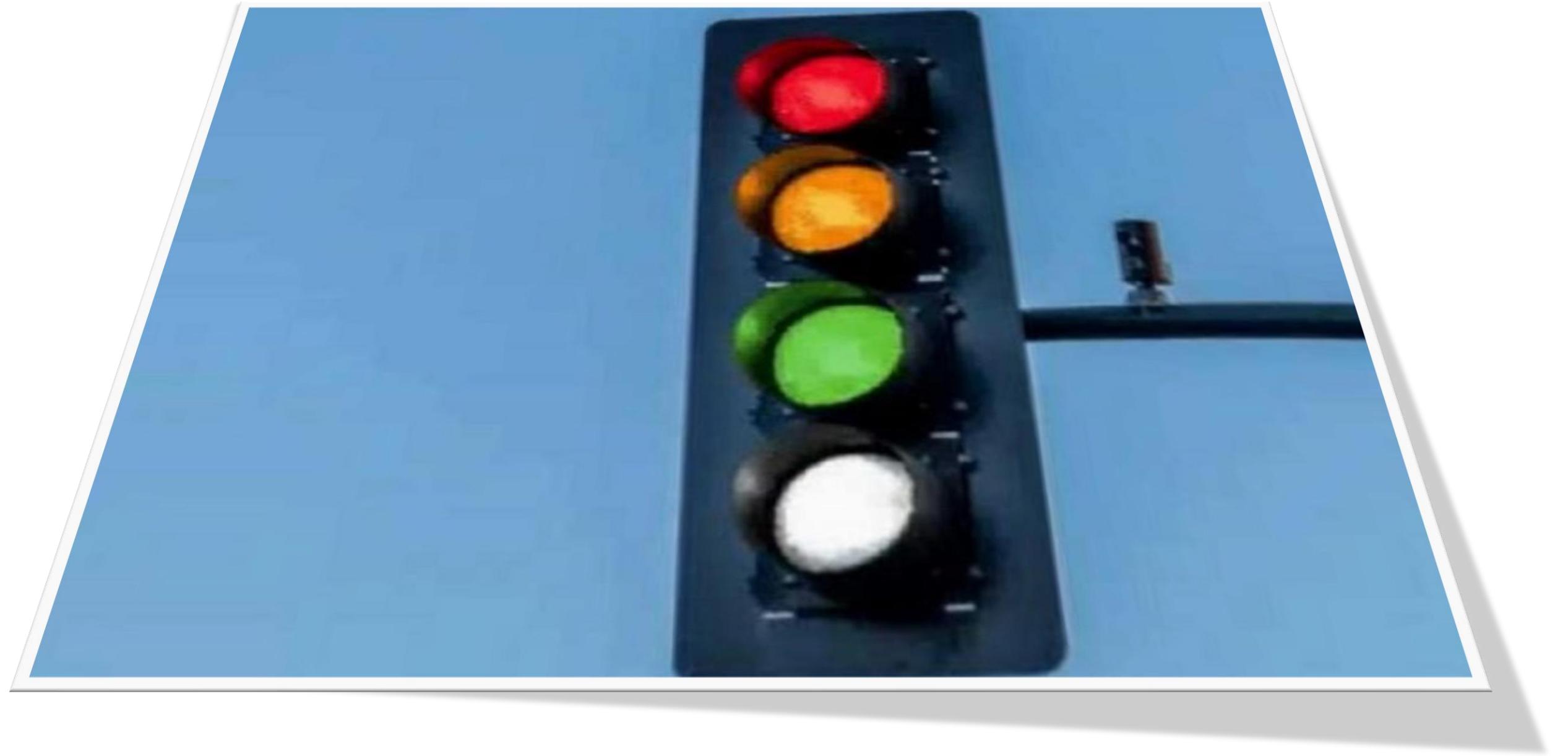
Observações:

BaseType_t Usado para receber valores como **pdTRUE** ou **pdFALSE**

xHigherPriorityTaskWoken : É uma variável de controle usada em ISRs para informar ao kernel do FreeRTOS se: “**A tarefa que foi acordada por essa interrupção tem prioridade maior do que a tarefa atual?**” Se **sim**, o sistema precisa realizar uma troca de contexto imediatamente ao sair da ISR.

O **semáforo binário** volta automaticamente a **0** quando uma tarefa consegue pegá-lo com sucesso usando **xSemaphoreTake()**.

**Exemplo de programa desenvolvido no FreeRTOS para
aprendizagem de Semáforo Contador.**



Exemplo com Semáforo de contagem “Registro de Eventos”

Este programa exemplifica o uso de um **semáforo de contagem** (counting semaphore) no FreeRTOS, aplicado na placa BitDogLab. O sistema também utiliza um display para exibir mensagens ao usuário.

Objetivo: Registrar e processar múltiplos eventos gerados pelo botão A (GPIO 5).

Cada vez que o botão é pressionado, o programa contabiliza o evento e atualiza o display com o total de eventos processados, mesmo que várias pressões ocorram em sequência rápida.

Semáforo de contagem: controla a fila de eventos aguardando processamento. Tarefa única (**vContadorTask**): consome os eventos e atualiza o display.

Funcionamento do Programa: O sistema inicializa e exibe: "**Aguardando evento...**" no display.

Quando o botão A (GPIO 5) é pressionado a ISR é acionada. O semáforo de contagem é incrementado com **xSemaphoreGiveFromISR()**. O semáforo pode acumular vários eventos consecutivos (até o limite definido, neste caso 10). A tarefa **vContadorTask** fica bloqueada em **xSemaphoreTake(...)** até que um evento esteja disponível.

Ao receber o semáforo, incrementa a variável **eventosProcessados**. Exibe no display **Evento recebido!** **Eventos: N.** Depois aguarda 1.5 segundos simulando tempo de processamento. Retorna à mensagem "**Aguardando evento...**".

FreeRTOS no RP2040

Exemplo 3: Semáforo de contagem

```
c Semaforo02Display.c x
c Semaforo02Display.c > main0
11
12 #include "pico/stdlib.h"
13 #include "hardware/i2c.h"
14 #include "hardware/gpio.h"
15 #include "lib/ssd1306.h"
16 #include "FreeRTOS.h"
17 #include "task.h"
18 #include "semphr.h"
19 #include "pico/bootrom.h"
20 #include "stdio.h"
21
22 #define I2C_PORT i2c1
23 #define I2C_SDA 14
24 #define I2C_SCL 15
25 #define ENDERECO 0x3C
26
27 #define BOTAO_A 5 // Gera evento
28 #define BOTAO_B 6 // BOOTSEL
29
30 ssd1306_t ssd;
31 SemaphoreHandle_t xContadorSem;
32 uint16_t eventosProcessados = 0;
33
```

```
42 // Tarefa que consome eventos e mostra no display
43 void vContadorTask(void *params)
44 {
45     char buffer[32];
46
47     // Tela inicial
48     ssd1306_fill(&ssd, 0);
49     ssd1306_draw_string(&ssd, "Aguardando ", 5, 25);
50     ssd1306_draw_string(&ssd, " evento...", 5, 34);
51     ssd1306_send_data(&ssd);
52
53     while (true)
54     {
55         // Aguarda semáforo (um evento)
56         if (xSemaphoreTake(xContadorSem, portMAX_DELAY) == pdTRUE)
57         {
58             eventosProcessados++;
59
59             // Atualiza display com a nova contagem
60             ssd1306_fill(&ssd, 0);
61             sprintf(buffer, "Eventos: %d", eventosProcessados);
62             ssd1306_draw_string(&ssd, "Evento ", 5, 10);
63             ssd1306_draw_string(&ssd, "recebido!", 5, 19);
64             ssd1306_draw_string(&ssd, buffer, 5, 44);
65             ssd1306_send_data(&ssd);
66
67             // Simula tempo de processamento
68             vTaskDelay(pdMS_TO_TICKS(1500));
69
70             // Retorna à tela de espera
71             ssd1306_fill(&ssd, 0);
72             ssd1306_draw_string(&ssd, "Aguardando ", 5, 25);
73             ssd1306_draw_string(&ssd, " evento...", 5, 34);
74             ssd1306_send_data(&ssd);
75
76         }
77     }
78 }
```

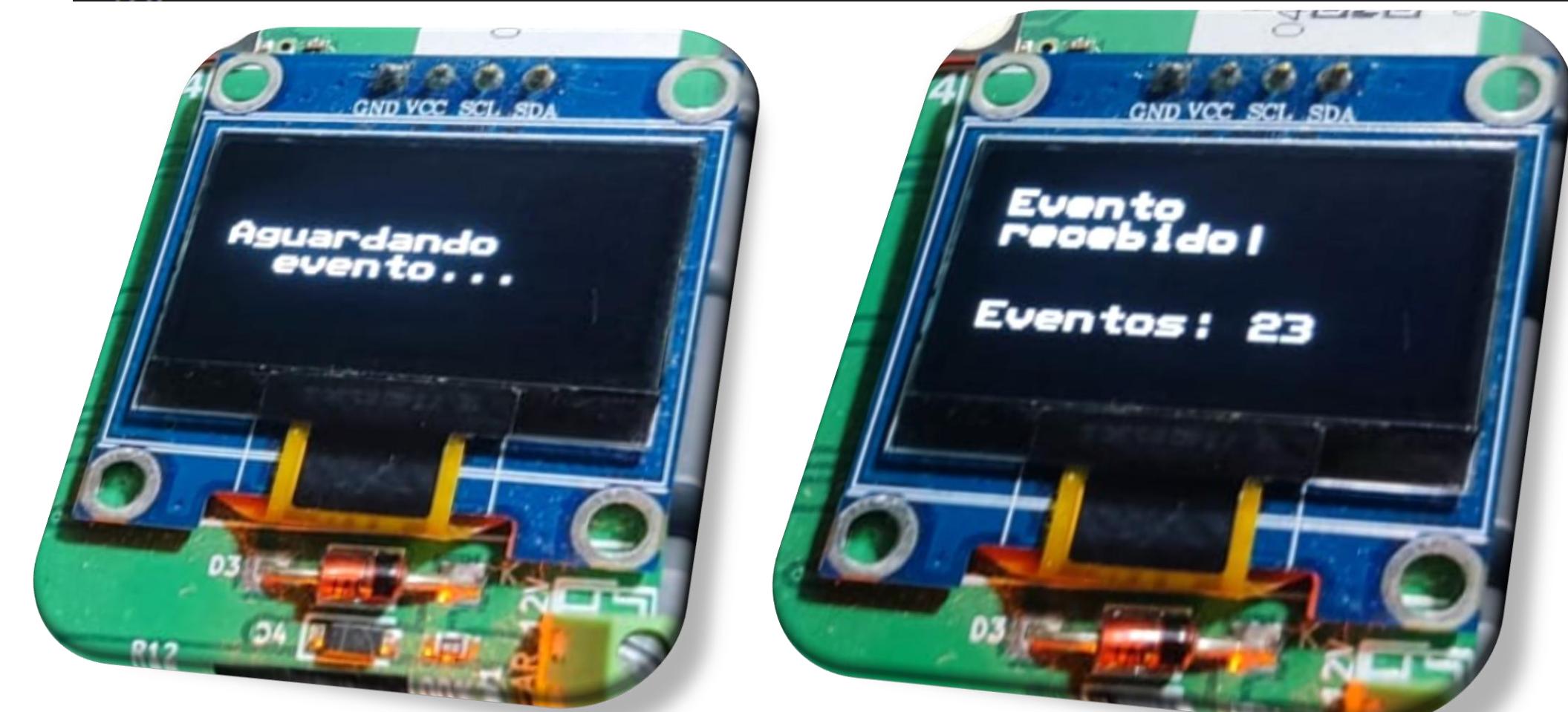
```
34 // ISR do botão A (incrementa o semáforo de contagem)
35 void gpio_callback(uint gpio, uint32_t events)
36 {
37     BaseType_t xHigherPriorityTaskWoken = pdFALSE;
38     xSemaphoreGiveFromISR(xContadorSem, &xHigherPriorityTaskWoken);
39     portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
40 }
```

FreeRTOS no RP2040

Exemplo 3: Semáforo de contagem

```
80 // ISR para BOOTSEL e botão de evento
81 void gpio_irq_handler(uint gpio, uint32_t events)
82 {
83     if (gpio == BOTAO_B)
84     {
85         reset_usb_boot(0, 0);
86     }
87     else if (gpio == BOTAO_A)
88     {
89         gpio_callback(gpio, events);
90     }
91 }
92
93 int main()
94 {
95     stdio_init_all();
96
97     // Inicialização do display
98     i2c_init(I2C_PORT, 400 * 1000);
99     gpio_set_function(I2C_SDA, GPIO_FUNC_I2C);
100    gpio_set_function(I2C_SCL, GPIO_FUNC_I2C);
101    gpio_pull_up(I2C_SDA);
102    gpio_pull_up(I2C_SCL);
103    ssd1306_init(&ssd, WIDTH, HEIGHT, false, ENDERECO, I2C_PORT);
104    ssd1306_config(&ssd);
105    ssd1306_send_data(&ssd);
106
107    // Configura os botões
108    gpio_init(BOTAO_A);
109    gpio_set_dir(BOTAO_A, GPIO_IN);
110    gpio_pull_up(BOTAO_A);
111
112    gpio_init(BOTAO_B);
113    gpio_set_dir(BOTAO_B, GPIO_IN);
114    gpio_pull_up(BOTAO_B);
```

```
115
116     gpio_set_irq_enabled_with_callback(BOTAO_A, GPIO_IRQ_EDGE_FALL, true, &gpio_irq_handler);
117     gpio_set_irq_enabled(BOTAO_B, GPIO_IRQ_EDGE_FALL, true);
118
119     // Cria semáforo de contagem (máximo 10, inicial 0)
120     xContadorSem = xSemaphoreCreateCounting(10, 0);
121
122     // Cria tarefa
123     xTaskCreate(vContadorTask, "ContadorTask", configMINIMAL_STACK_SIZE + 128, NULL, 1, NULL);
124
125     vTaskStartScheduler();
126
127 }
128
```



Neste exemplo, o semáforo de contagem captura todos os pulsos, inclusive os gerados por bounce mecânico do botão A. Isso evidencia o efeito do rebote e mostra a importância de implementar algum tipo de tratamento

Tarefa



Tarefa:

Painel de Controle Interativo com Acesso Concorrente

Enunciado

Consolidar os conhecimentos adquiridos sobre sincronização de tarefas com **mutex** e **semáforos** no FreeRTOS por meio do desenvolvimento de um sistema embarcado funcional e acessível.

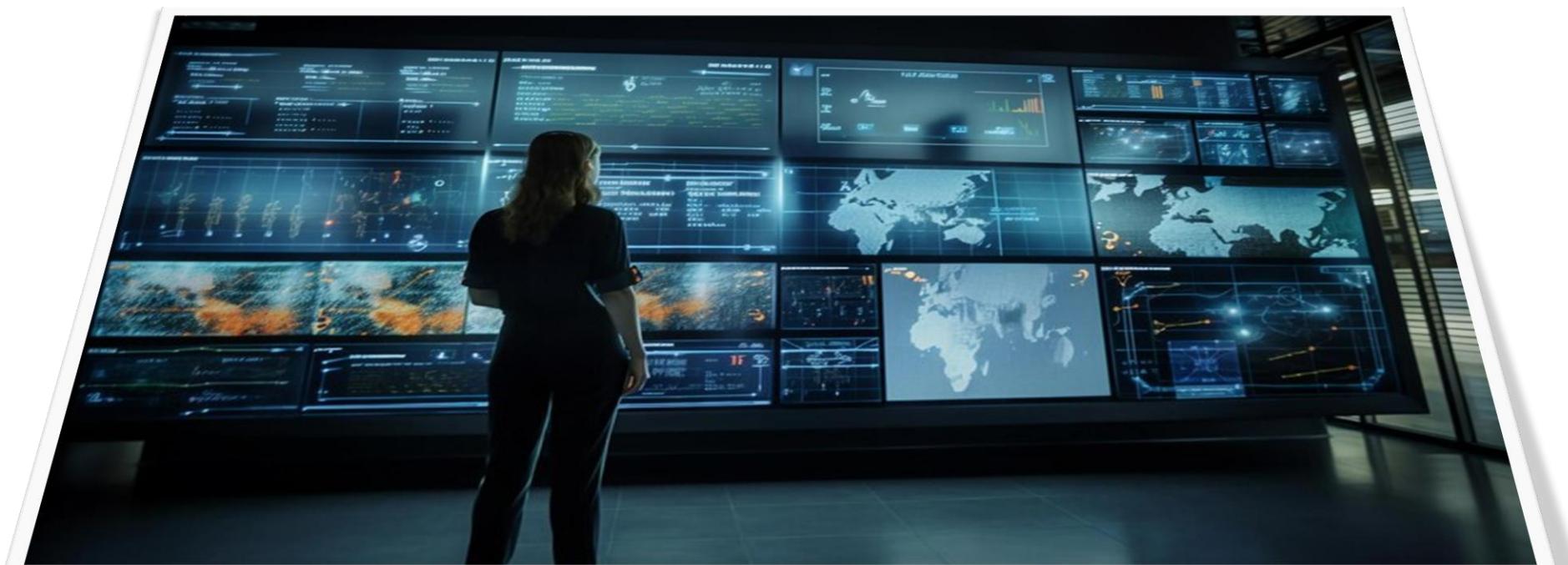
Descrição do Projeto

Você deverá implementar um painel de controle interativo, simulando o controle de acesso de usuários a um determinado espaço (ex: laboratório, biblioteca, refeitório). O sistema será baseado na placa BitDogLab com o RP2040 e utilizará:

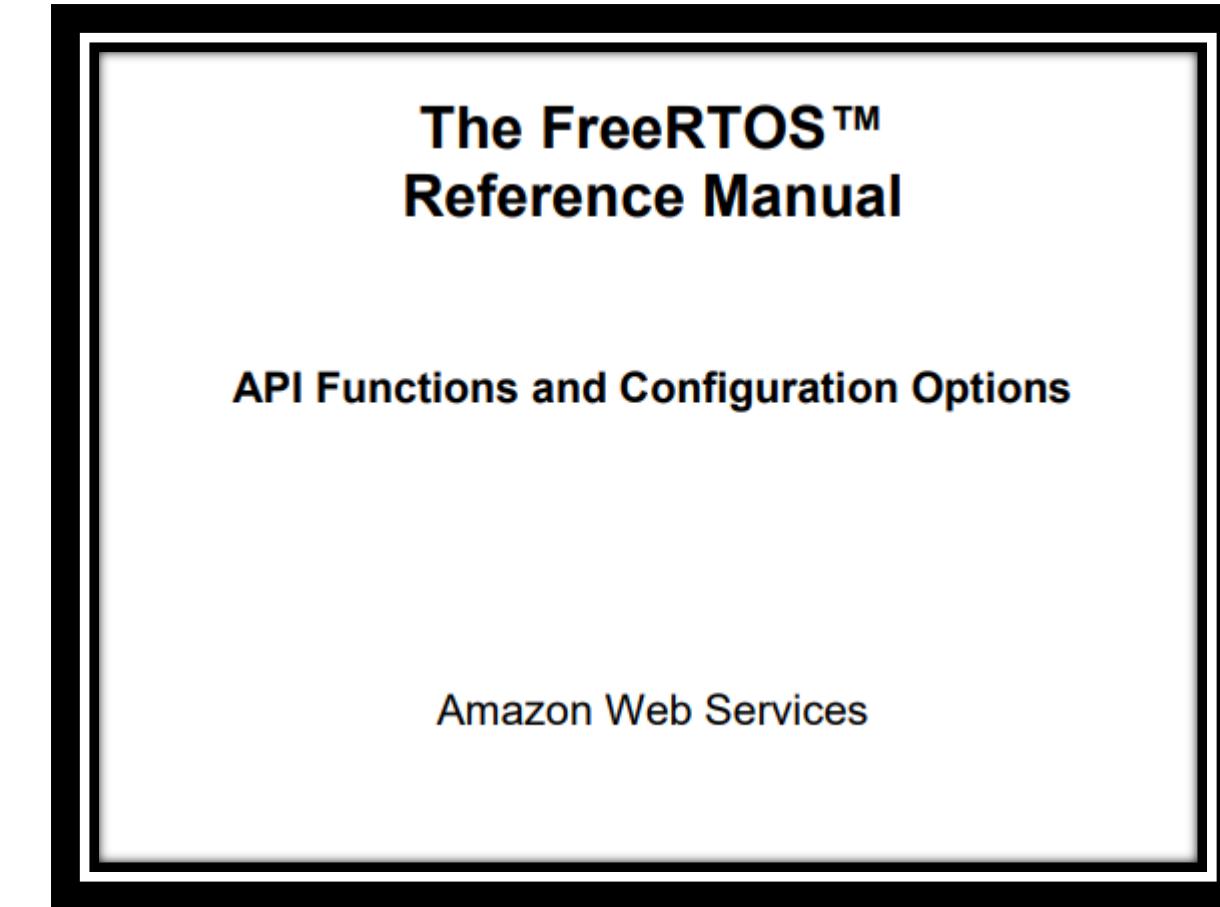
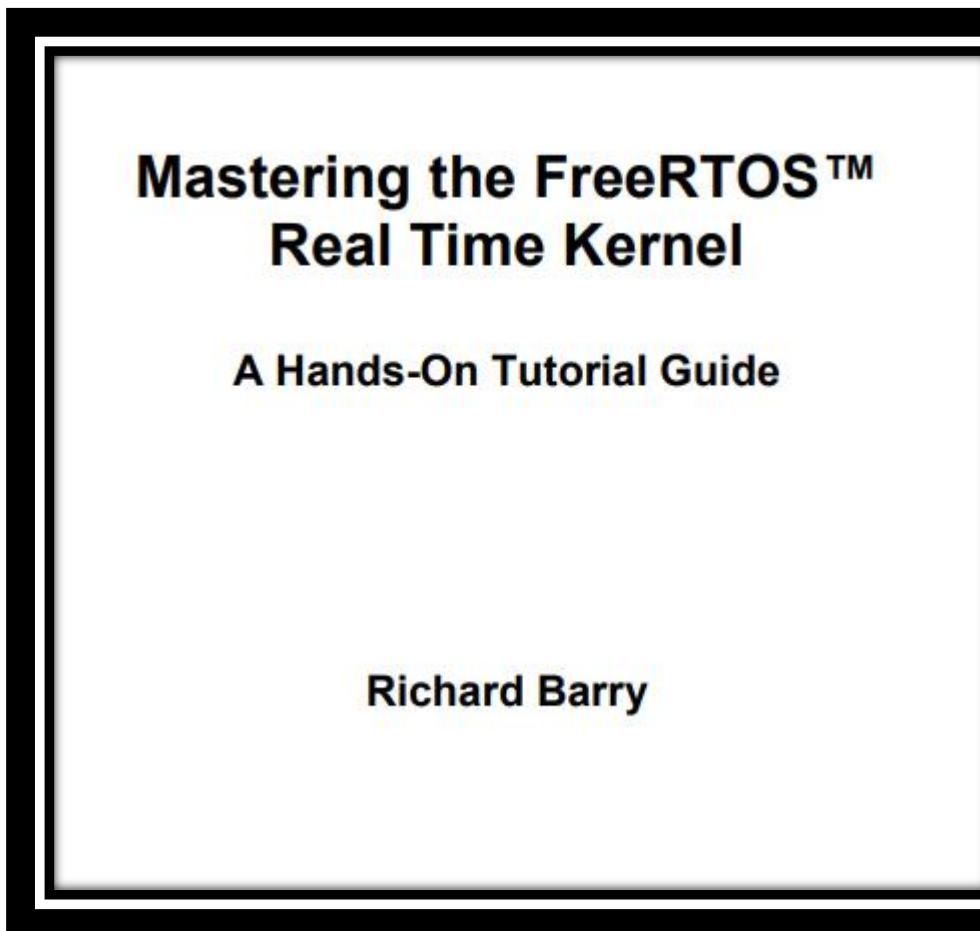
- **Semáforo de contagem** para controlar o número de usuários simultâneos.
- **Semáforo binário** com interrupção para resetar o sistema.
- **Mutex** para proteger o acesso ao display OLED.

O sistema deve oferecer:

- Feedback visual através do LED RGB, indicando a ocupação.
- Sinalização sonora com o buzzer.
- Exibição de mensagens e contagem no display OLED.



Referências



http://www.openrtos.net/Documentation/RTOS_book.html



Obrigado!

Executores:



Coordenação:



Iniciativa:

