SEL0630 - PROJETOS EM SISTEMAS EMBARCADOS

Prática 5: Introdução aos Sistemas Operacionais de Tempo Real (RTOS)

Resumo

Esta prática visa introduzir o conceito de RTOS, diferenciar sistemas embarcados de tempo real de sistemas convencionais e trabalhar com FreeRTOS utilizando a ESP32. Será abordado o uso de sistemas operacionais de tempo real (RTOS) para sistemas embarcados com foco para a execução de tarefas com prioridades bem definidas, garantindo previsibilidade e eficiência no uso dos recursos.

Conceitos Importantes

Real-Time Operating System, FreeRTOS, microcontroller, ESP32, tasks, mutex, semaphore, deadlock, round-robin, real time systems, hard real-time, soft real-time, scheduler, SBC, embedded linux.

Objetivo

O objetivo desta prática é compreender e implementar o conceito de tempo real em sistemas embarcados, utilizando o FreeRTOS na ESP32, sendo:

- Implementação de duas tasks utilizando FreeRTOS na ESP32. Cada task deve ter um propósito específico de livre escolha para demonstrar a capacidade do FreeRTOS em lidar com tarefas concorrentes de maneira previsível e eficiente.
- Poderá envolver o uso de sensores, conversor ADC e DAC, PWM, timer e interrupção, comunicação serial e sem fio via Bluetooth e Wi-Fi etc (exemplo: task1: PWM com controle de duty cycle via aplicativo móvel por Bluetooth; task2: sensor capacitivo).
- Poderá também envolver o desenvolvimento de um projeto de integração utilizando uma SBC com Linux embarcado (Raspberry Pi 3B+) como sistema principal e um microcontrolador ESP32 como sistema secundário, utilizando o FreeRTOS.
- Cada grupo poderá escolher a aplicação com base nas práticas anteriores.

Sistemas de Tempo Real e FreeRTOS

Sistemas de Tempo Real são caracterizados pela necessidade de reagir a eventos dentro de prazos específicos. Por exemplo, em um controle de motor, a tarefa de corrigir a posição deve ser realizada antes de um prazo definido, para evitar falhas no sistema. Diferentemente do Linux, que é um sistema operacional multitarefa sem garantias de tempo real, um RTOS (real-time operating system) permite a configuração de tarefas com prioridades específicas, garantindo que as mais importantes sejam executadas no tempo correto. A Raspberry Pi é uma SBC que roda Linux, sendo mais indicada para

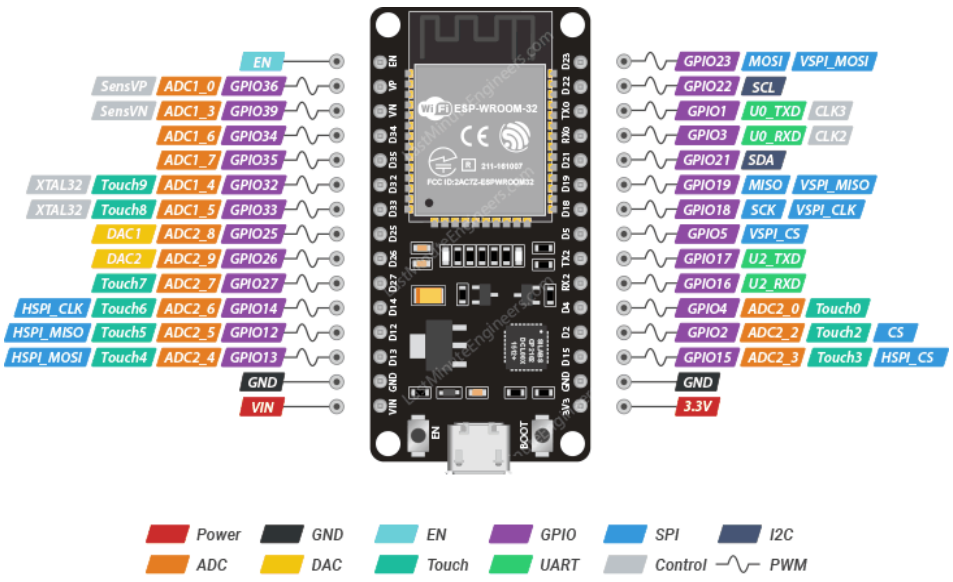
aplicações de alto nível, como interface gráfica ou processamento de dados. Já o microcontrolador ESP32 é uma plataforma mais adequada para controle direto de hardware em tempo real, sendo muito utilizada para comunicação e controle de dispositivos em projetos IoT

Para tanto, será utilizado o FreeRTOS, um RTOS de código aberto, projetado especificamente para microcontroladores. Sendo um RTOS, ele oferece diversas vantagens para o desenvolvimento de aplicativos em microcontroladores, tais como: (i) Multitarefa - executa várias tarefas simultaneamente, dividindo o tempo de processamento entre elas de forma eficiente; (ii) Escalonamento- possui um escalonador, isto é, tarefas de maior prioridade têm preferência sobre as de menor prioridade; e (iii) gerenciamento de memória - permite a alocação e liberação de memória de forma organizada em dispositivos com limitação de memória. A documentação e ebooks oficiais (com acesso aberto) sobre FreeRTOS podem ser consultados neste link.

ESP32

Como já abordado em disciplina anterior, a **ESP32** é um microcontrolador desenvolvido pela **Espressif Systems**, que se destaca por possuir um processador de 32 bits **dual-core Xtensa LX6** (ou single-core RISC-V) com clock de até 240 MHz, possui **Wi-Fi** e **Bluetooth** integrados, sendo ideal para aplicações **IoT** (Internet das Coisas). Além disso, possui diversos periféricos, como **GPIOs**, **Touch**, **sensor de efeito hall**, **ADC**, **DAC**, **PWM**, **SPI**, **I2C**, **CAN** e **UART**, que facilitam a interface com sensores e atuadores. O ESP32 também suporta o **FreeRTOS**, permitindo aplicações de tempo real. A documentação oficial da ESP32 encontra-se neste link. O pinout e uso dos periféricos por meio do framework do Arduino por ser consultado nesta página. Também foi disponibilizado no e-Disciplinas, na guia RTOS, um material de revisão (SEL0433/SEL0614) sobre hardware e programção do ESP32.

Uma vez que o foco desta prática é trabalhar com sistemas de tempo real e uso do FreeRTOS, será utilizado o framework do Arduino a fim de evitar uma preocupação a mais com uso de bibliotecas específicas e de maior complexidade inerentes à Espressif IDE (framework nativo da ESP32). No entanto, vamos trabalhar por meio do PlataformIO e sua integração com Visual Studio Code, visando um ambiente mais profissional do que a Arduino IDE.



Aplicação

As Tarefas (Tasks) são unidades básicas de execução no FreeRTOS, cada uma com seu próprio contexto e pilha de execução. O exemplo a seguir ilustra a estrutura básica de implementação de duas tasks (blink LED e uma contagem) por meio do FreeRTOS. Cada Task foi configurada com grau de prioridade diferente, o que irá influenciar na distribuição de tempo do escalonador para cada tarefa. Estudar e reproduzir o exemplo abaixo.

```
 ************************************************************/

/* Biblioteca do Arduino */
#include <Arduino.h>  // Inclui a biblioteca padrão do Arduino para
utilizar este framework

/* Bibliotecas FreeRTOS */
#include "freertos/FreeRTOS.h"  // Inclui a biblioteca do FreeRTOS para
a criação e manipulação de tasks
#include "freertos/task.h"      // Inclui funções específicas para
manipulação de tasks no FreeRTOS

/* Mapeamento de pinos */
#define LED 2  // Define o pino 2 como o pino ao qual o LED está
conectado

/* Variáveis para armazenamento do handle das tasks */
TaskHandle_t taks1Handle = NULL;  // Handle para a task 1, usado para
controle da task (interromper, reiniciar)
TaskHandle_t taks2Handle = NULL;  // Handle para a task 2, usado para
controle da task

/* Protótipos das Tasks */
void vTask1(void *pvParameters);  // Protótipo da função da task 1
void vTask2(void *pvParameters);  // Protótipo da função da task 2

/* Função setup */
void setup() {
  Serial.begin(9600);  // Configura a comunicação serial com uma taxa de
9600 bits por segundo

  /* Criação das tasks */
  xTaskCreate(
    vTask1,                   // Ponteiro para a função que define o
comportamento da task 1
    "TASK1",                  // Dar um nome da task (apenas para
referência)
    configMINIMAL_STACK_SIZE,  // Função que define tamanho mínimo de
```

```
pilha necessário para a task
    NULL,                       // Parâmetro passado para a task (neste
caso, nenhum)
    1,                          // Prioridade da task 1 (1 é uma
prioridade baixa)
    &taks1Handle                // Ponteiro para armazenar o handle da
task 1
  );

  xTaskCreate(
    vTask2,                     // Ponteiro para a função que define o
comportamento da task 2
    "TASK2",                    // Nome da task 2
    configMINIMAL_STACK_SIZE + 1024, // Tamanho da pilha aumentado para
a task 2, pois ela envolve a comunicação serial
    NULL,                       // Parâmetro passado para a task (nenhum
neste exemplo)
    2,                          // Prioridade da task 2 (prioridade maior
que a task 1)
    &taks2Handle                // Ponteiro para armazenar o handle da
task 2
  );
}



/* Função loop */
void loop() {
  vTaskDelay(3000);  // Libera a CPU por 3 segundos; esta função apenas
gera atraso enquanto as tasks executam
}

/*
  vTask1
  Função que inverte o estado do LED em intervalos de 200 ms
*/
void vTask1(void *pvParameters) {
  pinMode(LED, OUTPUT);  // Configura o pino LED como saída

  while (1) {  // Loop infinito para que a task continue executando
    digitalWrite(LED, !digitalRead(LED));  // Inverte o estado atual do
LED
    vTaskDelay(pdMS_TO_TICKS(200));        // Aguarda 200 ms antes de
inverter novamente (usando o FreeRTOS para atraso)
  }
}
```

```
/*
  vTask2
  Função que imprime o valor de uma contagem a cada 1 segundo
*/
void vTask2(void *pvParameters) {
  int cont = 0;  // Inicializa uma variável para contagem

  while (1) {  // Loop infinito para a task continuar executando
    Serial.println("Task 2: " + String(cont++));  // Imprime o valor
atual da contagem e incrementa em seguida
    vTaskDelay(pdMS_TO_TICKS(1000));              // Aguarda 1 segundo
antes de imprimir novamente
  }
}
```

O trecho de código a seguir ilustra como passar parâmetros para a Task. Neste caso, passa-se o número do pino associado ao LED1 para a variável pin por meio do ponteiro relacionado a Task1. As demais linhas de códigos e estruturas permanecem as mesmas do programa anterior.

```
// Trecho de código ilustrando como passar parâmetros para uma task

/*mapeamento de pinos*/

#define LED1 2

/*protótipos das Tasks*/
void vTaskBlink(void *pvParameters);

// demais linhas códigos e estruturas

void setup() {
  Serial.begin(9600);

xTaskCreate(vTaskBlink,"TASK1",configMINIMAL_STACK_SIZE,(void*)LED1,1,&t
ask1Handle);

void vTaskBlink(void *pvParameters)
{
    int pin = (int)pvParameters;
```

```
    pinMode(pin,OUTPUT);

    while (1)
    {
      digitalWrite(pin,!digitalRead(pin));
      vTaskDelay(pdMS_TO_TICKS(200));
    }
}
```

O trecho de código a seguir ilustra como trabalhar com o handle da tarefa, visando algum tipo de ação: deletar, suspender, reiniciar etc. mediante a alguma condição. Por exemplo, após uma contagem a Task1 será deletada/pausada ou será retomada quando um contador atingir determinado valor a partir do bloco de código da Task2. As demais linhas de códigos e estruturas permanecem as mesmas do programa anterior.

```
// Trecho de código ilustrando como trabalhar com o handle da task para
deletar, suspender e reiniciar…


// demais linhas de código e estruturas

TaskHandle_t task1Handle = NULL;

void vTask2(void *pvParameters)
{

// processamentos na task 2 e condições
// exemplo:
// iint cont = 0;
// if(cont==10){
//      Serial.println("Supendendo a Task 1...");

vTaskDelete(task1Handle); // deleta task1
vTaskSuspend(task1Handle); // suspende a task
vTaskResume(task1Handle); // reinicia a task

}
```

Por fim, o trecho de código a seguir ilustra como selecionar o núcleo do microcontrolador (considerando uma ESP32 dual-core, por exemplo), visando alocar um núcleo para cada Task e otimizar sua performance. Utiliza-se a função xTaskCreatePinnedToCore, cujo último parâmetro é número do núcleo: 0 (primeiro núcleo) ou 1 (segundo núcleo). As demais linhas de códigos e estruturas permanecem as mesmas do programa anterior.

```
// Trecho de código ilustrando como escolher núcleos do microcontrolador
para alocar as tasks usando a função xTaskCreatePinnedToCore


// demais linhas de código e estruturas

//Configuração de prioridade, alocação de memória e seleção do core da
ESP32:
 xTaskCreatePinnedToCore(Task1, "Temperatura", 2048, NULL, 5, NULL, 0);
 xTaskCreatePinnedToCore(Task2, "Bluetooth", 2048, NULL, 1, NULL, 1);

 //2048: É o valor máximo da pilha (stack) alocada para cada tarefa em
bytes. Determina a quantidade de memória disponível para a execução da
tarefa.
 // NULL - indica que nenhum parâmetro adicional está sendo usados para as
tarefas acima (algum rótulo adicional de configuração ou ponteiro
específico, que não se faz necessário aqui, ou alguma ação de
suspender/reiniciar a tarefa)
// os números no quinto parâmetro indicam a prioridade das tarefas e os
números no último parâmetro indicam o core da ESP32 (core0 ou core1)
}
```

A seguir são apresentados os parâmetros da função **xTaskCreatePinnedtoCore:**

xTaskCreatePinnedtoCore (pvTaskCode, pcName, usStackDepth, pvParameters, uxPriority, pvCreatedTask, xCoreID)

- **pvTaskCode:** ponteiro para a função de entrada da task, exemplo: “Task1” “TaskA” “TaskPWM” etc. (dependerá do nome dado na função, ex: void task1 (){})
- **pcName:** nome descritivo para a tarefa apenas para facilitar identificação, por exemplo: “TaskA – PWM”; (colocar entre aspas).
- **usStackDepth:** tamanho da pilha da tarefa em bytes (na ESP32 WROOM esse tamanho é 2048)
- **pvParameters (opcional):** ponteiro que permite retornar algum outro parâmetro/marcador específico (por ex.: número do pino de um LED) endereço de memória ou compartilhamento de dados entre tarefas, deixar em “NULL” se não for usar;
- **uxPriority:** prioridade da tarefa, por exemplo: 5 - prioridade alta; 0 – mais baixa prioridade
- **pvCreatedTask (opcional):** parâmetro que permite interagir posteriormente com a tarefa, caso desejar suspender, deletar etc. Deixar em “NULL” se não for usar..

- **xCoreID:** seleciona o núcleo em microcontroladores com processador dual-core. “NO_AFFINITY” indicará que a tarefa pode ser executada/escalonada em qualquer núcleo disponível. Valores 0 ou 1 selecionam o core da ESP32.

Aprofundamento

Além das “Tasks”, que são unidades básicas de execução em RTOS (similares a threads em sistemas operacionais de propósito geral), caso desejar se aprofundar, os seguintes recursos podem ser implementados para otimização de programas com FreeRTOS:

- **Queue**: As filas são usadas para a troca de mensagens ou dados entre tasks ou entre uma task e uma interrupção, permitindo sincronizar processos em um ambiente multitarefa.
    - Por exemplo: uma task que lê o sensor coloca o valor lido em uma fila (queue). Outra task, responsável por decidir os movimentos do robô, lê esses valores da fila e determina se deve parar, virar ou seguir em frente.
- **Semaphore**: são usados para gerenciar o acesso a recursos compartilhados. Existem dois tipos principais: binário, que tem dois estados (disponível ou ocupado) e contador, que pode contar recursos disponíveis.
- **Mutex**: Um tipo especial de semáforo(mutual exclusion), sendo usado para garantir exclusão mútua, permitindo que apenas uma task acesse um recurso crítico por vez. São usados para evitar condições de corrida, ajudando a prevenir inversão de prioridade (quando uma task de alta prioridade é bloqueada/fica aguardando por uma task de baixa prioridade que está utilizando um recurso compartilhado) ou deadlocks (situação em que duas tasks visam o mesmo recurso e ficam bloqueadas uma aguardando a outra).
- **Event Group**: conjunto de sinalizadores de eventos representado por um conjunto de bits (cada bit sinaliza um evento ou pode representar um sensor) facilitando a comunicação entre tasks, que podem esperar por um ou vários eventos ao mesmo tempo.
- **Notifications**: uma forma mais direta e simples de comunicação entre tasks. Por exemplo: uma task monitora (por meio de um sensor) o nível de luz ambiente e envia uma notificação para a task de acionamento da cortina.
- **Software Timers**: são geridos pelo RTOS (ao contrário dos temporizadores de hardware), e permitem que o sistema realize operações assíncronas sem bloquear uma task. Caso deseje acionar um alarme por 1 minuto, por exemplo, não exige a implementação de uma task só para realizar esse controle de tempo.

Roteiro (Parte 1) - Familiarização com FreeRTOS

- Nos computadores do Laboratório, utilizar a Arduino IDE ou, caso deseje utilizar um ambiente mais profissional, o VS Code e PlatformIO. Recomenda-se a programação nativa ESP-IDF.
- Conectar a placa ESP32 ao PC via cabo USB e verificar qual a porta COM alocada em Gerenciador de Dispositivos do Windows.
- Gravar um programa de exemplo para testar a comunicação entre o PC e a ESP32, bem como para verificar se a placa está funcionando corretamente. Recomenda-se testar o programa a seguir, que acende o LED onboard, presente na própria placa e conectado ao GPIO 2, ao tocar no pino 13 (sensor capacitivo touch0). Para tanto, criar um novo projeto, escolher o Framework do Arduino, digitar o programa abaixo, compilar e gravar na placa, observando o resultado.

```ino
#include <Arduino.h>

int capacitiveValue;

int threshold = 20; //limiar para ajustar

void setup()

{

  Serial.begin(115200); // taxa de transferência serial (baud rate) = 115200 bits por segundo (bps)

  pinMode(2, OUTPUT);  // pino correspondente ao LED onboard como saída

  digitalWrite(2, LOW); //  inicialmente desligado (LED apagado)

 Serial.println("ESP32 Touch Sensor Test"); // mensagem inicial impressa no monitor serial

 // A função Serial.println imprime uma mensagem ou valor no monitor serial e após isso ocorre quebra de linha

 // na função Serial.print não ocorre quebra de linha

} //end setup

// --- Loop Infinito ---

void loop(){

  capacitiveValue = touchRead(13); // armazena o valor lido do touch do pino13

  Serial.print("key:"); // imprimir a msg "key"...

  Serial.println(capacitiveValue); // e o valor lido no monitor serial

  delay(500); // delay de 500 ms para atualização

   if (touchRead(13) < threshold) {  // lógica para acender o LED onboard quando o pino13 for tocado

    digitalWrite(2, HIGH); // se sim, acende o LED (pino 2)...

    Serial.println("touched"); // msg p/ sinalizar quando houve toque

    }

    else {

      digitalWrite(2, LOW); // caso contrário, o LED permanece apagado

      }

} //end loop
```

- Para testar outras funcionalidades da ESP32, foi disponibilizado no e-Disciplinas, aba “RTOS”, um material de revisão sobre ESP32 e vários exemplos de programas que exploram sensores, conversor ADC e DAC, PWM, timer e interrupção, comunicação serial e sem fio via Bluetooth e Wi-Fi.
- Testar a estrutura básica do FreeRTOS Tasks ilustrada anteriormente por meio do framework do Arduino (opcional: realizar testes passando parâmetros para uma taks ou interagindo com ela: deletar, suspender, reiniciar). Testar também a estrutura que permite processamento multi núcleo, isto é, colocar cada Task em um núcleo da ESP32 por meio da função **xTaskCreatePinnedtoCore.**

Roteiro (Parte 2)

- **Processamento multitarefa com RTOS:** Implementar duas Tasks utilizando FreeRTOS na ESP32. Cada task deve ter um propósito específico, **de livre escolha**, para demonstrar a capacidade do FreeRTOS em lidar com tarefas concorrentes. Poderá envolver o uso de sensores, conversor ADC e DAC, PWM, timer e interrupção, comunicação serial e sem fio via Bluetooth e Wi-Fi etc (exemplo: task1: PWM com controle de duty cycle via aplicativo móvel por Bluetooth; task2: sensor capacitivo).
- **Processamento multinúcleo:** no contexto acima, a primeira Task deve ser configurada como alta prioridade e ser alocada no núcleo 1 da ESP32. Por outro lado, a segunda Task será de baixa prioridade de alocada no núcleo 0 da ESP32.
- Modificar o programa de exemplo inserindo a aplicação de sua escolha para cada Task, configurando a primeira como alta prioridade e núcleo 1, bem como a segunda como baixa prioridade no núcleo 0 da ESP32. Testar na placa e demonstrar o funcionamento. Caso preferir, testar o projeto antes no simulador Wokwi.

- Conforme sugerido no início deste documento, pode ser desenvolvido uma aplicação que explore a integração entre uma Raspberry Pi (SBC com Linux embarcado - programa em Python) e uma ESP32 (microcontrolador com RTOS e programa em C/C++), combinando a capacidade de processamento e interface do Linux com o controle em tempo real oferecido pelo FreeRTOS.

    - **Por exemplo:** o ESP32 coleta dados de sensores que precisam ser processados em tempo real ou que requerem conversor A/D por se tratar de leitura analógica, enquanto a Raspberry Pi realiza a classificação e análise dos dados. Neste caso, um programa na ESP32 lê continuamente os dados de sensores (ex.: sensor de movimento, distância, potenciômetro) e usa tasks do FreeRTOS para lidar com processamento multitarefa e situações independentes em diferentes aspectos da leitura. O envio dos dados para o Raspberry Pi poderá ser de forma serial I2C ou SPI, por exemplo). Do outro lado, a Raspberry Pi recebe os dados dos sensores do ESP32 e um programa em Python classifica esses dados (ex.: para detectar quedas ou reconhecer padrões de movimento, determinada distância, leitura analógica etc.), e então envia feedback ao ESP32 para ajustar a coleta de dados ou fazer algum tipo de controle.

- Apesar da sugestão acima, o/a aluno(a) pode escolher a aplicação prática que julgar mais conveniente, podendo inclusive envolver as práticas anteriores. No caso da implementação das Tasks com FreeRTOS na ESP32, também podem ser criadas situações de compartilhamento de

recursos entre mais de uma task, **sendo opcional o aprofundamento com uso de filas, semáforos, mutex, notificações, grupo de eventos e software timers**.

O laboratório dispõe dos seguintes recursos que podem ser utilizados nesta aplicação:

- **Microcontrolador:** placas ESP32 Devkit e STM32
- **SBC/Linux embarcado:** Raspberry Pi 3B+, Raspberry Pi Zero 2W, BeagleBone, SBC Labrador e Toradex CoM Colibri + Placa base Aster
- **Sensores, atuadores e periféricos e interfaces:** sensor de distância, sensor de movimento, sensor de luminosidade, potênciometro, buzzer, micro servomotor, Tag RFID, componentes discretos (LED, LED RGB, resistores, capacitores, regulador 3v3-5V, botões etc.), módulo da câmera para Raspberry Pi (interface via cabo flat), webCam (USB), microfone, display LCD, display OLED (I2C), sensor BME280 (temperatura, umidade e pressão com interface via I2C), cabo USB-TTL UART (Tx, Rx).

Entrega:

- Scripts dos programas desenvolvidos para a aplicação escolhida + documentação da prática.
- **Documentação:** incluir uma breve apresentação textual que descreva a aplicação desenvolvida nas Partes 1 e 2. Apresentar um diagrama esquemático do projeto desenvolvido na parte 2 com explicação textual (de 2 a 3 páginas, no máximo) dos conceitos envolvidos por meio do FreeRTOS e ESP32. Explicitar a diferença, por exemplo, entre trabalhar com tasks usando um RTOS e de usar processos e threads (recurso abordado em uma prática anterior). Apresentar os programas desenvolvidos com comentários/explicações sobre os principais blocos implementados. Apresentar fotografias (ou link para vídeo) da montagem prática do projeto em protoboard bem como prints das telas demonstrando o funcionamento da aplicação. Enviar na tarefa a documentação em arquivo PDF ou link para repositório do GitHub. **Opcional:** gravar um vídeo demonstrando o funcionamento do projeto (caso não o fizer durante a aula) e incluir o link para ele no relatório.

Critérios de avaliação

- Implementação dos programas e montagem prática dos projetos na protoboard com funcionamento demonstrado e atendimento ao roteiro em relação aos requisitos e funções que devem ser usadas, sequência lógica, uso correto das bibliotecas; participação; trabalho em equipe e boa organização dos experimentos;
- Documentação: atendimento ao formato e estrutura solicitada, explicação breve, objetiva e com precisão dos conceitos e passos implementados conforme os roteiros; fotografias das montagens práticas etc.

Bibliografia

- **FreeRTOS**
- **FreeRTOS - Documentação e Books**
- **ESP32 - Arduino**
- **ESP32 - Getting Started**
- **ESP32 -** Espressif IDE

- **Operating Systems Foundations with Linux on the Raspberry Pi - ARM Education Media.**
- PlatformIO
- PlatformIO - integração com Visual Studio Code
- **Material de aula disponibilizado no e-Disciplinas (**RTOS, Desenvolv. de Linux Embarcado, Projeto de Sist.Embarcados)