

Sistemas em Tempo Real com uso do ESP32

André Valim Pelz
Engenharia de computação
Universidade do Vale do Itajaí - Univali
Itajaí, Brasil
Andre.valim@edu.univali.br

Eduardo Vicente Caldeira
Engenharia de computação
Universidade do Vale do Itajaí - Univali
Itajaí, Brasil
Eduardo.c@edu.univali.br

Nicole Migliorini Magagnin
Engenharia de computação
Universidade do Vale do Itajaí - Univali
Itajaí, Brasil
Nicole.magagnin@edu.univali.br

Resumo— O presente trabalho se trata de uma implementação de sistema de tempo real para a disciplina de Sistemas de Tempo Real, a implementação atende aos requisitos de uma empresa fictícia que deseja um sistema de monitoramento de dutos. Neste relatório também se encontra a execução de duas listas de exercícios sobre o microcontrolador ESP32, usado para a implementação do sistema.

Palavras-chave—tempo real, sistemas de tempo real, sistema de monitoramento, esp32.

I. INTRODUÇÃO

O ESP32 é um microcontrolador desenvolvido pela Espressif e é o sucessor do ESP8266, esse possui não só o clássico módulo de Wi-fi dos controladores ESP, mas também um sistema de processamento Dual Core, Bluetooth híbrido e múltiplos sensores embutidos [1].

Esse microcontrolador, utilizado para a implementação descrita neste relatório, serve como ponte para a Internet of Things, definida por ser uma conexão entre objetos físicos, usuários e internet.

O presente relatório trata da implementação de um sistema de tempo real para a empresa fictícia Santa Catarina Petróleo Ltda. com a finalidade de consolidação dos conhecimentos adquiridos durante todo o semestre da disciplina de Sistemas em Tempo Real e fazendo o aproveitamento do tema apresentado na primeira média, utilizando a avaliação de processos críticos.

II. METODOLOGIA

A metodologia aplicada para a implementação deste trabalho foi a lógica de programação aplicada na linguagem C, além do uso de ferramentas do microcontrolador ESP através do *plugin* ESP-IDF na IDE Visual Code.

III. PROPOSTA

A primeira parte é a execução dos código das listas de exercícios de FreeRTOS e ESP-IDF, no caso a **Lista RTOS** (Exercícios 1,2,3,4,5 e 7) e **Lista 2 RTOS**. Deverão ser executados os experimentos com os códigos e feito uma análise de cada funcionalidade expressada em cada código - descrever a funcionalidade e análise de execução dentro do contexto de Sistema Operacional e Sistema Operacional de Tempo Real.

A segunda parte diz respeito ao desenvolvimento do código relativo ao **Exercício - Threads** (adaptação do código do primeiro trabalho para um contexto de código que seja executável no FreeRTOS). Reaproveite o enunciado do trabalho feito na M1.

Os códigos com Pthreads está disponível na pasta Exercícios (**Exercício - Threads**). Você deve emular a

existência do problema (interrupção causado sobre identificação de pressão elevada ou problema) usando o touch sensor com interrupção.

Poderá ser feito para um sensor em um duto gás, um sensor em um duto de petróleo e um sensor em um poço de petróleo. No caso, há uma sensor que identifica ambas as pressões no duto de gás e petróleo. Há também uma tarefa supervisora que é responsável por fazer o log de eventos (printar) na tela periodicamente. Os tempos de período de execução e apresentação pode ser os mesmos do trabalho da M1.

A captura do tempo de execução pode ser feita usando as bibliotecas da ESP32 (link).

Você poderá simular a comunicação de dados (via delay) ou usar algum sistema de comunicação de dados via Wi-Fi/Bluetooth. No caso da última opção, você terá a bonificação de até 2,0 pontos em uma das avaliações da M1 ou M2 (atribuição total ou não fica a critério do professor) ou 1 ponta na Média com a menor nota.

Deverá ser feita uma apresentação de no máximo 5 minutos demonstrando essa solução.

IV. LISTAS

Para a iniciação dos conhecimentos em ESP32 e o uso da ESP-IDF, foram realizadas duas listas de exercícios sobre o tema. Suas execuções se encontram nos próximos tópicos.

a. Lista I

1) Execute o código `hello_word_main.c` na placa ESP32 e descreva as funções usadas. Informe também se há alguma função do FreeRTOS? Há diferença na criação do código para ESP32 quando comparado a outros ports?

Para a execução do código `hello_world_main.c`, foi usado o plugin ESP-IDF na IDE Visual Studio Code e executado utilizando a função "flash" enquanto pressionado o botão de Boot da ESP32. Com o monitor foi possível visualizar os seguintes resultados:

Figura 1 - Execução do exercício 1

Figura 2 - Execução do exercício 2

```
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!  
Hello world!
```




Figura 3 - Execução do exercício 3

a) Qual a diferença entre as funções `xTaskCreatePinnedToCore()` e `xTaskCreate()`? `XTaskCreatePinnedToCore` é uma função para fixar uma tarefa específica em um núcleo específico, logo com isso pode-se executar duas tarefas diferentes de forma independente e de forma simultânea usando os

Todas têm a mesma prioridade? No código estão sendo executadas duas tarefas ao mesmo tempo, porém uma delas tem a prioridade muito mais alta que a outra, a *hello task 2* possui maior prioridade, porém também têm um maior *delay*, fazendo com que as duas *tasks* coincidam em frequência na hora da impressão.

A exclusão é feita a partir da operação `xSemaphoreCreateMutex`, onde é criado um mutex e é

`taskENTER_CRITICAL` é uma operação que tem início e fim, sendo respectivamente `taskENTER_CRITICAL()` e `taskEXIT_CRITICAL()` que funcionam para desabilitar interrupções, sendo de forma global ou com nível de prioridade de interrupção específico. Já `vTaskSuspendAll`: é uma operação que suspende o escalonador, enquanto `xSemaphoreCreateMutex`: é uma operação que cria um *mutex* e faz o retorno de um identificador pelo qual o *mutex* criado seja referenciado

Sim, há diferença, uma vez que o primeiro código monitora apenas um sensor e o segundo código monitora todos eles.

```

T0:[1490]
T0:[1490]
T0:[1490]
T0:[1491]
T0:[1491]
T0:[1491]
T0:[1491]
T0:[1491]
T0:[1491]
T0:[1490]
T0:[1490]
T0:[1490]
T0:[1490]
T0:[1488]
T0:[ 86]
Sistema de Emergência Acinado
T0:[ 72]
Sistema de Emergência Acinado
T0:[1488]
T0:[1489]
T0:[1489]
T0:[1489]
T0:[1488]
T0:[ 51]
Sistema de Emergência Acinado
T0:[1488]
T0:[ 50]
Sistema de Emergência Acinado
T0:[1489]
T0:[1489]
T0:[1489]
T0:[1489]
T0:[ 83]

```

Figura 6 – Touchpad Example

```

Usuario pediu para abrir a porta: 6
Usuario pediu para abrir a porta: 8
Usuario pediu para abrir a porta: 4
Usuario pediu para abrir a porta: 6
Usuario pediu para abrir a porta: 8
Hello World
Usuario pediu para abrir a porta: 4
Usuario pediu para abrir a porta: 6
Usuario pediu para abrir a porta: 8
Usuario pediu para abrir a porta: 4
Usuario pediu para abrir a porta: 6
Usuario pediu para abrir a porta: 8
Hello World
Usuario pediu para abrir a porta: 4
Usuario pediu para abrir a porta: 6
Usuario pediu para abrir a porta: 8
Usuario pediu para abrir a porta: 4
Usuario pediu para abrir a porta: 6
Usuario pediu para abrir a porta: 8

```

Figura 7 - Touchpad init

2) Execute o código `gpio_intr_example.c` na placa ESP32 e responda:

a) Há diferença no periférico quanto ao uso do periférico para o exemplo anterior? O uso do FreeRTOS é necessário? Sim, neste código o periférico usado é o botão, enquanto no anterior o periférico era o sensor. O uso do FreeRTOS é necessário pois o código faz uso das operações de

`xTaskResumeFromISR()` e `vTaskSuspend()` para controle de tarefas.

```

I (0) cpu_start: App cpu up.
I (210) cpu_start: Pro cpu start user code
I (210) cpu_start: cpu freq: 160000000
I (210) cpu_start: Application information:
I (215) cpu_start: Project name:   template-app
I (220) cpu_start: App version:    1
I (225) cpu_start: Compile time:   Nov 25 2021 21:07:00
I (231) cpu_start: ELF file SHA256: b65caed7f2626fc4...
I (237) cpu_start: ESP-IDF:        v4.3.1-dirty
I (242) heap_init: Initializing, RAM available for dynamic allocation:
I (249) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (255) heap_init: At 3FFB31B8 len 0002CE48 (179 KiB): DRAM
I (262) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (268) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (274) heap_init: At 4008AFC0 len 00015040 (84 KiB): IRAM
I (282) spi_flash: detected chip: gd
I (285) spi_flash: flash io: dio
I (290) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
Button pressed!!!
Button pressed!!!
Button pressed!!!
Button pressed!!!
Button pressed!!!
Button pressed!!!

```

Figura 8 - Execução do exercício 2 da lista II

3) Execute o código `library_and_timers.c` na placa ESP32 e responda:

a) O temporizador usado é de sistema ou de hardware? É de Hardware, uma vez que é utilizada a função `esp_timer_get_time()` do microcontrolador Esp32 para obter-se o tempo de execução em milissegundos.

b) Há necessidade de mexer em algum arquivo para modularizar o código? Sim, foi necessária a implementação de uma biblioteca com as funções do exercício 1.

c) Os temporizadores em software são melhores do que usado no exemplo? Não, os temporizadores em Hardware oferecem maior precisão.

```

Usuario pediu para abrir a porta: 1
Usuario pediu para abrir a porta: 4
Usuario pediu para abrir a porta: 6
Usuario pediu para abrir a porta: 8
Usuario pediu para abrir a porta: 10
Hello World
Usuario pediu para abrir a porta: 1
Usuario pediu para abrir a porta: 4
Usuario pediu para abrir a porta: 6
Usuario pediu para abrir a porta: 8
Usuario pediu para abrir a porta: 10
Usuario pediu para abrir a porta: 1
Usuario pediu para abrir a porta: 4
Usuario pediu para abrir a porta: 6
Usuario pediu para abrir a porta: 8
Usuario pediu para abrir a porta: 10
Hello World
Usuario pediu para abrir a porta: 1
Usuario pediu para abrir a porta: 4
Usuario pediu para abrir a porta: 6
Usuario pediu para abrir a porta: 8
Usuario pediu para abrir a porta: 10

```

Figura 9 - Execução do exercício 3 da lista II

4) Execute o código `queue_and_events_group.c` e responda:

a) Qual a diferença entre usar Mutex e usar Queues e Events Group?

O Mutex se trata de um bloqueio para que apenas um produtor ou consumidor acesse partes do código de uma vez, enquanto isso, Queue são filas que quando criadas retornam identificadores de tarefas para que as mesmas sejam localizadas posteriormente, já o Event Group funciona como uma fila, mas com a formação de agrupamento e não first-in-first-out.

b) Posso implementar uma abordagem parecida com monitor com Events Group?

É possível, porém é necessário o uso de mais controladores como um Mutex para que seja possível não só sinalizar como bloquear as atividades quando tornar-se necessário.

```

Item nao recebido, timeout expirou!
Item nao recebido, timeout expirou!
Item recebido: 1
Item nao recebido, timeout expirou!
Item nao recebido, timeout expirou!
Item recebido: 2
Item nao recebido, timeout expirou!
Item nao recebido, timeout expirou!
Item recebido: 3
Item nao recebido, timeout expirou!
Item nao recebido, timeout expirou!
Item recebido: 4
Item nao recebido, timeout expirou!
Item nao recebido, timeout expirou!
Item recebido: 5
Item nao recebido, timeout expirou!
Item nao recebido, timeout expirou!
Item recebido: 6
Item nao recebido, timeout expirou!
Item nao recebido, timeout expirou!
Item recebido: 7

```

Figura 10 - Execução do exercício 4 lista II

V. TRABALHO M3

a. Implementação

A implementação proposta consiste em implementar a partir de Threads e o uso do microcontrolador ESP32, um sensor para um duto de gás, um sensor para o duto de petróleo e um sensor para o poço de petróleo, onde os erros deveriam ser simulados usando o touch do sensor.

Para atender aos requisitos, o código foi dividido em três partes, sendo `main.c`, `touch_pad.c` e `touch_pad.h`.

1. Touch_pad.c e Touch_pad.h

A biblioteca `touch_pad` tem como principal objetivo a inicialização dos sensors utilizando o microcontrolador. Suas funções iniciam os sensores touch, capturam seus status e resetam os mesmos.

```

#ifndef TOUCH_PAD_H
#define TOUCH_PAD_H

#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/touch_sensor.h"
#include "soc/sens_periph.h"

#define WELL      0
#define PIPE_400  1
#define PIPE_800  2
#define PIPE_1200 3

void touch_pad_init_all();
int get_sensor_status(unsigned id);
void reset_sensor_status(unsigned id);

#endif // TOUCH_PAD_H

```

Figura 11 - touch_pad.h

Quanto ao uso do arquivo em .c referente ao touch pad, são definidos sensores para cada um dos sensores da proposta enunciada e inicializados os mesmos.

```
#include "touch_pad.h"

#define WELL_TOUCH_PAD_ID 0 //
#define PIPE_400_TOUCH_PAD_ID 4 // ids internos dos touch pads para cada sensor
#define PIPE_800_TOUCH_PAD_ID 7 //
#define PIPE_1200_TOUCH_PAD_ID 9 //
#define TOUCH_PAD_NO_USE 0 //
#define TOUCH_PAD_THRESHOLD 400 // variaveis de inicializacao dos sensores
#define TOUCH_PAD_FILTER_PERIOD 10 //

void touch_intr_handler(void* arg); // prototipo do tratador de interrupcao

int pipe_sensor_status[4] = {}; // vetor que guarda status dos sensores em uso quando ocorre interrupcao

void touch_pad_init_all() { // funcao para inicializacao dos 4 touchpads usados para simular os sensores (apenas duto 0)
    touch_pad_init(); // inicializacao padrao geral
    touch_pad_set_voltage(TOUCH_HVOLT_2V7, TOUCH_LVOLT_0V5, TOUCH_HVOLT_ATTEN_1V); // inicializacao padrao geral
    touch_pad_set_fsm_mode(TOUCH_FSM_MODE_TIMER); //

    touch_pad_config(WELL_TOUCH_PAD_ID, TOUCH_PAD_NO_USE); //
    touch_pad_config(PIPE_400_TOUCH_PAD_ID, TOUCH_PAD_NO_USE); //
    touch_pad_config(PIPE_800_TOUCH_PAD_ID, TOUCH_PAD_NO_USE); // configuracao de cada pad usado
    touch_pad_config(PIPE_1200_TOUCH_PAD_ID, TOUCH_PAD_NO_USE); //

    touch_pad_set_thresh(WELL_TOUCH_PAD_ID, TOUCH_PAD_THRESHOLD); //
    touch_pad_set_thresh(PIPE_400_TOUCH_PAD_ID, TOUCH_PAD_THRESHOLD); //
    touch_pad_set_thresh(PIPE_800_TOUCH_PAD_ID, TOUCH_PAD_THRESHOLD); //
    touch_pad_set_thresh(PIPE_1200_TOUCH_PAD_ID, TOUCH_PAD_THRESHOLD); //

    touch_pad_set_trigger_mode(TOUCH_TRIGGER_BELOW); //
    touch_pad_intr_enable(); // inicializacao e configuracao das interrupcoes
    touch_isr_register(&touch_intr_handler, NULL); //

    touch_pad_filter_start(TOUCH_PAD_FILTER_PERIOD); // configuracao do filtro do touchpad
}
```

Figura 12 - Touch_pad.c - Touch_pad_init_all

A função de inicialização é responsável por chamar a configuração dos sensores touch, dadas por define e logo em seguida é chamada a função touch_pad_set_thresh, responsável por passar o ID de início e o limite de contagem do sensor touch, podendo assim retornar a situação dos sensores. Então são chamadas funções para definir que o sensor deve ser pressionado em sua parte de baixo e funções de inicialização para o ESP32 [2].

```
void touch_intr_handler(void* arg) { // tratador de interrupcao do touchpad (sem display no x1)
    uint8_t touch_value;
    uint8_t pad_intr = touch_pad_get_status(); // adquire status de todos os touch pads

    touch_pad_read_filtered(WELL_TOUCH_PAD_ID, &touch_value); // le valor
    pipe_sensor_status[0] = (touch_value/100 + 1)* (pad_intr >> WELL_TOUCH_PAD_ID) & 0b01; // verifica se ativo, caso sim, de valor de 1 a 3 baseado no valor lido

    touch_pad_read_filtered(PIPE_400_TOUCH_PAD_ID, &touch_value); //
    pipe_sensor_status[1] = (touch_value/100 + 1)* (pad_intr >> PIPE_400_TOUCH_PAD_ID) & 0b01; //

    touch_pad_read_filtered(PIPE_800_TOUCH_PAD_ID, &touch_value); //
    pipe_sensor_status[2] = (touch_value/100 + 1)* (pad_intr >> PIPE_800_TOUCH_PAD_ID) & 0b01; //

    touch_pad_read_filtered(PIPE_1200_TOUCH_PAD_ID, &touch_value); //
    pipe_sensor_status[3] = (touch_value/100 + 1)* (pad_intr >> PIPE_1200_TOUCH_PAD_ID) & 0b01; //

    touch_pad_clear_status(); // reinicia status dos pads no esp
}
```

Figura 13 - Touch_pad.c touch_intr_handler

A função touch_intr_handler tem como objetivo a filtragem de leitura de sensores dos poços e dutos com a finalidade de fazer a passagem dos dados de contagem do sensor por um filtro IIR, nela são passados o índice do touch_pad e um ponteiro para acessar o valor do sensor, a função retorna se o sensor está ativo e um valor de 1 a 3 baseado no valor lido. Após, os status dos sensores são limpos.

Ao final do código se encontram funções de obter e resetar o status do sensor, baseadas em um parâmetro de id.

```
int get_sensor_status(unsigned id) { // retorna status do sensor com id requisitado
    return pipe_sensor_status[id];
}

void reset_sensor_status(unsigned id) { // reinicia status do sensor com id requisitado
    pipe_sensor_status[id] = 0;
}
```

Figura 14 - touch_pad.c

II. Main.c

A biblioteca main realiza as principais funções da implementação e em seu início possui os defines necessários para a observação de funcionamentos do código, onde é possível definir qual será o atual sensor, duto e o modo de debug, que pode ser alterado para que possam ser

demonstrados warnings em caso de perdas de *deadline* ou a impressão do tempo de execução a cada loop. São também definidos os *delays* do estado do sensor e do display, além da definição da função de conversão do tempo para milissegundos. Quando DEBUG_MODE vale 0, todos os sensores estarão sob análise, e o programa apenas indicará quando alguma *task* perder a *deadline* definida em DEBUG_EXPECTED_DELAY, quando DEBUG_MODE vale 1, apenas o sensor selecionado para análise terá o tempo de execução indicado, e o programa irá imprimir o tempo de execução de cada ciclo.

```
#ifndef DEBUG
#define DEBUG_CURRENT_PIPE 0 // seleciona sensor para analise
#define DEBUG_CURRENT_SENSOR PIPE_400 //

/*
 * A constante DEBUG_MODE define o tipo de analise de tempo realizada durante a execucao.
 * - Quando DEBUG_MODE vale 0, todos os sensores estao sob analise, e o programa
 * apenas indica quando alguma task perde a deadline definida em DEBUG_EXPECTED_DELAY.
 * - Quando DEBUG_MODE vale 1, apenas o sensor selecionado para analise tem o tempo de
 * execucao indicado, e o programa ira imprimir o tempo de execucao de cada ciclo.
 */
#define DEBUG_MODE 0

#define DEBUG_EXPECTED_DELAY 200 // deadline das tasks do sensor
#endif

#define SENSOR_STATUS_DELAY_MS 100 // constantes de delays
#define DISPLAY_DELAY_MS 5000 //

#define TIMESPEC_TO_MS(ts) ((uint64_t) ts.tv_sec*1000 + ts.tv_nsec/1000000) // macro para obter tempo em milissegundos das medicoes

const char* TAG = "PIPE_SENSORS"; // tag para log do esp

int sensors[3][4] = {}; // matriz com status de todos os sensores
```

Figura 15 - Defines main.c

A função seguinte é denominada *display_task()* e serve para a impressão e demonstração do estado dos sensores em console. A função *vTaskDelay* seta o delay para o display interagindo com o microcontrolador.

```
void display_task() { // task para display
    const char* nome[4] = {"Poço", "400m", "800m", "1200m"};

    while(1) {
        printf("\n");

        for (int i = 0; i < 3; i++){
            printf("Poco %d:\n", i + 1);

            for (int j = 0; j < 4; j++){
                printf("    Sensor %s:\t", nome[j]);

                if(!j)
                    printf("Estado: %s\n\n", (sensors[i][j] ? "Mal funcionamento" : "Estável") );
                else {
                    printf("Óleo: %s\n\t\t", (sensors[i][j]>2 ? "Mal funcionamento" : "Estável") );
                    printf("Gás: %s\n", (sensors[i][j]>2 ? "Mal funcionamento" : "Estável") );
                }

                printf("\n");
            }

            vTaskDelay(DISPLAY_DELAY_MS / portTICK_PERIOD_MS); // dorme pelo tempo de delay definido
        }
    }
}
```

Figura 16 - Display_task()

Para a checagem de status é obtido o status do sensor passando sua *Id* e utilizando o mesmo, são emitidos *logs* [3] de erros ou de informação, informando qual duto ou poço está instável e se a contramedida está sendo aplicada, após essa informação, os dados são resetados.

```
int check_sensor_status(int pipe_id, int sensor_id) { // funcao para verificar status do sensor
    int result;

    if(!pipe_id) // apenas duto 0 sendo simulado pelo esp
        result = get_sensor_status(sensor_id); // adquire status atual do sensor
    else // para outros dutos, define sensor como estavel
        result = 0;

    return result; // retorna resultado
}
```

Figura 17 - Check_sensor_status()


```

74 void apply_countermeasure(int pipe_id, int sensor_id) { // funcao que aplica contramedida para sensores instaveis
75     const int sensor_name = sensor_id*400;
76     char* pipe_name = (char *) malloc(100 * sizeof(char));
77
78     sprintf(pipe_name, "DUTO %d", pipe_id);
79
80     if(sensor_id) //
81         ESP_LOGI(TAG, "Sensor do poço instável (%s)", pipe_name); // mensagens diferentes para poço/duto
82     else
83         ESP_LOGI(TAG, "Sensor do duto em São instável (%s)", sensor_name, pipe_name); //
84
85     ESP_LOGI(TAG, "Aplicando contramedida..."); // aplica delay para contramedida
86     vTaskDelay(2 / portTICK_PERIOD_MS); //
87
88     reset_sensor_status(sensor_id); // reinicia status do sensor
89 }

```

Figura 18 - Função *apply_countermeasure*

Para o monitoramento das tarefas é usada uma função de execução de tasks capaz de mostrar um *log* de *warning* de qual task está sendo executada no momento com influência do *define Debug*. Dependendo dessas preferências, o tempo atual é ou não obtido através de função. São checados os status dos sensores e um delay é utilizado, para novamente ser mostrado o tempo de execução para que o usuário decida se condiz com o deadline.

Se o código estiver em modo de debug ativo, é mostrado o tempo de execução, caso contrário, são lançados *warnings* quando os deadlines são excedidos. Essa função também é capaz de a partir de estado instáveis de dutos ou do poço, chamar a função para que a contramedida seja aplicada.

```

89 void sensor_monitoring_task(void* arg) { // task para monitorar os sensores
90     int pipe_id = *(int*) arg/10;
91     int sensor_id = *(int*) arg%10;
92     TickType_t last_wake_up_tick;
93
94     #ifdef DEBUG
95     struct timespec start, finish; // variáveis de debug
96     uint64_t exec; //
97
98     if(pipe_id == DEBUG_CURRENT_PIPE || sensor_id == DEBUG_CURRENT_SENSOR)
99         ESP_LOGI(TAG, "Task atualizada sobre DEBUG: duto=%d, sensor=%d", pipe_id, sensor_id);
100     #endif
101 }

```

Figura 19 - *Sensor_monitoring_task* parte 1

```

while(1) {
    #ifdef DEBUG
    clock_gettime(CLOCK_REALTIME, &start); // captura tempo do início
    #endif

    last_wake_up_tick = xTaskGetTickCount(); // tick inicial da task

    if(sensor_id) // aplica delay que simula comunicação com o sensor
        vTaskDelay((20*sensor_id + 1) / portTICK_PERIOD_MS);
    else
        vTaskDelay((75 + 1) / portTICK_PERIOD_MS);

    sensors[pipe_id][sensor_id] = check_sensor_status(pipe_id, sensor_id); // capta status do sensor

    if(sensors[pipe_id][sensor_id]) // se instavel
        apply_countermeasure(pipe_id, sensor_id); // aplica contramedida

    vTaskDelayUntil(&last_wake_up_tick, SENSOR_STATUS_DELAY_MS / portTICK_PERIOD_MS); // espera ate fim do periodo

    #ifdef DEBUG
    clock_gettime(CLOCK_REALTIME, &finish); // capta tempo de fim

    exec = TIMESPEC_TO_MS(finish) - TIMESPEC_TO_MS(start); // calcula tempo de execucao

    #if DEBUG_MODE == 1
    if(pipe_id == DEBUG_CURRENT_PIPE || sensor_id == DEBUG_CURRENT_SENSOR)
        ESP_LOGI(TAG, "Tempo de execucao do sensor %d no duto %d: %llu", sensor_id, pipe_id, exec); // imprime tempo de execucao na tela
    #else
    // para DEBUG_MODE == 0
    if(exec > DEBUG_EXPECTED_DELAY) { // se perder deadline
        ESP_LOGW(TAG, "Tempo de execucao excedeu o limite de %d", DEBUG_EXPECTED_DELAY); // indica perda na tela
        ESP_LOGW(TAG, "Tempo de execucao do sensor %d no duto %d: %llu", sensor_id, pipe_id, exec); //
    }
    #endif

    #endif
}

```

Figura 20 - *Sensor_monitoring_task* parte 2

Por fim, a função *main* realiza o alocamento de tarefas e a criação das mesmas, envolvendo também a função *xTaskCreatePinnedToCore* [4] que cria a task adicionando se há ou não uma afinidade por núcleo. Por padrão, os parâmetros são os *ids* de sensores de dutos e poço.

```

void app_main(void)
{
    int argc=1; // vetor com id das tasks dos sensores

    touch_pad_init_all(); // inicializa touchpads

    for(int pipe_id = 0; pipe_id < 3; pipe_id++) {
        for(int sensor_id = 0; sensor_id < 4; sensor_id++) {
            char* task_name = (char *) malloc(100 * sizeof(char));
            sprintf(task_name, "sensor_monitoring_task_%d", argc[pipe_id][sensor_id]); // cria nome para task
            // cria task do sensor
            vTaskCreatePinnedToCore(sensor_monitoring_task, task_name, 2048, (void *) &argc[pipe_id][sensor_id], 4, -sensor_id*10, NULL, sensor_id*3);
        }
    }

    xTaskCreate(display_task, "display_task", 2048, NULL, 5, NULL); // inicia task para display

    while(1) {
        vTaskDelay(10000 / portTICK_PERIOD_MS); // delay permanente, serve como task idle
    }
}

```

Figura 21 – *main*

b. Execução

A execução do código foi dada em três partes onde durante a primeira o modo de debug 0 foi ativado, sendo assim, o programa apenas indicou perda de deadline e todos os sensores estiveram sob análise, gerando duas perdas de deadline para tempos acima de 200ms no sensor dois e três.

```

W (57738) PIPE_SENSORS: Tempo de execução escedeu o limite de 200!
W (57768) PIPE_SENSORS: Tempo de execução escedeu o limite de 200!
W (57778) PIPE_SENSORS: Tempo de execução do sensor 2 no duto 0: 204
W (57788) PIPE_SENSORS: Tempo de execução do sensor 3 no duto 2: 211

```

Figura 22 - Código executado com *Debug_mode* em 0

A segunda execução se deu de maneira que a função de debug estivesse em 1, sendo assim apenas o sensor selecionado para análise teve o tempo de execução indicado, foi selecionado o sensor 1.

```

9 #ifdef DEBUG
10 #define DEBUG_CURRENT_PIPE 0 // seleciona sensor para analise
11 #define DEBUG_CURRENT_SENSOR PIPE_400 //
12
13 /*
14  * A constante DEBUG_MODE define o tipo de analise de tempo realizada durante a execucao.
15  * - Quando DEBUG_MODE vale 0, todos os sensores estaraos sob analise, e o programa
16  * apenas indicara quando alguma task perder a deadline definida em DEBUG_EXPECTED_DELAY.
17  * - Quando DEBUG_MODE vale 1, apenas o sensor selecionado para analise tera o tempo de
18  * execucao indicado, e o programa ira imprimir o tempo de execucao de cada ciclo.
19  */
20 #define DEBUG_MODE 1

```

TERMINAL	PROBLEMS	OUTPUT	DEBUG CONSOLE
			Gás: Estável
Poço 3:			Estado: Estável
Sensor Poço:			Óleo: Estável
Sensor 400m:			Gás: Estável
Sensor 800m:			Óleo: Estável
Sensor 1200m:			Gás: Estável
			Óleo: Estável
			Gás: Estável

```

W (11140) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (11330) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (11520) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (11710) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (11900) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (12090) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (12280) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (12470) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (12660) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (12850) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (13040) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (13230) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (13420) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (13610) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (13800) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (13990) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (14180) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (14370) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (14560) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100
W (14750) PIPE_SENSORS: Tempo de execucao do sensor 1 no duto 0: 100

```

Figura 23 - Código executado com *Debug_mode* em 1

Por fim, na figura 24 é possível observar o display e a aplicação de contramedidas. Vale ressaltar que o display demonstra o último estado do sensor, sendo assim, por vezes a contramedida já pode ter sido aplicada, mas ainda assim o display ainda não demonstrará devido ao delay.

Os resultados obtidos foram satisfatórios e foi percebido que as únicas perdas de *deadline* apresentadas ocorreram devido as interrupções de impressão do código.

```

E (228918) PIPE_SENSORS: Sensor do poço instável! (DUTO_0)
I (228918) PIPE_SENSORS: Aplicando contramedida...
E (228918) PIPE_SENSORS: Sensor do duto em 400m instável! (DUTO_0)
I (228918) PIPE_SENSORS: Aplicando contramedida...

Poço 1:
  Sensor Poço:      Estado: Mal funcionamento

  Sensor 400m:      Óleo: Estável
                   Gás: Mal funcionamento

  Sensor 800m:      Óleo: Estável
                   Gás: Estável

  Sensor 1200m:     Óleo: Estável
                   Gás: Estável

Poço 2:
  Sensor Poço:      Estado: Estável

```

Figura 24 - Display e contramedidas

VI. CONCLUSÃO

Com a execução da lista e implementação deste trabalho foi possível entender sobre o funcionamento do microcontrolador ESP32, suas funções e algumas de suas inúmeras possibilidades.

Os conhecimentos da matéria de Sistemas em Tempo Real foram reunidos em conceitos de Escalonamento, Threads e Microcontroladores e pode-se ter uma melhor compreensão prática dos assuntos.

As implementações para este trabalho obtiveram o resultado esperado.

REFERÊNCIAS

- [1] ESP32. **The internet of things with ESP32**. Disponível em: <http://esp32.net/>. Acesso em: 20 nov. 2021.
- [2] ESPRESSIF SYSTEMS (SHANGHAI). **Sensor de toque**. Disponível em: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/touch_pad.html#_CPPv423touch_pad_read_filtered11touch_pad_tP8uint16_t. Acesso em: 25 nov. 2021.
- [3] ESPRESSIF SYSTEMS (SHANGHAI). **Biblioteca de registro**. Disponível em: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/log.html>. Acesso em: 22 nov. 2021.
- [4] NEVES, Felipe. **ESP32 – Lidando com Multiprocessamento – Parte II**. 2020. Disponível em: <https://www.embarcados.com.br/esp32-lidando-com-multiprocessamento-parte-ii/>. Acesso em: 01 dez. 2021.

OLIVEIRA, Rômulo Silva de. **Fundamentos dos Sistemas de Tempo Real**. 2. ed. Brasil: Edição do Autor, 2018.