UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

ENGENHARIA ELETRÔNICA

THIAGO MELCHER ARMÊNIO

VICTOR AUGUSTO DEL MONEGO

PROJETO DE SISTEMAS OPERACIONAIS:

IMPLEMENTAÇÃO DE LUZ DE EMERGÊNCIA COM SENSOR DE LUMINOSIDADE

CURITIBA

2024

THIAGO MELCHER ARMÊNIO

VICTOR AUGUSTO DEL MONEGO

PROJETO DE SISTEMAS OPERACIONAIS:
IMPLEMENTAÇÃO DE LUZ DE EMERGÊNCIA COM SENSOR DE LUMINOSIDADE

Este projeto foi realizado para fins de avaliação para a matéria de **Sistemas Operacionais**, ministrada pelo professor Luiz Fernando Copetti. Neste documento será abordada a implementação e execução do projeto final da matéria.

CURITIBA

2024

Introdução:

O projeto foi desenvolvido utilizando a placa Tiva C Series EK-TM4C1294XL, com o objetivo de implementar um LED de emergência que se acende quando o sensor de luminosidade identifica que o nível de luz ambiente está suficientemente baixo. Este projeto utiliza diversos conceitos apresentados na disciplina de Sistemas Operacionais, como a utilização de processos, threads, escalonamento de memória, GPIO, comunicação entre periféricos e CPU.

Além de explorar a teoria de sistemas operacionais, a implementação prática permitiu aprofundar o conhecimento em áreas como a criação e gerenciamento de tarefas, principalmente em um ambiente de tempo real utilizando o FreeRTOS. A configuração e utilização dos pinos GPIO foram essenciais para a comunicação entre o microcontrolador e o sensor de luminosidade, assim como para acionar o LED.

O projeto também enfatiza a importância da comunicação eficiente entre os componentes do sistema, garantindo que as leituras do sensor de luminosidade sejam processadas, e dessa forma, o LED será acionado quando necessário. A implementação do escalonamento de tarefas assegura que todas as funções críticas sejam executadas de forma adequada, mantendo a responsividade do sistema.

Este trabalho, portanto, não só reflete a aplicação prática dos conceitos de sistemas operacionais, mas também demonstra a capacidade de integrar hardware e software para resolver problemas reais de maneira eficaz.

Desenvolvimento:

Código do projeto:

A seguir está o código de funcionamento do projeto, ele está dividido em 5 funções, sendo uma *main*, 3 com funcionalidades de dispositivos IO e uma para a situação de erro. Abaixo são apresentadas as *tasks* do código, bem como as funções da parte principal do projeto.

- Começando pela *LDRRead* que é responsável pela leitura do sensor de luminosidade, conectado ao conversor Analógico-Digital da placa. No corpo da *task*, a função fará a leitura contínua do sensor e fará uma média enquanto este sistema estiver em execução. Após sua execução, a tarefa fica suspensa por 300 ms, realizando o processo de *yield*.
- A função *SerialSend* é responsável por enviar periodicamente o valor médio lido do sensor de luminosidade via UART. No corpo da *task*, a função envia o valor armazenado na variável global *sensorADCValue* a cada 500 ms.
- A função *UARTReceive* é responsável por receber comandos via UART e controlar o estado do LED com base nesses comandos. A *task* monitora o canal serial, armazenando os caracteres recebidos da aplicação *Python* em um *buffer*. Quando um comando completo é recebido, a função o processa e realiza um *strcmp*. Se for um comando válido (LED_ON ou LED_OFF), acende ou apaga o LED correspondente. Após processar cada comando, a tarefa é suspensa por 100 ms.
- A função *__error__* é utilizada para capturar falhas de assertividade definidas nas bibliotecas de depuração do TivaWare. Nesta situação, essa função entra em um loop, permitindo a colocação de um ponto de interrupção para a depuração do sistema.
- Por fim, a função *main*. Responsável por configurar o sistema e iniciar o seu escalonador. Primeiro, o *clock* do sistema é configurado para 120 MHz usando a função *ROM_SysCtlClockFreqSet*. Em seguida, os pinos *GPIO* são inicializados para uso com a placa, e a *UART* é configurada para comunicação serial, permitindo a interação com um terminal serial para *output* e controle. Em seguida a função habilita e configura o Conversor Analógico-Digital e os pinos *GPIO* necessários para a leitura do sensor de luminosidade e o controle do LED. O ADC é configurado para usar o sequenciador 3 com uma única amostra, e o pino PE3 é configurado como entrada analógica. Após este processo, as tarefas antes mencionadas são criadas, para, por fim, o escalonador ser

iniciado com a chamada *vTaskStartScheduler*, que começa a executá-las. A função *main* não deve retornar.

Abaixo, a figura 1 ilustra o código descrito acima.

```
#include <stdint.h>
#include <stdbool.h>
#include "main.h"
#include "drivers/pinout.h"
#include "utils/uartstdio.h"
// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "driverlib/uart.h"
#include "driverlib/inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/adc.h"
#define SAMPLE_COUNT 4 // Quantidade de amostras para média

volatile uint32_t sensorADCValue;
// Demo Task declarations
void LDRRead(void *pvParameters);
void SerialSend(void *pvParameters);
void UARTReceive(void *pvParameters);
// Main function
int main(void)
{
    // Initialize system clock to 120 MHz
    uint32_t output_clock_rate_hz;
    output_clock_rate_hz = ROM_SysCtlClockFreqSet(
                               (SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
                                SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),
                               SYSTEM_CLOCK);
    ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);
    // Initialize the GPIO pins for the Launchpad
    PinoutSet(false, false);
    // Set up the UART which is connected to the virtual COM port
    UARTStdioConfig(0, 57600, SYSTEM_CLOCK);
    // Ativa o ADC0 e o GPIO
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    // Configura o pino PE3 como entrada analógica (ADC0, canal 0)
```

```
    GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_3);
    // Configura o sequenciador 3 do ADC0 com prioridade 0 e uma única
amostra
    ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_CH0 | ADC_CTL_IE |
ADC_CTL_END);
    ADCSequenceEnable(ADC0_BASE, 3);
    // Limpa o flag de interrupção para o sequenciador 3
    ADCIntClear(ADC0_BASE, 3);
    // Configura o pino PE0 do LED de saída
    GPIOPinTypeGPIOOutput(GPIO_PORTE_BASE, GPIO_PIN_0);
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_0, 0);
    // Create demo tasks
    xTaskCreate(LDRRead, (const portCHAR *)"Sensor",
                configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    xTaskCreate(SerialSend, (const portCHAR *)"SendSerial",
                configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    xTaskCreate(UARTReceive, (const portCHAR *)"UARTReceive",
                configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    vTaskStartScheduler();
    // Code should never reach this point
    return 0;
}
// Flash the LEDs on the launchpad
void LDRRead(void *pvParameters)
{
    while(1){
    uint32_t ui32ADC0Value[SAMPLE_COUNT];
    uint32_t ui32TempAvg;
    int i;
    // Realiza a conversão de várias amostras e calcula a média
    for(i = 0; i < SAMPLE_COUNT; i++) {
        // Dispara a conversão
        ADCProcessorTrigger(ADC0_BASE, 3);
        // Espera a conversão ser concluída
        while(!ADCIntStatus(ADC0_BASE, 3, false)) {}
        // Lê o valor do ADC
        ADCSequenceDataGet(ADC0_BASE, 3, &ui32ADC0Value[i]);
        // Limpa o flag de interrupção para o próximo passo
        ADCIntClear(ADC0_BASE, 3);
    }
    // Calcula a média dos valores lidos
    ui32TempAvg = 0;
    for(i = 0; i < SAMPLE_COUNT; i++) {
        ui32TempAvg += ui32ADC0Value[i];
    }
    ui32TempAvg /= SAMPLE_COUNT;
    sensorADCValue = ui32TempAvg;
    vTaskDelay(300);
    }
}
// Write text over the Stellaris debug interface UART port
void SerialSend(void *pvParameters)
{
```

```
    for (;;)
    {
        UARTprintf("%d\n",sensorADCValue);
        vTaskDelay(500);
    }
}
// Task to receive commands from UART
void UARTReceive(void *pvParameters)
{
    char buffer[16]; // Ajuste o tamanho conforme necessário
    int i = 0;
    while (1)
    {
        if (UARTCharsAvail(UART0_BASE))
        {
            char receivedChar = UARTCharGet(UART0_BASE);
            // Verificar se é um caractere válido
            if (receivedChar != '\r' && receivedChar != '\n')
            {
                buffer[i++] = receivedChar;
                // Verificar se chegamos ao fim do buffer
                if (i >= sizeof(buffer))
                {
                    // Buffer cheio, tratar erro ou limpar buffer
                    i = 0;
                }
            }
            else if (i > 0)
            {
                // Fim do comando, adicionar null terminator
                buffer[i] = '\0';
                // Processar comando
                if (strcmp(buffer, "LED_ON") == 0)
                {
                    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_0,
GPIO_PIN_0); // Acender LED
                    UARTprintf("LED ligado\n");
                }
                else if (strcmp(buffer, "LED_OFF") == 0)
                {
                    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_0, 0); //
Apagar LED
                    UARTprintf("LED desligado\n");
                }
                // Limpar buffer para o próximo comando
                i = 0;
            }
        }
        vTaskDelay(100); // Ajuste conforme necessário
    }
}
/*  ASSERT() Error function
 *
```

```
 *  failed ASSERTS() from driverlib/debug.h are executed in this function
 */
void __error__(char *pcFilename, uint32_t ui32Line)
{
    // Place a breakpoint here to capture errors until logging routine is
finished
    while (1)
    {
    }
}
```

Figura 1: Código "main.c"

Para consultar os diversos códigos das bibliotecas utilizadas, vide os arquivos fonte na documentação em anexo a este relatório.

Servidor:

O seguinte código em *Python* é um servidor que recebe as informações via serial, e interpreta se deve ser enviado um comando de acender ou apagar o LED, de acordo com o nível de luminosidade lido pelo sensor.

```
import serial
import time
# Configurações da porta serial
SERIAL_PORT = 'COM3'
SERIAL_BAUDRATE = 57600
SERIAL_TIMEOUT = 1
# threshold para luminosidade
LUMINOSITY_THRESHOLD = 300
def serial_server():
    # Configuração da porta serial
    ser = serial.Serial(SERIAL_PORT, SERIAL_BAUDRATE,
timeout=SERIAL_TIMEOUT)
    print(f"Servidor serial iniciado na porta {SERIAL_PORT} com baudrate
{SERIAL_BAUDRATE}")
    led_on = False  # Estado atual do LED
    try:
        while True:
            # Lê dados da porta serial
            data = ser.readline().decode('utf-8').strip()
            if data:
                # Obtém o tempo atual
                current_time = time.strftime("%Y-%m-%d %H:%M:%S")
                # Imprime o valor recebido e o instante em que foi
recebido
                print(f"Recebido: {data} às {current_time}")
                # Tenta converter o valor recebido para um número
                try:
                    luminosity_value = float(data)
```

```
                    # Verifica se o valor está abaixo ou acima do limite
                    if luminosity_value < LUMINOSITY_THRESHOLD and not
led_on:
                        print("Luminosidade abaixo do limite. Enviando
comando para acender o LED.")
                        ser.write(b'LED_ON\n')  # Envia o comando para
acender o LED
                        led_on = True
                    elif luminosity_value >= LUMINOSITY_THRESHOLD and
led_on:
                        print("Luminosidade acima do limite. Enviando
comando para apagar o LED.")
                        ser.write(b'LED_OFF\n')  # Envia o comando para
apagar o LED
                        led_on = False
                except ValueError:
                    print("Valor recebido não é um número válido.")
    except KeyboardInterrupt:
        print("Servidor serial encerrado pelo usuário.")
    finally:
        ser.close()

if __name__ == "__main__":
    serial_server()
```

Figura 2: Código de servidor de resposta Python

Abaixo, temos também uma imagem do circuito externo montado, composto por um resistor de 1.5KΩ, um resistor de 220Ω, um sensor de luminosidade resistivo, e um LED branco.

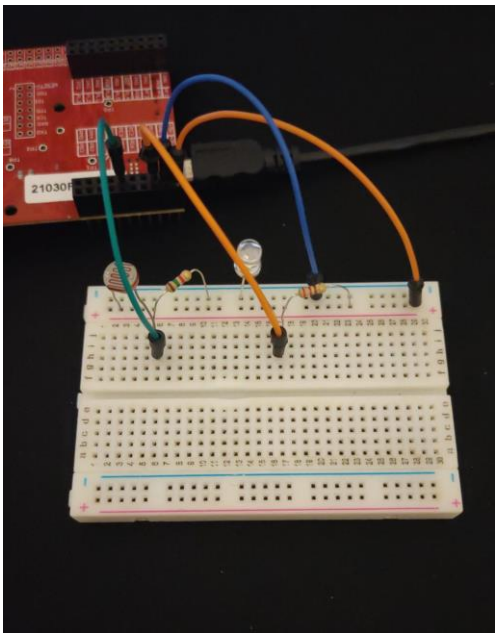

Figura 3: circuito utilizado

Na figura 4, tem-se uma imagem do circuito acoplado ao PC fazendo a comunicação.

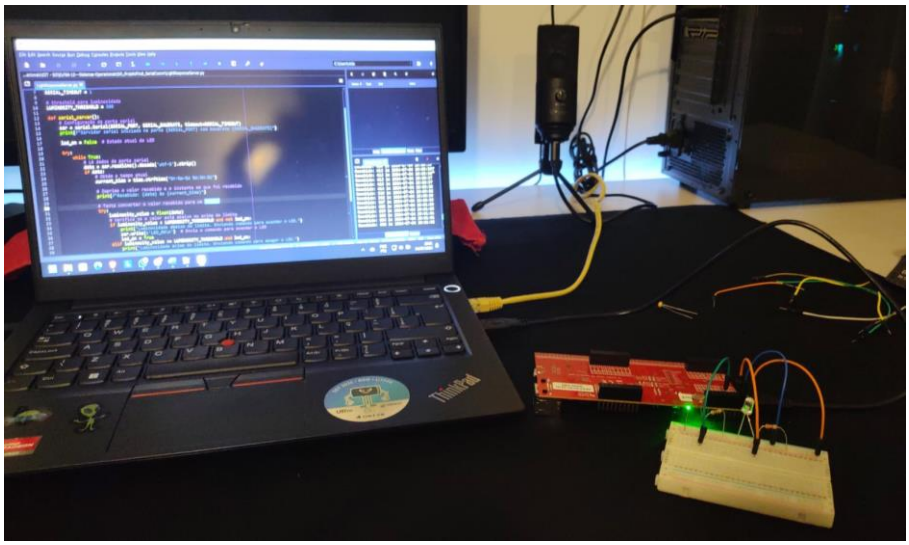

Figura 4: circuito acoplado

Em anexo a este relatório, um vídeo demonstrativo do circuito funcionando está contido.

Conclusão:

O desenvolvimento deste permitiu uma aplicação prática dos conceitos abordados na disciplina de Sistemas Operacionais. Sua implementação demonstrou a importância da integração entre *hardware* e *software* na criação de sistemas de tempo real eficientes e responsivos. Ao longo dele, foi possível explorar de forma mais aprofundada áreas fundamentais, como criação e gerenciamento de tarefas, o uso de *GPIO* para comunicação entre periféricos e CPU, e a aplicação de escalonamento de tarefas para garantir a responsividade do sistema.

A experiência adquirida com a utilização prática da placa TIVA reforçou a compreensão da importância de um *design* eficiente e da comunicação entre os componentes do sistema, cruciais para o funcionamento correto de sistemas embarcados, disciplina futura do curso de Engenharia Eletrônica. Em suma, o projeto não apenas consolidou os conhecimentos adquiridos na disciplina de Sistemas Operacionais, assim como ajuda na criação de uma base para futuros projetos e aplicações no campo da engenharia de sistemas embarcados.

Referências:

- KOBYL, A. (n.d.). ***TM4C129 FreeRTOS Demo***. GitHub. Disponível em:https://github.com/akobyl/TM4C129_FreeRTOS_Demo. Acesso em: julho de 2024.

- VITRORROR. (n.d.). ***ELF66-12 Sistemas Operacionais***. GitHub. Disponível em:https://github.com/Vitroror/ELF66-12-Sistemas-Operacionais. Acesso em: julho de 2024.

- ENERGIA. (n.d.). **EK-TM4C1294XL Pin Maps**. Disponível em:https://energia.nu/pinmaps/ek-tm4c1294xl/. Acesso em: julho de 2024.

- LUZ, P. D. G. **ELF52: SISTEMAS MICROCONTROLADOS**. Disponível em:http://www.elf52.daeln.com.br/. Acesso em: julho de 2024.

- FreeRTOS. (n.d.). **FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions**. Disponível em:https://www.freertos.org/index.html. Acesso em: julho de 2024.