Dispositivos Conectados
Input/Output Básico

2025/2026

Authors: Paulo C. Bartolomeu

1 Objetivos

- Configuração e utilização de portos de saída digital;
- Configuração e utilização de portos de entrada digital (*polling* e interrupção);
- Configuração e utilização de *Queues* para comunicação entre tarefas;
- Configuração e geração de sinais PWM;
- Configuração e leitura de portos de *input* analógico.

2 Introdução

Num contexto de Internet das Coisas (IoT), a capacidade de controlar e ler sinais elétricos através dos portos de entrada e saída digital e analógica é fundamental para criar dispositivos inteligentes e interativos. Saber programar estes portos permite que um microcontrolador se comunique com sensores, atuadores e outros periféricos, transformando sinais do mundo físico em dados digitais que podem ser processados e transmitidos, ou convertendo comandos digitais em ações concretas, como acender uma luz, movimentar um motor ou regular a intensidade de um LED. Esta habilidade não só proporciona um controlo direto sobre o ambiente, mas também abre caminho à automação, à monitorização em tempo real e à integração de múltiplos dispositivos numa rede IoT. Compreender o funcionamento e a programação dos portos de I/O é, portanto, essencial para desenvolver soluções robustas, eficientes e responsivas, permitindo aos criadores transformar ideias em aplicações funcionais que interagem de forma inteligente com o mundo ao seu redor.

Nesta aula iremos explorar a utilização do kit baseado no módulo ESP32-C6 para efetuar operações de controlo sobre um LED, ler o estado de um botão de pressão e monitorizar a tensão de um divisor resistivo implementado com um potenciómetro. Tratam-se de operações básicas de I/O, mas que são fundamentais para a construção de um sistema integrados IoT.

2.1 Material complementar

Os componentes indicados na tabela seguinte serão disponibilizados durante o período da aula para testes. A **negrito** identificam-se os componentes que não são de acesso livre e serão entregues em aula para ser devolvidos no final do semestre junto com os restantes kits.

Componente	Descrição
Pot. 10 KOhm	Potenciómetro

Componente	Descrição
R 1 KOhm	Resistência
LED Vermelho	Díodo emissor de luz
SW	Botão de pressão
knob	Pino para o potenciómetro

Deverão ser implementados na placa branca os três circuitos especificados na Figura 1.

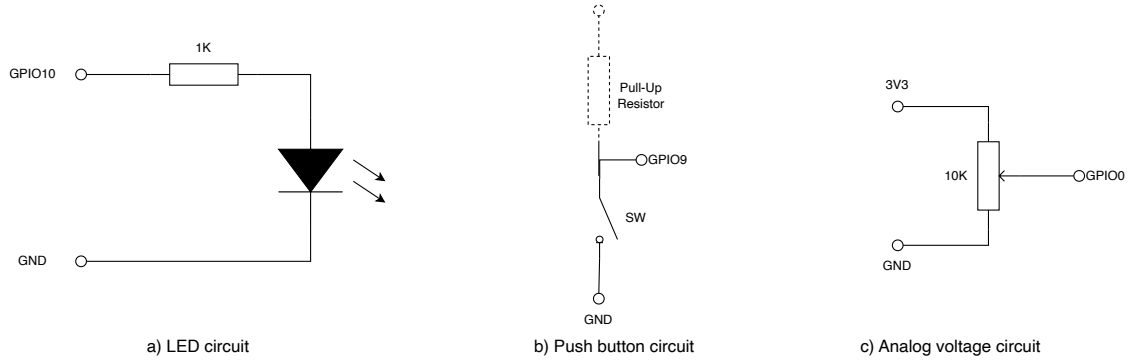

Figura 1: Circuitos de interface com o kit ESP32-C6

ATENÇÃO! Seja prudente e confirme as ligações na placa branca antes de alimentar o ESP32. Seja particularmente cauteloso a verificar as ligações ao kit ESP32-C6.

3 Trabalho Prático

O trabalho prático desta aula desenvolve-se com base na API de Referência para os periféricos do ESP32-C6 juntamente com o ESP32-C6 Technical Reference Manual.

3.1 Controlo de um LED

O passo mais elementar de qualquer sistema *embedded* é o de programar uma saída digital para alternar estre o estado "1" e o estado "0" com um determinado período.

Em seguida é facultado o "esqueleto" de um programa funcional que, conjugado com o circuito da Figura 1a) permite ativar periodicamente o respetivo LED.

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

#define BLINK_GPIO  GPIO_NUM_10   // GPIO10 on ESP32-C6

void app_main(void)
{
    // Configure GPIO10 as output
    gpio_config_t io_conf = {
        .pin_bit_mask = 1ULL << BLINK_GPIO, // bit mask for GPIO10
        .mode = GPIO_MODE_OUTPUT,
        .pull_up_en = GPIO_PULLUP_DISABLE,
```

```
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_DISABLE
    };
    gpio_config(&io_conf);

    bool level = false;

    while (1) {
        gpio_set_level(BLINK_GPIO, level);
        level = !level; // invert for next toggle
        vTaskDelay(pdMS_TO_TICKS(1000)); // delay 1 second
    }
}
```

TROBLESHOOTING: Caso encontre problemas em colocar o programa a funcionar devido à não satisfação de dependências pode editar o ficheiro `CMakeLists.txt` que está na sua pasta raiz (a que contém a sua "main") usando o bloco de código seguinte. Como não há `REQUIRES` nem `PRIV_REQUIRES`, não se declara dependências explícitas de nenhum outro componente, logo todos os componentes definidos no projeto serão compilados e ligados normalmente. Em desenvolvimento é recomendado que as dependências sejam identificadas de forma explícita recorrendo a `REQUIRES` e `PRIV_REQUIRES` no `idf_component_register`.

```
idf_component_register(SRCS "main.c"
                       INCLUDE_DIRS ".")
```

Consultando a secção GPIO & RTC GPIO da API de Referência, analise o programa e teste-o na placa observando o LED a "piscar" com um período de 2 segundos.

QUESTÃO 1: Altere o código para um período de refrescamento de 20 milisegundos e observe no osciloscópio o sinal produzido pela placa, confirmando quer a frequência pretendida quer o *duty-cycle* do sinal.

QUESTÃO 2: Reponha o período original (2 segundos) e acrescente uma impressão periódica do estado do LED de modo a que seja impressa para o terminal a seguinte informação.

```
~$ GPIO 10 set to 1
~$ GPIO 10 set to 0
~$ GPIO 10 set to 1
~$ GPIO 10 set to 0
~$ ...
```

3.2 Leitura do estado de um botão de pressão (*polling*)

Tendo como ponto de partida o circuito da Figura 1b) pretende-se agora ler o estado de um botão de pressão.

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

#define INPUT_GPIO   GPIO_NUM_9   // choose an available GPIO on your board

void app_main(void)
{
    // Configure the GPIO as input with pull-up enabled
```

```
    gpio_config_t io_conf = {
        .pin_bit_mask = 1ULL << INPUT_GPIO, // bit mask for selected GPIO
        .mode = GPIO_MODE_INPUT,
        .pull_up_en = GPIO_PULLUP_ENABLE,   // enable internal pull-up
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_DISABLE
    };
    gpio_config(&io_conf);

    while (1) {
        int level = gpio_get_level(INPUT_GPIO);  // read the pin state (0 or 1)

        // INCLUA AQUI O CODIGO PARA IMPRIMIR O ESTADO DO BOTAO

        vTaskDelay(pdMS_TO_TICKS(1000)); // check every 1s
    }
}
```

QUESTÃO 3: Analise as opções de configuração e complete o código de modo a que se visualize o estado do botão de pressão no monitor de acordo com o seguinte exemplo:

```
~$ GPIO 9 read 1
~$ GPIO 9 read 1
~$ GPIO 9 read 1
~$ GPIO 9 read 0
~$ ...
```

QUESTÃO 4: Prossiga com a compilação e execução do código desenvolvido e verifique o funcionamento. O que pode dizer sobre a lógica de funcionamento do botão de pressão? É direta ou invertida?

QUESTÃO 5: Qual o objetivo da resistência ligada ao botão de pressão e identificada no circuito da Figura 1b) a tracejado? Porque é que embora não tenha montado essa resistência, internamente ela é ligada?

3.3 Leitura do estado de um botão de pressão (*interrupt*)

Quando se utilizam interrupções em sistemas *embedded*, surge sempre o desafio de transferir informação de forma segura entre a rotina de serviço da interrupção (ISR) e a aplicação principal. Implementar operações como imprimir dentro de uma ISR não é recomendado, pois pode atrasar outros eventos importantes. As filas (queues) oferecem um mecanismo eficiente e seguro para esta comunicação. A ISR pode rapidamente colocar na fila um pequeno dado, como um sinal de evento ou uma leitura de sensor, e regressar de imediato. Mais tarde, uma tarefa do FreeRTOS, a correr fora do contexto da interrupção, retira essa informação da fila e processa-a sem restrições de tempo. Desta forma, as ISRs permanecem leves, ao mesmo tempo que se garante uma transferência de dados fiável e sincronizada entre os eventos de interrupção e a lógica da aplicação.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "driver/gpio.h"

#define INPUT_GPIO   GPIO_NUM_9   // example input pin

static QueueHandle_t gpio_evt_queue = NULL;
```

```c
// ISR handler
static void IRAM_ATTR gpio_isr_handler(void *arg)
{
    uint32_t gpio_num = (uint32_t) arg;
    // Send GPIO number to the queue (from ISR context)
    xQueueSendFromISR(gpio_evt_queue, &gpio_num, NULL);
}

// Task that handles GPIO events
static void gpio_task(void *arg)
{
    uint32_t io_num;
    for(;;) {
        if (xQueueReceive(gpio_evt_queue, &io_num, portMAX_DELAY)) {
            // INCLUA AQUI O CODIGO PARA IMPRIMIR O NUMERO DO GPIO ORIGEM DA INTERRUPCAO
        }
    }
}

void app_main(void)
{
    // Configure the GPIO as input with pull-up, interrupt on falling edge
    gpio_config_t io_conf = {
        .pin_bit_mask = 1ULL << INPUT_GPIO,
        .mode = GPIO_MODE_INPUT,
        .pull_up_en = GPIO_PULLUP_ENABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_NEGEDGE
    };
    gpio_config(&io_conf);

    // Create a queue to handle GPIO events
    gpio_evt_queue = xQueueCreate(10, sizeof(uint32_t));

    // Start the task that will process GPIO events
    xTaskCreate(gpio_task, "gpio_task", 2048, NULL, 10, NULL);

    // Install GPIO ISR service
    gpio_install_isr_service(0);

    // Hook ISR handler for specific GPIO
    gpio_isr_handler_add(INPUT_GPIO, gpio_isr_handler, (void*) INPUT_GPIO);

    printf("Waiting for GPIO %d falling edge interrupts...", INPUT_GPIO);
}
```

QUESTÃO 6: Consulte e analise as APIs de GPIO & RTC GPIO e da Queue. Estude o código fornecido e complete o programa de modo a imprimir a seguinte informação sempre que ocorra um flanco descendente no sinal do botão de pressão.

```
~$ Interrupt on GPIO 9
~$ Interrupt on GPIO 9
~$ Interrupt on GPIO 9
~$ Interrupt on GPIO 9
~$ ...
```

3.4 Controlo de uma saída de PWM

A biblioteca LEDC do ESP-IDF é uma ferramenta de eleição para gerar sinais PWM de forma precisa e eficiente em microcontroladores da família ESP32. Através dela é possível controlar a intensidade de um LED ajustando o ciclo ativo do sinal (*duty-cycle*), permitindo, por exemplo, criar

efeitos de *fade* suave ou simplesmente regular o brilho em diferentes níveis. No caso de um LED, o LEDC torna possível transformar um simples pino digital num regulador de intensidade, abrindo caminho para interfaces mais dinâmicas e interativas em projetos de IoT e sistemas embebidos.

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/ledc.h"

#define LED_GPIO       GPIO_NUM_10
#define LEDC_CHANNEL   LEDC_CHANNEL_0
#define LEDC_TIMER     LEDC_TIMER_0
#define LEDC_MODE      LEDC_LOW_SPEED_MODE
#define LEDC_DUTY_RES  LEDC_TIMER_13_BIT // 13-bit resolution
#define LEDC_FREQUENCY 5000              // 5 kHz PWM

void app_main(void)
{
    // 1. Configure timer
    ledc_timer_config_t ledc_timer = {
        .speed_mode      = LEDC_MODE,
        .timer_num       = LEDC_TIMER,
        .duty_resolution = LEDC_DUTY_RES,
        .freq_hz         = LEDC_FREQUENCY,
        .clk_cfg         = LEDC_AUTO_CLK
    };
    ledc_timer_config(&ledc_timer);

    // 2. Configure channel
    ledc_channel_config_t ledc_channel = {
        .gpio_num  = LED_GPIO,
        .speed_mode= LEDC_MODE,
        .channel   = LEDC_CHANNEL,
        .intr_type = LEDC_INTR_DISABLE,
        .timer_sel = LEDC_TIMER,
        .duty      = 0,
        .hpoint    = 0
    };
    ledc_channel_config(&ledc_channel);

    uint32_t max_duty = (1 << LEDC_DUTY_RES) - 1;
    uint32_t duty_10 = max_duty / 10;

    while (1) {
        ledc_set_duty(LEDC_MODE, LEDC_CHANNEL, duty_10);
        ledc_update_duty(LEDC_MODE, LEDC_CHANNEL);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

QUESTÃO 7: Estude a API LEDC e analise o funcionamento do código do exemplo.

QUESTÃO 8: Altere o código para diferentes valores de *duty-cycle* e observe o sinal de *drive* do LED no osciloscópio, confirmando-o. Como justifica a frequência usada para o PWM?

QUESTÃO 9: Altere o código de modo a que seja possível observar o LED com um efeito de "respirar", ou seja, vai incrementalmente aumentando o brilho até alcançar o máximo e depois diminui até chegar ao mínimo, repetindo este processo indefinidamente.

3.5 Configuração e utilização da ADC no modo One-shot

O modo de funcionamento oneshot da ADC aplica-se quando é necessário obter leituras pontuais e precisas de sinais analógicos, sem manter a conversão ativa de forma contínua. Este modo permite ao processador solicitar uma medição apenas quando necessário, reduzindo o consumo de energia e libertando recursos do sistema, o que é especialmente importante em dispositivos IoT alimentados a bateria. Além disso, o oneshot simplifica a integração com tarefas específicas, como a monitorização de sensores em intervalos definidos ou a leitura de valores críticos em resposta a eventos, garantindo eficiência e precisão no tratamento de sinais analógicos dentro de arquiteturas *embedded*.

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_adc/adc_oneshot.h"

#define ADC_CHANNEL     ADC_CHANNEL_0   // GPIO0 (check pin mapping for ESP32-C6)
#define ADC_UNIT        ADC_UNIT_1      // Use ADC1
#define ADC_ATTEN       ADC_ATTEN_DB_12  // 12 dB, ~1.1 V full-scale

void app_main(void)
{
    // ADC Oneshot driver handle
    adc_oneshot_unit_handle_t adc1_handle;
    adc_oneshot_unit_init_cfg_t init_config1 = {
        .unit_id = ADC_UNIT,
    };
    adc_oneshot_new_unit(&init_config1, &adc1_handle);

    // Configure channel
    adc_oneshot_chan_cfg_t config = {
        .atten = ADC_ATTEN,
        .bitwidth = ADC_BITWIDTH_DEFAULT, // default = 12-bit
    };
    adc_oneshot_config_channel(adc1_handle, ADC_CHANNEL, &config);

    while (1) {
        int adc_raw = 0;
        adc_oneshot_read(adc1_handle, ADC_CHANNEL, &adc_raw);

        // Convert raw value to voltage (approximate, no calibration)
        float voltage = (adc_raw / 4095.0f) * (1.1f/0.25f);

       // INCLUA AQUI O CODIGO PARA IMPRIMIR O VALOR DIGITAL LIDO E O CORREPONDENTE ANALOGICO

        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

QUESTÃO 10: Estude o driver Oneshot da ADC e complete o código de modo a que seja impressa a seguinte informação a cada segundo:

```
~$ ADC Raw: 4095, Voltage: 3.899 V
~$ ADC Raw: 4095, Voltage: 3.899 V
~$ ADC Raw: 4095, Voltage: 3.899 V
~$ ADC Raw: 4095, Voltage: 3.899 V
~$ ...
```

QUESTÃO 11: Observe o valor de tensão analógico e compare-o com o valor obtido pelo seu programa. Como justifica a discrepância?

QUESTÃO 12: Implemente um mecanismo básico de calibração e teste-o na prática,

avaliando o seu funcionamento.

QUESTÃO 13: Estude a API da ADC e identifique uma alternativa ao método empregue e implemente-a, comparando com o resultado da calibração básica na questão anterior.

QUESTÃO 14: O ESP-IDF permite criar logs para as diferentes funcionalidades através da Logging library. Repita os exercícios anteriores substituindo o uso da função `printf` pela macro `ESP_LOGI`.

4 Acknowledgements

Originally authored by Paulo C. Bartolomeu