UNIVERSIDADE FEDERAL DO RECÔNCAVO DA BAHIA
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

RELATÓRIO
MODULOS DE SISTEMA: ESP32

Gabriel Sampaio de Oliveira
Erik Montgomery de Britto

Cruz das Almas
2024

UNIVERSIDADE FEDERAL DO RECÔNCAVO DA BAHIA
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

RELATÓRIO
MODULOS DE SISTEMA: ESP32

Gabriel Sampaio de Oliveira
Erik Montgomery de Britto

Relatório 2 da disciplina de Sistemas Embarcados (GCET528) - sob a orientação do Professor Dr. Igor Dantas dos Santos Miranda.

Cruz das Almas
2024

Sumário

1 Introdução

O ESP32, um microcontrolador poderoso e versátil da Espressif, tem se destacado no desenvolvimento de soluções embarcadas, oferecendo múltiplas funcionalidades em um único chip. Suas capacidades de processamento dual-core, conectividade Wi-Fi e Bluetooth, e uma vasta gama de pinos de entrada e saída (GPIOs) tornam-no ideal para uma ampla variedade de aplicações, desde automação residencial até dispositivos IoT (Internet das Coisas).

Este relatório aborda as características essenciais do ESP32, com foco particular em suas interfaces de pinos e funcionalidades integradas. Discutimos o uso dos GPIOs, que suportam tanto entradas quanto saídas digitais, com a possibilidade de configuração de resistores de pull-up e pull-down via software. Exploramos as capacidades do ADC (Conversor Analógico-Digital) e DAC (Conversor Digital-Analógico), permitindo ao ESP32 interagir com sensores analógicos e gerar sinais de saída analógicos.

Além disso, abordamos o uso de interrupções, uma funcionalidade crucial para a criação de sistemas responsivos e eficientes, onde eventos externos podem ser capturados e processados em tempo real. A comunicação UART, fundamental para a integração com outros dispositivos, também é explorada, destacando a flexibilidade e facilidade de uso da interface serial do ESP32. Por fim, discutimos os temporizadores integrados, que permitem a execução de tarefas periódicas e o controle preciso de eventos temporais.

2 Experimentos

2.1 Lab 05 - Estudo de Portas digitais

2.1.1 Objetivo

Avaliar o funcionamento das portas digitais no ESP32.

2.1.2 Procedimentos

2.1.2.1 Parte 1 - Portas de entrada

A tensão máxima suportada pelas entradas do ESP32 é de 3,6V. Tentar aplicar uma tensão acima desse valor pode danificar o seu Chip. Para leituras de ADC, a faixa típica de operação é de 0 a 3,3V.

Foi feita a análise do ESP32,e constatado que as portas de entrada do ESP32 podem ter resistores de pull-up, pull-down ou nenhum resistor, dependendo da configuração desejada. Esses resistores são configuráveis por software para cada pino de entrada digital. Aqui está um resumo:

- Resistor de Pull-up: Pode ser ativado para manter o pino em um nível lógico alto (3,3V) quando não está conectado ou quando o dispositivo conectado ao pino está em alta impedância.
- Resistor de Pull-down: Pode ser ativado para manter o pino em um nível lógico baixo (0V) quando não está conectado ou quando o dispositivo conectado ao pino está em alta impedância.
- Sem Resistor: O pino pode ser configurado sem resistor de pull-up ou pull-down, deixando o pino flutuante (indeterminado) se não estiver conectado a uma fonte de sinal externo.

O código utilizado para este teste foi o seguinte:

Código C utilizado para o Lab 05 - Parte 1

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

#define TEST_GPIO 4
```

```
8
9  void test_gpio_pull_mode(gpio_num_t gpio_num, gpio_pull_mode_t pull_mode) {
10
11 gpio_config_t io_conf = {
12     .pin_bit_mask = (1ULL << gpio_num),
13
14     .mode = GPIO_MODE_INPUT,
15     .pull_down_en = GPIO_PULLDOWN_DISABLE,
16     .pull_up_en = GPIO_PULLUP_DISABLE,
17     .intr_type = GPIO_INTR_DISABLE
18 };
19
20    // Verifica se a configura o    para habilitar o resistor de pull-up
21 if (pull_mode == GPIO_PULLUP_ONLY) {
22     // Habilita o resistor de pull-up para o pino
23     io_conf.pull_up_en = GPIO_PULLUP_ENABLE;
24 }
25 // Verifica se a configura o    para habilitar o resistor de pull-down
26 else if (pull_mode == GPIO_PULLDOWN_ONLY) {
27     // Habilita o resistor de pull-down para o pino
28     io_conf.pull_down_en = GPIO_PULLDOWN_ENABLE;
29 }
30
31 gpio_config(&io_conf);
32 int value = gpio_get_level(gpio_num);
33
34     if (pull_mode == GPIO_PULLUP_ONLY) {
35         printf("GPIO %d configurado com resistor pull-up. Valor lido: %d\n"
    , gpio_num, value);
36     } else if (pull_mode == GPIO_PULLDOWN_ONLY) {
37         printf("GPIO %d configurado com resistor pull-down. Valor lido: %d\
    n", gpio_num, value);
38     } else {
39         printf("GPIO %d configurado sem resistor interno. Valor lido: %d\n"
    , gpio_num, value);
40     }
41 }
42
43 void app_main(void) {
44
45     printf("GPIO entrada diferentes configura es");
46
47     // Teste da porta com resistor pull-up
48     //Manda o numero da porta de entrada e o modo da porta de entrada
49     test_gpio_pull_mode(TEST_GPIO, GPIO_PULLUP_ONLY);
50     // Aguarda um pouco antes do pr ximo teste
51     vTaskDelay(pdMS_TO_TICKS(1000));
```

```
52
53      // Teste da porta com resistor pull-down
54      test_gpio_pull_mode(TEST_GPIO, GPIO_PULLDOWN_ONLY);
55      // Aguarda um pouco antes do pr ximo teste
56      vTaskDelay(pdMS_TO_TICKS(1000));
57
58      // Teste da porta sem resistor interno
59      test_gpio_pull_mode(TEST_GPIO, GPIO_FLOATING);
60      vTaskDelay(pdMS_TO_TICKS(1000));
61
62  }
```

2.1.2.2 Parte 2 - Porta de saída

A corrente máxima que uma porta de saída (GPIO) do ESP32 pode fornecer ou drenar é de 40 mA por pino. No entanto, para evitar sobrecarga e garantir a longevidade do dispositivo, é recomendável limitar a corrente a 20 mA por pino, sendo que pode ocorrer sobrecarga em caso de uso de conjunto de pinos.

2.1.2.3 Parte 3 - Frequência máxima

Foi gerado um código a fim de avaliar frequência máxima de saída das portas digitais e frequência máxima das portas entrada e medido com o auxílio do osciloscópio.

Código C utilizado para o Lab 05 - Parte 3

```
1  #include <stdio.h>
2  #include "freertos/FreeRTOS.h"
3  #include "freertos/task.h"
4  #include "driver/gpio.h"
5  #include <rom/ets_sys.h>
6
7  #define INPUT_GPIO 4  // Define o pino de entrada que voc  deseja testar
8
9  void app_main(void) {
10     // Vari veis para armazenar o tempo da  ltima  transi  o e o tempo
       atual
11     uint32_t  ultimo = xTaskGetTickCount();
12     uint32_t  atual = xTaskGetTickCount();
13     double frequencia;
14     uint32_t cont=0;
15     // Vari vel para armazenar o per odo entre transi  es
16     double  period = 0;
17     // Inicializa o n vel atual como 0 (baixo)
18     int current_level = 0;
19 while (1){
20     cont = 0;
```

```
21      ultimo = xTaskGetTickCount();
22
23  // Loop principal
24      while (cont<5000000) {
25          // L   o n vel  l gico atual do pino de entrada
26          int new_level = gpio_get_level(INPUT_GPIO);
27
28          // Verifica se houve t r a n s i   o de alto para baixo ou de baixo
    para alto
29          if (new_level != current_level) {
30              // Atualiza o tempo atual com o tempo de tick atual
31
32              // Calcula o per odo da onda como a diferen a entre o tempo
    atual e o tempo da  ltima  t r a n s i
33          // period += (atual - ultimo);
34                  // Calcula a frequ ncia a partir do per odo
35
36              // Atualiza o tempo da  ltima  t r a n s i  o com o tempo atual
37              //ultimo = atual;
38
39              // Atualiza o  n vel atual para o novo  n vel lido
40              current_level = new_level;
41              cont++;
42          }
43
44          // Adiciona um pequeno atraso para evitar alta carga na CPU
45          // ets_delay_us(50);
46      }
47          atual = xTaskGetTickCount();
48          period= (double)(atual-ultimo);
49          period /= 100*cont/2;
50
51          frequencia = 1/period;
52
53       // Imprime a frequ ncia detectada em Hz
54          printf("Frequ ncia detectada: %f Hz, %f s  \n", frequencia, period
    );
55
56  vTaskDelay(pdMS_TO_TICKS(500));
57
58  }
59  }
```

Para a frequência máxima de saída das portas digitais foi encontrado o valor de 9,89 kHz, sendo esta a frequência que ele opera sem overshoot. E para frequência máxima das portas de entrada o valor de 1,6 MHz.

Os testes com as portas digitais do ESP32 mostraram que o dispositivo é versátil e robusto para diferentes aplicações. A configuração correta das portas de entrada, especialmente com resistores pull-up e pull-down, é essencial para evitar problemas de leitura. No caso das portas de saída, vimos que respeitar os limites de corrente é crucial para a longevidade do chip. Por fim, o ESP32 demonstrou capacidade de operar em frequências relativamente altas, confirmando sua eficiência para projetos que exigem resposta rápida. Em resumo, o ESP32 se mostrou confiável, mas é importante ajustar suas configurações conforme a necessidade da aplicação.

2.2 Lab 06 - Conversão Analógico-Digital

2.2.1 Objetivo

Estudar o funcionamento dos conversores analógico-digital, utilizando o ESP32.

2.2.2 Procedimentos

2.2.2.1 Parte 1 - Entendendo o ADC do ESP32

Na documentação do ESP32, foi pesquisado as principais informações a respeito das características e configurações do ADC:

- Número de canais: 18 canais de ADC (10 canais no ADC1 e 8 canais no ADC2).
- Resolução: A resolução do ADC do ESP32 é configurável até 12 bits, o que proporciona uma contagem máxima de 4096 níveis.
- Faixa de tensão: A faixa de tensão de entrada para o ADC do ESP32 vai de 0 a 3,3.
- Taxa de amostragem máxima: Aproximadamente 6 ksps (kilo samples per second).
- Principais modos de uso: A biblioteca analogRead() é a função básica para leitura de valores do ADC. O ADC pode trabalhar com DMA (Direct Memory Access) para transferir dados diretamente para a memória sem a intervenção da CPU.

2.2.2.2 Parte 2 - Experimento com ADC

Foi gerado um código que usa a porta do ESP32 como leitor de tensão. O leitor consegue ler tensões que variam de 0 a 3,3 volt representados por 12 bits (0 a 4095).

Código C utilizado para o Lab 06

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

```
#include "esp_log.h"
#include "led_strip.h"
#include "sdkconfig.h"
#include "driver/adc.h"

static const char *TAG = "example";

#define delay(value)  vTaskDelay( value/portTICK_PERIOD_MS )
#define ADC_pin 4
#define BLINK_GPIO CONFIG_BLINK_GPIO
static uint8_t s_led_state = 0;
#ifdef CONFIG_BLINK_LED_STRIP
static led_strip_handle_t led_strip;

static void blink_led(void)
{
    /* If the addressable LED is enabled */
    if (s_led_state) {
        /* Set the LED pixel using RGB from 0 (0%) to 255 (100%) for each color */
        led_strip_set_pixel(led_strip, 0, 16, 16, 16);
        /* Refresh the strip to send data */
        led_strip_refresh(led_strip);
    } else {
        /* Set all LED off to clear all pixels */
        led_strip_clear(led_strip);
    }
}

static void configure_led(void)
{
    ESP_LOGI(TAG, "Example configured to blink addressable LED!");
    /* LED strip initialization with the GPIO and pixels number*/
    led_strip_config_t strip_config = {
        .strip_gpio_num = BLINK_GPIO,
        .max_leds = 1, // at least one LED on board
    };
#if CONFIG_BLINK_LED_STRIP_BACKEND_RMT
    led_strip_rmt_config_t rmt_config = {
        .resolution_hz = 10 * 1000 * 1000, // 10MHz
        .flags.with_dma = false,
    };
    ESP_ERROR_CHECK(led_strip_new_rmt_device(&strip_config, &rmt_config, &led_strip));
#elif CONFIG_BLINK_LED_STRIP_BACKEND_SPI
    led_strip_spi_config_t spi_config = {
        .spi_bus = SPI2_HOST,
```

```
50          .flags.with_dma = true,
51      };
52      ESP_ERROR_CHECK(led_strip_new_spi_device(&strip_config, &spi_config, &
        led_strip));
53  #else
54  #error "unsupported LED strip backend"
55  #endif
56      /* Set all LED off to clear all pixels */
57      led_strip_clear(led_strip);
58  }
59
60  #elif CONFIG_BLINK_LED_GPIO
61
62  static void blink_led(void)
63  {
64      /* Set the GPIO level according to the state (LOW or HIGH)*/
65      gpio_set_level(BLINK_GPIO, s_led_state);
66  }
67
68  static void configure_led(void)
69  {
70      ESP_LOGI(TAG, "Example configured to blink GPIO LED!");
71      gpio_reset_pin(BLINK_GPIO);
72      /* Set the GPIO as a push/pull output */
73      gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
74  }
75
76  #else
77  #error "unsupported LED type"
78  #endif
79
80  void app_main(void)
81  {
82      int adc_values[20] = {0}; // Array para armazenar os  ltimos  20
        valores
83      int idx = 0; //  ndice  atual no array circular
84      int sum = 0; // Soma dos  ltimos  20 valores
85
86      while (1) {
87          // Leitura do valor anal gico
88          int adc_value = adc1_get_raw(ADC_pin);
89
90          // Atualiza a soma e o array circular
91          sum = sum - adc_values[idx] + adc_value;
92          adc_values[idx] = adc_value;
93          idx = (idx + 1) % 20;
94
```

```
95          // Calcula a m dia
96          float avg_voltage = (sum / (float)20) ;
97
98          printf("M dia dos  ltimos  20 valores: %.2f\n", avg_voltage);
99          delay(100);
100     }
101 }
```

Utilizando um potenciômetro de 100K Ohn e um multímetro, foi realizada a averiguação da linearidade da conversão, variando valores em toda a faixa (0 a 4095) conforme a tabela abaixo e plotado um gráfico que representa a mesma.

Bits	0	200	470	900	1500	2000	2500	3000	3300	3700	4095
Tensão	0	0,243	0,36	0,564	0,879	1,5	1,8	2,21	2,47	2,78	3,26

Tabela 1 – Tensão vs Bits

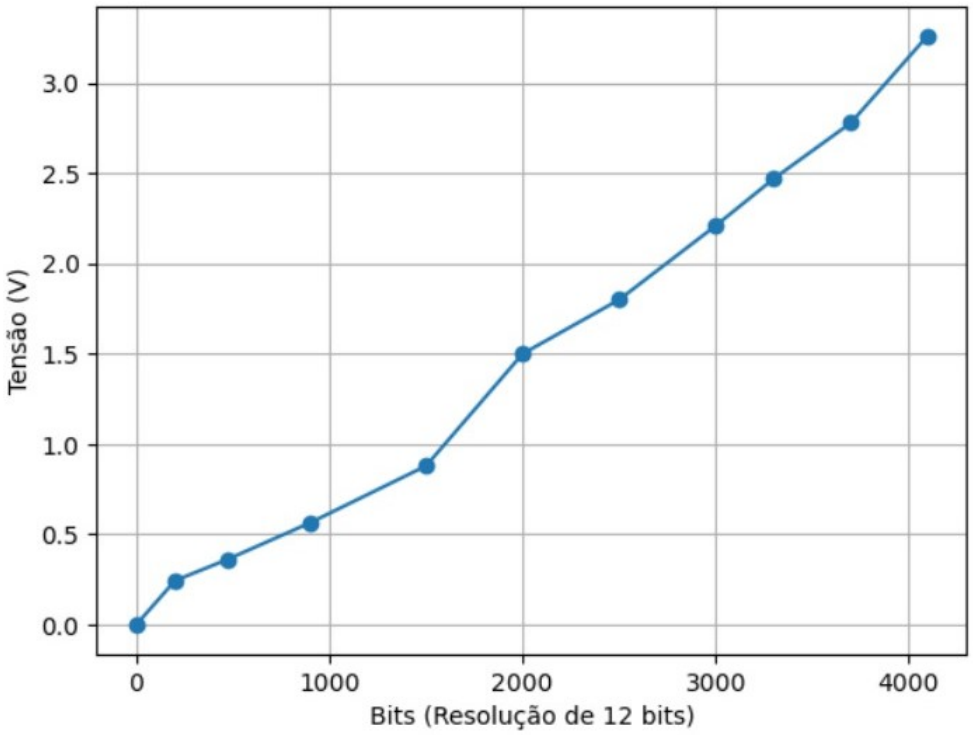

Figura 1 – Gráfico de Tensão vs Bits

Não foi possível observar a menor variação de tensão que provoca a variação de 1 bit variando o potenciômetro manualmente, devido a isso foi feito o cálculo utilizando os valore encontrados na tabela, para cada 1 bit alterado, a tensão varia em torno de 0,8 mV.

Os testes com o conversor analógico-digital (ADC) do ESP32 mostraram que ele é preciso e eficiente para medir sinais analógicos. A resolução de 12 bits e a faixa de tensão de 0 a 3,3V permitem medições detalhadas, embora seja difícil perceber manualmente variações

muito pequenas. No geral, o ESP32 se mostrou confiável para conversão de sinais em projetos de sistemas embarcados.

2.3 Lab 07 - Interrupções e Contadores de Pulso

2.3.1 Objetivo

Estudar o funcionamento das Interrupções e dos Contadores de Pulso no ESP32.

2.3.2 Procedimentos

2.3.2.1 Parte 1 - Interrupção básica

Foi configurado uma interrupção associada a um porta externa do microcontrolador de forma que toda vez que o botão for pressionado, gere uma mensagem no console confirmando o funcionamento da interrupção.

Código C utilizado para o Lab 07 - Parte 1

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

#define RED GPIO_NUM_13
#define DELAY_TIME_MS 200

volatile bool button_pressed = false;

void IRAM_ATTR gpio_isr_handler(void* arg) {
    button_pressed = true;
}

void button_config() {
    printf("Configuring button\n");
    gpio_install_isr_service(0);
    gpio_set_direction(RED, GPIO_MODE_INPUT);
    gpio_pullup_en(RED);
    gpio_set_intr_type(RED, GPIO_INTR_POSEDGE);
    gpio_isr_handler_add(RED, gpio_isr_handler, NULL);
    printf("Button config complete\n");
}

void app_main(void) {
    int cont = 0;
    printf("Start\n");
```

```
    button_config();

    while (1) {
        printf("Counter: %d\n", cont);
        if (button_pressed) {
            button_pressed = false; // Resetar o flag ap s  detec  o
            cont = 0; // Resetar o contador
            printf("OPA\n");
        }
        else {
            cont++; // Incrementar o contador se o bot o  n  o estiver
   pressionado
        }
        vTaskDelay(pdMS_TO_TICKS(DELAY_TIME_MS));
    }
}
```

2.3.2.2 Parte 2 - Medidor de Frequências

Na parte 2 foi implementado um sistema de medidor de frequência usando o módulo PCNT do ESP32 e conectado ao gerador de sinal com uma onda quadrada de 100kH sendo medidos 99540 pulsos por segundo em média.

Em seguida foi variada a frequência de forma a encontrar a frequência máxima que pode ser medida com erro menor que 5%. E encontrado o valor de 228kHz para uma frequência de 240Khz gerada.

Código C utilizado para o Lab 07 - Parte 2

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

#define RED GPIO_NUM_13
#define PCNT_INPUT_SIG_IO 4
#define DELAY_TIME_MS 200

volatile bool button_pressed = false;

void IRAM_ATTR gpio_isr_handler(void* arg) {
    button_pressed = true;
}

static void pcnt_example_init(void) {
```

```
20      pcnt_config_t pcnt_config = {
21          .pulse_gpio_num = PCNT_INPUT_SIG_IO,
22          .ctrl_gpio_num = PCNT_PIN_NOT_USED, // N o usar pino de controle
23          .channel = PCNT_CHANNEL_0,
24          .unit = PCNT_UNIT_0,
25          .pos_mode = PCNT_COUNT_INC,   // Contar pulsos na borda de subida
26          .neg_mode = PCNT_COUNT_DIS,   // N o contar pulsos na borda de
    descida
27          .lctrl_mode = PCNT_MODE_KEEP, // Ignorar sinal de controle
28          .hctrl_mode = PCNT_MODE_KEEP, // Ignorar sinal de controle
29          .counter_h_lim = 0,           // Sem limite superior
30          .counter_l_lim = 0,           // Sem limite inferior
31      };
32
33      // Inicializa a unidade de contagem de pulsos
34      pcnt_unit_config(&pcnt_config);
35
36      // Configura o filtro de pulsos (opcional)
37      pcnt_set_filter_value(PCNT_UNIT_0, 100);
38      pcnt_filter_enable(PCNT_UNIT_0);
39
40      // Inicializa o contador
41      pcnt_counter_pause(PCNT_UNIT_0);
42      pcnt_counter_clear(PCNT_UNIT_0);
43
44      // Inicia o contador
45      pcnt_counter_resume(PCNT_UNIT_0);
46  }
47
48  void button_config() {
49      printf("Configuring button\n");
50      gpio_install_isr_service(0);
51      gpio_set_direction(RED, GPIO_MODE_INPUT);
52      gpio_pullup_en(RED);
53      gpio_set_intr_type(RED, GPIO_INTR_POSEDGE);
54      gpio_isr_handler_add(RED, gpio_isr_handler, NULL);
55      printf("Button config complete\n");
56  }
57
58  void app_main(void) {
59      int16_t count = 0;
60      printf("Start\n");
61      button_config();
62      pcnt_example_init();
63
64      while (1) {
65          pcnt_counter_clear(PCNT_UNIT_0);
```

```
66          vTaskDelay(pdMS_TO_TICKS(50));
67          pcnt_get_counter_value(PCNT_UNIT_0, &count);
68          printf("Pulse count: %d\n", count);
69      }
70  }
```

Os experimentos realizados demonstraram a eficiência das interrupções e dos contadores de pulso no ESP32 para tarefas que exigem resposta rápida e medição precisa de sinais. A implementação da interrupção básica mostrou como o ESP32 pode detectar e reagir a eventos externos, enquanto o medidor de frequência revelou a capacidade do módulo PCNT de contar pulsos com alta precisão, mesmo em frequências elevadas. Esses recursos são essenciais para projetos que requerem sincronização e controle precisos em tempo real.

2.4 Lab 08 - Temporizadores e Conversão A/D

2.4.1 Objetivo

Estudar o funcionamento dos temporizadores e da conversão analógica-digital no ESP32.

2.4.2 Procedimentos

2.4.2.1 Parte 1 - Temporizador básico

Na primeira etapa do projeto, foi configurado um temporizador que gera uma interrupção a cada 1 segundo, toda vez que a interrupção acontece, o estado do LED é alterado. Segue o código desenvolvido:

Código C utilizado para o Lab 08 - Parte 1

```
1  #include <stdio.h>
2  #include "freertos/FreeRTOS.h"
3  #include "freertos/task.h"
4  #include "freertos/timers.h"
5  #include "driver/gpio.h"
6  #include "esp_system.h"
7  #include "esp_timer.h"
8
9  // Definindo os pinos e vari veis
10 #define LED_GPIO_PIN GPIO_NUM_2
11
12 static int segundos = 0, minutos = 0, horas = 0;
13 bool casa = 1 ;
14
15 static void IRAM_ATTR timer_callback(void* arg) {
16
17     segundos++;
```

```
18
19      casa = !casa;
20      gpio_set_level(LED_GPIO_PIN, casa);
21
22      if (segundos > 59) {
23          segundos = 0;
24          minutos++;
25      }
26      if (minutos > 59) {
27          minutos = 0;
28          horas++;
29      }
30      if (horas > 23) {
31          horas = 0;
32      }
33
34      printf("%02d:%02d:%02d\n", horas, minutos, segundos);
35  }
36
37  void app_main() {
38
39      // Configura o pino do LED como sa da
40      gpio_reset_pin(LED_GPIO_PIN);
41      gpio_set_direction(LED_GPIO_PIN, GPIO_MODE_OUTPUT);
42
43      // Configura e inicia o temporizador
44      const esp_timer_create_args_t timer_args = {
45          .callback = &timer_callback,
46          .name = "periodic"
47      };
48
49      esp_timer_handle_t periodic_timer;
50      esp_timer_create(&timer_args, &periodic_timer);
51      esp_timer_start_periodic(periodic_timer, 1000000); // 1 s em ms
52  }
```

2.4.2.2 Parte 2 - Conversão analógica digital

Foi desenvolvido um código em C para fazer leituras de um sinal analógico usando o ADC do ESP32. Também foi utilizado um potenciômetro para variar a tensão na entrada do ADC. Em seguida a variação foi observada imprimindo o resultado a cada 1 segundo.

2.4.2.3 Parte 3 - Coleta de forma onda

O código a seguir faz 1000 leituras do ADC em um segundo (taxa de amostragem de 1kHz), armazenando os valores em um vetor.

Código C utilizado para o Lab 08 - Parte 3

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "esp_log.h"
#include "sdkconfig.h"
#include "esp_timer.h"
#include <driver/adc.h>
#include <esp_adc_cal.h>

int cont = 0;
uint32_t voltage[1000];
esp_timer_handle_t timer_handler;
esp_adc_cal_characteristics_t adc_cal;

static const char *TAG = "ADC CAL";

static void periodic_timer_callback(void* arg){
    uint32_t adc_reading = adc1_get_raw(ADC1_CHANNEL_0);
    voltage[cont] = esp_adc_cal_raw_to_voltage(adc_reading, &adc_cal);
    cont ++;
    if(cont==1000){
        for (int i = 0; i < cont; i++){
            printf("%lu, ", (unsigned long)voltage[i]);
        }
        esp_timer_stop(timer_handler);
    }
}

void timer_setup(){
    const esp_timer_create_args_t periodic_timer_args ={
        .callback = &periodic_timer_callback,
        .name = "periodic"
    };
    esp_timer_create(&periodic_timer_args, &timer_handler);

}

void app_main(void)
{
    adc1_config_width(ADC_WIDTH_BIT_12);
    adc1_config_channel_atten(ADC1_CHANNEL_0, ADC_ATTEN_DB_12);
    esp_adc_cal_characterize(ADC_UNIT_1, ADC_ATTEN_DB_12, ADC_WIDTH_BIT_12,
    1100, &adc_cal);
    timer_setup();
    esp_timer_start_periodic(timer_handler,(uint64_t) 1000);
```

```
46
47 }
```

Os dados coletados pelo script criado foram utilizados em um novo script em python para ser plotado em gráfico. Observando como entrada uma onda senoidal, uma onda quadrada e uma onda dente de serra, todas de 100 Hz, uma de cada vez. Cada forma de onda está identificada a seguir.

Neste Laboratório, exploramos como o ESP32 lida com temporizadores e conversão A/D. Ao configurar o temporizador, vimos como é fácil criar eventos precisos, como fazer um LED piscar a cada segundo. Já na parte de conversão analógica-digital, usamos o ESP32 para captar sinais analógicos e convertê-los em valores digitais, permitindo analisar diferentes formas de onda. Esses experimentos destacaram o quanto o ESP32 é prático e eficiente para controlar e monitorar sinais em tempo real.

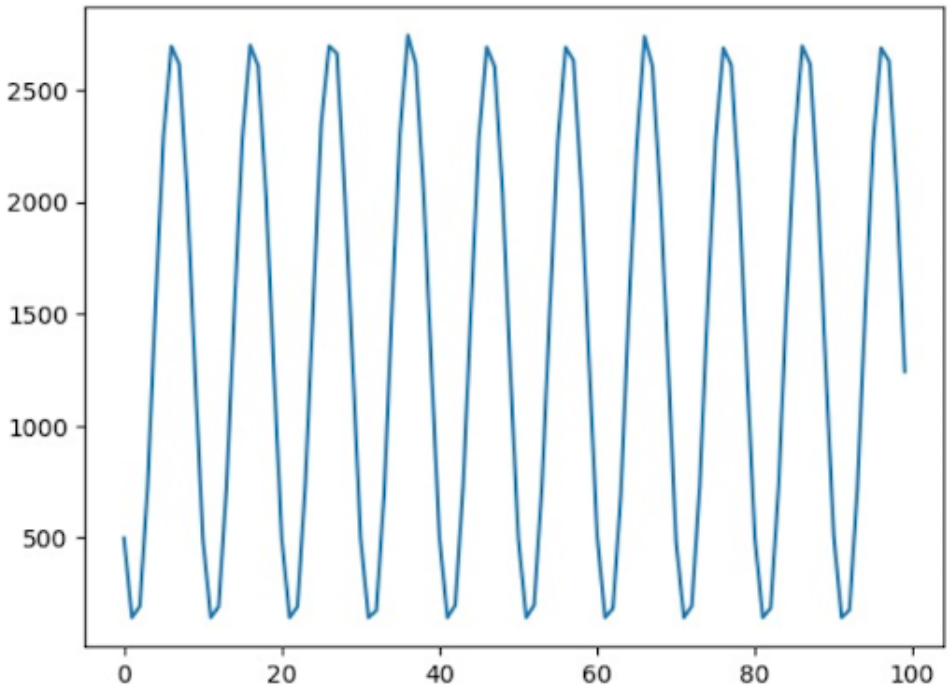

Figura 2 – Onda senoidal

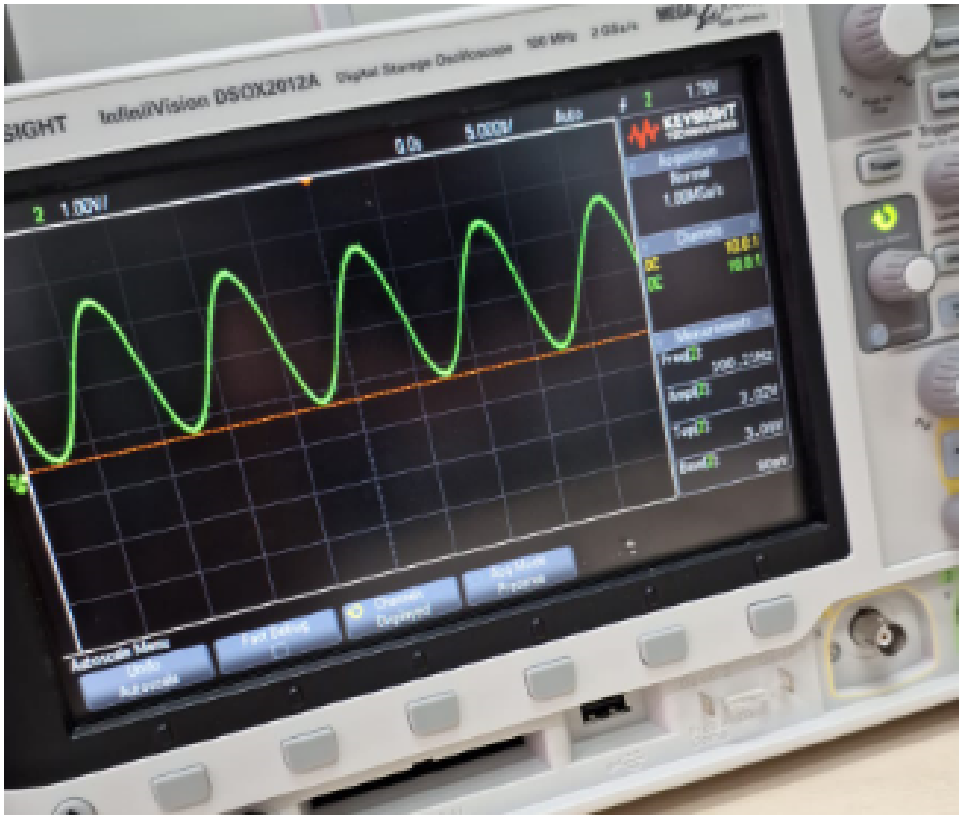

Figura 3 – Onda senoidal no osciloscópio

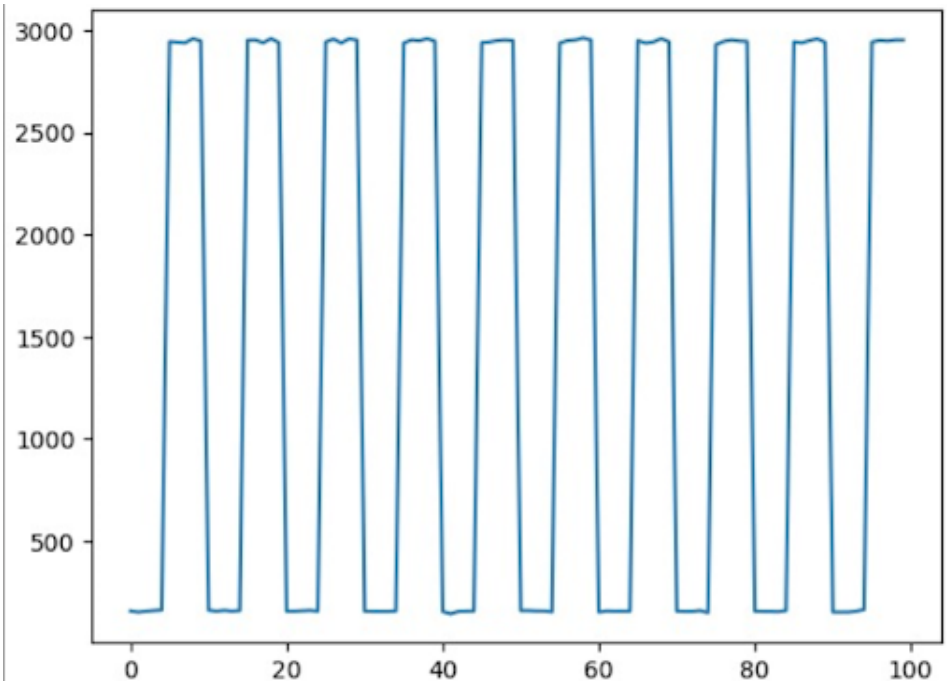

Figura 4 – Onda quadrada

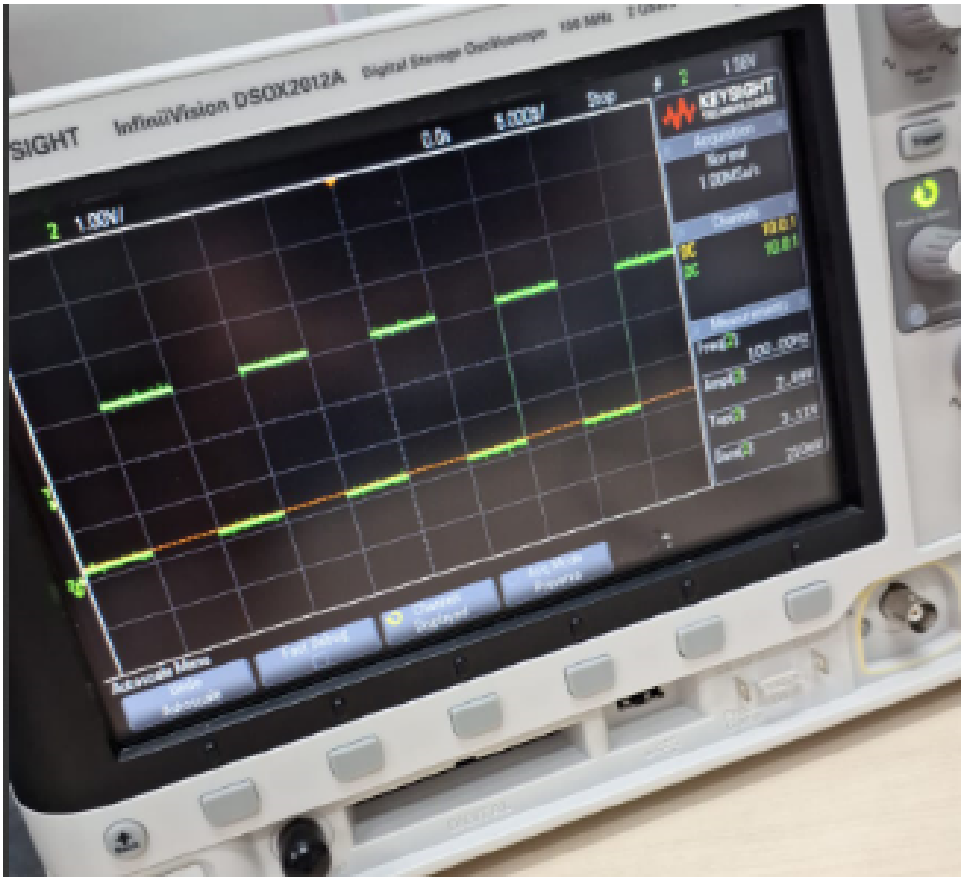

Figura 5 – Onda quadrada no osciloscópio

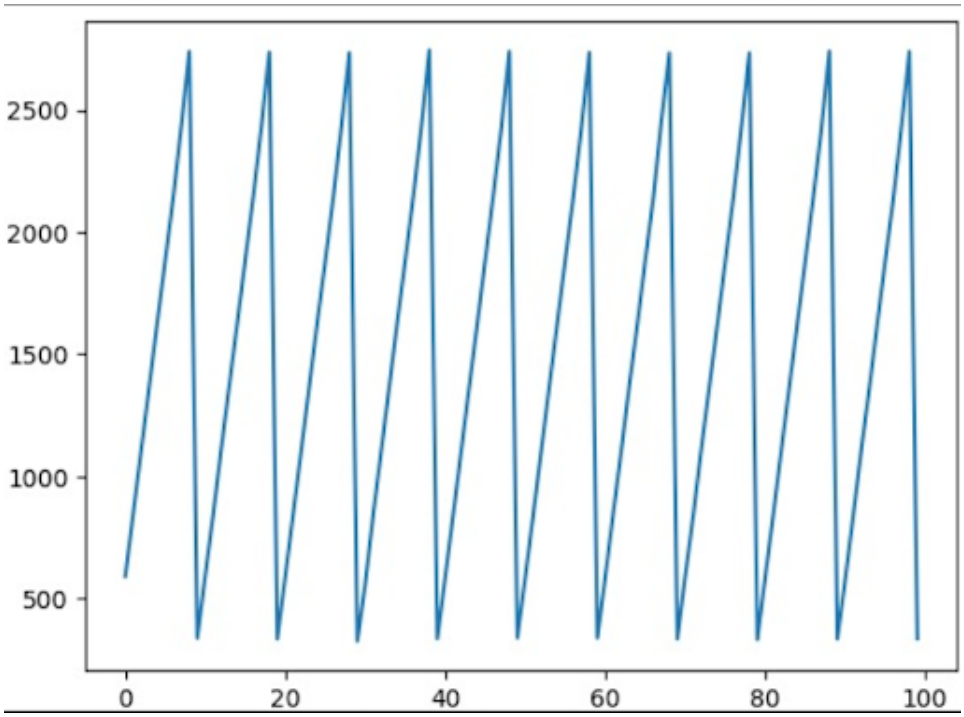

Figura 6 – Onda dente de serra

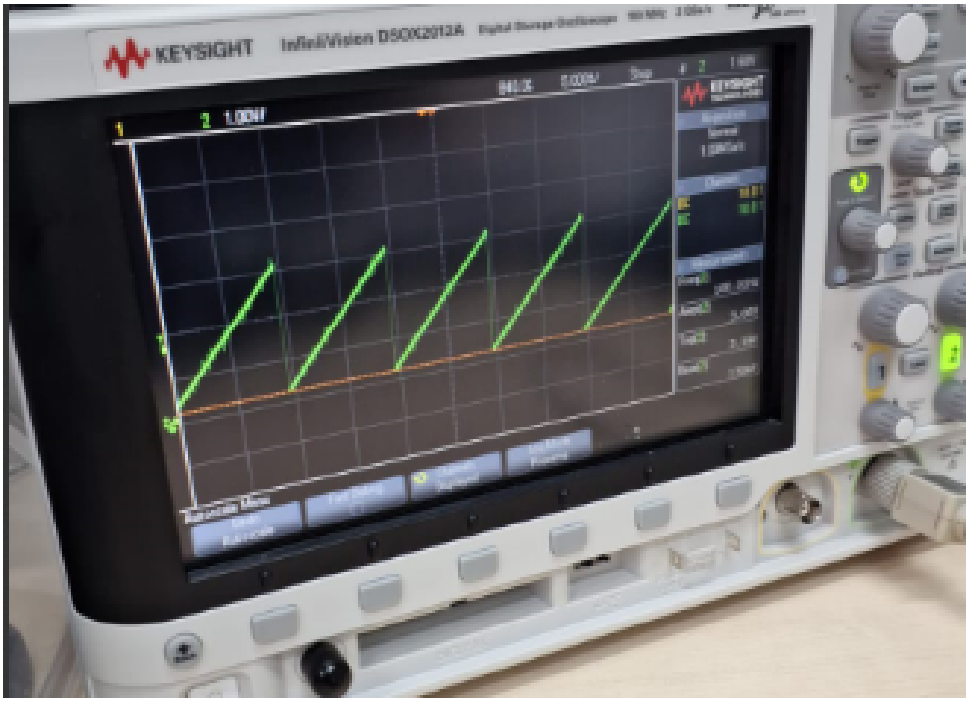

Figura 7 – Onda dente de serra no osciloscópio

2.5 Lab 09 - Protocolo UART

2.5.1 Objetivo

Analisar o funcionamento do protocolo de comunicação UART.

2.5.2 Procedimentos

2.5.2.1 Parte 1 - Comunicação UART em loopback

O desenvolvimento da primeira parte desse experimento foi a criação de um firmware que ao apertar um botão, envia uma letra do alfabeto maiúsculo pela UART indo de 'A' a 'Z' de forma cíclica. A recepção da própria UART pega o valor recebido e envia para o terminal.

Para a conexão entre o transmissor e o receptor foi utilizado um único fio. Segue o código da primeira etapa do Lab 9:

Código C utilizado para o Lab 09

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

#define UART_NUM UART_NUM_1
```

```
#define TX_PIN (GPIO_NUM_17)
#define RX_PIN (GPIO_NUM_16)
#define BUF_SIZE (1024)

static const char *TAG = "UART_LOOP";

void uart_init() {
    const uart_config_t uart_config = {
        .baud_rate = 9600,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_APB,
    };

    uart_driver_install(UART_NUM, BUF_SIZE * 2, BUF_SIZE * 2, 0, NULL, 0);
    uart_param_config(UART_NUM, &uart_config);
    uart_set_pin(UART_NUM, TX_PIN, RX_PIN, UART_PIN_NO_CHANGE,
   UART_PIN_NO_CHANGE);
}

 char tx_data[] = {'A', 'D', 'G'};
    int tx_index = 0;
    char rx_data;
    uint8_t rx_buffer[BUF_SIZE];

void uart_loop_task(void *arg) {

    while (1) {

        // Envia o caractere via UART
        ESP_LOGI(TAG, "Enviando: %c", tx_data[tx_index]);
        uart_write_bytes(UART_NUM, &tx_data[tx_index], 1);

        // Recebe o caractere via UART
         int rx_bytes = uart_read_bytes(UART_NUM, rx_buffer, 1,
   pdMS_TO_TICKS(1000));
            //int rx_bytes = 0;
            //rx_buffer[0]=0;

            if (rx_bytes  > 0) {
            rx_data = rx_buffer[0];
            // Imprime o caractere recebido
            ESP_LOGE(TAG, "Recebido %c\n", rx_data);

        } else {
```

```
54              ESP_LOGE(TAG, "Nenhum dado recebido.");
55          }
56           tx_index++;
57          // Incrementa o caractere para o pr ximo no alfabeto
58
59          if (tx_index >= sizeof(tx_data)) {
60              tx_index = 0;  // Reinicia o  ndice  se atingir o final do
    vetor
61          }
62
63          vTaskDelay(pdMS_TO_TICKS(5000));
64      }
65  }
66
67  void app_main(void) {
68      uart_init();
69      xTaskCreate(uart_loop_task, "uart_loop_task", 2048, NULL, 10, NULL);
70  }
```

2.5.2.2 Parte 2 - Observação do sinal enviado usando Osciloscópio

Foi executado o sistema implementado na parte 1 e registrado na tela do osciloscópio para as letras A , D e G conforme as figuras abaixo. Para a confirmação, foi feito o cálculo dos bits do sinal . Para letra A , 01000001 , que em decimal significa 65, confirmando na tabela ASCII, Para letra D , 01000100 , que em decimal significa 68, confirmando na tabela ASCII e para letra G , 01000111, que em decimal significa 71, confirmando na tabela ASCII.

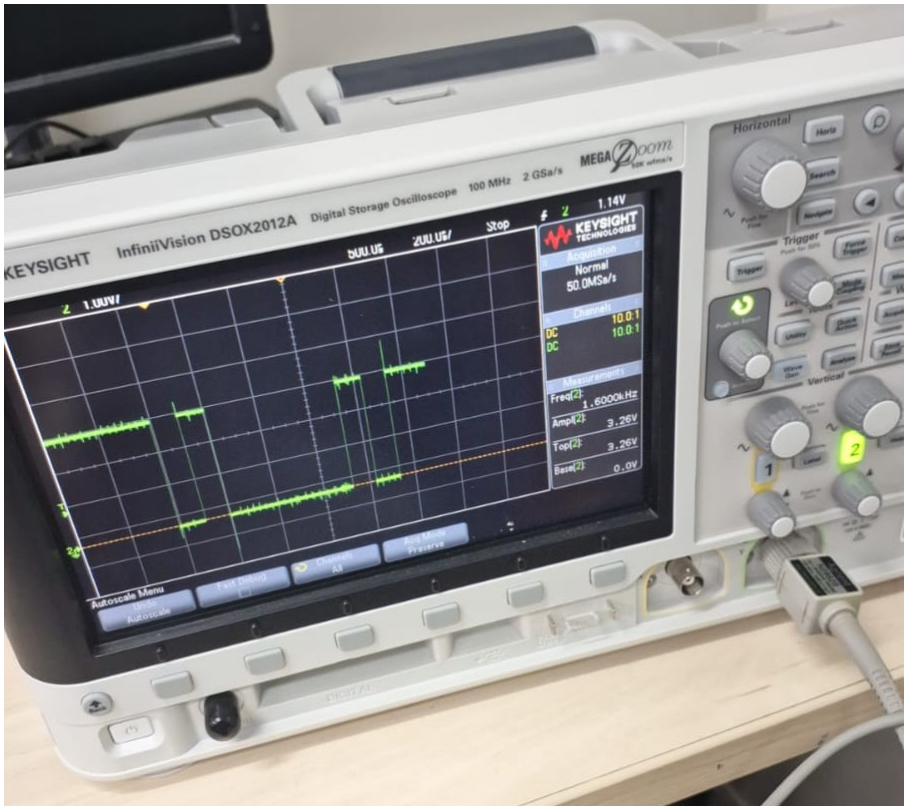

Figura 8 – Letra “A” comunicação via UART

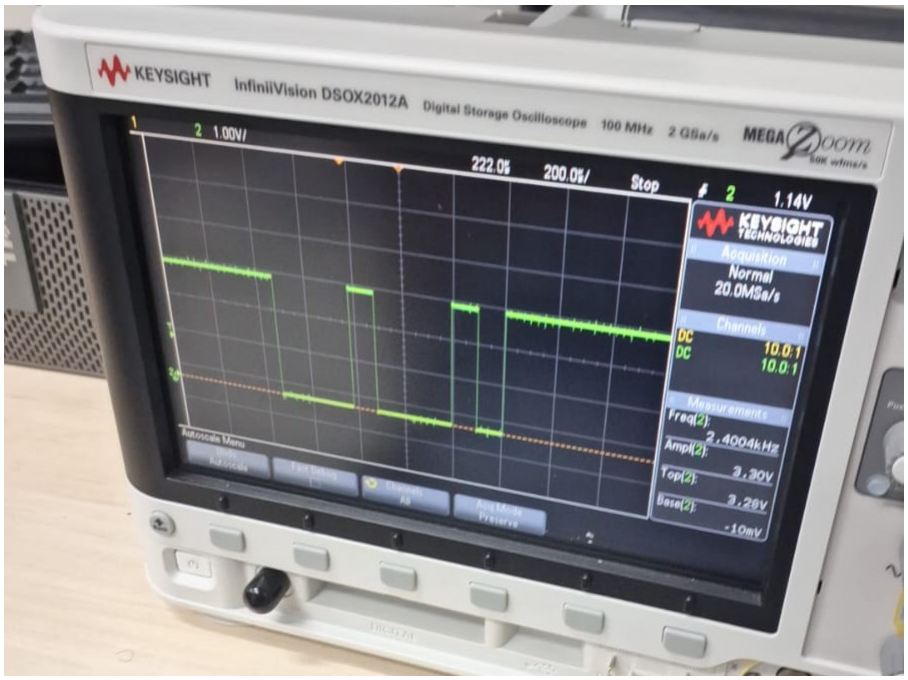

Figura 9 – Letra “D” comunicação via UART

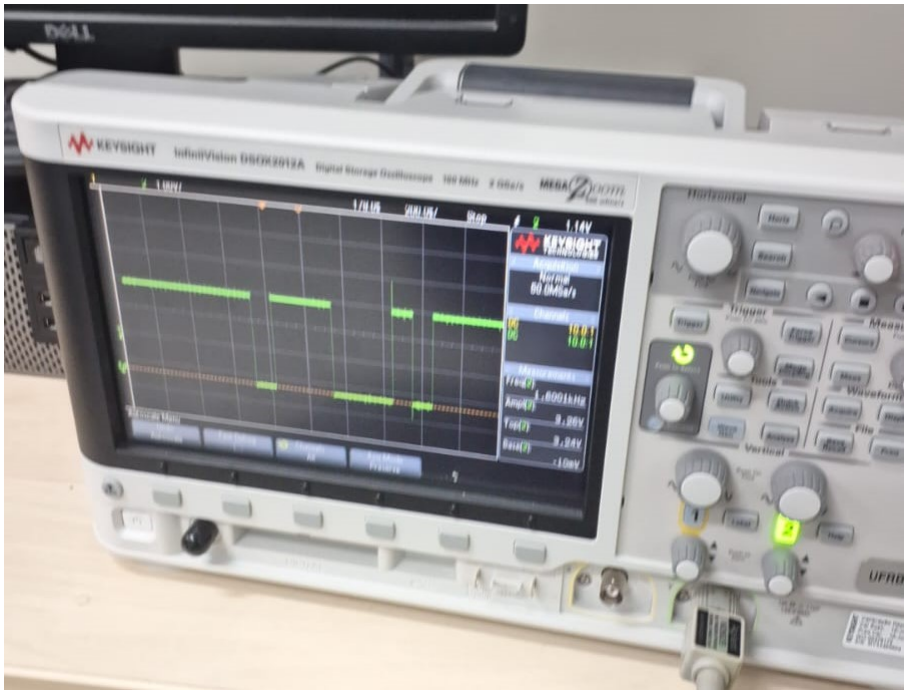

Figura 10 – Letra “G” comunicação via UART

Neste experimento, exploramos o funcionamento do protocolo UART, que é essencial para comunicação serial em diversos dispositivos. Na primeira parte, desenvolvemos um firmware simples que envia letras do alfabeto de forma cíclica quando um botão é pressionado, recebendo os dados através de uma conexão loopback. Este exercício demonstrou a eficiência da UART para transmitir e receber dados de forma confiável.

Na segunda parte, usamos um osciloscópio para observar os sinais enviados. Comparamos os sinais capturados com os valores esperados na tabela ASCII, confirmando que o sistema estava funcionando corretamente. Essa análise prática com o osciloscópio reforçou a importância da precisão na configuração dos parâmetros de comunicação serial, como taxa de bits e paridade, para garantir uma transmissão de dados eficaz e sem erros.

Referências

ESPRESSIF SYSTEMS. *ESP32 Series Datasheet*. [S.l.], 2024. Disponível em: <https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf>. Nenhuma citação no texto.

ESPRESSIF SYSTEMS. *ESP32 Technical Reference Manual Version 5.2*. [S.l.], 2024. Disponível em: <https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdff>. Nenhuma citação no texto.