This is the code for blinking away this code starts a task, start a loop and turns a pin onn, delays the sets inverts the state and the sets the pin to that state

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

# define DELAY (1000 / portTICK_PERIOD_MS)
# define PIN (33)
void TaskBlink (void *pvParameters)
{
    uint8_t state = 1;
    while (1) {
        gpio_set_level(PIN, state);
        vTaskDelay(DELAY);
        state = !state;
    }
}

void app_main(void)
{
    gpio_reset_pin(PIN);
    gpio_set_direction(PIN, GPIO_MODE_OUTPUT);
    xTaskCreate(TaskBlink, "Blink", 4096, NULL, 1, NULL);
}
```

This is the code for serial count this code uses UART (Universal Asynchronous Receiver-Transmitter) and sets this up, it takes an interger as inputs and passes it to the countdown function and starts counting down through its serial port, it uses UART 0 which is the uart port assoicated with the usb input and is used ofr firmware / debugging of the esp32

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>  // For atoi()
#include "driver/uart.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

#define BUF_SIZE 1024

void countDown(uint8_t start) {
    for (uint8_t i = start; i > 0; i--) {
        char buf[32];
        snprintf(buf, sizeof(buf), "%d\n", i);
        uart_write_bytes(UART_NUM_0, buf, strlen(buf));
        vTaskDelay(1000 / portTICK_PERIOD_MS); // Delay for 1 second
```

```
    }
    // Print zero separately to avoid potential underflow issues with
uint8_t
    uart_write_bytes(UART_NUM_0, "0\n", 2);
}

void app_main() {
    uint8_t start = 0;
    char input_buf[BUF_SIZE];

    // Initialize UART
    const uart_config_t uart_config = {
        .baud_rate = 115200,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
    };
    uart_driver_install(UART_NUM_0, BUF_SIZE * 2, 0, 0, NULL, 0);
    uart_param_config(UART_NUM_0, &uart_config);

    // No need to set pins for UART_NUM_0 connected to USB

    // Read input from UART
    while (1) {
        int len = uart_read_bytes(UART_NUM_0, (uint8_t *)input_buf,
BUF_SIZE - 1, 20 / portTICK_PERIOD_MS);
        if (len > 0) {
            input_buf[len] = '\0'; // Null-terminate the string
            start = atoi(input_buf); // Convert input to integer
            countDown(start);
        }
    }
}
```

This is the code for reporting change This code setsup the uart and setes a pin to input mode to read the state of the input pin, the pin only prints when changes happen meaning if the pin is high for 1 minute continously it will only print 1 once

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

#define PIN (18)
static const char *TAG_POLLING = "GPIO_POLLING"; // New TAG
static int last_pin_state = 0; // Keep track of the last state

void task_polling(){ // Renamed task for clarity
```

```
    // Initialize the pin state once
    last_pin_state = gpio_get_level(PIN);
    ESP_LOGI(TAG_POLLING, "Polling started. Initial state of PIN %d: %d",
PIN, last_pin_state);

    while (1){
        int current_state = gpio_get_level(PIN);

        // Only print when a change is detected
        if (current_state != last_pin_state){
            if (current_state == 1){
                printf("1\n"); // Print '1' only on a rising edge (0 -> 1)
                ESP_LOGI(TAG_POLLING, "Rising edge detected (0 -> 1) on PIN
%d", PIN);
            } else { // current_state must be 0
                printf("0\n"); // Print '0' only on a falling edge (1 -> 0)
                ESP_LOGI(TAG_POLLING, "Falling edge detected (1 -> 0) on
PIN %d", PIN);
            }
            last_pin_state = current_state; // Update the last state
        }
        vTaskDelay(pdMS_TO_TICKS(100)); // Use pdMS_TO_TICKS for
clarity/portability
    }
}

void app_main(void)
{
    gpio_reset_pin(PIN);
    gpio_set_direction(PIN,GPIO_MODE_INPUT);
    gpio_set_pull_mode(PIN, GPIO_PULLDOWN_ONLY); // Pin will be 0 by
default, 1 when connected to 3.3V

    xTaskCreate(task_polling, "gpio_polling_task", 4096, NULL, 1, NULL); //
Renamed task and added stack size
}
```

This is the command and concoure code this code reads an input, takes it appart counts out all commands, and seperates each word / entry out the code initlizes two structs and inputs the name and function of each function below, the frist word determines what functioned is called.

```
/*
 * SPDX-FileCopyrightText: 2010-2022 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: CC0-1.0
 */

#include "driver/uart.h"
#include "driver/gpio.h"
#include "esp_log.h"
#include <string.h>
```

```
#include <stdlib.h>
#include <stdio.h>

#define COMMAND_COUNT (4)
#define SIZE (255)

void writeText(char* text) {
    char newline[] = "\r";

    // Random log i had to include, because if i do not it will remove  the
first letter of the first string i write over UART cannot figure out why.
    ESP_LOGI("UART", "Sending");

    uart_write_bytes(UART_NUM_0, newline, strlen(newline));
    uart_write_bytes(UART_NUM_0, text, strlen(text));
    uart_write_bytes(UART_NUM_0, newline, strlen(newline));
}
struct dispatch_entry_t {
    const char* name; // Stores the command name (e.g., "pin-high")
    union {           // A union allows storing different function pointer
types
        void (*function)(int, char**); // For commands that take argc/argv
        // Other function pointer types could go here if needed
    } function;
};

struct dispatch_entry_t command_dispatch_table[COMMAND_COUNT];

void init(void) {
    uart_config_t uart_config = {
        .baud_rate = 115200,
        .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_DEFAULT,
    };

    ESP_ERROR_CHECK(uart_param_config(UART_NUM_0, &uart_config));
    ESP_ERROR_CHECK(uart_driver_install(UART_NUM_0, 1024 * 2, 0, 0, NULL,
0));
}

int dispatch(char* input){

    char* argv[SIZE];
    int argc = 0;

    int nextString[20];
    nextString[argc] = 0;

    char intermediateString[SIZE];
    int length = strlen(input);
    strcpy(intermediateString, input);
```

```
    for (int i = 0; i <= length; i++) {
        // When we find a delimiter or end of string
        if (input[i] == ' ' || input[i] == '\0') {
            intermediateString[i] = '\0';
            argc++;
            nextString[argc] = i+1;
        }
    }

    for (int i = 0; i <argc; i++){
        //printf("%s\n", &intermediateString[nextString[i]]);
        argv[i] = &intermediateString[nextString[i]];
        //printf("%s\n", argv[i]);
    }

    for (int i = 0; i<COMMAND_COUNT; i++){
        if (strcmp(command_dispatch_table[i].name, argv[0]) == 0){
            command_dispatch_table[i].function.function(argc,argv);
            //printf("%s\n", "The name matches");
            return 1;
        }
    }

    printf("\n Unknown command: %s\n", argv[0]);
    return 0;
}

void echo_command (int argc, char* argv[]) {
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
}

void pin_high(int argc, char* argv[]){
    for (int i = 1; i < argc; i++){
        int pin = atoi(argv[i]);
        gpio_reset_pin(pin);
        gpio_set_direction(pin,GPIO_MODE_OUTPUT);
        gpio_set_level(pin, 1);
    }
}

void pin_low(int argc, char* argv[]){
    for (int i = 1; i < argc; i++){
        int pin = atoi(argv[i]);
        gpio_reset_pin(pin);
        gpio_set_direction(pin,GPIO_MODE_OUTPUT);
        gpio_set_level(pin, 0);
    }
}
```

```
void pin_read(int argc, char* argv[]){
    for (int i = 1; i < argc; i++){
        int pin = atoi(argv[i]);
        gpio_reset_pin(pin);
        gpio_set_direction(pin,GPIO_MODE_INPUT);
        gpio_set_pull_mode(pin, GPIO_PULLDOWN_ONLY);
        int level = gpio_get_level(pin);
        char number[4];
        sprintf(number, "%d", level);
        writeText(number);
    }
}


void init_dispatch_table(void){
    command_dispatch_table[0].name = "echo";
    command_dispatch_table[0].function.function = echo_command;
    command_dispatch_table[1].name = "pin-high";
    command_dispatch_table[1].function.function = pin_high;
    command_dispatch_table[2].name = "pin-low";
    command_dispatch_table[2].function.function = pin_low;
    command_dispatch_table[3].name = "pin-read";
    command_dispatch_table[3].function.function = pin_read;
};

char* readData() {
    char* data = (char*)malloc(SIZE);

    while (1) {
        int length = 0;
        ESP_ERROR_CHECK(uart_get_buffered_data_len(UART_NUM_0, (size_t
*)&length));
        length = uart_read_bytes(UART_NUM_0, data, length, 20 /
portTICK_PERIOD_MS);

        if (length > 0) {
            printf("Received on port %d: ", UART_NUM_0);

            for (int i = 0; i < length; i++){
                printf("%c", data[i]);
            }
            printf("\n");
            data[length] = '\0';
            return data;
        }
        vTaskDelay(10 / portTICK_PERIOD_MS);
    }
}



void task(void *arg){
while (1){
```

```
        char* testString = readData();
        dispatch(testString);
        free(testString);
        vTaskDelay(10 / portTICK_PERIOD_MS);
    }
    }

    void app_main(void)
    {
    init();
    init_dispatch_table();

    xTaskCreate(task, "Command-And-Conqoure", 8192, NULL, 1, NULL);
    }
```