this is the code for the guessing game

```
/*
 * SPDX-FileCopyrightText: 2022-2023 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
 #include "freertos/FreeRTOS.h"
 #include "freertos/task.h"
 #include "soc/soc_caps.h"
 #include "esp_log.h"
 #include "esp_adc/adc_oneshot.h"
 #include "esp_adc/adc_cali.h"
 #include "esp_adc/adc_cali_scheme.h"
 #include "esp_system.h"

 const static char *TAG = "GUESSING_GAME";

 /*---------------------------------------------------------------
         ADC General Macros
 ---------------------------------------------------------------*/
 //ADC1 Channels
 #if CONFIG_IDF_TARGET_ESP32
 #define EXAMPLE_ADC1_CHAN0          ADC_CHANNEL_4
 #define EXAMPLE_ADC1_CHAN1          ADC_CHANNEL_5
 #else
 #define EXAMPLE_ADC1_CHAN0          ADC_CHANNEL_2
 #define EXAMPLE_ADC1_CHAN1          ADC_CHANNEL_3
 #endif

 #define EXAMPLE_ADC_ATTEN           ADC_ATTEN_DB_12

 // Game configuration
 #define MAX_TRIES                   8
 #define MAX_NUMBER                  17  // Maximum number in range (0 to
MAX_NUMBER inclusive)
 #define INPUT_BUFFER_SIZE           64

 static bool example_adc_calibration_init(adc_unit_t unit, adc_channel_t
channel, adc_atten_t atten, adc_cali_handle_t *out_handle);
 static void example_adc_calibration_deinit(adc_cali_handle_t handle);

 /**
  * Perform a single ADC sample
  *
  * @param adc_handle The ADC handle to use
  * @param channel The ADC channel to sample
  * @param result Pointer to store the raw result
  * @param cali_handle Calibration handle (NULL if no calibration)
```

```
 * @param voltage Pointer to store calibrated voltage (NULL if not needed)
 *
 * @return ESP_OK on success, error code otherwise
 */
static esp_err_t sample(adc_oneshot_unit_handle_t adc_handle,
    adc_channel_t channel,
    int *result,
    adc_cali_handle_t cali_handle,
    int *voltage)
{
    esp_err_t ret;

    // Read the raw value
    ret = adc_oneshot_read(adc_handle, channel, result);
    if (ret != ESP_OK) {
        return ret;
    }

    // Convert to voltage if calibration handle is provided and voltage
pointer is not NULL
    if (cali_handle != NULL && voltage != NULL) {
        ret = adc_cali_raw_to_voltage(cali_handle, *result, voltage);
    }

    return ret;
}

/**
 * Read a line from stdin and parse it to an integer
 *
 * @param result Pointer to store the parsed integer
 * @return true if parsing was successful, false otherwise
 */
static bool read_integer_from_stdin(int *result) {
    char input_buffer[INPUT_BUFFER_SIZE];
    char *endptr;

    // Read line from stdin
    if (fgets(input_buffer, sizeof(input_buffer), stdin) == NULL) {
        return false;
    }

    // Remove newline character if present
    size_t len = strlen(input_buffer);
    if (len > 0 && input_buffer[len - 1] == '\n') {
        input_buffer[len - 1] = '\0';
    }

    // Parse to integer
    *result = strtol(input_buffer, &endptr, 10);

    // Check if parsing was successful (entire string was consumed)
    return (*endptr == '\0' && endptr != input_buffer);
}
```

```
/**
 * Read a single character response (y/n)
 *
 * @return 'y', 'n', or '\0' for invalid input
 */
static char read_yn_response(void) {
    char input_buffer[INPUT_BUFFER_SIZE];

    if (fgets(input_buffer, sizeof(input_buffer), stdin) == NULL) {
        return '\0';
    }

    // Check first character and make it lowercase
    char response = input_buffer[0];
    if (response >= 'A' && response <= 'Z') {
        response += 32; // Convert to lowercase
    }

    if (response == 'y' || response == 'n') {
        return response;
    }

    return '\0';
}

void app_main(void)
{
    //-------------ADC1 Init---------------//
    adc_oneshot_unit_handle_t adc1_handle;
    adc_oneshot_unit_init_cfg_t init_config1 = {
        .unit_id = ADC_UNIT_1,
    };
    ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_config1, &adc1_handle));

    //-------------ADC1 Config---------------//
    adc_oneshot_chan_cfg_t config = {
        .atten = EXAMPLE_ADC_ATTEN,
        .bitwidth = ADC_BITWIDTH_DEFAULT,
    };
    ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle,
EXAMPLE_ADC1_CHAN0, &config));

    //-------------ADC1 Calibration Init---------------//
    adc_cali_handle_t adc1_cali_chan0_handle = NULL;
    bool do_calibration1_chan0 = example_adc_calibration_init(ADC_UNIT_1,
EXAMPLE_ADC1_CHAN0, EXAMPLE_ADC_ATTEN, &adc1_cali_chan0_handle);

    // Main game loop
    while (1) {
        // Step 3: Greeting
        printf("Shall we play a game? (y/n): ");
        fflush(stdout);
```

```
            char response = read_yn_response();

            if (response == 'n') {
                printf("Goodbye!\n");
                esp_restart();
            } else if (response == 'y') {
                // Step 4: Produce a Number
                int adc_raw = 0;
                int voltage = 0;
                ESP_ERROR_CHECK(sample(adc1_handle, EXAMPLE_ADC1_CHAN0,
&adc_raw,
                                       do_calibration1_chan0 ?
adc1_cali_chan0_handle : NULL,
                                       &voltage));

                // Convert ADC reading to game number (0 to MAX_NUMBER)
                int secret_number = adc_raw % (MAX_NUMBER + 1);

                // Step 5: Welcome Message
                printf("I am thinking about a number between 0 and %d (both
included). "
                       "I bet you can't guess it in %d tries!\n", MAX_NUMBER,
MAX_TRIES);

                // Step 6: Game Loop setup
                bool done = false;
                uint8_t count = 0;

                while (!done) {
                    count++;

                    // Step 7: Query User
                    printf("You have guessed %d %s. Please try again: ",
                           count - 1,
                           (count - 1) == 1 ? "time" : "times");
                    fflush(stdout);

                    // Step 8: Fetch Answer
                    int guess;
                    if (!read_integer_from_stdin(&guess)) {
                        printf("Error: Please enter a valid integer.\n");
                        count--; // Don't count invalid inputs
                        continue;
                    }

                    // Step 9 & 10: Evaluate Response and Produce Feedback
                    if (guess == secret_number) {
                        if (count <= MAX_TRIES) {
                            printf("You got it! Congratulations, you managed
it within %d tries!\n", MAX_TRIES);
                        } else {
                            printf("You got it! But it took you more than %d
tries.\n", MAX_TRIES);
                        }
```

```
                    done = true;
                } else {
                    if (guess < secret_number) {
                        printf("Too low!\n");
                    } else {
                        printf("Too high!\n");
                    }

                    // Check if max tries reached
                    if (count >= MAX_TRIES) {
                        // Step 12: Bonus cheeky remark
                        printf("The only winning move is not to play. The
number was %d!\n", secret_number);
                        done = true;
                    }
                }
            }

            printf("\n"); // Add some spacing between games
        } else {
            printf("Please answer with 'y' or 'n'.\n");
        }

        vTaskDelay(pdMS_TO_TICKS(100)); // Small delay to prevent flooding
    }

    // Cleanup (this code will never be reached due to infinite loop and
esp_restart)
    ESP_ERROR_CHECK(adc_oneshot_del_unit(adc1_handle));
    if (do_calibration1_chan0) {
        example_adc_calibration_deinit(adc1_cali_chan0_handle);
    }
}

/*---------------------------------------------------------------
        ADC Calibration
---------------------------------------------------------------*/
static bool example_adc_calibration_init(adc_unit_t unit, adc_channel_t
channel, adc_atten_t atten, adc_cali_handle_t *out_handle)
{
    adc_cali_handle_t handle = NULL;
    esp_err_t ret = ESP_FAIL;
    bool calibrated = false;

#if ADC_CALI_SCHEME_CURVE_FITTING_SUPPORTED
    if (!calibrated) {
        ESP_LOGI(TAG, "calibration scheme version is %s", "Curve
Fitting");
        adc_cali_curve_fitting_config_t cali_config = {
            .unit_id = unit,
            .chan = channel,
            .atten = atten,
            .bitwidth = ADC_BITWIDTH_DEFAULT,
        };
```

```
            ret = adc_cali_create_scheme_curve_fitting(&cali_config, &handle);
            if (ret == ESP_OK) {
                calibrated = true;
            }
        }
    #endif

    #if ADC_CALI_SCHEME_LINE_FITTING_SUPPORTED
        if (!calibrated) {
            ESP_LOGI(TAG, "calibration scheme version is %s", "Line Fitting");
            adc_cali_line_fitting_config_t cali_config = {
                .unit_id = unit,
                .atten = atten,
                .bitwidth = ADC_BITWIDTH_DEFAULT,
            };
            ret = adc_cali_create_scheme_line_fitting(&cali_config, &handle);
            if (ret == ESP_OK) {
                calibrated = true;
            }
        }
    #endif

        *out_handle = handle;
        if (ret == ESP_OK) {
            ESP_LOGI(TAG, "Calibration Success");
        } else if (ret == ESP_ERR_NOT_SUPPORTED || !calibrated) {
            ESP_LOGW(TAG, "eFuse not burnt, skip software calibration");
        } else {
            ESP_LOGE(TAG, "Invalid arg or no memory");
        }

        return calibrated;
    }

    static void example_adc_calibration_deinit(adc_cali_handle_t handle)
    {
    #if ADC_CALI_SCHEME_CURVE_FITTING_SUPPORTED
        ESP_LOGI(TAG, "deregister %s calibration scheme", "Curve Fitting");
        ESP_ERROR_CHECK(adc_cali_delete_scheme_curve_fitting(handle));

    #elif ADC_CALI_SCHEME_LINE_FITTING_SUPPORTED
        ESP_LOGI(TAG, "deregister %s calibration scheme", "Line Fitting");
        ESP_ERROR_CHECK(adc_cali_delete_scheme_line_fitting(handle));
    #endif
    }
```

This C code, built for **ESP32 microcontrollers** using the **ESP-IDF framework**, creates an interactive **number guessing game** played over the serial (UART) port. What makes it unique is how it generates the "secret number": it uses an **Analog-to-Digital Converter (ADC) reading** from a connected sensor or variable resistor to add a random element to the game.

Let's break down how this guessing game works.

Game Concept

The program simulates a classic "guess the number" game:

1. **Start a Game:** It first asks the player if they want to play.
2. **Generate Secret Number:** If the player agrees, the ESP32 takes a reading from an analog input pin (like from a potentiometer or a fluctuating sensor). It then uses this reading to determine the secret number.
3. **Player Guesses:** The player inputs their guesses via the serial terminal.
4. **Feedback:** The game tells the player if their guess is too high, too low, or correct.
5. **Try Limit:** The player has a limited number of tries to guess the number.
6. **Game End:** The game ends when the player guesses correctly or runs out of tries. After a game, it prompts the player to play again.

Code Breakdown

1. Includes and Configuration Macros

This section brings in all the necessary libraries:

- Standard C libraries (`stdio.h`, `stdlib.h`, `string.h`) for input/output, string conversion, and memory.
- FreeRTOS headers (`freertos/FreeRTOS.h`, `freertos/task.h`) for managing tasks and delays.
- ESP-IDF specific headers for logging (`esp_log.h`), system functions (`esp_system.h` for `esp_restart`), and crucial ADC functionality (`esp_adc/adc_oneshot.h`, `esp_adc/adc_cali.h`, `esp_adc/adc_cali_scheme.h`).
- **ADC Channel Definitions:** Macros like `EXAMPLE_ADC1_CHAN0` specify which physical GPIO pin (analog input) is used for the ADC reading. The specific pin numbers might change based on the ESP32 board you're using.
- **ADC Attenuation (`EXAMPLE_ADC_ATTEN`):** This sets the input voltage range for the ADC, allowing it to measure up to around 3.9V in this case.
- **Game Configuration Macros:**
  - `MAX_TRIES`: The maximum number of guesses the player gets (set to 8).
  - `MAX_NUMBER`: The upper limit for the secret number (the number will be between 0 and 17, inclusive).
  - `INPUT_BUFFER_SIZE`: A buffer size for reading player input from the serial terminal.

2. `sample` Function (ADC Reading Helper)

This helper function takes care of reading the analog input from a specified ADC channel.

- It performs a **raw ADC conversion**, which gives you an integer value representing the voltage.
- Crucially, if you have set up **ADC calibration** (which this code does), it will then use the calibration data to convert that raw reading into a more accurate **voltage value in millivolts (mV)**. This helps ensure consistent readings across different ESP32 chips.

3. Input Reading Helper Functions

This code includes two specific helper functions to make reading user input robust:

- `read_integer_from_stdin(int *result)`:
  - This function reads a line of text typed by the user into the serial monitor.
  - It then attempts to convert that text into an **integer** using `strtol()`.
  - It includes checks to ensure that the entire input line was a valid number, returning `true` on success or `false` if the input wasn't a clean integer. This helps prevent crashes from bad user input.
- `read_yn_response(void)`:
  - This function reads a single character response from the user (e.g., 'y' or 'n').
  - It converts the input to lowercase to handle both 'Y' and 'y' (or 'N' and 'n').
  - It returns 'y', 'n', or a null character if the input wasn't valid.

4. `app_main` Function (The Game Logic)

This is the main function where the ESP32 starts executing the game.

- **ADC Initialization:**
  - It first initializes **ADC Unit 1** on the ESP32.
  - It configures the specific ADC channel (`EXAMPLE_ADC1_CHAN0`) with the chosen attenuation and bit width.
  - It then attempts to set up **ADC calibration** for this channel. If calibration is successful (meaning the ESP32 has calibration data in its eFuses), the readings will be more accurate. If not, it will still proceed, but voltage conversion won't be as precise.
- **Main Game Loop (`while (1)`)**: The game runs continuously in an infinite loop.
  - **Step 3: Greeting:** It prompts the user: "Shall we play a game? (y/n): ".
  - **User Response:** It waits for the user to input 'y' or 'n' using `read_yn_response()`.
    - If 'n' is entered, it prints "Goodbye!" and **restarts the ESP32** (`esp_restart()`).
    - If 'y' is entered, the game begins.
    - For any other input, it asks the user to re-enter 'y' or 'n'.
  - **Step 4: Produce a Number (The ADC Trick!)**
    - If the user agrees to play, the ESP32 takes a single **ADC sample** from `EXAMPLE_ADC1_CHAN0`. This reading will be influenced by whatever is connected to that analog input pin (e.g., a potentiometer's position, a sensor's output).
    - The `secret_number` for the game is then generated by taking the **raw ADC reading modulo (`%`)** `(MAX_NUMBER + 1)`. This effectively maps the potentially large raw ADC value into a number between 0 and `MAX_NUMBER` (e.g., 0 to 17). This is where the "randomness" (or input-driven variation) comes from.
  - **Step 5: Welcome Message:** It prints a welcome message, telling the player the range of numbers and how many tries they have.

- **Step 6: Inner Game Loop (`while (!done)`)**: This loop handles the actual guessing process for a single game.
    - **Step 7: Query User:** It prompts the user for their guess, indicating how many guesses they've already made.
    - **Step 8: Fetch Answer:** It uses `read_integer_from_stdin()` to get the player's numerical guess. If the input is invalid, it prints an error and doesn't count it as a guess.
    - **Step 9 & 10: Evaluate Guess & Feedback:**
        - If the `guess` matches the `secret_number`:
            - It prints a congratulatory message, noting if they stayed within the `MAX_TRIES` limit.
            - The `done` flag is set to `true` to end the inner game loop.
        - If the guess is incorrect:
            - It tells the player if their guess was "Too low!" or "Too high!".
            - It then checks if the player has reached `MAX_TRIES`. If so, it reveals the secret number with a "cheeky remark" (from the movie WarGames) and sets `done` to `true` to end the game.
- **Game End:** Once the inner loop ends, it adds some spacing and then the main `while(1)` loop starts again, prompting the user to play another game.
- A small `vTaskDelay` is added to prevent the ESP32 from flooding the serial port and to allow other FreeRTOS tasks to run.

- **Cleanup (Unreachable Code):** Like the previous ADC example, there's code after the main infinite loop to de-initialize ADC resources. In practice, this code is never reached because the `while(1)` loop runs forever, or the system restarts.

5. ADC Calibration Functions (`example_adc_calibration_init` and `example_adc_calibration_deinit`)

These are helper functions, largely identical to the previous ADC example:

- **`example_adc_calibration_init`:** Attempts to create an ADC calibration handle. It tries to use the more accurate **Curve Fitting** scheme first. If that's not supported, it falls back to **Line Fitting**. This is crucial for getting accurate voltage readings if you were to convert the ADC value to a voltage. It also logs if calibration succeeds, fails, or is skipped because the calibration data isn't "burnt" into the ESP32's eFuses.
- **`example_adc_calibration_deinit`:** Releases the resources associated with the calibration handle.

How to Play

1. **Hardware:** Connect a simple potentiometer (variable resistor) to the analog input pin defined by `EXAMPLE_ADC1_CHAN0` (e.g., GPIO4). Connect one end of the pot to 3.3V, the other to GND, and the middle wiper to the ADC pin.
2. **Flash:** Compile and flash this code onto your ESP32 board using the ESP-IDF tools.
3. **Serial Monitor:** Open a serial monitor connected to your ESP32 (usually UART0) at 115200 baud.

4. **Interact:**
    - The game will prompt you to play. Type `y` and press Enter.
    - Before you start guessing, **rotate the potentiometer to a random position**. This will influence the secret number.
    - The game will tell you the range and number of tries.
    - Type your guess (an integer) and press Enter.
    - The game will give you feedback ("Too high!", "Too low!", or "You got it!").
    - Keep guessing until you win or run out of tries!

This example beautifully combines serial communication, basic input parsing, game logic, and an interesting use of the ESP32's ADC capabilities to generate a dynamic secret number.