TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

ESP32-C6 SENSOR TESTBENCH

Comparing sensor values with datasheet

Gomez Cuesta, Alvar

December 12, 2024

Contents

1 INTRODUCTION **3**
1.1 What can you do with this program? . . . 3
1.2 Concepts . . . 3

2 SENSORS **5**
2.1 HC-SR04/HC-SR05 . . . 5
2.1.1 Measurement procedure . . . 5
2.1.2 Datasheet . . . 6
2.2 DHT-11 . . . 6
2.2.1 Measurement procedure . . . 6
2.2.2 DATASHEET . . . 7

3 ESP-32 C6 **8**

4 CODE **9**
4.1 constants.c . . . 9
4.2 hc_sr05.c . . . 9
4.3 dht11_w.c . . . 16
4.4 main.c . . . 19

5 ANALYSIS **21**

6 LEARNING ACHIEVEMENTS **22**

1 INTRODUCTION

This project forms part of the Signals and Sensor Conditioning course, therefore some information provided here might be redundant if you already did the course, but I think it's important to include it here for external readers.

In this document I am going to explain the process I have been through while learning how to use the ESP-32 micro-controller and the HC-SR05 and DHT-11 sensors. Also, I will write down all my notes regarding the C language and what I learn during the realization of this project, as well as things that I think that are important to explain in order to understand how the code works.

1.1 What can you do with this program?

My initial idea, and it remains the same, was to create an independent test bench to which you could connect through the UART protocol, meaning that you wouldn't need to install any extra software in your computer in order to be able to perform tests and check the results. A kind of a "plug and play" device.

With this idea in mind, I ended up doing a console based GUI that can be controlled by sending commands through the UART protocol. Everything is executed in the ESP32 micro-controller. Later in this document the operation of this GUI will be explained.

Things you can do:

- Choose between multiple sensors.
- Choose the length and frequency of the test you want to run.
- Print the last test results in “serial" mode so you can later put it into a Matlab, Python or other language scripts.

Things that are not implemented yet, but that are planned to be implemented in the future:

- Run multiple test with different sensors at the same time. This way I will learn how to use tasks in FreeRTOs and how to manage them.
- Connect to the device through the WiFi chip, for example with a mobile phone, and be able to control it from there.

1.2 Concepts

In this section, a series of relevant concepts for this report are going to be explained. Even though some of them are implemented, the main target of this project is to allow the user to test whatever sensor that uses a “common" interface to communicate with the board.

- **Precision:** refers to the ability of a sensor/device to measure the same quantity multiple times with a low spread in the output value.
  Precision can be expressed as:
  - **Repeatability:** closeness of the output readings when the same input is applied under the same conditions in a short period of time. We are going to be focusing on this.
  - **Reproducibility:** closeness of the output readings when the same input is applied under different conditions, like location or time of measurement for example.

- **Resolution:** smallest change that can be measured by a sensor. This typically depends on the bit depth of the ADC or the controller capabilities. It is usually expressed in bits, with the least significant bit being the equivalent to the smallest change measurable.

- **Uncertainty:** error or doubt in the measurement due to multiple factors. There are 2: type A, which is calculated based on statistical methods, and type B, which is evaluated by other means, like calibration data, someone opinion, manufacturer specifications, or other possible factors that we will see later for each sensor.

2 SENSORS

In this section all the sensors available to be used in this project are going to be explained. Even though this project aims to let the user use whatever sensor he wants, as long as it has a good protocol to communicate with the board

2.1 HC-SR04/HC-SR05

This sensor, also known as an **Ultrasonic Ranging Module**, allows the user to measure distances in a close range by using ultrasonic waves. The way this sensor works is as follows:

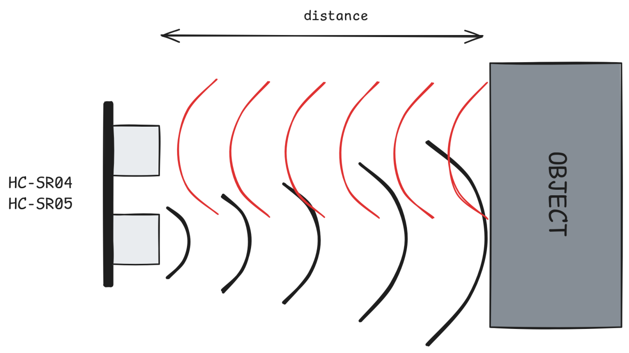


Figure 1: Ultrasonic sensor operation

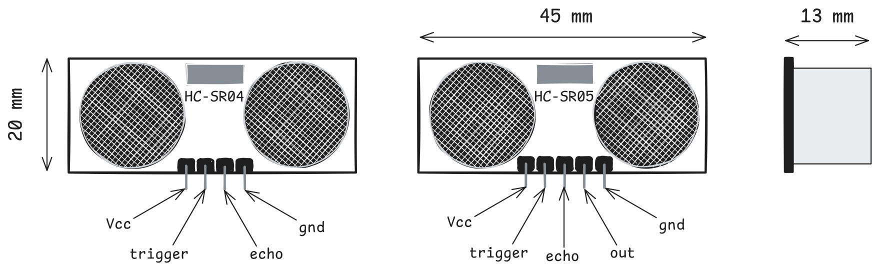


Figure 2: HC-SR04/HC-SR05 pin out and dimensions

2.1.1 Measurement procedure

As seen in the image above, in order to generate the data, the sensor sends some ultrasonic waves and wait for them to bounce off the nearest object. The time elapsed between the moment the waves are sent and the moment they are received gives as a result the distance to the object.

The timing diagram of the sensor is as follows:

1. Send a short **HIGH** value to the **trigger** pin of the sensor with a width of $10\mu S$.
2. The sensor will send 8 40khz waves that will bounce in the nearest object in front of the sensor.
3. A **HIGH** signal will be sent through the echo pin to the board. The width of this signal is equivalent to the time taking for the waves to bounce in the object and return back to the sensor.

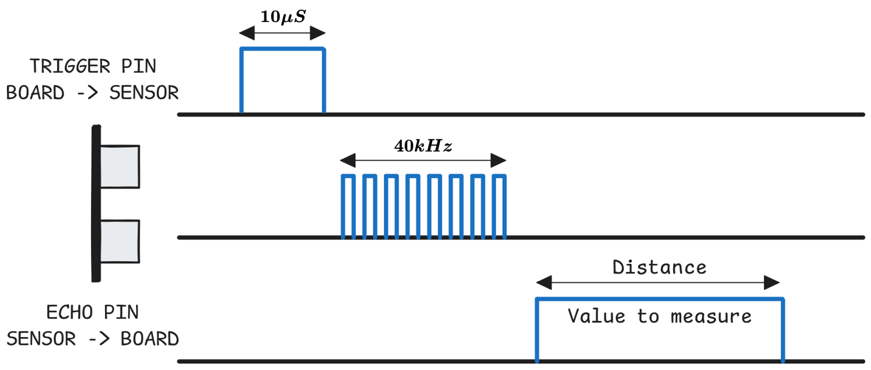


Figure 3: Timing diagram

2.1.2 Datasheet

The sensor datasheet gives us the following specifications:
We will try to prove how accurate this specifications are.

Resolution	0.3cm
Angle	15º
Range	2-450cm
Supply current	10-40 mA

2.2 DHT-11

DHT stands for “**Digital Humidity Temperature**".

The DHT-11 measures **Temperature** and **Humidity** sensor. It can measure both values at the same time. It’s not one of the best in the market, but is a cheap sensor and is included in almost every beginner kit. In order to measure both values, it uses a capacitive humidity sensor , and a thermistor.

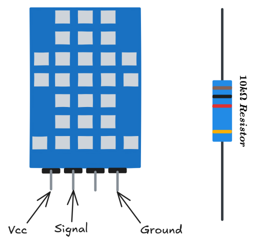


Figure 4: DHT-11 sensor pinout

In order to make the sensor work properly, we need to put a 10kΩ resistor as a pullup resistor that will go from the data pin to the Vcc.

2.2.1 Measurement procedure

DHT is a slow sensor, so we have to be careful with the timings, otherwise we will get errors in the measurement. We can divide the measurement procedure in 3 steps:

1. **Request to DHT**: the micro-controller sends a signal to the DHT as a request.
2. **Respond from DHT**: the DHT sensor responds with another signal that is processed by the micro-controller.
3. **Start of data transmission**: DHT starts sending the data.

In the next picture we can see the needed timings:

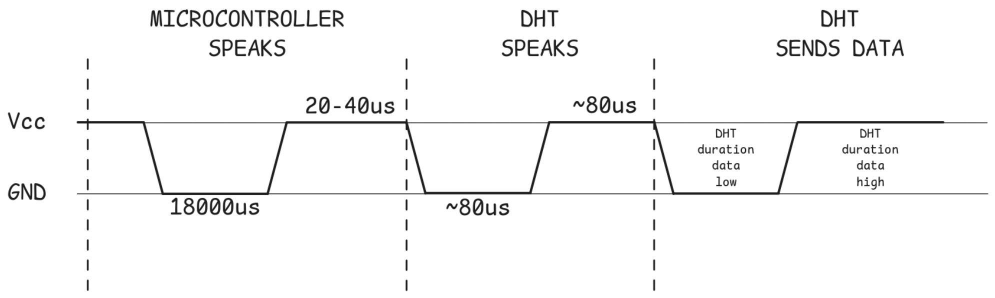


Figure 5: DHT-11 timing diagram

Now, the data structure used by the DHT to transmit the values is: 40 bits transmitted sequentially, with one high and one low signal to calculate the value of each bit. The 40 bits are grouped in 5 bytes, the first and third one store the Humidity and Temperature value respectively. Bytes 2 and 4 are filled with zeroes, and byte 5 is the checksum. The checksum is calculated as follows:

byte_5 == (byte_1 + byte_2 + byte_3 + byte_4) & 0xFF

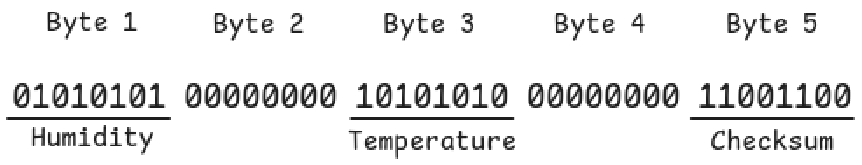


Figure 6: DHT-11 data structure for read values

More information can be found here.

Now, how does the sensor send the bits values to the micro-controller? As mentioned before, for each bit, we have a low duration value, and a high duration value. First, the DHT sets the signal to low for approximately 50us, and then let it float back to HIGH. If the duration of the HIGH signal is longer than the duration of LOW signal, then it means the sensor is transmitting a logic '1' value. '0' if it's the opposite case.

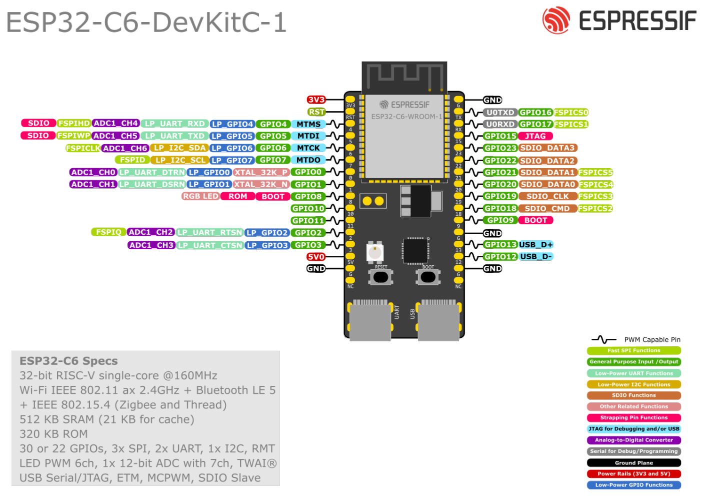
2.2.2 DATASHEET

The sensor datasheet gives us the following specifications:
We will try to prove how accurate this specifications are.

Resolution	1ºC/1%
Temperature Range	0-50ºC
Temperature Accuracy	±2ºC
Humidity Range	20-90%
Humidity Accuracy	±5%
Supply current	10-40 mA

3 ESP-32 C6

The board we are going to be using is the **ESP-32 C6 DEVKIT**. The schematics of the board are:


ESP32-C6-DevKitC-1
ESPRESSIF
3V3
RST
SDIO FSPIHD ADC1_CH4 LP_UART_RXD LP_GPIO4 GPIO4 MTMS
SDIO FSPIWP ADC1_CH5 LP_UART_TXD LP_GPIO5 GPIO5 MTDI
FSPICLK ADC1_CH6 LP_I2C_SDA LP_GPIO6 GPIO6 MTCK
FSPID LP_I2C_SCL LP_GPIO7 GPIO7 MTDO
ADC1_CH0 LP_UART_DTRN LP_GPIO0 XTAL_32K_P GPIO0
ADC1_CH1 LP_UART_DSRN LP_GPIO1 XTAL_32K_N GPIO1
RGB LED ROM BOOT GPIO8
GPIO10
GPIO11
FSPIQ ADC1_CH2 LP_UART_RTSN LP_GPIO2 GPIO2
ADC1_CH3 LP_UART_CTSN LP_GPIO3 GPIO3
5V0
GND
GND
U0TXD GPIO16 FSPICS0
U0RXD GPIO17 FSPICS1
GPIO15 JTAG
GPIO23 SDIO_DATA3
GPIO22 SDIO_DATA2
GPIO21 SDIO_DATA1 FSPICS5
GPIO20 SDIO_DATA0 FSPICS4
GPIO19 SDIO_CLK FSPICS3
GPIO18 SDIO_CMD FSPICS2
GPIO9 BOOT
GND
GPIO13 USB_D+
GPIO12 USB_D-
GND
PWM Capable Pin
Fast SPI Functions
General Purpose Input /Output
Low-Power UART Functions
Low-Power I2C Functions
SDIO Functions
Other Related Functions
Strapping Pin Functions
JTAG for Debugging and/or USB
Analog-to-Digital Converter
Serial for Debug/Programming
Ground Plane
Power Rails (3V3 and 5V)
Low-Power GPIO Functions

ESP32-C6 Specs
32-bit RISC-V single-core @160MHz
Wi-Fi IEEE 802.11 ax 2.4GHz + Bluetooth LE 5 + IEEE 802.15.4 (Zigbee and Thread)
512 KB SRAM (21 KB for cache)
320 KB ROM
30 or 22 GPIOs, 3x SPI, 2x UART, 1x I2C, RMT
LED PWM 6ch, 1x 12-bit ADC with 7ch, TWAI®
USB Serial/JTAG, ETM, MCPWM, SDIO Slave

A picture of the board with the sensors mounted.

4 CODE

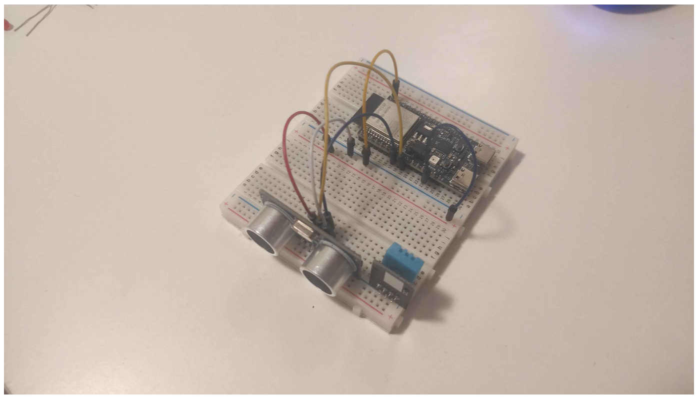
In this section I will be explaining the different parts of the code, as well as how the interface works. Multiple flowcharts will be include, as well as references to some parts of the code.

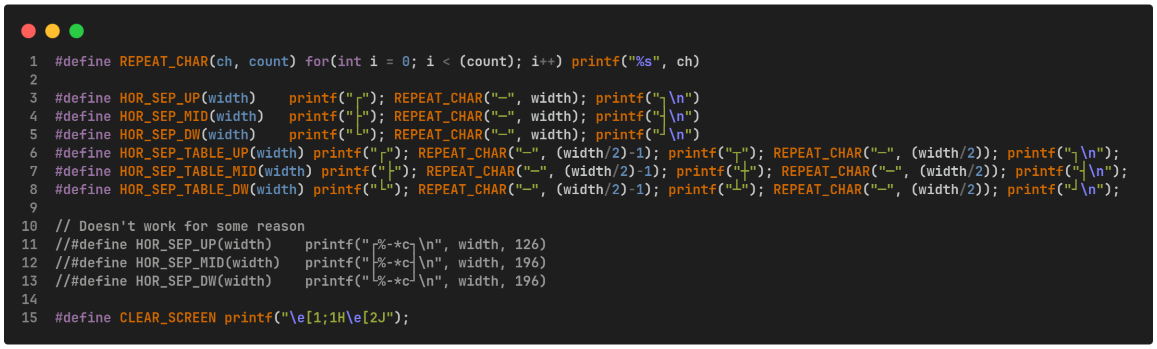
In order to keep the code organised and avoid long and confusing files, I have divided the code in multiple files, each of them with it purpose.

4.1 constants.c

This file contains all the necessary strings for the menus to work.
The *#define* statements are later used to print the borders of the GUI.

WAIT ENTER
This function waits for the user to press a key. Checks every 100ms if there's any new character in the **stdin**




```c
#define REPEAT_CHAR(ch, count) for(int i = 0; i < (count); i++) printf("%s", ch)

#define HOR_SEP_UP(width)    printf("┌"); REPEAT_CHAR("─", width); printf("┐\n")
#define HOR_SEP_MID(width)   printf("├"); REPEAT_CHAR("─", width); printf("┤\n")
#define HOR_SEP_DW(width)    printf("└"); REPEAT_CHAR("─", width); printf("┘\n")
#define HOR_SEP_TABLE_UP(width) printf("┌"); REPEAT_CHAR("─", (width/2)-1); printf("┬"); REPEAT_CHAR("─", (width/2)); printf("┐\n");
#define HOR_SEP_TABLE_MID(width) printf("├"); REPEAT_CHAR("─", (width/2)-1); printf("┼"); REPEAT_CHAR("─", (width/2)); printf("┤\n");
#define HOR_SEP_TABLE_DW(width) printf("└"); REPEAT_CHAR("─", (width/2)-1); printf("┴"); REPEAT_CHAR("─", (width/2)); printf("┘\n");

// Doesn't work for some reason
//#define HOR_SEP_UP(width)    printf("┌%-*c┐\n", width, 126)
//#define HOR_SEP_MID(width)   printf("├%-*c┤\n", width, 196)
//#define HOR_SEP_DW(width)    printf("└%-*c┘\n", width, 196)

#define CLEAR_SCREEN printf("\e[1;1H\e[2J");
```

buffer.

When a new character is detected, it checks if its either an enter key, or another type of case. In this case only the arrows are mapped and they always start with the code. **\033**. The ANSI scape codes can be found here. The code for each arrow is:

- **Arrow up**: "\033[A"
- **Arrow down**: "\033[B"
- **Arrow right**: "\033[C"
- **Arrow left**: "\033[D"

All the other arrays of strings in the file are already documented.

4.2 hc_sr05.c

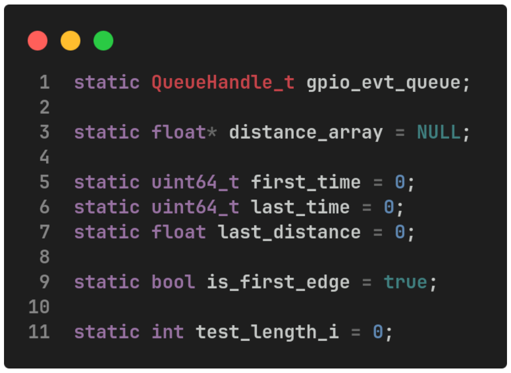
For the ultrasonic ranging module there was no driver available on the internet, so I decided to do it by myself. I already explained how to use the HC_SR05 previously in the 2.1.1 section.

I'm going to try to explain the code the best way possible. However, I highly recommend opening the file alongside this PDF in order to fully understand the dataflow and maybe read some annotations in the code.

On one side, we need to somehow detect when a edge change occurs in the GPIO pin selected. For this I decided to use interruptions.

Interruptions are a mechanism implemented in almost every micro-controller/CPU. This allows to stop whatever process the CPU is running and do another short and more important task. There are **hardware**

and **software** interruptions. In our case, GPIO interruptions are hardware type.



```c
static QueueHandle_t gpio_evt_queue;

static float* distance_array = NULL;

static uint64_t first_time = 0;
static uint64_t last_time = 0;
static float last_distance = 0;

static bool is_first_edge = true;

static int test_length_i = 0;
```

As global variables for the code of this file, I declared a *Queue Handler* which will be explained later; a float pointer that will later store the address to the allocated space in the heap for the test measurements; *first_time* and *last_time*, two variables for the timer values captured in order to calculate the distance; *last_distance*, that will store the last 16 values average; *is_first_edge*, the boolean variable for the interruption generated by the GPIO pin; *test_lenght_i*, an integer variable that will store the lenght of the test.

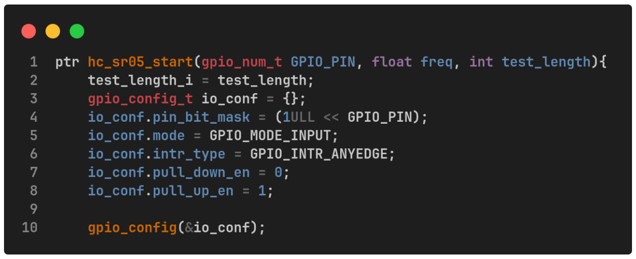
Below you will see the applied use of each of these variables.

HC-SR05 START

Parameters:

- **GPIO_PIN**: pin used for data connection with the sensor
- **freq**: frequency of measurement
- **test_lenght**: total number of samples of the test

So, first step to enable GPIO interruptions is going to be configuring the GPIO selected pin.

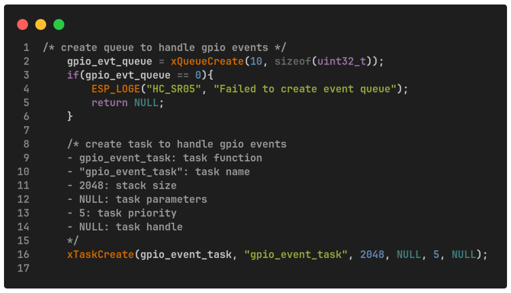


```c
ptr hc_sr05_start(gpio_num_t GPIO_PIN, float freq, int test_length){
    test_length_i = test_length;
    gpio_config_t io_conf = {};
    io_conf.pin_bit_mask = (1ULL << GPIO_PIN);
    io_conf.mode = GPIO_MODE_INPUT;
    io_conf.intr_type = GPIO_INTR_ANYEDGE;
    io_conf.pull_down_en = 0;
    io_conf.pull_up_en = 1;

    gpio_config(&io_conf);
```

At the beginning of the ***hc_sr05_start*** function, we declare and initialize the *gpio_config_t* struct. Set the mode to INPUT and the interruption type to **ANYEDGE**, since we are going to be checking both the rising and the falling edges.

Last step is calling *gpio_config* and passing the structure as a parameter. Now the GPIO pin is configured.

Second step is going to be creating a queue for all the events generated. Events and interruptions “are" the same in this context. The reason why we need this queue is because we might later want to recover some information that was generated in those interruptions, as we will see later.



```
/* create queue to handle gpio events */
    gpio_evt_queue = xQueueCreate(10, sizeof(uint32_t));
    if(gpio_evt_queue == 0){
        ESP_LOGE("HC_SR05", "Failed to create event queue");
        return NULL;
    }

    /* create task to handle gpio events
    - gpio_event_task: task function
    - "gpio_event_task": task name
    - 2048: stack size
    - NULL: task parameters
    - 5: task priority
    - NULL: task handle
    */
    xTaskCreate(gpio_event_task, "gpio_event_task", 2048, NULL, 5, NULL);

```

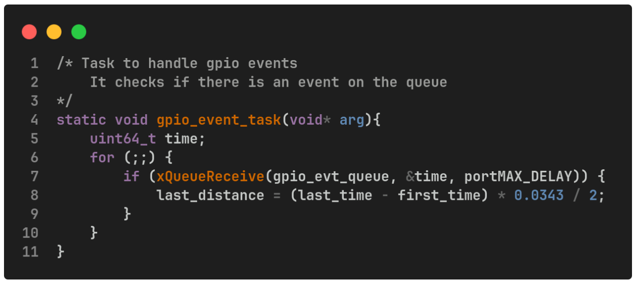
By using the function xQueueCreate, we are going to create a queue that can store up to 10 events(without being dispatched), and the size of each event, the information we are going to store, is going to be of type uint_32. For this case, since we only generate 2 interruptions per sensor activation, a size of 2 for the queue should be enough if we use this queue exclusively for this sensor. A size of 10 is a safer approach though.

Then we have to create the task, using the function xTaskCreate. We pass as parameters:

- **gpio_event_task**: function that is going to be in charge of dispatching the queue events.
- **“gpio_event_task"**: kind of an identifier name for the task.
- **2048**: stack depth. Not enough time to fully understand how exactly this parameter affects, but 2048 is a safe value.
- **NULL(task parameteres)**: since we don’t need to send any parameters to the task, we set it to NULL.
- **5**: task priority.
- **NULL()**: *Used to pass a handle to the created task out of the xTaskCreate() function. pxCreatedTask is optional and can be set to NULL.*

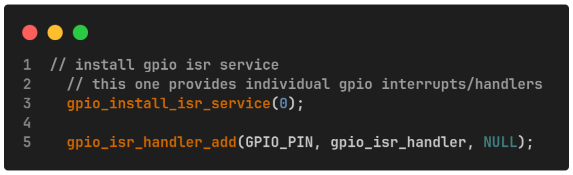
The new task will be the one in charge or manipulating the data generated by the interruptions, or executing whatever code we put in there.

Now, queues are allocated in the RAM(or heap memory in this case) that is available for the FreeRTOS system, so we have to be very careful with the amount of resources we assign to the queues, and we should always try to **optimize** the space, since we are working with embedded systems.



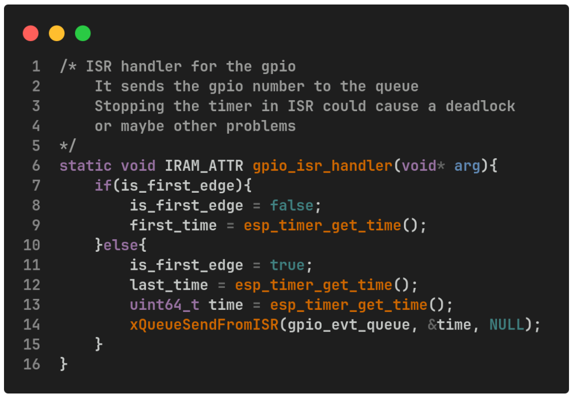
```
/* Task to handle gpio events
    It checks if there is an event on the queue
*/
static void gpio_event_task(void* arg){
    uint64_t time;
    for (;;) {
        if (xQueueReceive(gpio_evt_queue, &time, portMAX_DELAY)) {
            last_distance = (last_time - first_time) * 0.0343 / 2;
        }
    }
}
```

gpio_event_taks is the function in charge of dispatching the events from the queue and performing the necessary calculations for the distance.



```
// install gpio isr service
  // this one provides individual gpio interrupts/handlers
  gpio_install_isr_service(0);

  gpio_isr_handler_add(GPIO_PIN, gpio_isr_handler, NULL);
```

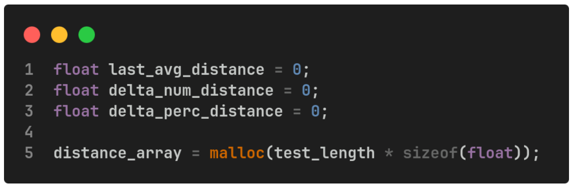
Third step is installing the *isr service* for the GPIO pins. This allows each GPIO pin to have its own interruption handler. Call ***gpio_install_isr_service*** and then add a handler for the selected pin with ***gpio_isr_handler_add***. An ISR handler is a short and small function that we call/execute when the interruption occurs. We should avoid passing arguments to it, and make it as short and fast as possible.



```c
/* ISR handler for the gpio
    It sends the gpio number to the queue
    Stopping the timer in ISR could cause a deadlock
    or maybe other problems
*/
static void IRAM_ATTR gpio_isr_handler(void* arg){
    if(is_first_edge){
        is_first_edge = false;
        first_time = esp_timer_get_time();
    }else{
        is_first_edge = true;
        last_time = esp_timer_get_time();
        uint64_t time = esp_timer_get_time();
        xQueueSendFromISR(gpio_evt_queue, &time, NULL);
    }
}
```

This function, as shown in the image attached, will be triggered every time a change in any of the edges is detected of the chosen GPIO pin. The HC-SR05 data signal starts when it pull up the signal to HIGH, thus a rising edge; and ends the signal with dropping it back to LOW, thus a falling edge.
When we detect the first edge(rising), we call the function ***esp_timer_get_time*** (explanation in ***Distance calculation*** section) and store the value in a variable called "*first_time*". When we detect the second edge(falling), we follow the same procedure but store the value in "*last_time*". And **only** when a falling edge is detected, a new event is sent to the queue, by using ***xQueueSendFromISR***, indicating the queue we want to use, a parameter we would like to pass, and in this case as NULL, the priority of the event.

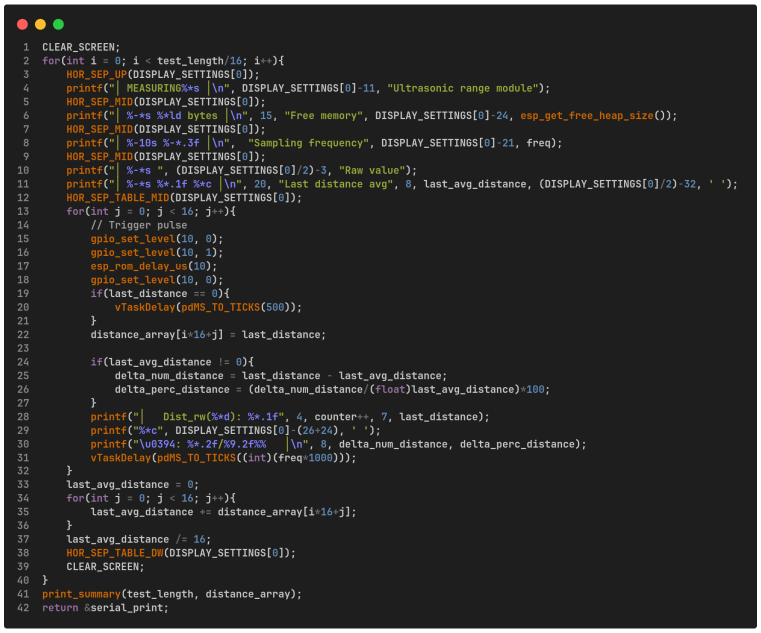
In order to store some temporary data, a few new variables are created.



```c
float last_avg_distance = 0;
float delta_num_distance = 0;
float delta_perc_distance = 0;

distance_array = malloc(test_length * sizeof(float));
```

- **last_avg_distance** stores the average of the last 16 measured values.
- Both *delta* variables will store the delta calculated to the previous average.
- **distance_array** allocates as much space needed to store all the measurements from the test.

Now is time to format and print the GUI and display the data to the user. CLEAR_SCREEN is executed. Then a for loop starts, which will iterate a total of $test\_lenght/16$. This is just a personal preference for the GUI, since this will make the program show at maximum 16 measurements at the same time.



```c
CLEAR_SCREEN;
for(int i = 0; i < test_length/16; i++){
    HOR_SEP_UP(DISPLAY_SETTINGS[0]);
    printf("│ MEASURING%*s │\n", DISPLAY_SETTINGS[0]-11, "Ultrasonic range module");
    HOR_SEP_MID(DISPLAY_SETTINGS[0]);
    printf("│ %-*s %*ld bytes │\n", 15, "Free memory", DISPLAY_SETTINGS[0]-24, esp_get_free_heap_size());
    HOR_SEP_MID(DISPLAY_SETTINGS[0]);
    printf("│ %-10s %-*.3f │\n",  "Sampling frequency", DISPLAY_SETTINGS[0]-21, freq);
    HOR_SEP_MID(DISPLAY_SETTINGS[0]);
    printf("│ %-*s ", (DISPLAY_SETTINGS[0]/2)-3, "Raw value");
    printf("│ %-*s %*.1f %*c │\n", 20, "Last distance avg", 8, last_avg_distance, (DISPLAY_SETTINGS[0]/2)-32, ' ');
    HOR_SEP_TABLE_MID(DISPLAY_SETTINGS[0]);
    for(int j = 0; j < 16; j++){
        // Trigger pulse
        gpio_set_level(10, 0);
        gpio_set_level(10, 1);
        esp_rom_delay_us(10);
        gpio_set_level(10, 0);
        if(last_distance == 0){
            vTaskDelay(pdMS_TO_TICKS(500));
        }
        distance_array[i*16+j] = last_distance;

        if(last_avg_distance != 0){
            delta_num_distance = last_distance - last_avg_distance;
            delta_perc_distance = (delta_num_distance/(float)last_avg_distance)*100;
        }
        printf("│   Dist_rw(%*d): %*.1f", 4, counter++, 7, last_distance);
        printf("%*c", DISPLAY_SETTINGS[0]-(26+24), ' ');
        printf("\u0394: %*.2f/%9.2f%%   │\n", 8, delta_num_distance, delta_perc_distance);
        vTaskDelay(pdMS_TO_TICKS((int)(freq*1000)));
    }
    last_avg_distance = 0;
    for(int j = 0; j < 16; j++){
        last_avg_distance += distance_array[i*16+j];
    }
    last_avg_distance /= 16;
    HOR_SEP_TABLE_DW(DISPLAY_SETTINGS[0]);
    CLEAR_SCREEN;
}
print_summary(test_length, distance_array);
return &serial_print;
```

Test header is printed and a new for loop starts, this time it will only iterate 16 times, for the reason explained before.

It's time to activate the sensor. To do this, we send a short HIGH signal through the *trigger* pin, and wait for the sensor to send a signal back through the *echo* pin.

After this, the interruptions will be triggered and the distance will be calculated. The new value is stored in it's position of the array allocated before, and the deltas are calculated. If previous average is 0, we of course can't calculate a delta, so it remains to 0.

Last step is printing a new line with the measurement and the calculated deltas. As we exit the for loop, the average of the last 16 values is calculated.

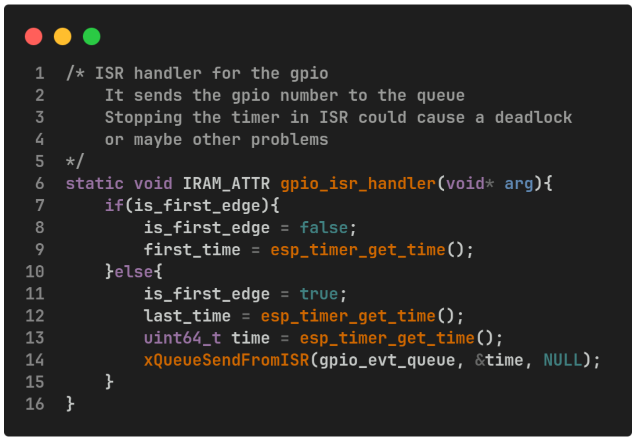
Eventually, when the test is finished, the summary of the test is printed and we return the pointer to the *serial_print* function.

Distance calculation

In order to calculate the distance, I have to find a way to measure timings precisely. As we saw in the measurement procedure of the HC-SR05, the distance measured is proportional to the amount of time the signal is HIGH. This is the perfect usage case for a **Timer**.

Now, at the beginning, while reading and searching the documentation about **General Purpose Timers**, my idea was to start a timer when the rising edge was detected, and stop it when the falling edge was detected. However, these timers are not really designed to be started and stopped continuously, but to be triggered automatically when the setted limit time is reached and, let's put an example, a device connection was not established in a reasonable amount of time.

I encountered multiple problems while testing and trying to understand how they work. In none of the cases I managed to stop then in an interruption without the system crashing. Probably, with more time and testing, I could end up with a code that works with this aproach. However, I decided to discard **General Purpose Timers** and use the **built-in timer** that is always running since the system boots. It is a 64-bit timer with 1 microsecond precision.

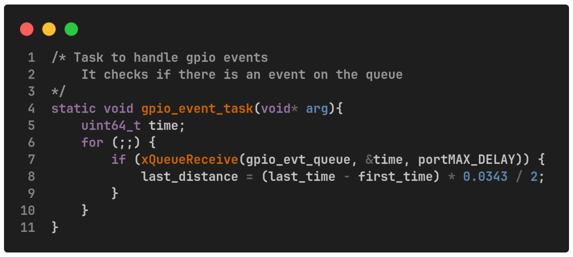


```c
/* ISR handler for the gpio
    It sends the gpio number to the queue
    Stopping the timer in ISR could cause a deadlock
    or maybe other problems
*/
static void IRAM_ATTR gpio_isr_handler(void* arg){
    if(is_first_edge){
        is_first_edge = false;
        first_time = esp_timer_get_time();
    }else{
        is_first_edge = true;
        last_time = esp_timer_get_time();
        uint64_t time = esp_timer_get_time();
        xQueueSendFromISR(gpio_evt_queue, &time, NULL);
    }
}
```

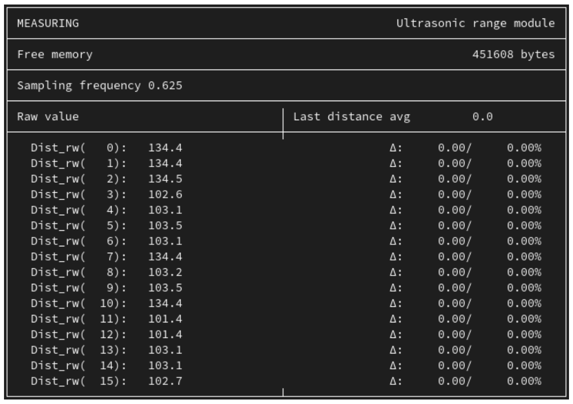
Calling ***esp_timer_get_time*** function will return the timer value, that we store in a variable. To calculate the distance, I apply the following formula:

$$Distance = \frac{(last\_time - first\_time) \times 0.0340}{2}$$

- 0.0340 is the speed of sound in m/s (340m/s) divided by 100 to convert the output to centimetres.
- *We divide distance by 2 because the sensor returns the round trip time, which doubles the distance measurement*



```c
/* Task to handle gpio events
    It checks if there is an event on the queue
*/
static void gpio_event_task(void* arg){
    uint64_t time;
    for (;;) {
        if (xQueueReceive(gpio_evt_queue, &time, portMAX_DELAY)) {
            last_distance = (last_time - first_time) * 0.0343 / 2;
        }
    }
}
```

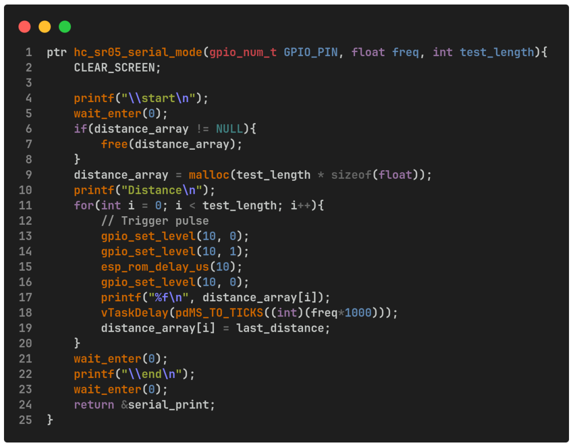


```
MEASURING                                        Ultrasonic range module
Free memory                                                 451608 bytes
Sampling frequency 0.625
Raw value                           Last distance avg       0.0
  Dist_rw(   0):   134.4                        Δ:     0.00/     0.00%
  Dist_rw(   1):   134.4                        Δ:     0.00/     0.00%
  Dist_rw(   2):   134.5                        Δ:     0.00/     0.00%
  Dist_rw(   3):   102.6                        Δ:     0.00/     0.00%
  Dist_rw(   4):   103.1                        Δ:     0.00/     0.00%
  Dist_rw(   5):   103.5                        Δ:     0.00/     0.00%
  Dist_rw(   6):   103.1                        Δ:     0.00/     0.00%
  Dist_rw(   7):   134.4                        Δ:     0.00/     0.00%
  Dist_rw(   8):   103.2                        Δ:     0.00/     0.00%
  Dist_rw(   9):   103.5                        Δ:     0.00/     0.00%
  Dist_rw(  10):   134.4                        Δ:     0.00/     0.00%
  Dist_rw(  11):   101.4                        Δ:     0.00/     0.00%
  Dist_rw(  12):   101.4                        Δ:     0.00/     0.00%
  Dist_rw(  13):   103.1                        Δ:     0.00/     0.00%
  Dist_rw(  14):   103.1                        Δ:     0.00/     0.00%
  Dist_rw(  15):   102.7                        Δ:     0.00/     0.00%
```

Figure 7: Running a test

SERIAL MODE

If we would like to be able to capture the data and later put it into Matlab, as we will do later, this mode will print the values in the terminal without any extra formatting.



```c
ptr hc_sr05_serial_mode(gpio_num_t GPIO_PIN, float freq, int test_length){
    CLEAR_SCREEN;

    printf("\\start\n");
    wait_enter(0);
    if(distance_array != NULL){
        free(distance_array);
    }
    distance_array = malloc(test_length * sizeof(float));
    printf("Distance\n");
    for(int i = 0; i < test_length; i++){
        // Trigger pulse
        gpio_set_level(10, 0);
        gpio_set_level(10, 1);
        esp_rom_delay_us(10);
        gpio_set_level(10, 0);
        printf("%f\n", distance_array[i]);
        vTaskDelay(pdMS_TO_TICKS((int)(freq*1000)));
        distance_array[i] = last_distance;
    }
    wait_enter(0);
    printf("\\end\n");
    wait_enter(0);
    return &serial_print;
}
```

A "start" message is printed and the program waits for an Enter input. This is the moment when we start logging the output. Also, we make sure the pointer to the array that will store the data is NULL before allocating new memory, and if that's the case, we *free* it.

The test starts and prints all the values. The measurement procedure is exactly the same as in the main function.

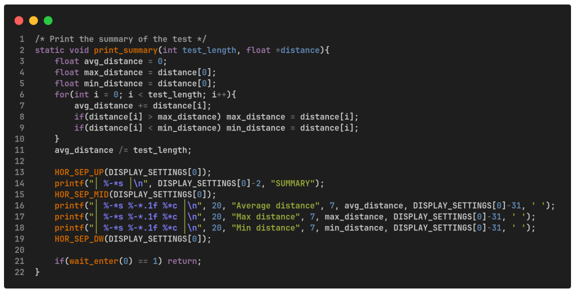
Eventually, when the test is finished, the program waits for the user to press the enter key. Here is when we stop the logging and save the file in the project directory. The program returns a pointer to the *serial_print* function.

CAPTURING TERMINAL OUTPUT: To capture the terminal output, we can use the monitor tool included in **idf.py**. By pressing *Ctrl + T* we enter into menu mode, where we have a lot of options.

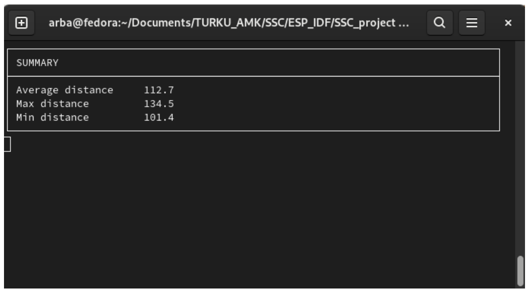
Ctrl + L starts or stop the logging and saves the file in our project directory.

PRINT SUMMARY

It calculates the max, min and average value of the test. It also prints a header for a better looking.



```c
/* Print the summary of the test */
static void print_summary(int test_length, float *distance){
    float avg_distance = 0;
    float max_distance = distance[0];
    float min_distance = distance[0];
    for(int i = 0; i < test_length; i++){
        avg_distance += distance[i];
        if(distance[i] > max_distance) max_distance = distance[i];
        if(distance[i] < min_distance) min_distance = distance[i];
    }
    avg_distance /= test_length;

    HOR_SEP_UP(DISPLAY_SETTINGS[0]);
    printf("│ %-*s │\n", DISPLAY_SETTINGS[0]-2, "SUMMARY");
    HOR_SEP_MID(DISPLAY_SETTINGS[0]);
    printf("│ %-*s %-*.1f %*c │\n", 20, "Average distance", 7, avg_distance, DISPLAY_SETTINGS[0]-31, ' ');
    printf("│ %-*s %-*.1f %*c │\n", 20, "Max distance", 7, max_distance, DISPLAY_SETTINGS[0]-31, ' ');
    printf("│ %-*s %-*.1f %*c │\n", 20, "Min distance", 7, min_distance, DISPLAY_SETTINGS[0]-31, ' ');
    HOR_SEP_DW(DISPLAY_SETTINGS[0]);

    if(wait_enter(0) == 1) return;
}
```

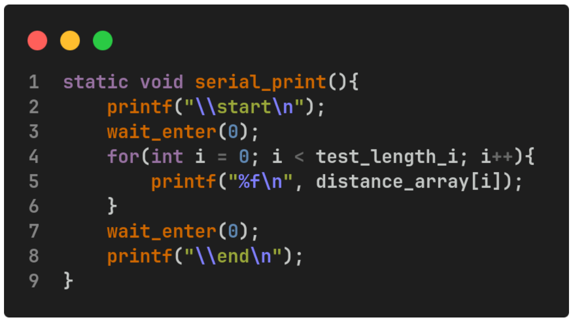


arba@fedora:~/Documents/TURKU_AMK/SSC/ESP_IDF/SSC_project ...

```
SUMMARY

Average distance     112.7
Max distance         134.5
Min distance         101.4
```

SERIAL PRINT

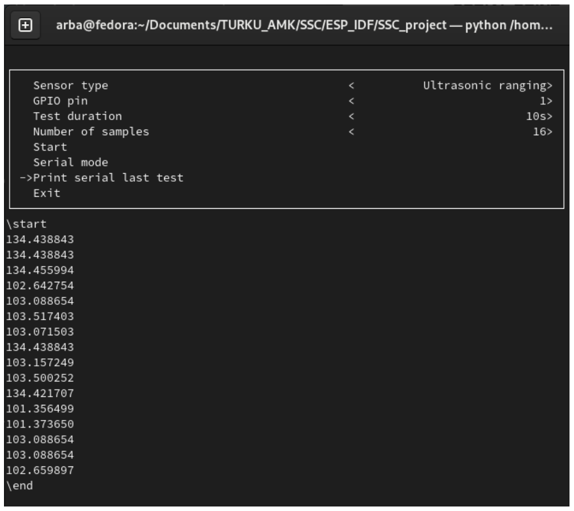
This small function prints the last test results. It stops before and after printing, so the user can save everything in a file.



```c
static void serial_print(){
    printf("\\start\n");
    wait_enter(0);
    for(int i = 0; i < test_length_i; i++){
        printf("%f\n", distance_array[i]);
    }
    wait_enter(0);
    printf("\\end\n");
}
```

The reason why I decided to make this function only accessible by returning the pointer to it, as seen in the previous code explained, is to unify for all sensors that are added to the code later. Let's explain this better with an example:

- Some sensors might measure multiple values at the same time, like the DHT-11, others will only measure one at once, like the HC-SR05. Others might even want to use a different type of data structure.
- By returning a pointer to the function, we avoid having to check from what type of sensor the data comes from, or its necessary formatting when printing the data. This might not be the best way, but since I program each sensor in a different file, I found this procedure the best fitting.



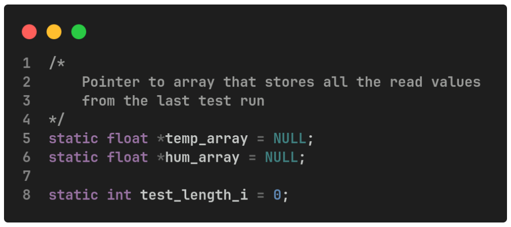
arba@fedora:~/Documents/TURKU_AMK/SSC/ESP_IDF/SSC_project — python /hom...

```
   Sensor type                             <         Ultrasonic ranging>
   GPIO pin                                <                          1>
   Test duration                           <                        10s>
   Number of samples                       <                        16>
   Start
   Serial mode
 ->Print serial last test
   Exit
\start
134.438843
134.438843
134.455994
102.642754
103.088654
103.517403
103.071503
134.438843
103.157249
103.500252
134.421707
101.356499
101.373650
103.088654
103.088654
102.659897
\end
```

4.3 dht11_w.c

This file pretends to be used as a wrapper of the original
DHT-11 driver provided by the ESP-IDF team in their github, which can be found here.

We create 2 variables that will later store the address to the float array that stores the last test values. The reason why we instantiate them here, out of any function, is because we need them to be accessible by multiple functions.
However, with the **static** option we limit the scope of this variables to only this file, making them "private".



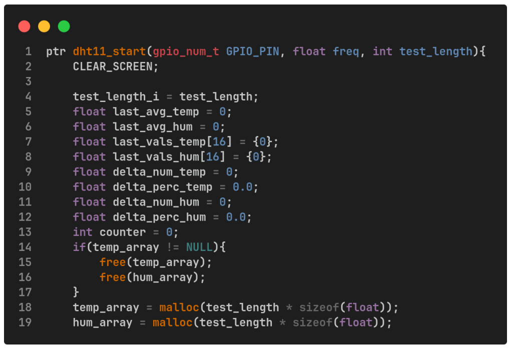
```c
/*
    Pointer to array that stores all the read values
    from the last test run
*/
static float *temp_array = NULL;
static float *hum_array = NULL;

static int test_length_i = 0;
```

DHT-11 START

Parameters:

- **GPIO_PIN**: pin used for data connection with the sensor
- **freq**: frequency of measurement
- **test_lenght**: total number of samples of the test

For each read value(temperature and humidity) there will be a variable storing the last value read, and array containing the last 16 values, and a delta to the last average in numeric and percentage type.

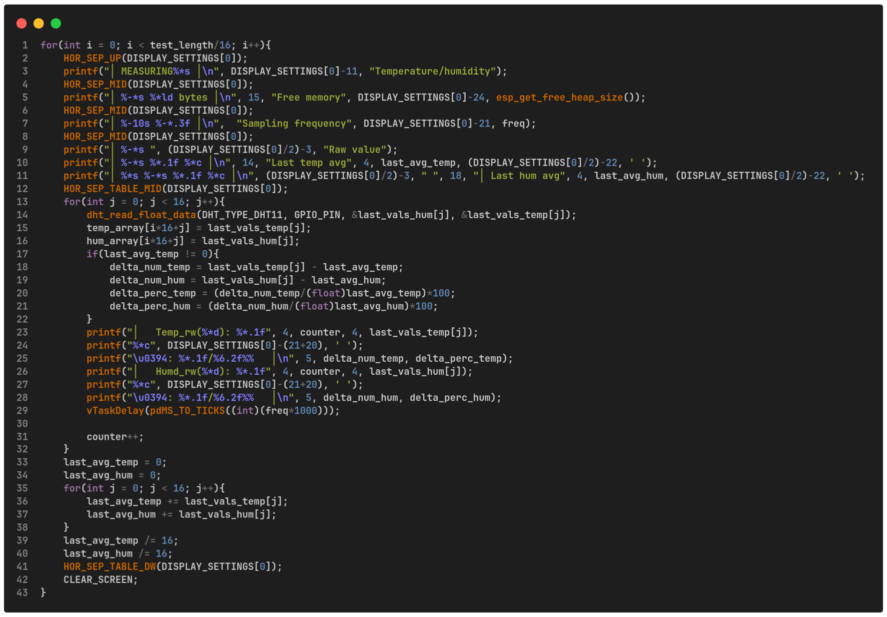


```c
ptr dht11_start(gpio_num_t GPIO_PIN, float freq, int test_length){
    CLEAR_SCREEN;

    test_length_i = test_length;
    float last_avg_temp = 0;
    float last_avg_hum = 0;
    float last_vals_temp[16] = {0};
    float last_vals_hum[16] = {0};
    float delta_num_temp = 0;
    float delta_perc_temp = 0.0;
    float delta_num_hum = 0;
    float delta_perc_hum = 0.0;
    int counter = 0;
    if(temp_array != NULL){
        free(temp_array);
        free(hum_array);
    }
    temp_array = malloc(test_length * sizeof(float));
    hum_array = malloc(test_length * sizeof(float));
```

Note here: the reason why **temp_array** and **hum_array** are initialized using malloc, is because we need these arrays to be stored in the heap, since we are later going to need access from outside of the function. If we initialize the same way as we did in the previous ones, then once we exit the function scope, they are no longer accessible, because they are stored in the stack.

In case the previous arrays were already allocated with malloc, we first free them to avoid possible problems with data in memory.

A for loop starts, and it iterates a total of $test\_lenght/16$ times. Then, the header of the GUI is printed T in



```c
for(int i = 0; i < test_length/16; i++){
    HOR_SEP_UP(DISPLAY_SETTINGS[0]);
    printf("│ MEASURING%*s │\n", DISPLAY_SETTINGS[0]-11, "Temperature/humidity");
    HOR_SEP_MID(DISPLAY_SETTINGS[0]);
    printf("│ %-*s %*ld bytes │\n", 15, "Free memory", DISPLAY_SETTINGS[0]-24, esp_get_free_heap_size());
    HOR_SEP_MID(DISPLAY_SETTINGS[0]);
    printf("│ %-10s %-*.3f │\n",  "Sampling frequency", DISPLAY_SETTINGS[0]-21, freq);
    HOR_SEP_MID(DISPLAY_SETTINGS[0]);
    printf("│ %-*s ", (DISPLAY_SETTINGS[0]/2)-3, "Raw value");
    printf("│ %-*s %*.1f %*c │\n", 14, "Last temp avg", 4, last_avg_temp, (DISPLAY_SETTINGS[0]/2)-22, ' ');
    printf("│ %*s %-*s %*.1f %*c │\n", (DISPLAY_SETTINGS[0]/2)-3, " ", 18, "│ Last hum avg", 4, last_avg_hum, (DISPLAY_SETTINGS[0]/2)-22, ' ');
    HOR_SEP_TABLE_MID(DISPLAY_SETTINGS[0]);
    for(int j = 0; j < 16; j++){
        dht_read_float_data(DHT_TYPE_DHT11, GPIO_PIN, &last_vals_hum[j], &last_vals_temp[j]);
        temp_array[i*16+j] = last_vals_temp[j];
        hum_array[i*16+j] = last_vals_hum[j];
        if(last_avg_temp != 0){
            delta_num_temp = last_vals_temp[j] - last_avg_temp;
            delta_num_hum = last_vals_hum[j] - last_avg_hum;
            delta_perc_temp = (delta_num_temp/(float)last_avg_temp)*100;
            delta_perc_hum = (delta_num_hum/(float)last_avg_hum)*100;
        }
        printf("│   Temp_rw(%*d): %*.1f", 4, counter, 4, last_vals_temp[j]);
        printf("%*c", DISPLAY_SETTINGS[0]-(21+20), ' ');
        printf("\u0394: %*.1f/%6.2f%%    │\n", 5, delta_num_temp, delta_perc_temp);
        printf("│   Humd_rw(%*d): %*.1f", 4, counter, 4, last_vals_hum[j]);
        printf("%*c", DISPLAY_SETTINGS[0]-(21+20), ' ');
        printf("\u0394: %*.1f/%6.2f%%    │\n", 5, delta_num_hum, delta_perc_hum);
        vTaskDelay(pdMS_TO_TICKS((int)(freq*1000)));

        counter++;
    }
    last_avg_temp = 0;
    last_avg_hum = 0;
    for(int j = 0; j < 16; j++){
        last_avg_temp += last_vals_temp[j];
        last_avg_hum += last_vals_hum[j];
    }
    last_avg_temp /= 16;
    last_avg_hum /= 16;
    HOR_SEP_TABLE_DW(DISPLAY_SETTINGS[0]);
    CLEAR_SCREEN;
}
```

the terminal. A new for loop starts, and it iterates 16 times, since I thought that 16 values are good enough to be shown at the same time in the GUI. An average of these 16 values is later calculated.

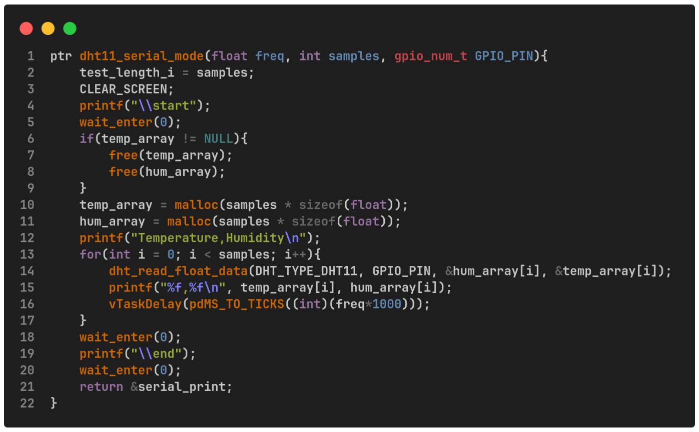
Next step is reading data from the sensor. We call the ***dht_read_float_data*** function, and pass as parameters: the name of the sensor, the pin used for the sensor, and a pointer to the temperature variable and another one for the humidity variable.

Store the new data in the test array, and calculate the delta to later display the values properly.
After everything has been printed and the 16 values round is finished, calculate the average of the last 16 values and start again.

SERIAL MODE

Parameters:

- **freq**: frequency of measurement
- **samples**: total number of samples of the test
- **GPIO_PIN**: pin used for data connection with the sensor



```c
ptr dht11_serial_mode(float freq, int samples, gpio_num_t GPIO_PIN){
    test_length_i = samples;
    CLEAR_SCREEN;
    printf("\\start");
    wait_enter(0);
    if(temp_array != NULL){
        free(temp_array);
        free(hum_array);
    }
    temp_array = malloc(samples * sizeof(float));
    hum_array = malloc(samples * sizeof(float));
    printf("Temperature,Humidity\n");
    for(int i = 0; i < samples; i++){
        dht_read_float_data(DHT_TYPE_DHT11, GPIO_PIN, &hum_array[i], &temp_array[i]);
        printf("%f,%f\n", temp_array[i], hum_array[i]);
        vTaskDelay(pdMS_TO_TICKS((int)(freq*1000)));
    }
    wait_enter(0);
    printf("\\end");
    wait_enter(0);
    return &serial_print;
}
```

This function prints all the values in serial mode, this is without a GUI, so that the user can capture all the values in a file and later put it into matlab or other scripts to analyse the results, as we will do later.

A start message is printed and the program wait for the user to press enter. Then, the values are measured and printed and, after the for loop is finished, the program waits again for the user to press enter, prints and end message, and returns back to the previous menu.

Returns:

- pointer to the *serial_print* function, that will be explained later.

PRINT SUMMARY

Parameters:

- **test_lenght**: total number of samples of the test
- ***temp**: pointer to the temperature array
- ***hum**: pointer to the humidity array

Same operation as 4.2

Returns:

- pointer to the *serial_print* function, that will be explained later.

SERIAL PRINT

See 4.2. Operation is the same.

4.4 main.c

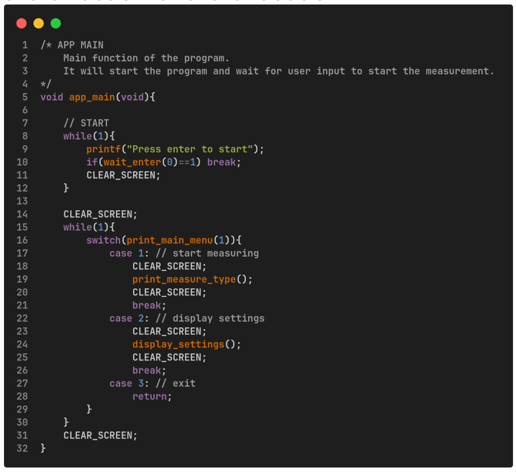
This file contains the main code of the program, as well as the functions in charge of printing the GUI and the menus.

APP MAIN

This is the main program function. All ESP-IDF must have a function named ***app_main***.

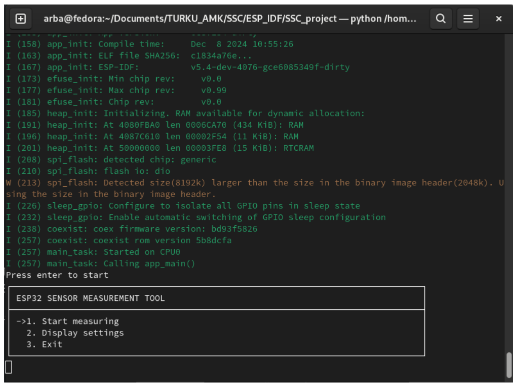
First, a “Press enter to start" is printed to let the user know the micro-controller is ready to be used. This message remains in the screen until the user press the key *Enter*.

After this, the main menu function is called and the returned value is evaluated in a switch, that will decide the direction of the execution.



```c
/* APP MAIN
    Main function of the program.
    It will start the program and wait for user input to start the measurement.
*/
void app_main(void){

    // START
    while(1){
        printf("Press enter to start");
        if(wait_enter(0)==1) break;
        CLEAR_SCREEN;
    }

    CLEAR_SCREEN;
    while(1){
        switch(print_main_menu(1)){
            case 1: // start measuring
                CLEAR_SCREEN;
                print_measure_type();
                CLEAR_SCREEN;
                break;
            case 2: // display settings
                CLEAR_SCREEN;
                display_settings();
                CLEAR_SCREEN;
                break;
            case 3: // exit
                return;
        }
    }
    CLEAR_SCREEN;
}
```

(a) app_main code

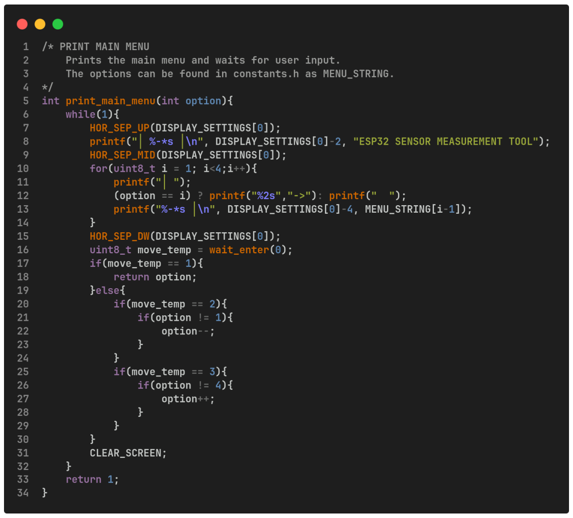


arba@fedora:~/Documents/TURKU_AMK/SSC/ESP_IDF/SSC_project — python /hom...

```
I (158) app_init: Compile time:     Dec  8 2024 10:55:26
I (163) app_init: ELF file SHA256:  c1834a76e...
I (167) app_init: ESP-IDF:          v5.4-dev-4076-gce6085349f-dirty
I (173) efuse_init: Min chip rev:     v0.0
I (177) efuse_init: Max chip rev:     v0.99
I (181) efuse_init: Chip rev:         v0.0
I (185) heap_init: Initializing. RAM available for dynamic allocation:
I (191) heap_init: At 4080FBA0 len 0006CA70 (434 KiB): RAM
I (196) heap_init: At 4087C610 len 00002F54 (11 KiB): RAM
I (201) heap_init: At 50000000 len 00003FE8 (15 KiB): RTCRAM
I (208) spi_flash: detected chip: generic
I (210) spi_flash: flash io: dio
W (213) spi_flash: Detected size(8192k) larger than the size in the binary image header(2048k). U
sing the size in the binary image header.
I (226) sleep_gpio: Configure to isolate all GPIO pins in sleep state
I (232) sleep_gpio: Enable automatic switching of GPIO sleep configuration
I (238) coexist: coex firmware version: bd93f5826
I (257) coexist: coexist rom version 5b8dcfa
I (257) main_task: Started on CPU0
I (257) main_task: Calling app_main()
Press enter to start
 ESP32 SENSOR MEASUREMENT TOOL

 ->1. Start measuring
   2. Display settings
   3. Exit
```

(b) Start and main menu

PRINT MAIN MENU



```c
/* PRINT MAIN MENU
    Prints the main menu and waits for user input.
    The options can be found in constants.h as MENU_STRING.
*/
int print_main_menu(int option){
    while(1){
        HOR_SEP_UP(DISPLAY_SETTINGS[0]);
        printf("│ %-*s │\n", DISPLAY_SETTINGS[0]-2, "ESP32 SENSOR MEASUREMENT TOOL");
        HOR_SEP_MID(DISPLAY_SETTINGS[0]);
        for(uint8_t i = 1; i<4;i++){
            printf("│ ");
            (option == i) ? printf("%2s","->"): printf("  ");
            printf("%-*s │\n", DISPLAY_SETTINGS[0]-4, MENU_STRING[i-1]);
        }
        HOR_SEP_DW(DISPLAY_SETTINGS[0]);
        uint8_t move_temp = wait_enter(0);
        if(move_temp == 1){
            return option;
        }else{
            if(move_temp == 2){
                if(option != 1){
                    option--;
                }
            }
            if(move_temp == 3){
                if(option != 4){
                    option++;
                }
            }
        }
        CLEAR_SCREEN;
    }
    return 1;
}
```

Prints the main menu options. These are:

- **Start measuring**
- **Display settings**
- **Exit**

More entries could be added here, like a settings menu to configure the WiFi settings for example. The strings used can be found in the file *constants.c*, in the string array called *MENU_STRINGS*.

By evaluating the output from *wait_enter* we can know what key the user pressed, and therefore moving the selection arrow up or down.

DISPLAY SETTINGS

This entry is not really functional, since you can only change the GUI width. The minimum width is 60 and the maximum 120.

In the future I would like to add more options. I don't paste an image with the code here because it too long, but the way it works is similar to the menu shown before.

PRINT TEST TYPE

This is the test configuration screen. Again, I won't paste an image of the code because the way it works is similar to the previous explained and is too long. By reading the code alongside this PDF should be enough to understand it.

The available settings for the test are:

- **Sensor type**: choose between multiple sensors. So far only the **HC-SR04/5** and the **DHT-11** are working.
- **GPIO pin**: not fully functional, but it let you choose the GPIO pin you would like to use for the data line.
- **Test duration**: lets you choose the test length in time units. The values can be found in the *constans.c* file, in the array *TEST_LENGTH/TEST_LENGTH_VALUES*
- **Number of samples**: lets you choose the amount of samples you want to take.
  The frequency is $(test\ length)/(number\ of\ samples)$

The available options are:

- **Start**: runs the test in GUI mode.
- **Serial mode**: runs the test in serial mode.
- **Print serial last test**: calls the function stored in the variable *last_test_print_serial* if it's not NULL.
- **Exit**: returns to the main menu.

5 ANALYSIS

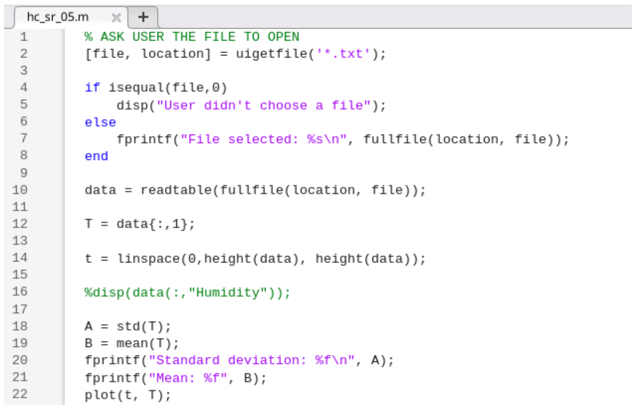
In this section I am going to analyse the data gathered from the HC-SR05 sensor.

I put the sensor at a distance of 55cm from a wall and run a test of 512 samples and 10 minutes lenght.

TYPE A UNCERTAINTY

Type A uncertainty is used to quantify the variation or the error in the measurement of the sensor in a space of time. We aim to show how good the repeatability of the sensor is.

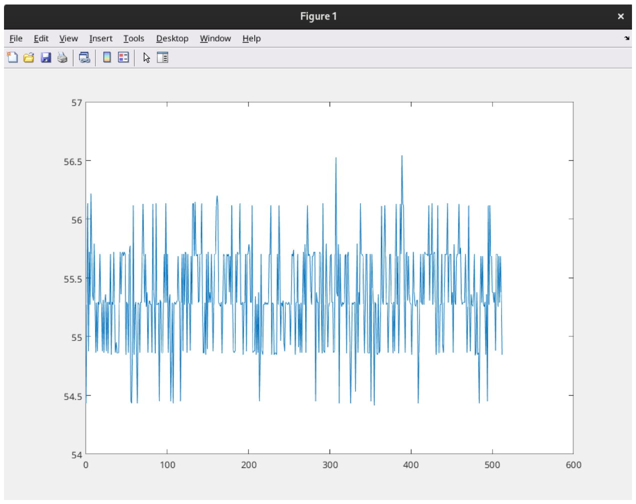
The matlab code is the following:



```
% ASK USER THE FILE TO OPEN
[file, location] = uigetfile('*.txt');

if isequal(file,0)
    disp("User didn't choose a file");
else
    fprintf("File selected: %s\n", fullfile(location, file));
end

data = readtable(fullfile(location, file));

T = data{:,1};

t = linspace(0,height(data), height(data));

%disp(data(:,"Humidity"));

A = std(T);
B = mean(T);
fprintf("Standard deviation: %f\n", A);
fprintf("Mean: %f", B);
plot(t, T);
```

It first prompt the user a UI to choose the file in the file explorer. That file is later loaded, converted into a table and then the values are calculated.
The output from the execution with the file “test.txt" that can be found in the github is:

```
>> hc_sr_05
File selected: /home/arba/Documents/TURKU_AMK/SSC/ESP_IDF/SSC_project/test.txt
Standard deviation: 0.399204
Mean: 55.353266
```



The mean value is **55.35**. And the standard deviation is **0.399204**.
This value is really close to the precision of the sensor, which clearly indicates than in a stable environment, the sensor repeatability is really good.

However, is important to now that in some environments we might need to calibrate the sensor.

TYPE B UNCERTAINTY

For this sensor, Type B uncertainties are slightly different to the typical ones:

- **Air Temperature**: this sensor uses ultrasonic waves, which travel at the speed of the sound. This is speed varies depending on multiple factors, and one of them is the air temperature. If, for example, the temperature varies between the sensors and the object(imagine there is air flowing), the measurements will probably not be consistent.
  That also means that it might be necessary to calibrate the sensor.
- **Surface of the object**: if the object surface is not flat, waves might bounce in different directions, therefore making the measurement less precise, and even induce more error. Always try to point to a flat surface.
- **Timer resolution**: the resolution of the timer used to measure the time of flight have great impact in the results.
- **Resolution of the sensor**: the smallest change the sensor can measure.

6 LEARNING ACHIEVEMENTS

Thanks to this project, I've learned a lot of things.

- **Introduction to the ESP-32 micro-controllers**: I have always wanted to learn about them, but never got the chance to test one. I think they are really powerful micro-controllers and have a lot of potential usages.
- **C programming**: I already knew a bit about C programming, since I did some projects in the past. However, with this project I learnt more things about pointers, memory management, and how to do a bit more efficient code.
- **Measurement analysis**: during this course and the realization of this project, I learned about measurement procedures, uncertainty and more important concepts.