A simple fitness tracker

In this lab, you will build a simple fitness tracker, a sort of proto-Fitbit.

The device will:

- Acquire acceleration from an accelerometer
- Detect steps from acceleration and count them
- When the user presses a button, an LED will signal if the user has reached a certain number of steps (a "goal" for the day) or not

You will need:

- An ESP32 board
- An MPU6050 accelerometer
- A pushbutton
- An LED
- A breadboard and a bunch of wires

In terms of software, we are going to use the ESP-IDF and focussing on these aspects:

- Buffers and queues
- FreeRTOS tasks
- I2C communication
- Power management and sleep modes

Let's get started!

Getting acceleration

You will need to sample acceleration using the MPU6050 digital accelerometer. For this, you will need to develop a little module that initialises the I2C hardware of the ESP32, initialise the MPU6050 and get samples from it when requested.

I suggest you to go through the code examples where we have already seen how to communicate with the MPU6050. I expect you to set the power mode of the accelerometer and the sampling rate when you initialise it, and to read the ACCEL_XOUT, ACCEL_YOUT, and ACCEL_ZOUT registers when requesting a sample. We don't need the gyroscope in this lab.

Given that sending messages over the I2C bus requires a lot of boilerplate code, I also expect you to create a small module with a bunch of functions to simplify the initialisation, writing and reading from the bus. Something like:

```
/**
 * Initialises I2C bus, uses controller 0
 */
void initI2C(int sdapin, int sclpin);
```

```
/**
 * Writes one byte into a register
 */
void writeI2C(uint8_t address, uint8_t reg, uint8_t data);

/**
 * Reads len bytes and places them into a buffer, buffer must be pre-allocated
 */
void readI2C(uint8_t address, uint8_t register, uint8_t *buffer, int len);
```

In addition, the same module, or maybe another one, could abstract the accelerometer and have a function that initialises the accelerometer and another that retrieves the acceleration from the device. I'll let you think about how this last function can be developed: for example, you can have 3 functions, one per axis (x, y and z), or one single function returning a struct containing the acceleration for all the 3 axes.

Make sure you test your code! Place an infinite loop at the end of app_main() where you retrieve the samples and print them out. Do the numbers make sense? Two axes should be about 0 and one should give you an almost constant value (the gravitational acceleration). Try placing the accelerometer upside down: does the acceleration change sign on one axis?

At the end of this step, I would expect you to have a code similar to this:

```
void app_main()
{
    // initialise the I2C bus and the MPU6050

    while(1){
        // get acceleration
        // print acceleration

        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

Remove the infinite loop at the end once you're satisfied with your results.

Sleeping

Our fitness tracker should, in principle, last for days if not months on a single battery. This means that we need to put it into sleep mode when it's not doing anything.

Using the deep-sleep would be ideal, but remember that waking up from deep sleep implies a complete reboot, which takes, under the best conditions, at least 150ms. That may be too much if we need to sample the acceleration signal at a frequency higher than 6Hz. So, we'll go with light sleep instead, which only "freezes" the CPU and does not require full boot-up.

But how can I trigger light sleep, and when? Luckily for us, the ESP-IDF has recently introduced an automatic power management system that implements the FreeRTOS "tickless

idle" mode, which basically sets the ESP32 into light sleep when no task is executed. Activating this needs a bit of tweaking, so, first, I'd suggest you to read how power management works in the ESP32 extensively.

Then, let's setup the project to activate it. Let's skip the clock frequency change and focus on the automatic light sleep mode only. If you read the documentation carefully, you'll see that you need to set 2 options in the configuration of the project: CONFIG_PM_ENABLE and CONFIG_FREERTOS_USE_TICKLESS_IDLE. This is done thanks to a utility included in the ESP-IDF, called **menuconfig**.

Open the terminal: 

run: `pio run -t menuconfig`

You should see something like this:

```
(Top)
                                    Espressif IoT Development Framework Configuration
    SDK tool configuration  --->
    Build type  --->
    Application manager  --->
    Bootloader config  --->
    Security features  --->
    Serial flasher config  --->
    Partition Table  --->
    Compiler options  --->
    Component config  --->
    vvvvvvvvvvvvvv
[Space/Enter] Toggle/enter  [ESC] Leave menu           [S] Save
[O] Load                    [?] Symbol info            [/] Jump to symbol
[F] Toggle show-help mode   [C] Toggle show-name mode  [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Navigate in the menu with arrows and select items with spacebar. Esc to cancel. If your arrows don't work (some students have experienced this problem in Windows) you can use the "/" key to jump to a specific configuration option.

Set these the two configuration options:

- Component config -> Power management -> Enable "Support for power management" (only that)
- Component config -> FreeRTOS -> Enable "Tickless Idle support"

Save with S, then quit with Q.

What these settings do is telling the ESP-IDF how the running environment should be initialised. In practice, these set a bunch of #define that are buried under the thousands of files of the ESP-IDF framework, which will configure your board and initialise it properly.

Now compile to make sure everything is OK.

Once the automatic power management is activated, we need to configure it using esp_pm_configure() in your app_main(). You need to pass it a pointer to an *esp_pm_config_esp32_t* variable you properly initialise. The variable should set the *light_sleep_enable* field to true and the frequencies to something else. The frequencies should, in principle, be ignored, but a possibility could be setting max_freq_mhz to 80 and

min_freq_mhz to 13. Don't forget to make sure that calling the function has worked OK by checking its return value.

To test the low power mode, create a little program with a periodic FreeRTOS task with a long period (like 2 seconds) that keeps the CPU busy for a long period, like half second, with the ets_delay_us() function. Also, remember that app_main() is executed in the idle task, don't put any infinite loop there or the ESP will never go to sleep!

With this setup, you can use an amperemeter to measure current going through the circuit. Connect the amperemeter between power supply and the board. For power supply, you cannot use the regular USB plug, you will need to use the 3.3V pin (power supply must be regulated! Higher voltages can kill the board!). Once you power the board, you should observe the current going up and down on the amperemeter, depending if the task is being executed or not. When no task is being executed, the ESP will go into light sleep and the current consumption will be lower.

If you don't have access to an amperemeter, you could enable the CONFIG_PM_TRACE parameter with menuconfig (Component config > Power Management > Enable debug tracing of PM using GPIOs). This will activate some pins when power management is being triggered. Those pins are documented in pm_trace.c. You are probably interested in pin 27, which is switched on when light sleep is activated. Try placing an LED between that pin and GND: does it switch on regularly? If that's the case, it means that the ESP is entering light sleep mode: success!

If you activate the CONFIG_PM_TRACE flag, please remember to avoid using the debug pins later on. By default, these are pins: 4, 5, 16, 17, 18, 19, 25, 26 and 27. Otherwise, disable the flag before you continue.

At the end of this section, I would expect your code to look like this:

```
void app_main()
{
    // configure light sleep mode with esp_pm_configure()

    // initialise the I2C bus and the MPU6050
}
```

Sampling acceleration

Now we need to take samples from the accelerometer. First of all, you need to decide the sampling frequency to use. Please be considerate: you want to balance the precision of your device with the need to save power. What would be your Nyquist frequency? Check what would be a reasonably high number of steps per second a human being can do and do your computations.

Once you have decided what your sampling frequency is, please write down explicitly how your decision was made in the comments of your code.

As for the sampling approach, we could use a timer, as done in another lab, but here I want you to practice FreeRTOS tasks, so let's create a periodic task instead. The idea is that the

task periodically takes the samples and stores them into a buffer, which you can implement as a circular queue. The buffer will be then later analysed by the steps-counting algorithm, which runs on a separate task. The sampling task and the steps-counting tasks will need to share the buffer and cooperate in order to make sure they don't create race conditions.

In summary:

- The sampling task periodically takes a sample of acceleration and adds it to the buffer.
- The steps-counting algorithm task periodically analyses the buffer, computes the number of steps, and empties the buffer.

This should work smoothly because the sampling task only adds to the buffer and the steps-counting algorithm only removes from the buffer, so race conditions should be avoided.

One additional note: the steps-counting algorithm does not need to know the direction of the acceleration vector, it only cares about its intensity. For this reason, and to save space on the buffer, we can let the task compute the module (or magnitude) of the acceleration vector: $acc_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$ and save that instead of (x ,y, z) on the buffer. Remind that, even if the accelerometer is still, there will always be some acceleration due to gravity, so this quantity will rarely be zero (question for you: when can it be zero?).

When implementing the module, be aware that the square root is easily done in the ESP32, thanks to its floating-point coprocessor. The mathematical functions are inside math.h. Be also aware that pow() (the function that computes the power) and sqrt() (square root) use doubles, so it may be tempting to convert everything to double and store doubles in the buffer. Now, it is safe to assume that the magnitude of the acceleration vector will fit inside 32 bits integers because its components fit inside 16 bits, therefore, instead of having a buffer of doubles, which take space and are hard to compute, you can convert the result of the sqrt to an int32 or uint32 (magnitude is always positive!) and store that instead.

As a first test, let's ignore the buffer for now and let's create the periodic task, with a low frequency (half hertz?), and let it get samples from the accelerometer, compute the magnitude and print all of them (x, y, z and magnitude). When you develop the periodic task, make sure you use TaskDelayUntil() to guarantee a (somewhat) precise sampling rate. Alternatively, you can also use FreeRTOS software timers.

If you have done everything correctly, you should observe that when you rotate the accelerometer the x, y and z should change, while the magnitude should roughly stay constant. To give you an idea of the scale of these numbers, in my tests I get a magnitude that oscillates between 16800 and 17500 when still.

If this first test is successful, it's time to introduce the buffer. You should already know how to program a buffer: just take the code from past exercises. A circular queue will work fine. The only recommendation is that you need the following functions:

- Initialisation of the buffer (up to you if you want to initialise the data into the heap with malloc() or into the stack with a static variable)

- Add an element into the buffer. Here it would be preferable to use the version that overwrites the oldest value when full.
- Read an element at position X (without removing it from the buffer).
- Pop an element: reads the value of the oldest element (the tail) and removes it.
- Give the current number of samples stored.
- Print the content of the buffer.

Make sure your buffer implementation has all of these and add it to the project. Then let the sampling task add a new value to the buffer each time it computes a new magnitude.

But, wait, how long should the buffer be? This depends on how fast it is filled in and how fast it is emptied. You will be able to compute this detail later, so for now let's leave it very small (like 10), just for testing.

To test this last part, let the sampling task add a new element to the buffer and print out the content of the buffer. You should see the new values being added to the buffer and, when the buffer is full, the oldest positions being replaced by the new samples.

At the end of this section, I would expect your code to look like this:

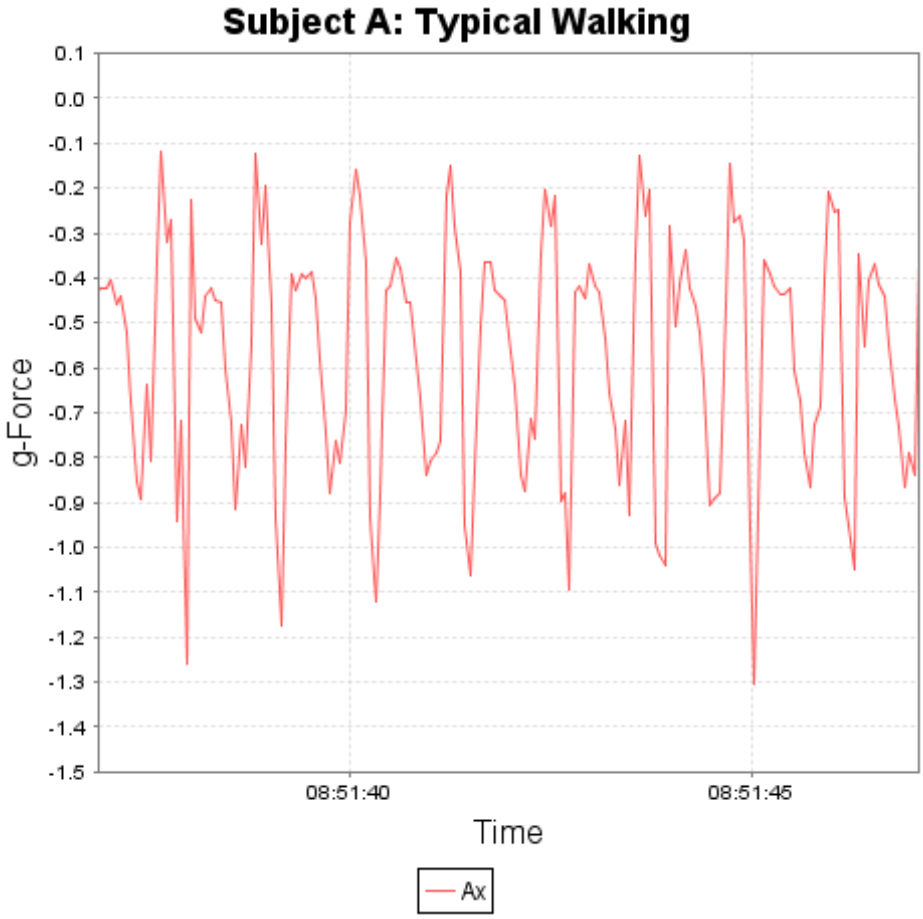
```
// the reason why I have chosen this sampling frequency is because .....
#define SAMPLING_PERIOD 500

#define BUFF_SIZE 10

buf_handle_t buffer;

static void sampling_task(void *arg)
{
    TickType_t xLastWakeTime = xTaskGetTickCount();

    while (1)
    {
        // get the acceleration

        // compute the magnitude

        // place the magnitude into the buffer

        // print the content of the magnitude

        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(SAMPLING_PERIOD));
    }
}

void app_main()
{
    // configure light sleep mode with esp_pm_configure()

    // initialise the I2C bus and the MPU6050
```

```
    // create sampling task
    xTaskCreate(sampling_task, "sampling", 2048, NULL, TASK_PRIORITY, NULL);
}
```

Computing the steps

Now that we can get a bunch of acceleration samples and save them onto a buffer, we need to find the steps in them. Remember, the idea is that we have a separate steps detection task that periodically analyses the buffer, computes the number of steps, and empties the buffer.

Detecting steps is far more complicated than it seems, but we are going to use a simple approach here. The accelerometer of a walking person looks like this:



It's basically a periodical signal: each time your heel hits the ground, it gives an acceleration to your body that counters the direction of the hit. So, a way for counting steps could be by detecting those "hits".

The algorithm could work this way:

1. When you run the step-detection task, you will have collected N samples, depending on both the sampling frequency and the frequency of your step-detection task.

2. On these samples compute the mean ($m = \frac{1}{N}\sum_{i=1}^{i=N} acc_i$) and the standard deviation ($sd = \sqrt{\frac{\sum_{i=1}^{i=N}(acc_i - m)^2}{N}}$) of the module of the acceleration vector.
3. Once you know the mean and the SD, then, for each sample, check which one is above the mean + the standard deviation. In other words: $acc_i > m + sd$.
4. Those samples above the threshold that are far apart at least a number of milliseconds d can be marked as steps and thus can increase a counter of the total number of steps (in other words, "debounce" your steps!).

This is a lot of condensed information. Re-read it carefully and think about how this can be implemented. You can even start drafting the code, but you'll need to decide the value of some constants before being able to proceed.

Once you have an idea, there are other things you need to balance and consider:

- How often do you run this algorithm? A short period would allow to keep the buffer small, but the standard deviation needs at least a couple of full periods of the signal to be significant. On the other hand, a long period would lead to more precise results but would require more buffer space and more computation. Do some considerations (and tests) and justify your choice. For example, you may decide to accumulate enough data into the buffer equivalent to about 10 steps at walking speed in it. Whatever the decision, write your reasoning as a comment in the code.
- Based on this period, you will know how big the buffer must be to accommodate the samples collected within that period. Leave some extra space as well, because the steps counting algorithm may take some time and more samples may be added while it is executing.
- I would keep the standard deviation always above a little value. This is because when you leave the accelerometer still, the SD will converge towards a very small amount, and the step-counting algorithm would be detecting steps that are, in reality, noise.
- You don't want to stop sampling when computing steps, so make sure you assign proper priorities to both tasks.
- There is one possible race condition: when the steps counter task needs to decide how many samples of the buffer to analyse, it can use the variable that indicates the head of the buffer. Make sure you only read this value <u>once</u> (read it and store in a local variable), because this variable is, simultaneously, also updated by the sampling task.
- To fine tune your algorithm, instead of detecting a step when: $acc_i > m + sd$ you can detect a step when : $acc_i > m + K \cdot sd$, where K needs to be chosen wisely. If you choose a small K, then you could detect steps when there are no actual steps. If K is too big, you may miss steps with a lower intensity. Choosing this parameter is crucial, but it's not easy and will need a bit of experimenting. A value between 1 and 3 is reasonable.
- Be careful when you "debounce" your steps: you are analysing samples stored in the past, not live! To understand the difference in time between two samples, just multiply the difference of their position in the buffer by the sampling rate.

To test your code, you can first just check what are the mean and the SD when the accelerometer is kept still. You should see the mean being almost constant (if you convert the

raw value to g, this should be exactly g) and the standard deviation to be very small. In my experiments I have noticed that a still accelerometer has a mean magnitude of 17000 and an SD of 100. With very little movement the SD increases to 300 and when walking I get an SD of about 3000. If you observe these values, you should be on the right track.

Then you could test the step counting. Stepping on your feet holding the accelerometer, print out the value of those magnitude samples that are above the threshold. This may help you identify if the threshold is too low or too high, and thus fine tune it. Try to compute the accuracy of your algorithm: for example, "walk" (staying still) 10 steps and see how many steps are actually counted. Write the accuracy (as percentage of detected steps over actual steps) in the comments.

In the end always remind that this a bad algorithm and it's unlikely that you will ever get a 100% accuracy. Do your best to fine tune it, but stop whenever you see your improvements get smaller and smaller. Remember that you're here to learn, not to win a competition!

> If you cannot walk (for any reason) try asking someone around to help you. If you don't have anybody who can help, then you can simulate walking moving the accelerometer up and down (in this case write it in the comments to let me know!).

At the end of this step, I expect your code to look like this:

```
// the reason why I have chosen this sampling frequency is because .....
#define SAMPLING_PERIOD ???

// I have chosen to run the algorithm every YYY because ....
#define ALGO_PERIOD ???

// number of samples to he held inside the buffer: this is because ...
#define BUFF_SIZE ???

// minimum SD to avoid converging to 0
#define MIN_SD ???

// constant applied to SD to detect steps
#define K ???

// minimum time between steps, this value is chosen because...
#define MIN_INTRA_STEP_TIME ???

buf_handle_t buffer;

int step_count = 0;

static void sampling_task(void *arg)
{
    TickType_t xLastWakeTime = xTaskGetTickCount();

    while (1)
    {
        // get the acceleration
```

```
        // compute the magnitude

        // place the magnitude into the buffer

        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(SAMPLING_PERIOD));
    }
}

static void algo_task(void *arg)
{
    TickType_t xLastWakeTime = xTaskGetTickCount();

    while (1)
    {
        // get size of the buffer

        if (size > 0)
        {
            // compute mean, here do NOT empty the queue when reading it!

            // compute SD, here queue must not be emptied yet

            if (sd < MIN_SD)
                sd = MIN_SD;

            // now do the step counting, while also emptying the queue
            int lastStepTS = -MIN_INTRA_STEP_TIME;

            for (int i = 0; i < size; i++)
            {
                // get sample, removing it from queue

                // if sample > mean + K * sd
                // AND if time between last step and this sample is >
MIN_INTRA_STEP_TIME
                // step found! step_count++;
            }
        }
        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(ALGO_PERIOD));
    }
}

void app_main()
{
    // configure light sleep mode with esp_pm_configure()

    // initialise the I2C bus and the MPU6050

    // create sampling task
    xTaskCreate(sampling_task, "sampling", 2048, NULL, SAMPLING_PRIORITY, NULL);
```

```
    xTaskCreate(algo_task, "algo", 2048, NULL, ALGO_PRIORITY, NULL);
}
```

Putting it all together

If you have got so far, the rest will be easy. Now we need to add the user interface aspect of it. First, we need to configure two pins, one as an output, to switch on the LED and another as an input. The use case goes like this: if the user presses the button, the LED will blink with a different sequence depending on if a certain number of steps has been reached or not.

To make things more interesting, let's use FreeRTOS tasks and semaphores. We need to add a task that, when activated, checks the value of the counter and flashes the LED for, say, 2 seconds. To stop and activate the task, we will use a semaphore. When the button is pressed, the ISR of the button will "give" the semaphore and unblock the task. When the task has completed, it will "take" the semaphore and stop again. Check the code examples given in the lecture about FreeRTOS, you'll find everything you need there.

And this is how the code looks like at the end:

```
// the reason why I have chosen this sampling frequency is because .....
#define SAMPLING_PERIOD ? ? ?

// I have chosen to run the algorithm every YYY because ....
#define ALGO_PERIOD ? ? ?

// number of samples to he held inside the buffer: this is because ...
#define BUFF_SIZE ? ? ?

// minimum SD to avoid converging to 0
#define MIN_SD ? ? ?

// constant applied to SD to detect steps
#define K ? ? ?

// minimum time between steps, this value is chosen because...
#define MIN_INTRA_STEP_TIME ? ? ?

// daily goal of steps
#define STEPS_GOAL ? ? ?

buf_handle_t buffer;

int step_count = 0;

SemaphoreHandle_t xSemaphore = NULL;

static void sampling_task(void *arg)
{
    TickType_t xLastWakeTime = xTaskGetTickCount();
```

```
    while (1)
    {
        // get the acceleration

        // compute the magnitude

        // place the magnitude into the buffer

        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(SAMPLING_PERIOD));
    }
}

static void algo_task(void *arg)
{
    TickType_t xLastWakeTime = xTaskGetTickCount();

    while (1)
    {
        // get size of the buffer

        if (size > 0)
        {
            // compute mean, here do NOT empty the queue when reading it!

            // compute SD, here queue must not be emptied yet

            if (sd < MIN_SD)
                sd = MIN_SD;

            // now do the step counting, while also emptying the queue
            int lastStepTS = -MIN_INTRA_STEP_TIME;

            for (int i = 0; i < size; i++)
            {
                // get sample, removing it from queue

                // if sample > mean + K * sd
                // AND if time between last step and this sample is >
MIN_INTRA_STEP_TIME
                // step found! step_count++;
            }
        }
        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(ALGO_PERIOD));
    }
}

// button pressed ISR
void button_isr_handler(void *arg)
{
    // give semaphore (pay attention: this is in an ISR!)
}
```

```
void led_task(void *arg)
{
    while (1)
    {
        // wait for semaphore
        if (xSemaphoreTake(xSemaphore, portMAX_DELAY) == pdTRUE)
        {
            // flash LED with sequence depending on if step_count > STEPS_GOAL
        }
    }
}

void app_main()
{
    // configure light sleep mode with esp_pm_configure()

    // initialise the I2C bus and the MPU6050

    // create sampling task
    xTaskCreate(sampling_task, "sampling", 2048, NULL, SAMPLING_PRIORITY, NULL);
    xTaskCreate(algo_task, "algo", 2048, NULL, ALGO_PRIORITY, NULL);
}
```

If your code looks differently, but does what is supposed to do, then perfect. An important learning outcome of this course is that you are able to organise the code in a way that makes sense.

Once you have completed, upload your code to Canvas. Don't forget to add a few comments to tell us how you have chosen some constants (see comments in the example code above).

I know, this was quite a journey. But if you have completed this you should feel proud of yourself. By the way, the next time you check the steps on your smartwatch, think about all the work that is needed to develop it. Maybe you can even imagine those poor developers spending hours working on the product you are wearing without knowing all the effort they put into it!

When you are done with this exercise, please reconfigure your board and **remove the power management mode** or it will interfere with your next lab.

If you have time

As always, clean code, comments and nicely designed modules help. The device you have built up to here is functional but somewhat basic. Here are some ideas for improving it.

- When counting steps, computing the mean takes one for loop of all the stored samples. Computing the standard deviation takes another for loop. Identifying the

steps after that takes another for loop. Can you do better than that? There are formulas to compute mean and variance incrementally.

- What scale do you actually need when using the accelerometer? Do you need the full 16g scale, or is a 2g scale enough? Configure the accelerometer accordingly.
- You can put the accelerometer in sleep mode when not sampling, see registers PWR_MGMT_1 and PWR_MGMT_2, for this.
- You can rely on the accelerometer's sampling instead of using the CPU. Each time a new sample is available an interrupt can be raised on the interrupt pin (which you need to plug to the ESP32 and handle with an ISR). See interrupt registers INT_PIN_CFG, INT_ENABLE and INT_STATUS of the MPU6050.
- (PRO tip!) You read the data from the accelerometer every N samples. The accelerometer can store its samples in a FIFO queue and the ESP could read the queue and empty it every now and then: this would allow you to keep the ESP in light sleep longer. Notice, though, that the queue has only 1024bytes (with each acceleration sample being 2 bytes x 3 axes) and that the minimum sampling frequency of the accelerometer is about 30Hz. If you want to explore this option, check the FIFO registers FIFO_EN, FIFO_COUNT end FIFO_R_W of the MPU6050.
- Instead of using periodic tasks, you could use a (more reliable) hardware timer and let the timer "wake up" the task that collects data using a FreeRTOS task notification. So, the data collection task would go to sleep (xTaskNotifyWait) and the timer would regularly wake it up: when the task wakes up, it gets the samples, computes the magnitude, stores it into the buffer and go back to sleep again.
- You can also use the FreeRTOS queues instead of your buffer, they have the advantage of being able to block a process. For example, you could make the algorithm task wait for a new sample in the queue, but only start processing it when N have been accumulated.
- Wanna try a "real" step counting algorithm? Check this one: https://oxford-step-counter.github.io/ there's also a C implementation. Be aware that it needs a bit of tweaking in order to work.