Rapport du projet d'Internet of Things

Groupe 10

Nathalie RALIJAONA

Armelle RIVIERE

Mélanie ROUDE

Pour le 15/05/18

Mr Aomar Osmani

Mr Massinissa Hamidi

Promo ISCE 2019

Table des matières

Matériel

ESP 32

C'est une série de système à faible coût. C'est un micro – contrôleur avec une puce Wi FI et Bluetooth. Ceci inclut également des commutateurs d'antenne intégrés, un balun RF, un amplificateur de puissance, un amplificateur de réception à faible bruit, des filtres et des modules de gestion de l'alimentation. Celui-ci comporte également une mémoire flash de 4 MB. Grâce à des fonctions d'économie d'énergie, il consomme une énergie extrêmement faible, par exemple la synchronisation de l'horloge à haute résolution.

Ces cartes de développement sont très souvent utilisées pour le développement d'objet connecté et se démocratise de plus en plus dans la création d'objet DIY.

BME 280

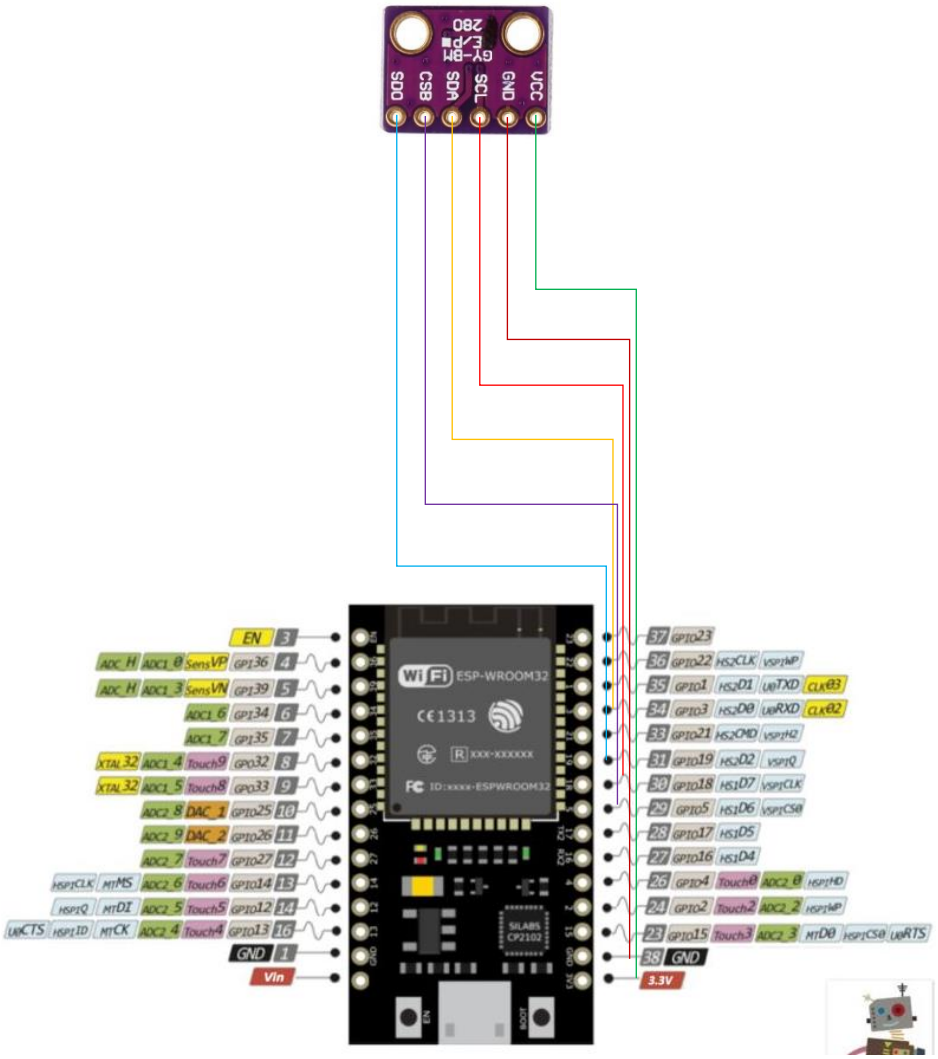
Un ESP 32 est un micro – contrôleur à puce permettant une connexion Wi Fi et Bluetooth. Pour la connexion il contient 4 SPI :

- SPIO : il est utilisé pour la mémoire flash, sa connexion avec la mémoire permet le transfert d'information.
- SPI1 : il est également utilisé pour écrire sur la puce flash différentes informations
- HSPI et VSPI sont libres d'utilisations.

Ce micro – contrôleur comporte également un capteur de température, qui nous sera utile pour l'ensemble du TP

Connexion

Ces 2 composants doivent être liés par la connectique. Les connexions sur les ports de l'ESP32 sont importantes car dans la programmation le rôle des bus du BME 280 sont liés aux numéros des ports de l'ESP 32. Voici donc les liaisons effectuées entre ces deux composants :



Projet

Généralité

Fichiers

Notre projet est composé de différent fichier .c chacun est utiliser dans un cadre spécifique.

Le fichier *app_layer.c* n'est pas utiliser dans le stade où nous en sommes mais aurait été utiliser lors des échange http.

Le fichier *connectivity.c* permet de débuter la connectivité du Wi – FI au sein du projet.

Le fichier *formatting.c* permet de récupérer l'ensemble des données captées par le BME 280 pour les formater de manière à ce que celles – ci soient transportable de manière rapide et sécuriser.

Le fichier *main.c* comprend la fonction maîtresse du programme, donc de lancer la connexion WI – Fi, de récupérer les données atmosphériques et d'échanger les données par la suite.

Le fichier *net_layer.c* montre dans un premier temps les paramètres de la connexion effectuer avec le réseau Wi- Fi. Il aurait été utiliser par la suite lors des échanges avec la page web.

Le fichier *phy_layer.c* permet de débuter les connexions Wi – Fi ou Bluetooth.

Le fichier *sensor_reading.c* permet d'initialiser le BME 280 ainsi que les bus SPI associé. C'est également ici que les fonctions pour récupérer l'ensemble des données atmosphériques sont implémenter.

Le fichier *trans_layer.c* n'a pas été utiliser pour les fonctions dont nous avions besoin. Il était présent pour les échanges des données vers internet.

Le fichier *transmission.c* comprend différentes fonctions pré – paramétré pour permettre l'affichage et l'échange d'informations au sein de la queue.

Librairies utilisées

Le projet est codé en C. Nous importons donc les librairies usuelles comme *string.h*, *stdio.h*, *stdint.h* et *stdlib.h*. La librairie *errno.h* permet de retourner les codes erreurs.

Pour permettre l'utilisation du BME 280 nous incluons *bme280.h*.

Les headers provenant de FreeRTOS permettent de gérer les interactions avec le kernel de l'ESP 32.

Pour l'implémentation de JSON nous importons *cJSON.h*.

La connexion de l'ESP 32 avec le Wi – Fi requiert les librairies *esp_wifi.h* et *esp_event_loop.h*.

Répartition du travail

Le projet se compose en 3 grandes parties. Nous avons divisé la charge de travail de manière équitable entre nous 3. Armelle et Nathalie ont instancié la connexion WI -FI. Pour le capteur de température cela a été réfléchit par Mélanie et Nathalie. La transmission des informations a principalement été codée par Armelle et Mélanie. Toutes trois avons fait énormément de recherche et étudié l'ensemble de la documentation. Que ce soit pour comprendre le fonctionnement de l'ESP ou du capteur ; mais également pour comprendre comment implémenter le wifi, la température, ou encore de passer d'un JSON à un string.

La vidéo a été faite par Armelle et Nathalie. Tandis que Mélanie a écrit le rapport.

Connexion au Wi-Fi

Fonctions

L'ensemble des fonctions sont situés dans le fichier *phy_layer.c* .

Au tout début de ce fichier, juste après les déclarations des headers nous pouvons voir ceci :

```
#define WIFI_SSID   "Honor 8" /
#define WIFI_PASS   "armoule23"
```

Cela définit le nom de réseau Wi – Fi et le mot de passe associé à celui – ci. C'est ce qui permettra par la suite de relier e BME 280 au réseau.

*static esp_err_t event_handler(void *ctx, system_event_t *event)*

Par la suite, nous avons cette première fonction :

```
static esp_err_t event_handler(void *ctx, system_event_t *event){
    switch(event->event_id) {

        case SYSTEM_EVENT_STA_START:
            //xTaskCreate(smartconfig_example_task, "smartconfig_example_task", 4096, NULL, 3, NULL);
            esp_wifi_connect();
            break;

        case SYSTEM_EVENT_STA_GOT_IP:
            xEventGroupSetBits(wifi_event_group, IPv4_CONNECTED_BIT);
            break;

        case SYSTEM_EVENT_AP_STA_GOT_IP6:
            xEventGroupSetBits(wifi_event_group, IPv6_CONNECTED_BIT);
            break;

        case SYSTEM_EVENT_STA_DISCONNECTED:
            esp_wifi_connect();
            xEventGroupClearBits(wifi_event_group, IPv4_CONNECTED_BIT);
            xEventGroupClearBits(wifi_event_group, IPv6_CONNECTED_BIT);
            break;

        default:
            break;
    }

    return ESP_OK;
}
```

Le switch permet lors de la connexion d'envisager les différentes possibilités.
Le premier cas est utilisé lors du début de la connexion pour créer le lien.
Le deuxième et troisième cas sont pour respectivement les protocoles IPv4 ou IPv6, lors de l'obtention d'une adresse IP.
Enfin le dernier cas, est lors d'une déconnexion, une reconnexion est immédiatement retentée, ainsi qu'une remise à niveau des bits concernant les protocoles IP.

La fonction *event_handler()* permet donc d'initialiser la connexion Wi – Fi et de gérer les protocoles IP.

int8_t _wifi_init()

```
int8_t _wifi_init(){
    int8_t rc = ESP_OK;

    wifi_event_group = xEventGroupCreate();

    esp_event_loop_init(event_handler, NULL);
    if (rc < 0) {
    }

    wifi_init_config_t init_config = WIFI_INIT_CONFIG_DEFAULT();

    esp_wifi_init(&init_config);
    if (rc < 0) {
    }
    esp_wifi_set_storage(WIFI_STORAGE_RAM);
    if (rc < 0) {
    }
    esp_wifi_set_mode(WIFI_MODE_STA);
    if (rc < 0) {
    }

    wifi_config_t config = {
        .sta={
            .ssid = WIFI_SSID,
            .password = WIFI_PASS
        }
    };
    esp_wifi_set_config(ESP_IF_WIFI_STA, &config);

    return rc;
}
```

L'ensemble des fonctions sont utilisées pour configurer le réseau Wi Fi de l'ESP 32.

La configuration même du Wi Fi se fait dans la structure : *wifi_config_t config{}.* C'est ici que sont renseigné le nom du réseau et son mot de passe définit au début du fichier.

Retour sur console

Lors de la compilation nous obtenons alors ceci :

```
[IoT-Labs] connected!
[IoT-Labs] IP Address:  192.168.43.38
[IoT-Labs] Subnet mask: 255.255.255.0
[IoT-Labs] Gateway:     192.168.43.1
[setup_connectivity] starting transport layer
[setup_connectivity] starting application layer
[setup_connectivity] END
```

```
I (5925) wifi: state: auth -> init (2)
I (5925) wifi: n:11 0, o:11 0, ap:255 255, sta:11 0, prof:1
I (6045) wifi: n:11 0, o:11 0, ap:255 255, sta:11 0, prof:1
I (6045) wifi: state: init -> auth (b0)
I (6055) wifi: state: auth -> assoc (0)
I (6065) wifi: state: assoc -> run (10)
I (6085) wifi: connected with Honor 8, channel 11
```

Nous pouvons alors observer l'état de connexion au réseau puis identifier l'ensemble des informations du réseau sur lequel l'ESP 32 est connecté.

Capteur de température

Fonction

L'ensemble des fonctions se situent dans le fichier sensor_reading.c. Auparavant dans le fichier les fonctions certaines fonctions sont implémenter. Elles permettent de :

- Initialiser les bus SPI ;
- Initialiser le BME 280 ;
- Créer les fonctions de lectures et d'écritures sur les SPI.

Les ports des branchements à l'ESP 32 sont définit au début de ce fichier. Ils sont extrêmement importants pour paramétrer les échanges entre les 2 composants.

```
#define PIN_NUM_MISO 19
#define PIN_NUM_MOSI 3
#define PIN_NUM_CLK  18
#define PIN_NUM_CS   5
```

*void setup_sensors(void *pvParameters)*

```
void setup_sensors(void *pvParameters){
    // TODO haint8_t _perform_sensor_readings(struct bme280_dev *dev, struct b
    // TODO perform, then, sensor fusion, etc.

    int8_t rc;
    struct sensors_config_t *cfg;
    cfg = (struct sensors_config_t*) pvParameters;

    // TODO iterate over provided devices, in case we implement multiple
    //      sensors handling!
    switch (cfg->device) {
        case BME280:
        rc = ENOSYS;
            rc = initialize_spi_sensor(); //nous
            if (rc < 0) {
                printf("[IoT-Labs] Error while initializing spi bus\n");
                break;
            }

            struct bme280_dev dev;
            rc = initialize_bme_device(&dev); //nous
            if (rc < 0) {
                printf("[IoT-Labs] Error while initializing bme280\n");
                break;
            }

            /* each time a sensor is setup, trigger a sensor_readings task */
            xTaskCreate(&perform_sensor_readings, "bme280_readings", 2048,
                (void*) &dev, 1, NULL);

            break;

            case DHT11:
            rc = ENOSYS;
            break;

            default:
            rc = ENOSYS;
            break;
        }

        vTaskDelete(NULL);
    }
```

Cette fonction permet l'initialisation du BME 280 et de ses bus. Elle est essentielle pour l'ensemble de la communication et connaître les paramètres atmosphériques.

*int8_t _perform_sensor_readings(struct bme280_dev *dev, struct bme280_data *data)*

```
int8_t _perform_sensor_readings(struct bme280_dev *dev, struct bme280_data *data){
    // TODO failwith "Students, this is your job!"
    int8_t rc;

//on recupre les data dans rc
    rc = bme280_get_sensor_data(BME280_ALL, data, dev); //nous

    /* debug */
    printf("temperature sensor_readings = %f DegC\n", (float)data->temperature/100);
    printf("humidity  sensor_readings = %f %%RH\n", (float)data->humidity/1024);
    printf("pressure  sensor_readings = %f Pa\n", (float)data->pressure/256);

    return rc;
}
```

La fonction *bme280_get_sensor_data()* permet de se connecter au BME280 et d'y renseigner la pression, la température et l'humidité capté par les capteurs. L'ensemble des données sont calculées et stockées dans la structure du BME 280.

Par la suite, nous affichons l'ensemble des données.

*void perform_sensor_readings(void *pvParameters)*

```
void perform_sensor_readings(void *pvParameters){
    int8_t rc;
    struct bme280_dev *dev;
    struct bme280_data *data;

    dev = (struct bme280_dev*) pvParameters;
    data = (struct bme280_data*) malloc(sizeof(struct bme280_data));

    while (1) {
        /* get sensor reading */
        rc = _perform_sensor_readings(dev, data); //nous

        /* construct adequate data structure in order to encapsulate sensor reading*/
        struct a_reading reading;
       rc = make_a_reading(&reading, data);//nous

        /* debug */
        //print_a_reading(&reading);

        /* save obtained reading in transmission queue */
        rc = transmission_enqueue(&reading);//nous
        if (rc < 0) {
            // No need to break here! or TODO break after a given number of trials
            printf("[IoT-Lab] Error while enqueuing a reading for transmission\n");
            goto end;
        }

        vTaskDelay(1000/portTICK_RATE_MS);
    }

end:
    vTaskDelete(NULL);
}
```

Cette fonction permet d'initialiser le BME 280 et ses connexions, en commençant par le *_perform_sensor_reading()*. L'appel de la fonction *make_a_reading()* permet l'encapsulation des données et la vérification de cette encapsulation. L'utilisation de *transmission_enqueue()* code l'envoie des informations dans une queue dans la structure, ce qui permettra par la suite de les utiliser facilement pour d'autres action.

Dans *transmission.c* nous retrouvons les fonctions appelées ci – dessus.

```c
int8_t make_a_reading(struct a_reading *reading, struct bme280_data *data){
    int8_t rc = 0;

    reading->timestamp = xTaskGetTickCount();
    memcpy(&reading->data, data, sizeof(*data));

    /* debug */
    printf("[transmission / make_a_reading] reading->timestamp = %d\n", reading->timestamp);
    printf("[transmission / make_a_reading] reading->data->temperature = %d\n", reading->data.temperature);

    return rc;
}
```

```c
int8_t transmission_dequeue(struct a_reading *reading){
    int8_t rc;

    rc = xQueueReceive(sensor_readings_queue, reading, portMAX_DELAY);
    if (rc != pdTRUE) {
        printf("[IoT-Labs] Error while dequeuing a reading\n");
        return rc;
    }

    return rc;
}
```

Retour sur console

Lors de l'appel de l'ensemble de ces fonctions nous obtenons alors ceci :

```
temperature = 18.280001 DegC
humidity = 100.000000 %RH
pressure = 42968.750000 Pa
```

La température semble cohérente, les autres données le sont moins. Cependant, le paramétrage étant fait par une fonction universelle nous suspectons le capteur d'envoyer des données erronées.

Pour vérifier que le capteur fonctionnait correctement, nous l'avons placé dans un réfrigérateur quelques instants. Nous avons pu observer un changement de température conséquent, les données semblent incorrectes au vu de la température afficher, mais un changement notable est présent.

```
[sensor_readings] temperature = -40.000000 DegC
[sensor_readings] humidity  = 0.000000 RH
[sensor_readings] pressure = 51.373341 hPa
```

Transmission des informations

Fonction

*int8_t json_of_reading(cJSON *json, const struct a_reading *reading)*

```c
int8_t json_of_reading(cJSON *json, const struct a_reading *reading){
    int8_t rc = 0;
    cJSON *timestamp;
    cJSON *bme280_reading;

    printf("[json_of_reading] ########################\n");
    print_a_reading(reading);

    /* create root node */
    json = cJSON_CreateObject();

    timestamp = cJSON_CreateNumber(reading->timestamp);
    cJSON_AddItemToObject(json, "timestamp", timestamp);
    bme280_reading = cJSON_CreateObject();

    if (cJSON_AddNumberToObject(bme280_reading, "temperature",reading->data.temperature/100) == NULL) {
        goto end;
    }

    if (cJSON_AddNumberToObject(bme280_reading, "pressure", reading->data.pressure) == NULL) {
        goto end;
    }

    if (cJSON_AddNumberToObject(bme280_reading, "humidity", reading->data.humidity/1024) == NULL) {
        goto end;
    }
    cJSON_AddItemToObject(json,"bme",bme280_reading);

    /* TODO failwith "Students, this is your job!" */

    /* debug */
    printf("%s\n", cJSON_Print(json));

    return rc;

    end:
    cJSON_Delete(json);
    return -1;
}
```

Cette fonction permet de transferer les données sauvegardées dans le BME280 dans du JSON.

Nous commencont par utiliser la fonction *cJSON_CreateNumber()* qui permet de convertir la donnée stockée dans le BME280 en un format approprié pour le JSON. Par la suitee l'appel de *cJSON_AddItemToObject()* permet de stockée l'objet devenu constant par la conversion dans le JSON.

Pour simplifier cette transaction, nous utiliserons par la suite la fonction *cJSON_AddNumberToObect()* qui permet de combiner les 2 appels en un seul pour le même résultat.

*int8_t json_of_readings(cJSON *json, const struct a_reading **readings)*

```
int8_t json_of_readings(cJSON *json, const struct a_reading **readings){
    int8_t rc = 0;
    print_a_reading(&readings); //nous
    return rc;
}
```

Cette fonction est utilisée lorsque nous avons plusieurs capteurs en même temps, ce qui n'est pas notre cas pour le moment. La fonction est incomplète pour la gestion de plusieurs entrées en parallèle.

*int8_t string_of_json(char *buff, cJSON *json)*

```
int8_t string_of_json(char *buff, cJSON *json){
    int8_t rc=0;
    memcpy(buff,json,sizeof(*json));//nous
    return rc;
}
```

```
int8_t string_of_json(char *buff, cJSON *json){
    int8_t rc;
    buff = cJSON_Print(json); //nous
    if (buff == NULL) {
        rc = -1;
    }
    return rc;
}
```

Cette fonction nous permet de passer de JSON en string. Nous cherchons donc à transformer la structure du JSON en un string avec l 'ensemble des informations.

Nous avons essayé plusieurs manières mais aucune ne fonctionnait. Le *memcpy* renvoie une chaîne de caractère spéciaux illisibles.
L'essai avec *cJSON_PRINT()* nous renvoyait un buffer vide.

Suite à un dysfonctionnement de notre capteur, nous n'avons pas pu continuer les recherches.

*void perform_transmissions(void *pvParameters)*

```
void perform_transmissions(void *pvParameters){
    int8_t rc;
    struct a_reading reading;
    char representation[REPR_MAX];
    //cJSON json; // FIXME very bad! very, very bad! make code modular! get rid of porosity

    // TODO make this function wait for the connection event
    while (1) {
        /* get a reading from transmission queue */
        /* debug */
       // print_a_reading(&reading);

        // format the dequeued reading and get the string representation corresponding to it
        rc = transmission_dequeue(&reading); //nous

        /* send the formatted reading */
        rc = reading_formatting(representation, &reading); //nous
    }
    vTaskDelete(NULL);
}
```

Ici nous formattons l'ensemble des données pour pouvoir effectuer une lecture des données atmosphérique en queue.

La fonction *transmission_dequeue()* permet de récupérer chaque donnée stockée lors de la réception des données atmosphériques. Enfin, *reading_formating()* permet de créer un JSON puis de le mettre en format texte.

*int8_t reading_formatting(char *representation, const struct a_reading *reading)*

```
int8_t reading_formatting(char *representation, const struct a_reading *reading){
    int8_t rc;
    cJSON json;

    // make a json object from dequeued reading
    rc = json_of_reading(&json, reading); // passe des datas en json//nous
    if (rc < 0) {
        printf("[IoT-Labs] Error while making json from a reading\n");
        return rc;
    }

    // get string representation of json object
    rc = string_of_json(representation, &json); //passe de json en string //nous
    if (rc < 0) {
        printf("[IoT-Labs] Error while getting string repr. from json\n");
        return rc;
    }

    //rc = ENOSYS;

    return rc;
}
```

Comme dit ci – dessus, cette fonction permet dans un premier temps de créer une structure JSON de l'ensemble des données atmosphériques. Puis de les traduire en un texte dans un string. L'objectif de cette transformation est de pouvoir l'envoyer avec un HTPP request par la suite.

Retour sur console

Nous pouvons voir ici la structure du JSON avec l'enregistrement des données du capteur de température. Nous observons le transfert de donnée des informations atmosphériques est correctement effectué entre le BME 280 et la structure JSON.

```
[sensor_readings] temperature = 18.320000 DegC
[sensor_readings] humidity  = 100.000000 RH
[sensor_readings] pressure = 110.000000 hPa
[make_a_reading] reading->timestamp transmission = 408
[make_a_reading] reading->data->temperature transmission = 1832
[json_of_reading] ############################
{
        "timestamp":    408,
        "bme":  {
                "temperature":  18,
                "pressure":     11000000,
                "humidity":     100
        }
}
```

Conclusion

Ce projet nous a permis de manipuler ces nouveaux outils et d'enfin pouvoir effectuer un travail concret pour la première fois de notre vie d'étudiantes du supérieur. A cause de problème matériel nous n'avons pas pu poursuivre le projet. Nous avons cependant beaucoup appris par ce projet. Le plus gros du travail a été un travail de recherche. Cela a été un travail très fastidieux, avec souvent peu de résultat étant donné qu'il n'y a pas beaucoup de documentation et que nous n'utilisions pas Arduino. Mais nos efforts n'ont pas été vain puisque nous pouvons vous présenter notre travail.

Il était très intéressant de découvrir le fonctionnement de cette technologie qui est de plus en plus répandu. Nous avons été étonnées de voir qu'avec un composant de la taille du BME280 nous pouvions récupérer autant d'informations aussi rapidement et facilement.