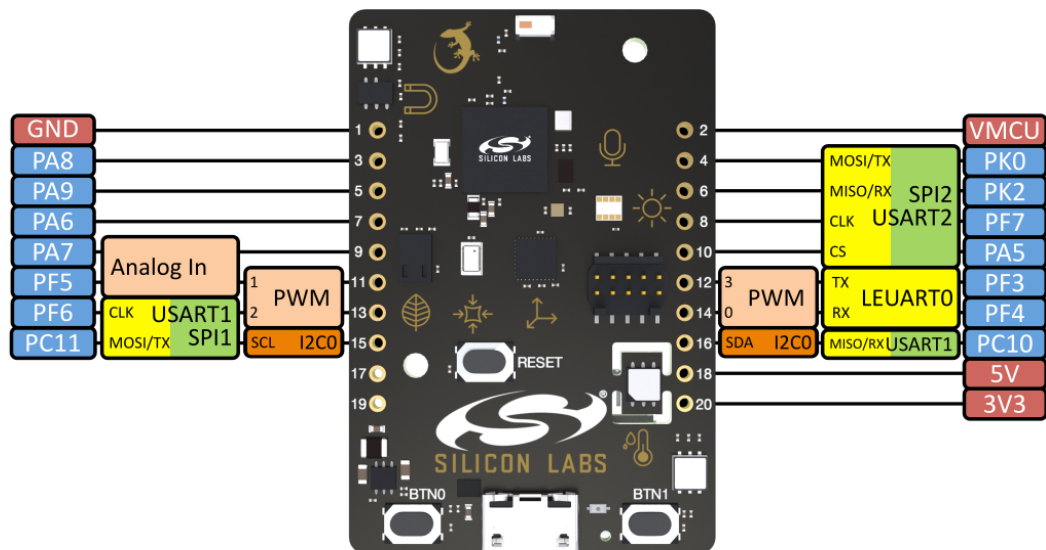


4E-IA3: IoT TP 2



SLTB004A Thunderboard Sense 2



Pinout du Thunderboard Sense 2 de Silicon Labs (source image: os.mbed.com)

NB: le repository du projet est disponible ici: https://github.com/avmolaei/repo_iot

1. PRÉREQUIS & MODALITES

N/A

2. CRÉATION DU PROJET

Cette étape étant triviale (suivi d'étapes succinctes d'un énoncé), nous n'allons pas la décrire en détail. Il en est de même pour la création du repository github (disponible ici: https://github.com/avmolaei/repo_iot) et la configuration du projet et des GATT.

3. RAJOUTER DES FONCTIONNALITÉS BLE & LOGICIELLES

IDEM

4. TESTER LES MODIFICATIONS GATT & LOGICIELLES

IDEM

5. RETOURNER UNE VALEUR CARACTÉRISTIQUE

Initialiser et dé-initialisation le capteur de température

- 1) Les APIs `sl_sensor_rht_init()` et `sl_sensor_rht_deinit()` permettent d'initialiser le capteur une fois la connexion établie ou coupée (elles sont appelées quand ces événements surviennent).

```
// This event indicates that a new connection was opened.
case sl_bt_evt_connection_opened_id:
    sl_sensor_rht_init();
    app_log_info("%s: connexion ouverte!\n", FUNCTION);

    break;

// -----
// This event indicates that a connection was closed.
case sl_bt_evt_connection_closed_id:
    sl_sensor_rht_deinit();
    app_log_info("%s: connexion fermee!\n", FUNCTION);
```

```
[I] app_init
[I] app_init
[I] app_init
[I] app_init
[I] app_init
[I] sl_bt_on_event: connexion ouverte!
[I] sl_bt_on_event: connexion fermee!
```

- 2) On crée la fonction `get_temp` afin de lire la température du capteur idoine avec `sl_sensor_rht_get()`:

```
case sl_bt_evt_gatt_server_user_read_request_id:
    tempValue = get_temp();
    if(evt->data.evt_gatt_server_user_read_request.characteristic == gattdb_temperature){
        app_log_info("%s: reading temp and humidity...\n", FUNCTION);
        app_log_info("%sTemperature: %d degC\n", FUNCTION, tempValue);
        app_assert_status(sl_bt_gatt_server_send_user_read_response(evt->data.evt_gatt_server_user_read_request.connection,
                                                                    gattdb_temperature,
                                                                    0,
                                                                    sizeof(tempValue),
                                                                    (uint8_t*)&tempValue,
                                                                    NULL));
    }
}
```

Voici le corps de la fonction `get_temp()`:

```
long get_temp(){
    int32_t rawVal =0;
    int32_t degC =0;
    uint32_t rh=0;
    //sl_status_t sc;

    app_assert_status(sl_sensor_rht_get(&rh, &rawVal));
    app_log_info("%sRawval: %d \n", FUNCTION, rawVal);
    degC = (rawVal * 1 *0.01*1)*10;
    app_log_info("%sTemperature: %d degC\n", FUNCTION, degC/100);
    return (long)degC;
}
```

On initialise nos variables à 0 puis on calcule la température que l'on affichera et calcule grâce à la rawValue obtenue par le capteur.

- 3) La valeur arrive sous forme de int32 pour la température et en uint32 pour l'humidité.

Formater la température selon le format BLE

- 4) Dans la documentation BLE, on trouve le calcul suivant:

2.1 Scalar values

When a characteristic field represents a scalar value and unless otherwise specified by the characteristic definition, the represented value is related to the raw value by the following equations, where the M coefficient, d, and b exponents are defined per field of characteristic:

$$R = C * M * 10^d * 2^b$$

Where:

R = represented value

C = raw value

M = multiplier, positive or negative integer (between -10 and +10)

d = decimal exponent, positive or negative integer

b = binary exponent, positive or negative integer

The default values are: M = 1, d = 0 and b = 0.

C est la température au format BLE, et R la represented value.

Ainsi, avec M = 1, d = -2, et b = 0, on a R=0,01.C

Il y a un rapport 100 entre la température en degrés Celsius et la température au format BLE.

- 5) Avec quelques log, on obtient:

```
[I] sl_bt_on_event: connexion ouverte!
[I] sl_bt_on_event condition check!!!
[I] characteristic access: temperature
[I] status_flags: 215273[I] get_tempRawval: 28965
[I] get_tempTemperature: 28 degC
```

- 6) On peut par exemple utiliser des librairies .h mais aussi grâce à des variables globales externes.
- 7) La seconde semble la manière la plus appropriée dans notre cas.
- 8) Pour isoler la portée des variables à un seul fichier, on peut utiliser le mot clé "static". Sinon, on peut envisager l'utilisation de structures.

Vérifier que l'accès en lecture concerne bien la caractéristique température

9) L'identifiant de la caractéristique température est 21.

```
#define gattdb_manufacturer_name_string 18
#define gattdb_system_id 18
#define gattdb_env_sensing 19
#define gattdb_temperature 21
#define gattdb_ota 22
```

10) La condition suivante permet de tester si on lit bien la caractéristique température.
On vérifie simplement si le champ characteristic de l'événement lu est égal à gattdb_temperature.

```
////////////////////////////////////
case sl_bt_evt_gatt_server_user_read_request_id:
    if(evt->data.evt_gatt_server_user_read_request.characteristic == gattdb_temperature){
        app_log_info("%s: PASSAGE DANS LA CONDITION!\n", FUNCTION );
        get_temp();
    }
break;
```

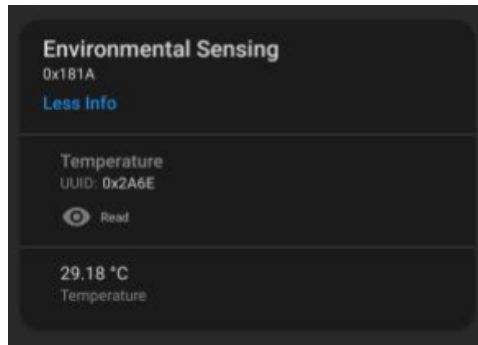
11) Sur la console:

```
[I] app_init
[I] sl_bt_on_event: connexion ouverte!
[I] sl_bt_on_event: PASSAGE DANS LA CONDITION!\n[I]
```

6. LIRE LA TEMPÉRATURE AVEC NOTIFICATION

Renvoyer ladite température

- 12) La température affichée sur l'application correspond bien à la température lue par le capteur.

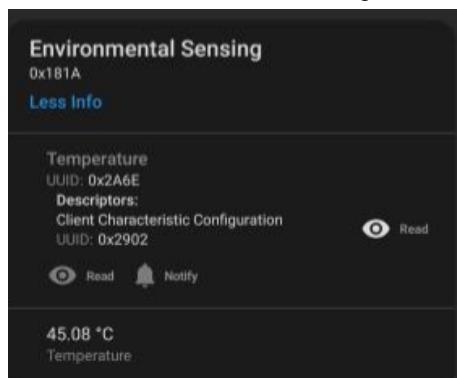


```
[I] sl_bt_on_eventTemperature: 29 degC
[I] get_tempRawval: 29834
[I] get_tempTemperature: 29 degC
[I] sl_bt_on_event: reading temp and humidity....
[I] sl_bt_on_eventTemperature: 29 degC
```

- 13) On peut prouver le bon fonctionnement du capteur et du code: en le plaçant à l'intérieur d'un ordinateur haute performance (Ryzen 7 3700x, RTX 3070), et lancer un jeu très gourmand en ressources comme Cyberpunk 2077; on atteint des températures d'air ambiant de presque 46°C:



la thunderboard Sense 2, à proximité du GPU et du ventilad CPU. Certes, il se peut que l'auteur de cette photo soit légèrement un fanboy de la marque Noctua





```
[I] sl_bt_on_eventTemperature: 4508 degC
[I] get_tempRawval: 45085
[I] get_tempTemperature: 45 degC
[I] sl_bt_on_event: reading temp and humidity....
[I] sl_bt_on_eventTemperature: 4508 degC
```

A l'inverse, on peut le placer au frigo, à côté du camembert et du pâté Hénaff. En attendant quelques minutes, on peut lire une température ambiante d'environ 10°C:



Temperature
UUID: 0x2A6E
Descriptors:
Client Characteristic Configuration
UUID: 0x2902

 Read  Notify

10.22 °C
Temperature

```
get_tempRawVal: 10228  
get_tempTemperature: 10 degC  
sl_bt_on_event: reading temp and humidty....  
sl_bt_on_eventTemperature: 1022 degC
```

Vérifier que le paramètre Notify de la caractéristique est bien pris en compte

14) On rentre bien dans ce case:

```
//obtention des événements sur le CCCD
case sl_bt_evt_gatt_server_characteristic_status_id:
    app_log_info("%s condition check!!!\n", FUNCTION);
    if(evt->data.evt_gatt_server_user_read_request.characteristic == gattdb_temperature){
        app_log_info("characteristic access: temperature\n", FUNCTION);
    }
}
```

15) Effectivement, comme sur le bout de code ci-dessus, on rentre bien dans le cas de la caractéristique température:

```
[I] sl_bt_on_event temperature: 2002 degC
[I] sl_bt_on_event condition check!!!
[I] characteristic access: temperature
```

16) Selon la datasheet:

uint8_t	status_flags	Enum <code>sl_bt_gatt_server_characteristic_status_flag_t</code> . Describes whether Client Characteristic Configuration was changed or if a confirmation was received. Values: <ul style="list-style-type: none">• <code>sl_bt_gatt_server_client_config (0x1)</code>: Characteristic client configuration has been changed.• <code>sl_bt_gatt_server_confirmation (0x2)</code>: Characteristic confirmation has been received.
---------	--------------	---

on peut, dans `sl_bt_api.h`, le voir aussi:

```
sl_bt_gatt_server_client_config = 0x1, /**< (0x1) Characteristic client
configuration has been changed. */
sl_bt_gatt_server_confirmation = 0x2, /**< (0x2) Characteristic confirmation
```

Il indique s'il y a eu un changement sur la configuration de la caractéristique client; c.a.d si on passe de notification à lecture simple.

17) Quand on clique sur Notify, on change donc la characteristic client config, et ce bit prend la valeur 0x1.

18) On ajoute donc un simple if: si le status flag vaut 0x01, on notifie (sans mauvaise allusion à notify) le changement de la configuration client. En plus, on dispose dans `sl_bt_api_compatibility.h` d'une jolie enum, qui permet de rendre le code plus lisible dans 2 semaines lorsque l'on aura tout oublié:

```
if (evt->data.evt_gatt_server_characteristic_status.status_flags == gatt_server_client_config) {
```

```
typedef enum
{
    gatt_server_client_config = 0x1,
    gatt_server_confirmation = 0x2
} gatt_server_characteristic_status_flag_t;
```

19) On fait le même procédé pour `client_config_flags`:
On utilise l'enum `gatt_notification` comme précédemment.

```
if (evt->data.evt_gatt_server_characteristic_status.client_config_flags == gatt_notification) {
    app_log_info("\n-----MODE NOTIF-----\n", FUNCTION);
    tempValue = get_temp();
    app_log_info("%sTemperature: %d degC\n", FUNCTION, tempValue);
    app_assert_status(sl_bt_gatt_server_send_notification(evt->data.evt_gatt_server_characteristic_status.connection,
                                                         gattdb_temperature,
                                                         sizeof(tempValue),
                                                         (uint8_t*)&tempValue
                                                         ));
}
```

On utilise aussi `sl_bt_gatt_server_send_notification()`, fonction de `sl_bt_api.h` similaire à celle utilisée pour le read simple:

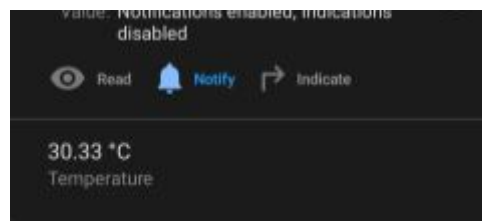
```
sl_status_t sl_bt_gatt_server_send_notification(uint8_t connection,
                                                uint16_t characteristic,
                                                size_t value_len,
                                                const uint8_t* value);

/*****
```

On récupère bien la température sur le log...

```
[I] status_flags: 216089[I]
-----MODE NOTIF-----
[I] get_tempRawval: 30156
[I] get_tempTemperature: 30 degC
[I] sl_bt_on_eventTemperature: 3015 degC
```

Et en notify:



Mettre en oeuvre le Timer

20) Le timer appelle le code lorsque qu'il finit son compte (timeout), et exécute le contenu du `sl_simple_callback_t`

```
2 sl_status_t sl_simple_timer_start(sl_simple_timer_t *timer,  
3                                     uint32_t timeout_ms,  
4                                     | sl_simple_timer_callback_t callback,  
5                                     void *callback_data,  
6                                     bool is_periodic);
```

Ce callback, disponible dans le même fichier, est le prototype de la fonction callback qu'on écrit, un peu comme sur plateforme STM32 de ST Microelectronics:

```
*****  
* Expected prototype of the user's callback function which is called when a  
* timer expires.  
*  
* @param timer Pointer to the timer handle.  
* @param data An extra parameter for the user application.  
*****  
typedef void (*sl_simple_timer_callback_t)(sl_simple_timer_t *timer, void *data);
```

21) On appelle donc le timer de la sorte:
on crée un `sl_simple_timer_t` appelé `mon_timer`

```
sc = sl_simple_timer_start(&mon_timer,  
                            1000,  
                            mon_callback,  
                            NULL,  
                            true);  
  
app_assert_status(sc);
```

et une fonction callback afin de surveiller la fréquence de rafraîchissement du notify:

```
static void mon_callback()  
{  
    int n = 0;  
    app_log_info("\\n-----\\nn=%d", n);  
    n++;  
}
```

On observe bien (les petits n = 1, 2, etc) qu'après le clic sur notify, on a effectivement un rafraîchissement de la température toutes les 1 secondes:

```
[I] get_tempRawval: 32547
-----MODE NOTIF-----
[I] get_tempTemperature: 32 degC
[I] sl_bt_on_eventTemperature: 3254
[I] get_tempRawval: 32580
[I] get_tempTemperature: 32 degC
[I]

n=0[I] get_tempRawval: 32558
[I] get_tempTemperature: 32 degC
[I]

n=1[I] get_tempRawval: 32569
[I] get_tempTemperature: 32 degC
[I]

n=2[I] get_tempRawval: 32580
[I] get_tempTemperature: 32 degC
[I]

n=3[I] get_tempRawval: 32580
[I] get_tempTemperature: 32 degC
[I]

n=4[I] get_tempRawval: 32601
[I] get_tempTemperature: 32 degC
[I]

n=5[I] get_tempRawval: 32580
[I] get_tempTemperature: 32 degC
[I]

n=6[I] get_tempRawval: 32580
```

22) On renvoi la température en notify avec `sl_bt_gatt_send_notification`:

```
app_assert_status(sl_bt_gatt_server_send_notification(evt->data.evt_gatt_server_user_read_request.connection,
gattdb.temperature,
sizeof(tempValue),
(uint8_t*)&tempValue
));
```

7. AJOUTER UN SERVICE AUTOMATION IO

Repérer un accès en écriture sur une caractéristique

23) Pour lire la valeur écrite:

```
case sl_bt_evt_gatt_server_user_write_request_id:
    app_log_info("%s condition check!!!\n", FUNCTION );
    if (evt->data.evt_gatt_server_user_write_request.characteristic == gattdb_digital) {
        app_log_info("characteristic access: digital\n", FUNCTION );
```

```
[I] sl_bt_on_event: connexion fermée!
[I] app_init
[I] sl_bt_on_event: connexion ouverte!
[I] sl_bt_on_event condition check!!!
[I] characteristic access: digital
```

On accède au sous champ data du sous champ value du sous champ `evt_gatt_server_user_write_request`. En effet, `value` est une structure, et comme pour la lecture du capteur, il contient 5 attributs:

```
*****/
PACKSTRUCT( struct sl_bt_evt_gatt_server_attribute_value_s
{
    uint8_t    connection; /**< Connection handle */
    uint16_t   attribute; /**< Attribute Handle */
    uint8_t    att_opcode; /**< Enum @ref sl_bt_gatt_att_opcode_t. Attribute
                           opcode that informs the procedure from which the
                           value was received. */
    uint16_t   offset; /**< Value offset */
    uint8array value; /**< Value */
});
```

Ce qui nous intéresse, c'est le dernier: `value`. la valeur du champ qu'entre l'utilisateur dans l'application EFR.

Utiliser une API de LEDs simple

24) Dans nos premiers tests, on initialise le driver des LED de la sorte, dans `app_init()`:

```
sl_led_init(&sl_led_led0);
```

Cependant, dans `sl_simple_led.c`, il existe une procédure qui initialise la `led0` sans avoir à préciser son nom (pratique lorsque l'on change la configuration gatt et que les fichiers sont modifiés automatiquement, ou lorsque l'on dispose de plusieurs LED):

```
1 void sl_simple_led_init_instances(void)
2 {
3     sl_led_init(&sl_led_led0);
4 }
5
```

ainsi, on remplace, toujours dans `app_init()`:

```
4 SL_WEAK void app_init(void)
5 {
6     //////////////////////////////////////
7     // Put your additional application init code here
8     app_log_info("%s\n", FUNCTION);
9     //sl_led_init(&sl_led_led0);
10    sl_simple_led_init_instances();
11 }
```

25) actuellement, il n'y a qu'une seule led dans l'instance. On l'a vu précédemment, la seule led initialisée dans `sl_simple_led_init_instances` est la `led0`. Aussi, on peut jeter un coup d'oeil au contextes (infos hardware) déclarés dans `sl_simple_led_instances.c`: il n'y en a qu'un seul.

```
2 sl_simple_led_context_t simple_led0_context = {
3     .port = SL_SIMPLE_LED_LED0_PORT,
4     .pin = SL_SIMPLE_LED_LED0_PIN,
5     .polarity = SL_SIMPLE_LED_LED0_POLARITY,
6 };
```

26) l'accès aux instances se fait en appelant `sl_led_led0`:

```
const sl_led_t sl_led_led0 = {
    .context = &simple_led0_context,
    .init = sl_simple_led_init,
    .turn_on = sl_simple_led_turn_on,
    .turn_off = sl_simple_led_turn_off,
    .toggle = sl_simple_led_toggle,
    .get_state = sl_simple_led_get_state,
};
```

Puisque `sl_led_led0` est une structure (type: `sl_led_t`):

```
7 /// A LED instance
8 typedef struct {
9     void *context;           ///< The context for this LED instance
10    sl_status_t (*init)(void *context);    ///< Member function to initialize LED instance
11    void (*turn_on)(void *context);        ///< Member function to turn on LED
12    void (*turn_off)(void *context);       ///< Member function to turn off LED
13    void (*toggle)(void *context);         ///< Member function to toggle LED
14    sl_led_state_t (*get_state)(void *context); ///< Member function to retrieve LED state
15 } sl_led_t;
```

On peut facilement éteindre et allumer la led en appelant son adresse dans les fonctions de la bibliothèque Simple LED Driver:

```
app_log_info("sl_led0: ON\n");
sl_led_turn_on(&sl_led_led0);
```

De même, on peut avoir accès à toutes les LED avec le tableau `sl_simple_led_array`:

```
extern const sl_led_t *sl_simple_led_array[];
```

Il est ici déclaré comme ayant une seule LED, `led0`:

```
const sl_led_t *sl_simple_led_array[] = {
    &sl_led_led0
};
```

Plutôt suspect, pour une carte arborant des LED RGB tel un PC de gamer... (en fait, ce n'est pas suspect du tout! Nous n'avons instancié que `led0`.)

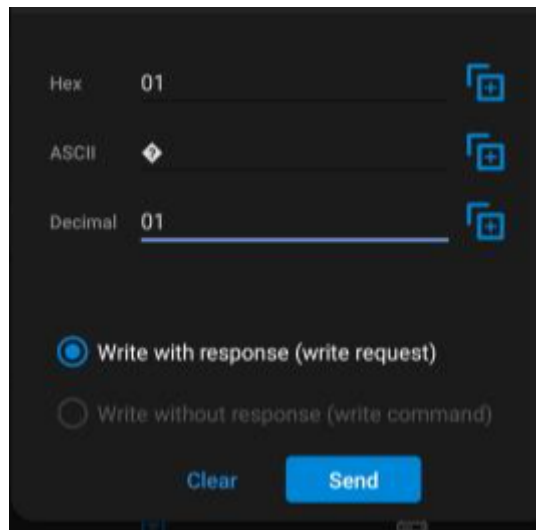
27) On peut maintenant facilement allumer ou éteindre les led: il suffit de vérifier ce que tape l'utilisateur dans le champ digital de l'appli. Si il envoie 0, on éteint les LED, si il envoie un autre chiffre, on les allume:

```
if(evt->data.evt_gatt_server_attribute_value.value.data[0] != 0){
    app_log_info("sl_led0: ON\n");
    sl_led_turn_on(*sl_simple_led_array);
}
else{
    app_log_info("sl_led0: OFF \n");
    sl_led_turn_off(*sl_simple_led_array);
}
```

[Petite démonstration disponible ici](#)

(NB: la vidéo de démo est au format `.webm`, un petit format ouvert oublié devant le `.mp4`, qui semble seulement persister sur les forums d'antan comme 4chan; mais il est très pratique: il est ouvert, et produits des fichiers qui peuvent être longs et très légers! dans le cas d'un simple codec vidéo ouvert, efficace et frugal, il serait intéressant dans le cadre de l'IoT...)

28) Dans EFR connect, on dispose de plusieurs formats pour write, mais surtout 2 options avec 2 boutons radio:



The screenshot shows a dark-themed interface for writing data. It has three input fields: 'Hex' with the value '01', 'ASCII' with a diamond symbol, and 'Decimal' with the value '01'. Each field has a copy icon to its right. Below these fields are two radio buttons: 'Write with response (write request)' which is selected, and 'Write without response (write command)'. At the bottom are 'Clear' and 'Send' buttons.

Write with response et write without response.

29) Dans les propriétés du GATT, on observe que l'option grisée peut être activée. Cela fait penser aux questions du 1er TP sur les valeurs pollées ou pushées.

Properties	Authenticated	Bonded	Encrypted
<input type="checkbox"/> Read	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> Write	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/> Write without response	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/> Reliable write	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

30) Dans notre code, il manque la partie ACK (acknowledge)/accusé de réception, qui part du MCU vers l'application EFR pour valider la bonne réception. L'omission de ce bout de code signifie que l'application attend l'ACK, et pense que le MCU a planté, est hors tension, ou quoi que ce soit: il timeout.

31) Effectivement: dans `sl_bt_api.h`, il est noté explicitement:
“Send a response to a `@ref sl_bt_evt_gatt_server_user_write_request` event when parameter `@p att_opcode` in the event is `@ref sl_bt_gatt_write_request` or `@ref sl_bt_gatt_execute_write_request` (see `@ref sl_bt_gatt_att_opcode_t`). The response needs to be sent within 30 seconds, otherwise no more GATT transactions are allowed by the remote side [...]”
Il faut donc lui renvoyer le `att_opcode` sous 30 secondes pour ne pas timeout.

```
sl_status_t sl_bt_gatt_server_send_user_write_response(uint8_t connection,  
                                                       uint16_t characteristic,  
                                                       uint8_t att_errorcode);
```

32) on rassure donc EFR connect, on lui dit que tout est OK (littéralement, SL_STATUS_OK)

```
sl_bt_gatt_server_send_user_write_response(evt->data.evt_gatt_server_user_write_request.connection,  
                                           gattdb_digital,  
                                           SL_STATUS_OK);
```

33) Après recompilation, on remarque que l'écriture sans réponse (commande au lieu de requête) est disponible.

