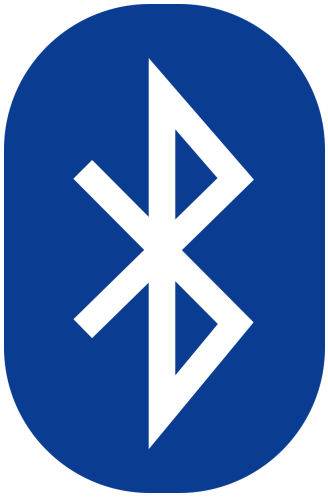
LIM
ZHANG
1E

Dylan
Laurent
E4FE

RAPPORT DE TP2
Objet Connecté



M.BENABOU

2023/2024

SOMMAIRE

I. Création du nouveau projet BLE

Création du projet

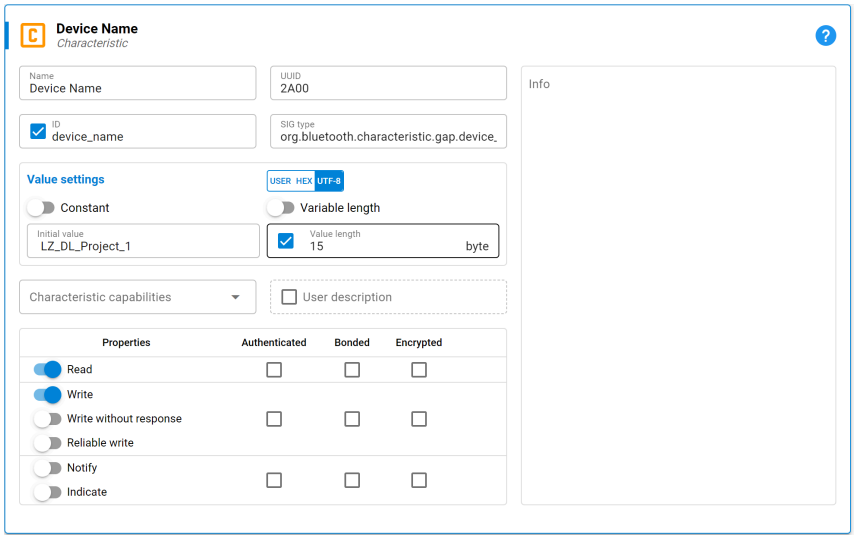
Dans cette partie, on commence par créer un nouveau projet en suivant l'exemple donné dans l'énoncé du TP. On a à disposition un exemple appelé "Bluetooth – SoC Empty", que l'on peut utiliser comme base pour développer l'application.

Versionner le projet sur Github

Dans un second temps, on fait en sorte de rendre le projet accessible sur Github grâce aux commandes suivantes tapées dans le terminal :
git init
git add .
git commit -m "commit initial groupe LZ_DL"

Configurer le GATT initial

On peut ensuite s'intéresser au GATT ("Generic ATTribute Profile"). Il va nous servir pour renseigner les fonctionnalités ou services, dans une liste (profile), précisant ces caractéristiques.

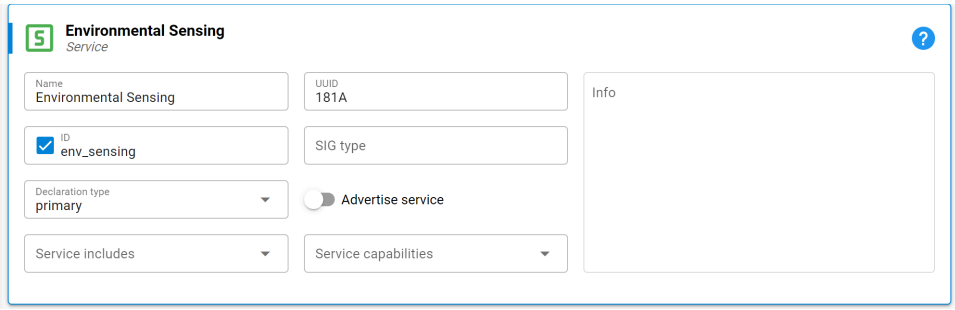


II. Rajouter des fonctionnalités BLE et logicielles

a. Rajouter un service et une caractéristique standards

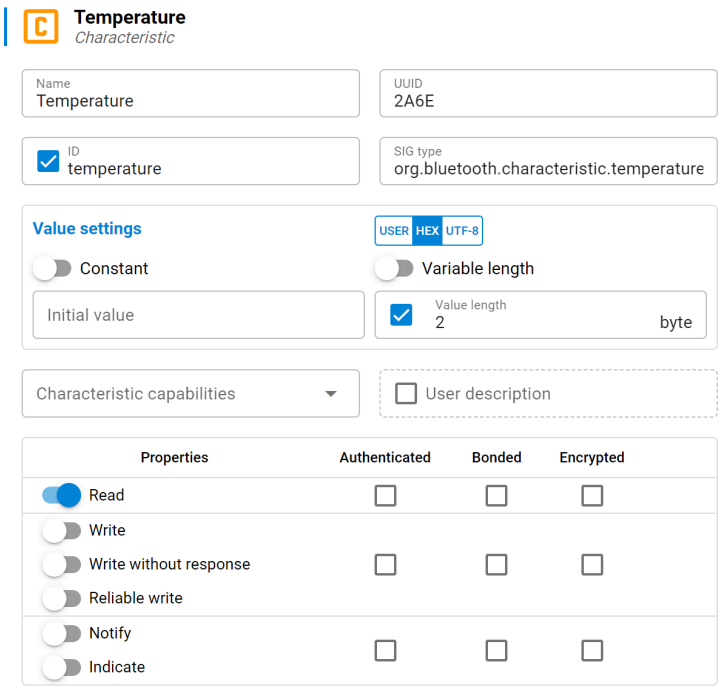
On doit rajouter un premier service applicatif, il doit permettre de donner des informations venant des capteurs d'environnement. Tout ce qui est à définir est le nom, l'id du service,

l'UUID, le type de déclaration. Comme indiqué, on a mis les UUID standard pour un capteur d'environnement.

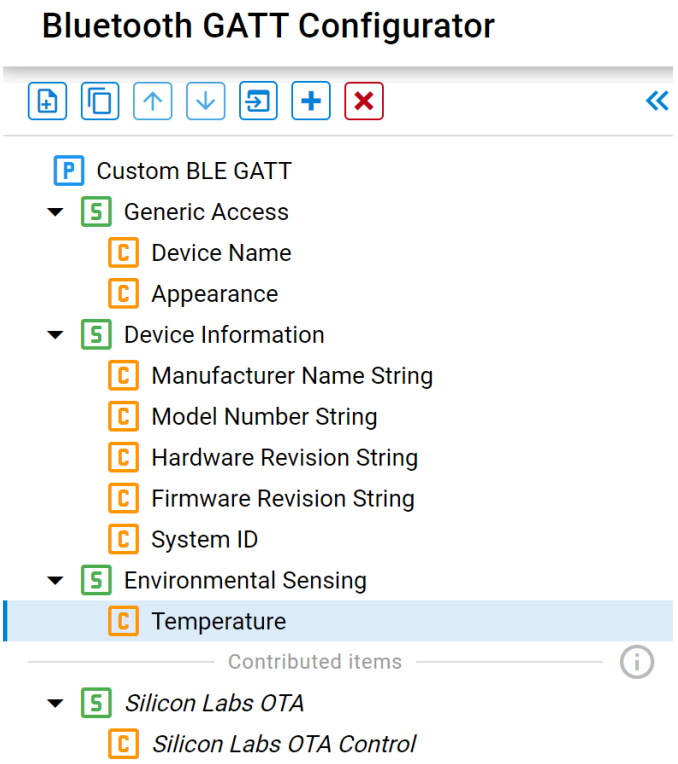


Il nous est proposé différentes caractéristiques à associer à notre service. On a défini une caractéristique de température avec la possibilité de l'activer en mode Read

Voici comment se présente notre caractéristique temperature :



A la fin de notre configuration , on peut observer tout ce que comporte le GATT. On retrouve dans cette liste, tous les services par la lettre “S” , les caractéristiques par la lettre “C”, la lettre “P” représente la différence par rapport à la configuration du protocole d’origine.

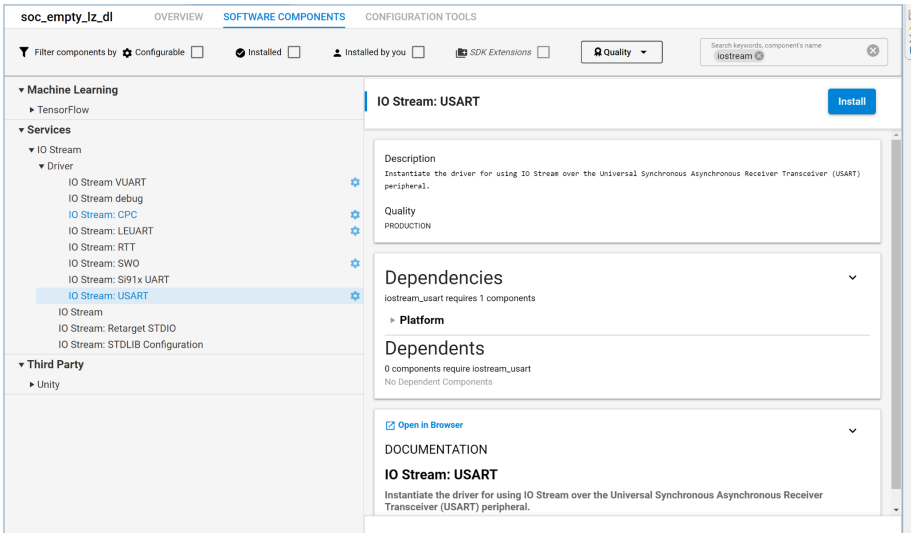


b. Rajouter des composants logiciels

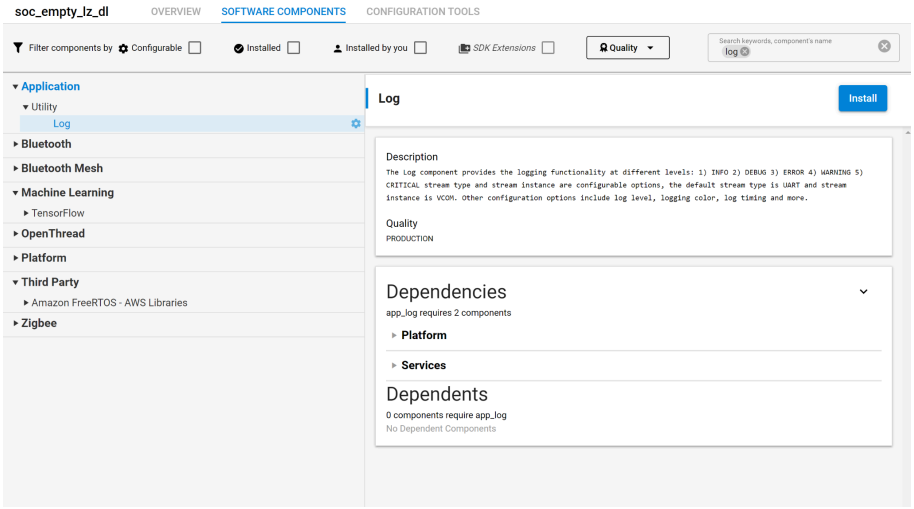
Dans cette sous-partie, on s’intéresse à installer ou configurer des interfaces d’affichages de logs qui nous seront utiles par la suite. Les logs sont une source essentielle pour obtenir un retour rapide sur l’application en cours de développement. Ils permettent également de repérer judicieusement la chronologie des processus grâce à l’horodatage des timers.

Activer le framework d’entrées sorties sur l’interface série

Ici, nous montrons la configuration de l’interface série. Nous utilisons l’USART pour la transmission des caractères de la carte vers le PC que l’on peut recevoir et afficher.



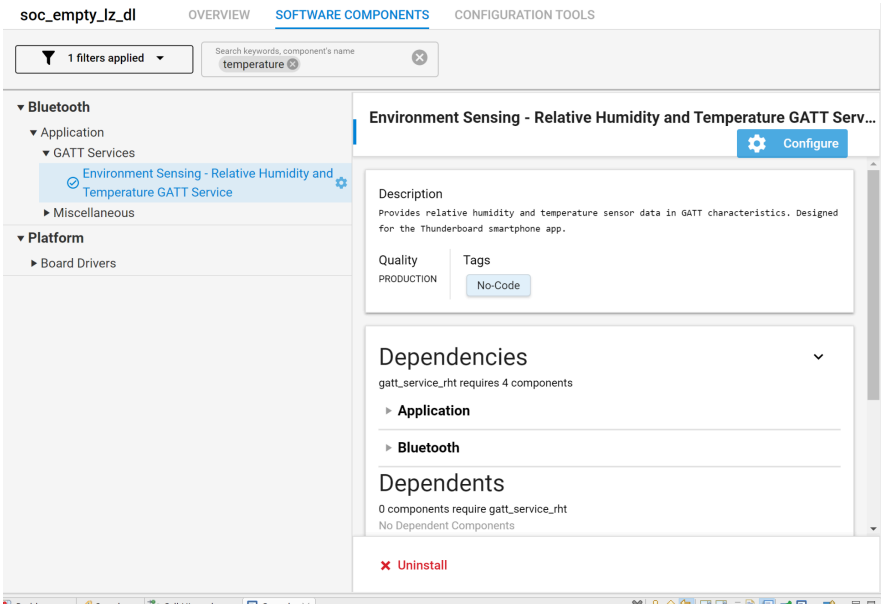
Activer le framework de log applicatif



Voici, un composant de log applicatif, permettant de facilement utiliser la communication USART et les streams de données à logger. Celui-ci est appelé par la fonction “app_log”

c. Rajouter un pilote de capteur

On a rajouté les pilotes des capteurs d'humidité et de température à partir de l’onglet software components. Puis de saisir dans la barre de recherche temperature et trouver *Relative Humidity and Temperature Sensor* et télécharger ce pilote.



III. Tester les modifications GATT et logicielles

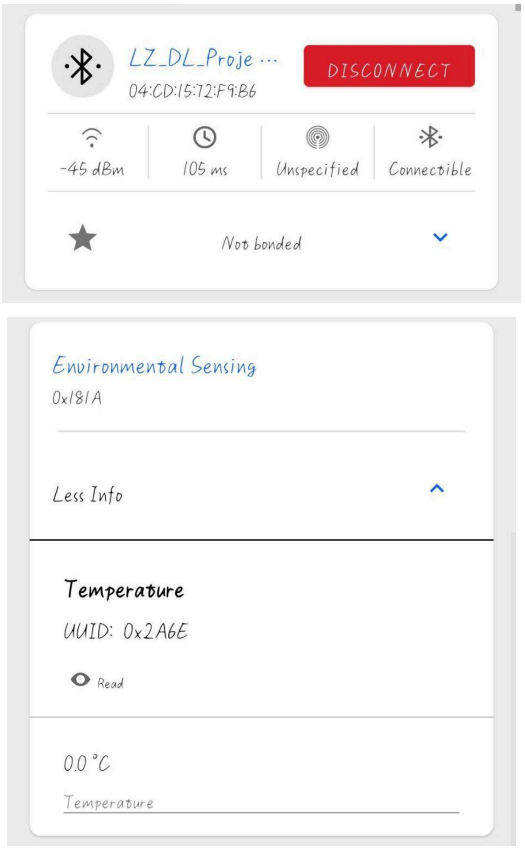
a. Vérifier la présence des services

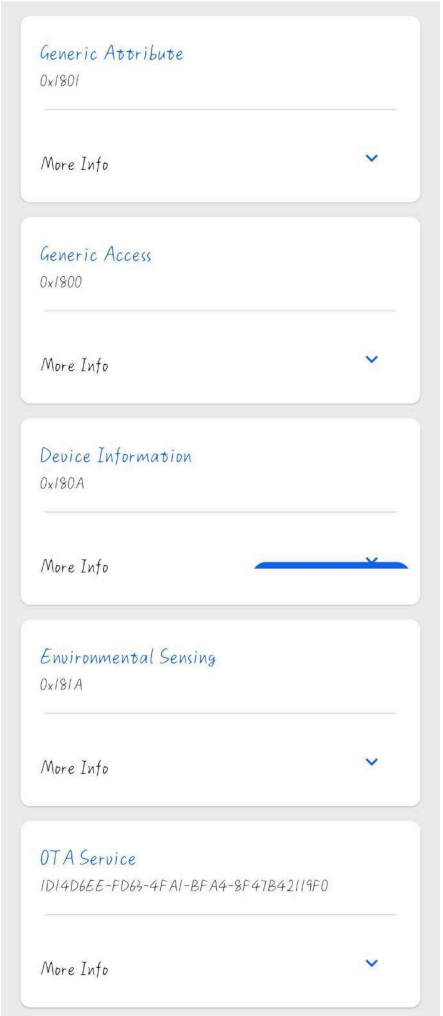
Après avoir compilé le code, puis flashé sur la TS2, on utilise l'application EFR Connect. Cette application va reconnaître non seulement toutes les TS2 via BLE, mais aussi tous les devices utilisant des services développés via Simplicity Studio/Silicon Labs. Après un scan, on peut retrouver notre application qui porte le nom qu'on lui a donné.

Lorsqu'on ouvre notre application, on trouve tous les champs qu'on a développés, les fameux GATT. On trouve aussi d'autres champs possibles mais ceux-ci sont inconnus.

Si on souhaite connaître davantage sur certaines caractéristiques, il est possible d'obtenir plus de détails concernant l'UUID, ou encore sur la valeur de caractéristique, des actions de lecture/écriture.

Les services retrouvés BLE sont alors les suivants : Generic Access, Device Information, Environmental Sensing, OTA Service.





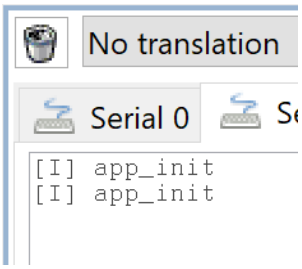
b. Tester le composant de log

Nous réalisons un simple test pour voir comment se comporte la fonctionnalité de log sur notre application. Le test proposé consiste à afficher un message lors de l'appel app_init (démarrage application). Il suffit donc d'obtenir un affichage d'un message spécifique à la connexion de l'application puis à nouveau le même message à la déconnexion-reconnexion. Le test suivant permet de gérer l'affichage de messages différents à la connexion et déconnexion en BLE.

En débutant notre application, nous avons l'extrait du code suivant qui a l'initialisation doit renvoyer le nom de notre fonction.

```
30 #include "em_common.h"
31 #include "app_assert.h"
32 #include "sl_bluetooth.h"
33 #include "app.h"
34 #include "app_log.h"
35
36 // The advertising set handle allocated from Bluetooth stack.
37 static uint8_t advertising_set_handle = 0xff;
38
39 /**************************************************************************//**
40  * Application Init.
41  *****************************************************************************/
42 SL_WEAK void app_init(void)
43 {
44   /////////////////////////////////////////////////////////////////////////////
45   // Put your additional application init code here!                         //
46   app_log_info("%s\n",__FUNCTION__);
47   // This is called once during start-up.                                    //
48   /////////////////////////////////////////////////////////////////////////////
49 }
50
51 /**************************************************************************//**
```

En flashant ce code et à l'aide de la console, nous observons que nous obtenons bien sur la console le nom de la fonction d'initialisation app_init.

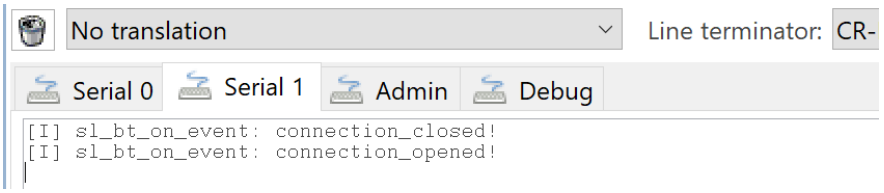


De plus, en appuyant sur le bouton Reset nous réinitialisons la carte et donc nous arrivons à obtenir une deuxième fois sur la console l'appel de la fonction app_init().

Dans le code de base nous avons aussi ces deux cas de figure:

```
// ------------------------------
// This event indicates that a new connection was opened.
case sl_bt_evt_connection_opened_id:
  app_log_info("%s: connection_opened!\n",__FUNCTION__);
  break;

// ------------------------------
// This event indicates that a connection was closed.
case sl_bt_evt_connection_closed_id:
  app_log_info("%s: connection_closed!\n",__FUNCTION__);

  // Generate data for advertising
  sc = sl_bt_legacy_advertiser_generate_data(advertising_set_handle,
                                             sl_bt_advertiser_general_discoverable);
  app_assert_status(sc);

  // Restart advertising after client has disconnected.
  sc = sl_bt_legacy_advertiser_start(advertising_set_handle,
                                     sl_bt_legacy_advertiser_connectable);
  app_assert_status(sc);
  break;
```

Ces deux cas de figure sont défini par la détection de sl_bt_evt_connection_opened_id ou sl_bt_evt_connection_closed_id, c'est-à-dire si nous nous connectons sur la carte depuis l'application EFR Connect. Nous obtenons les log suivants après une connexion et une déconnexion de la carte.



Cela nous renvoie bien le nom de l'événement.

c. Repérer un accès en lecture d'une caractéristique

Pour réussir à lire une caractéristique, l'utilisateur doit passer par les informations présentes dans la base de données GATT où le serveur possède tous ses champs de caractéristiques. Or, la demande se fait par l'événement Read Request, elle est initiée par le champ sl_bt_evt_gatt_server_user_read_request . Nous observons une description détaillée de celui-ci dans le code :

```
/**
 * @addtogroup sl_bt_evt_gatt_server_user_read_request
sl_bt_evt_gatt_server_user_read_request
 * @{
 * @brief Indicates that a remote GATT client is attempting to read a value of
 * an attribute from the local GATT database, where the attribute was defined in
 * the GATT database XML file to have the type="user"
 *
 * The parameter @p att_opcode informs which GATT procedure was used to read the
 * value. The application needs to respond to this request by using the @ref
 * sl_bt_gatt_server_send_user_read_response command within 30 seconds,
 * otherwise further GATT transactions are not allowed by the remote side.
 */
```

Il semble donc qu'on passe par cet événement lors de la lecture d'une caractéristique Température depuis EFR Connect.
On verra par la suite qu'on utilise un case (switch) pour détecter la présence de l'événement Read Request, en examinant l'id.

IV. Retourner une valeur de caractéristique

a. Initialisation du capteur de température

QUESTION 1: Observez les APIs sl_sensor_rht_init() et sl_sensor_rht_deinit(). À quoi servent-elles ? Dans app.c, à quel moment devriez-vous appeler ces fonctions ? Rajoutez les appels nécessaires.

Les API correspondant aux sl_sensor_rht_init() et sl_sensor_rht_deinit() sont “Initialize Relative Humidity and Temperature sensor”.

Dans sl_sensor_rht_init(), on observe des appels de fonctions permettant de sélectionner le capteur, la vérification des statuts du capteur, la présence ou non du capteur. En conclusion, elle permet de gérer facilement l'initialisation du capteur et son activation.
Contexte d'utilisation: Il faut l'utiliser au démarrage de l'application et dans app_init.

sl_sensor_rht_deinit() permet de désactiver le capteur et dire que le capteur n'est plus init.*
Contexte d'utilisation: Il faut l'utiliser quand on quitte l'application.

b. Lire la température depuis son capteur

QUESTION 2 : Quelle est la fonction qui lit la température du capteur idoine ?

Nous pouvons utiliser la fonction sl_status_t sl_sensor_rht_get(uint32_t *rh, int32_t *t) pour récupérer la valeur de la température.

QUESTION 3 : Sous quel format arrive cette température ?

La température nous arrive sous la forme d'une valeur en hexadécimal soit une suite de valeur binaire que nous devons par la suite faire la conversion pour obtenir la bonne valeur de la température.

c. Formater la température selon le standard BLE

QUESTION 4: Observez la Represented Value. Comment convertir une température quelconque (supposons-la en degrés, flottante) vers le format BLE ?

D'après le fichier GATT Specification Supplement, nous avons pu trouver l'information suivante concernant la température:

3.213 Temperature

The Temperature characteristic is used to represent a temperature.

The structure of this characteristic is defined below.

Field	Data Type	Size (in octets)	Description
Temperature	sint16	2	Base Unit: org.bluetooth.unit.thermodynamic_-temperature.degree_celsius Represented values: M = 1, d = -2, b = 0 Unit is degrees Celsius with a resolution of 0.01 degrees Celsius. Allowed range is: -273.15 to 327.67. A value of 0x8000 represents 'value is not known'. All other values are prohibited.

Table 3.324: Structure of the Temperature characteristic

La température est une valeur sur 2 octets de type sint16 pouvant mesurer les températures dans une plage de température de -273,15 à 327,67 °C. Ces valeurs peuvent être décrypter d'après l'équation suivante: $R = C * M * 10^d * 2^b$. Pour le cas de la température M=1, d=-2 et b=0 donc la conversion devras réaliser l'équation suivante:
$Température = Mesure * 1 * 10^{-2} * 1.$

QUESTION 5 : Vérifiez leur fonctionnement avec du log.

D'après les log, nous observons une valeur de température de 212236 °C, ce qui nous indique que la conversion n'est pas encore réalisé, mais nous pouvons déduire la valeur de la température de 21.22°C. Car l'unité est en °C avec une résolution de 0,01 °C.

```
app_init
sl_bt_on_event: connection_opened!
212236:temperature
```

QUESTION 6 : Quelles façons simples connaissez-vous, pour rendre des variables accessibles à plusieurs fichiers C ? Note : nous sommes dans un système

Parmi les méthodes connues permettant à l'accessibilité, on peut citer : les variables globales, les pointeurs, des librairies ainsi que des structures.

QUESTION 7 : Laquelle de ces façons est la plus plausible ?

Comme on utilise une application réservée à un système embarqué, nous sommes ainsi limités en ressource et la manière la plus adéquate est alors de se restreindre seulement à l'utilisation des librairies. Ainsi, on peut importer des fichiers contenant des "define" et les réutiliser dans les autres fichiers. Toutefois, il est tout à fait possible de varier les techniques en combinant avec des pointeurs et des structures pour plus de maintenabilité de code.

QUESTION 8 : Comment isoler leur portée (le mieux possible) ?

Leur portée peut être isolée dans des librairies spécifiques qui utilisent uniquement les variables nécessaires, englobe des prototypes de calculs locaux.

C'est ainsi que nous avons mis en place la librairie temperature.h et temperature.c pour nous permettre de mesurer la température.

Le fichier temperature.c:

```
#include "temperature.h"
#include "sl_sensor_rht.h"
int getTemperature(){
  uint32_t rh;
  uint32_t t;
  sl_sensor_rht_init();
  sl_sensor_rht_get(&rh,&t);
  sl_sensor_rht_deinit();
  return t/10;
}
```

Dans le fichier, nous utilisons le principe vu dans la partie précédente qu'on appellera par une librairie. Pour une librairie un fichier .c ne suffit pas, nous devons aussi déclarer les header dans le fichier temperature.h, c'est ce fichier que nous allons l'utiliser dans app.c.

Le fichier temperature.h:

```
#ifndef TEMPERATURE_H_
#define TEMPERATURE_H_
int getTemperature();
#endif
```

Après avoir fait #include "temperature.h" dans app.c, nous pouvons faire appel à la fonction getTemperature().

```
case sl_bt_evt_gatt_server_user_read_request_id:
     app_log_info("temperature:%dC\n",getTemperature()/100);
break
```

Nous observons bien sur le terminal la valeur de la température mesurée par la carte.

```
[I] app_init
[I] sl_bt_on_event: connection_opened!
[I] temperature:31C
[I] sl_bt_on_event: connection_closed!
```

d. Vérifier que l'accès en lecture concerne bien la caractéristique Température

L'événement que la fonction sl_bt_on_event(sl_bt_msg_t * evt) reçoit contient un champ data qui contient une union vers plusieurs types. Celui qui nous intéresse est evt→data.evt_gatt_server_user_read_request, de type

sl_bt_evt_gatt_server_user_read_request_t. Vous trouverez sa documentation dans sl_bt_api.h, disponible ici : gecko_sdk_x.x.x/protocol/bluetooth/inc/sl_bt_api.h 1.

QUESTION 9 : Observez votre fichier autogen/gatt_db.h. A votre avis, quel est l'identifiant de la caractéristique température ?

En regardant dans le fichier gatt_db.h, nous observons que la caractéristique température possède comme identifiant la valeur 27 stockée dans la variable gattdb_temperature.

```
#ifndef __GATT_DB_H
#define __GATT_DB_H
#include "sli_bt_gattdb_def.h"
extern const sli_bt_gattdb_t gattdb;
#define gattdb_generic_attribute              1
#define gattdb_service_changed_char           3
#define gattdb_database_hash                  6
#define gattdb_client_support_features        8
#define gattdb_device_name                    11
#define gattdb_device_information             14
#define gattdb_manufacturer_name_string       16
#define gattdb_model_number_string            18
#define gattdb_hardware_revision_string       20
#define gattdb_firmware_revision_string       22
#define gattdb_system_id                      24
#define gattdb_env_sensing                    25
#define gattdb_temperature                    27
#define gattdb_ota                            28
#define gattdb_ota_control                    30
#endif // __GATT_DB_H
```

QUESTION 10 : Créez une condition de test pour vérifier que l'accès de lecture est bien pour la caractéristique température.

On peut voir la nouvelle condition de test dans le switch permettant de vérifier si nous effectuons bien une lecture de la caractéristique température.

```
case sl_bt_evt_gatt_server_user_read_request_id:
  app_log_info("test_case\n",__FUNCTION__);
  switch (evt->data.evt_gatt_server_user_read_request.characteristic) {
    case gattdb_temperature:
      app_log_info("temperature:%dC\n",getTemperature());
      break;
```

Dans cet extrait, nous renvoyons la valeur de la température par la librairie que nous avons créé précédemment.

QUESTION 11 : Vérifiez-la avec du log et faites un screenshot de votre console.

Après avoir flashé le nouveau code, et avoir appuyé sur le bouton read sur l'application EFR Connect, nous obtenons le screenshot de la console suivante:

```
app_init
sl_bt_on_event: connection_opened!
test_case
temperature:3347C
```

Nous observons donc que la température est de 33,47°C, lors de cette compilation nous n'avons pas effectué la conversion en °C donc la console nous affiche la valeur de la température par unité de 0,01°C.

e. Renvoyer ladite température

QUESTION 12 : Avec les fonctions que vous avez précédemment écrites, complétez le code du case, puis appelez la fonction ci-dessus avec les bonnes valeurs d'argument. On suppose qu'il n'y a pas d'erreur, donc att_errcode vaudra 0.

D'après ce que nous avons vu avec la fonction sl_status_t sl_bt_gatt_server_send_user_read_response(uint8_t connection, uint16_t characteristic, uint8_t att_errorcode, size_t value_len, const uint8_t* value, uint16_t *sent_len); nous permet d'afficher la valeur de la température sur l'application EFR Connect.

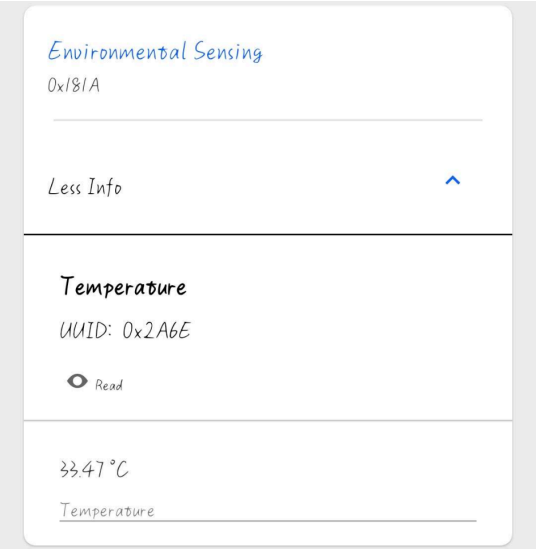
```
case sl_bt_evt_gatt_server_user_read_request_id:
    app_log_info("test_case\n",__FUNCTION__);
    switch (evt->data.evt_gatt_server_user_read_request.characteristic) {
      case gattdb_temperature:
        int temperature=getTemperature();
        app_log_info("temperature:%dC\n",temperature);
sc=sl_bt_gatt_server_send_user_read_response(evt->data.handle,gattdb_temperature,0,si
zeof(temperature),&temperature,&sent_lent);
          app_assert_status(sc);
          break;
```

Les paramètres à choisir sont donc evt->data.handle elle correspond à la valeur que nous souhaitons modifier, gattdb_temperature vu précédemment correspond à la caractéristique de la température. Elle est à 0 car on suppose que nous n'avons pas d'erreur et pour finir nous devons lui attribuer la valeur de la température mesurée par le capteur de température

de la carte, on utilisera sizeof() même si nous savons la taille de notre valeur de température.

QUESTION 13 : Faites des screenshots pertinents pour prouver votre succès !

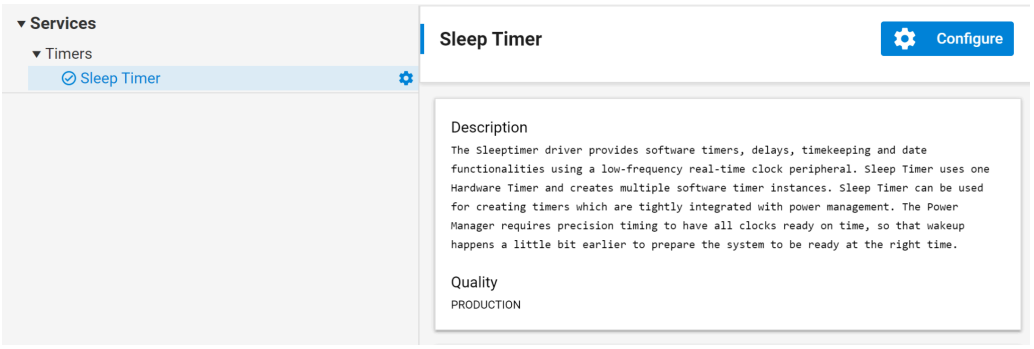
Nous observons sur EFR Connect que la valeur a bien été modifiée par la valeur de la température.



V. Lire la température avec notification

a. Rajouter un composant logiciel timer

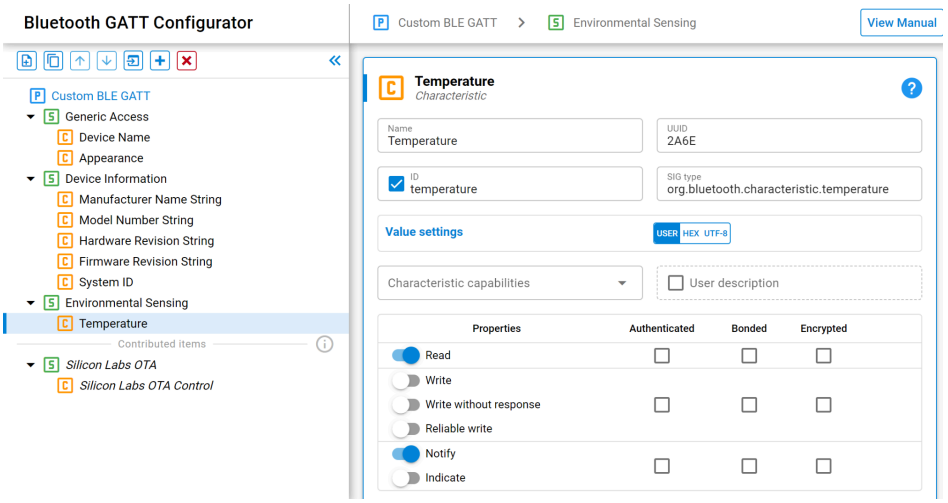
Comme pour l'ajout du capteur de température et d'humidité, nous allons rajouter cette fois le composant de timer, qui se nomme Sleep Timer.



Nous l'utiliserons plus tard dans cette partie.

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

Nous allons à présent travailler avec Notify, donc dans les paramètres de la caractéristique de la température, nous devons autoriser l'intervention de Notify.

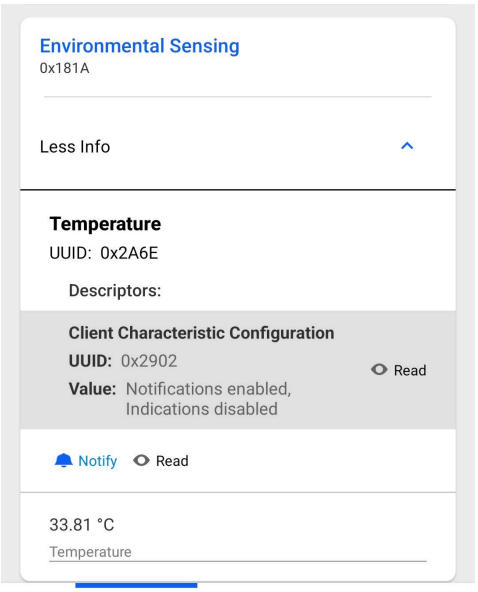


QUESTION 14 : Quand vous appuyez sur Notify avec EFR Connect, rentrez-vous bien dans ce case ?

Après avoir rajouté sl_bt_evt_gatt_server_characteristic_status_id et souhaitons afficher le message test_notify sur la console.

```
case sl_bt_evt_gatt_server_characteristic_status_id:
   app_log_info("test_notify\n",__FUNCTION__);
break;
```

En appuyant sur Notify, sur la console nous pouvons observer le log test_notify. Ce qui nous permet de confirmer que nous rentrons bien dans le case.



```
[I] app_init
[I] sl_bt_on_event: connection_opened!
[I] test_notify
```

QUESTION 15 : Vérifiez que ce paramètre Notify concerne bien la caractéristique Temperature. Faites des logs et screenshotez-les.

En appliquant le même procédure dans notify avec la caractéristique de la température, nous arrivons à obtenir le code suivant:

```
case sl_bt_evt_gatt_server_characteristic_status_id:
    app_log_info("test_notify\n",__FUNCTION__);
    switch (evt->data.evt_gatt_server_characteristic_status.characteristic){
         case gattdb_temperature:
            int temperature=getTemperature();
            app_log_info("temperature_notify:C coucou!\n");
            break;
         default:
            break;
```

Une fois que nous avons flashé le code, nous pouvons vérifier le fonctionnement de notify dans ce cas en affichant le message “temperature_notify:C coucou!”.

```
[I] test_notify
[I] temperature_notify:C coucou!
```

Sur la console, nous observons bien le message souhaité. Donc nous confirmons que le notify correspond à la caractéristique de la température. Nous pouvons rajouter cette fois l’affichage de la valeur de la température.

```
case sl_bt_evt_gatt_server_characteristic_status_id:
 app_log_info("test_notify\n",__FUNCTION__);
 switch (evt->data.evt_gatt_server_characteristic_status.characteristic){
   case gattdb_temperature:
     int temperature=getTemperature();
     app_log_info("notify:Coucou!\n");
     app_log_info("temperature_notify:%dC\n",temperature);
     break;
   default:
     break;
```

```
[I] app_init
[I] sl_bt_on_event: connection_opened!
[I] test_notify
[I] notify:Coucou!
[I] temperature_notify:2988C
```

c. Vérifier la raison pour laquelle la caractéristique est modifiée

Dans la librairie sl_bt_api.h, nous trouver la structure sl_bt_evt_gatt_server_characteristic_status_s:

```
PACKSTRUCT( struct sl_bt_evt_gatt_server_characteristic_status_s
{
  uint8_t  connection;          /**< Connection handle */
  uint16_t characteristic;      /**< GATT characteristic handle. This value is
                                     normally received from the
                                     gatt_characteristic event. */
  uint8_t  status_flags;        /**< Enum @ref
                                     sl_bt_gatt_server_characteristic_status_flag_t.
                                     Describes whether Client Characteristic
                                     Configuration was changed or if a
                                     confirmation was received. Values:
                                       - <b>sl_bt_gatt_server_client_config
                                         (0x1):</b> Characteristic client
                                         configuration has been changed.
                                       - <b>sl_bt_gatt_server_confirmation
                                         (0x2):</b> Characteristic confirmation
                                         has been received. */
  uint16_t client_config_flags; /**< Enum @ref
                                     sl_bt_gatt_server_client_configuration_t.
                                     This field carries the new value of the
                                     Client Characteristic Configuration. If the
                                     status_flags is 0x2 (confirmation
                                     received), the value of this field can be
                                     ignored. */
  uint16_t client_config;       /**< The handle of client-config descriptor. */
});
```

Nous pouvons observer le champ status_flags.
Et notamment le bit sl_bt_gatt_server_client_config

QUESTION 16 : À quoi sert-il ?

Le bit sl_bt_gatt_server_client_config est un bit servant à indiquer si nous avons fini ou non de configurer la caractéristique du client. Si ce bit est à 1, c'est que nous avons terminé la configuration de la caractéristique. Le cas contraire à 0.

QUESTION 17 : Quand vous cliquez sur Notify, qu'observez-vous vis à vis de ce champ?

Nous observons que le fait de cliquer sur Notify affecte une nouvelle valeur au client_config_flags en particulier, c'est le champ sl_bt_gatt_server_client_configuration_t, que l'on retrouve dans un fichier spécifique. Il décrit l'état de notification (désactivé, à notifier, à indiquer, les deux)

Dans sl_bt_api.h, nous observons les informations suivantes :

```
typedef enum
{
  sl_bt_gatt_server_disable                    = 0x0, /**< (0x0) Disable
                                                         notifications and
                                                         indications. */
  sl_bt_gatt_server_notification               = 0x1, /**< (0x1) The
                                                         characteristic value
                                                         shall be notified. */
  sl_bt_gatt_server_indication                 = 0x2, /**< (0x2) The
                                                         characteristic value
                                                         shall be indicated. */
  sl_bt_gatt_server_notification_and_indication = 0x3  /**< (0x3) The
                                                         characteristic value
                                                         notification and
                                                         indication are
                                                         enabled, application
                                                         decides which one to
                                                         send. */
} sl_bt_gatt_server_client_configuration_t;
```

QUESTION 18 : Créez toutes les conditions pour être sûr que quand on passe dans sl_bt_on_event(sl_bt_msg_t * evt), on détecte précisément le fait qu'on appuie sur le bouton Notify de la caractéristique Temperature.

En suivant ce que nous avons observé, nous pouvons rajouter cette condition et afficher cette valeur sur la console.

```
if(evt->data.evt_gatt_server_characteristic_status.status_flags && 0x1)//end switch
app_log_info("config_flag:%d\n",evt->data.evt_gatt_server_characteristic_status.client_config_flags);
```

Nous observons que lorsque Notify est active, config_flag est à 1 alors que lorsqu'elle est désactivée, config_flag est à 0.

Avec notify:

```
[I] test_notify
[I] notify:Coucou!
[I] temperature_notify:3074C
[I] config_flag:1
```

Sans Notify:

```
[I] test_notify
[I] notify:Coucou!
[I] temperature_notify:3092C
[I] config_flag:0
```

d. Vérifier le statut de Notify

QUESTION 19 : Complétez votre code ci-dessus pour afficher la valeur reçue de Notify.

En complétant le code, nous pouvons afficher la valeur reçue de Notify.

```
if(evt->data.evt_gatt_server_characteristic_status.status_flags && 0x1){
app_log_info("config_flag:%d\n",evt->data.evt_gatt_server_characteristic_status.client_config_flags);
    if(evt->data.evt_gatt_server_characteristic_status.client_config_flags){
        int temperature=getTemperature();
        app_log_info("temperature:%dC\n",temperature);

sc=sl_bt_gatt_server_send_user_read_response(evt->data.handle,gattdb_temperature,0,sizeof(temperature),&tempera
ture,&sent_lent);
        app_assert_status(sc);
```

e. Renvoyer la température périodiquement

QUESTION 20: Observez bien le prototype de sl_sleeptimer_start_periodic_timer_ms(). Comment le timer appelle-t-il votre code quand « c'est l'heure »? Comment lui envoyez-vous vos variables de travail ?

Le timer dispose d'une fonction callback qui sert à répéter périodiquement une partie du code. Par exemple, ici, on veut faire afficher un simple message pour chaque appel de timer. Ce callback dispose de deux paramètres qui permettent de passer des variables, notamment on peut le faire via le paramètre callback_data. Le timer fonctionne tel qu'une interruption.

QUESTION 21 : Quand Notify est activé, démarrez le timer et faites en sorte qu'il appelle votre code. Pour le moment, il doit afficher «Timer step 1», «Timer step 2 », «timer step … » sur la console à chaque coup de timer. Faites des screenshots de votre console.

Nous souhaitons effectuer un affichage de la valeur de Timer par incrément de celle-ci lors que notify est active.
En prenant le programme précédent en rajouter les fonctions servant à contrôler le timer nous obtenons l'extrait de code suivant:

```
if(evt->data.evt_gatt_server_characteristic_status.client_config_flags){
    sl_sleeptimer_start_periodic_timer_ms(&handle,1000,callback,NULL,0,0);
    app_log_info("timer on\n");}
else {
    app_log_info("timer off\n");
    sl_sleeptimer_stop_timer(&handle);
```

Cette fonction fait appel à la callback() que nous devons créer pour qu'il affiche sur la console la valeur de l'incrémentation par le timer pour afficher "Timer step 1", "Timer step 2" etc…

```
void callback(sl_sleeptimer_timer_callback_t *handle, void * data){
  handle = handle;
  data = data;
  x++;
  app_log_info("timer step %d\n",x);
}
```

Sur la console, nous observons que le timer s'active quand notify est active, que la valeur du step s'incrémente. Et quand nous désactivons notify, le timer se désactive.

```
I] timer on
I] timer step 1
I] timer step 2
I] timer step 3
I] timer step 4
I] timer step 5
I] timer step 6
I] timer step 7
I] timer step 8
I] timer step 9
I] timer step 10
I] timer step 11
I] timer step 12
I] timer step 13
I] timer step 14
I] timer step 15
I] timer step 16
I] timer step 17
I] timer step 18
I] timer step 19
I] timer step 20
I] test_notify
I] notify:Coucou!
I] temperature_notify:3149C
I] config_flag:0
I] timer off
```

Ce signal est un signal périodique qui nous permettra d'actualiser la valeur de la température tous les 1s. C'est grâce au Sleep Timer qui nous permet d'avoir ce fonctionnement d'actualisation.

On nous demande cette fois de rajouter en haut de app.c, #define TEMPERATURE_TIMER_SIGNAL (1<<0), et la fonction sl_bt_external_signal(TEMPERATURE_TIMER_SIGNAL) dans la callback de timer:

```
void sl_bt_priority_handle(void);

/**
 * @brief Signal the Bluetooth stack that an external event has happened.
 *
 * Signals can be used to report status changes from interrupt context or from
 * other threads to application. Signals are bits that are automatically cleared
 * after application has been notified.
 *
 * If the Platform Core Interrupt API has been configured to use the
 * CORE_ATOMIC_METHOD_BASEPRI as the implementation method of atomic sections,
 * this function must not be called from an interrupt handler with a priority
 * higher than CORE_ATOMIC_BASE_PRIORITY_LEVEL.
 *
 * @param signals is a bitmask defining active signals that are reported back to
 *   the application by system_external_signal-event.
 * @return SL_STATUS_OK if the operation is successful,
 *   SL_STATUS_NO_MORE_RESOURCE indicating the request could not be processed
 *   due to resource limitation at the moment, or SL_STATUS_INVALID_STATE when
 *   the on-demand start feature is used and the stack is currently stopped.
 */
sl_status_t sl_bt_external_signal(uint32_t signals);
```

Quel diantre est-ce ce signal ? Pourquoi doit-on faire ceci ?

Ce signal nous permettra de générer un signal externe permettant de simuler un changement d'état. Nous devons effectuer cela pour actualiser la valeur de la température sur l'application EFR Connect car grâce à ce changement nous pourrons afficher à nouveau la nouvelle valeur.

Question 22 : Modifiez votre code pour renvoyer la température.

En modifiant le code avec ce que nous avons vu précédemment, et en ajoutant la prise de la mesure de la température ainsi que son affichage sur l'application EFR Connect.

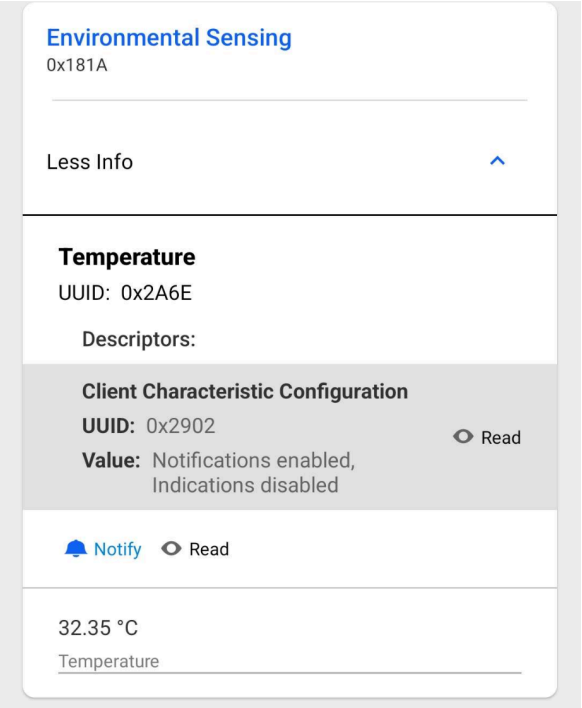
```
#define TEMPERATURE_TIMER_SIGNAL (1<<0)

void callback(sl_sleeptimer_timer_callback_t *handle, void * data);
sl_sleeptimer_timer_handle_t handle;
int x=0;
uint8_t connection_temp;
uint16_t characteristic_temp;
```

```
void callback(sl_sleeptimer_timer_callback_t *handle, void * data){
  handle = handle;
  data = data;
  sl_bt_external_signal(TEMPERATURE_TIMER_SIGNAL);
}
```

```
case (sl_bt_evt_system_external_signal_id):
    switch (evt->data.evt_system_external_signal.extsignals){
      case(TEMPERATURE_TIMER_SIGNAL):
        app_log_info("external_signal\n");
        app_log_info("temp=%d\n",getTemperature());
        int temp=getTemperature();
        sl_bt_gatt_server_send_notification(connection_temp,characteristic_temp,sizeof(temp),(const uint8_t*)&temp);
        break;
      default:
        break;
          }
```

Nous devons créer une nouvelle condition afin de vérifier si notre signal externe généré par le timer est identifié et si cette valeur correspond à celle que nous générons pour la température alors nous allons relire et re afficher la valeur de la température.
Nous observons donc sur la console que nous affichons plusieurs valeurs de la température automatiquement sans que nous ayons à appuyer sur un bouton.

```
[I] external_signal
[I] temp=3238
[I] external_signal
[I] temp=3238
[I] external_signal
[I] temp=3238
[I] external_signal
[I] temp=3240
[I] external_signal
```



Sur l'application EFR Connect, le changement de température s'effectue bien toute les secondes.

VI. Ajouter un service Automation IO

a. Repérer un accès en écriture sur une caractéristique

QUESTION 23 : Comment lit-on la valeur écrite par une caractéristique ?

La caractéristique peut être lue dans le champ de la structure sl_bt_gatt_server_user_write_request_s.

```
PACKSTRUCT( struct sl_bt_evt_gatt_server_user_write_request_s
{
  uint8_t    connection;     /**< Connection handle */
  uint16_t   characteristic; /**< GATT characteristic handle. This value is
                                  normally received from the gatt_characteristic
                                  event. */
  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 */
});
```

La variable value donne la valeur de la caractéristique. Lors d'un accès en écriture, il faut alors récupérer la valeur en pointant sur la variable value de la structure sl_bt_gatt_server_user_write_request_s.

b. Utiliser une API de LEDs simple

On observe les différents fichiers d'implémentation de LEDs. Ceux-ci décrivent comment sont instanciés les différentes leds notamment par des instances ayant chacun leur propres adresses, état, etc.

led.h

```c
#ifndef SL_SIMPLE_LED_INSTANCES_H
#define SL_SIMPLE_LED_INSTANCES_H

#include "sl_simple_led.h"

extern const sl_led_t sl_led_led0;

extern const sl_led_t *sl_simple_led_array[];

#define SL_SIMPLE_LED_COUNT 1
#define SL_SIMPLE_LED_INSTANCE(n) (sl_simple_led_array[n])

void sl_simple_led_init_instances(void);

#endif // SL_SIMPLE_LED_INIT_H
```

Ce fichier permet de déclarer les constantes, structures et la fonction décrite dans les autres fichiers.

led.c

```c
#include "sl_simple_led.h"
#include "em_gpio.h"
#include "sl_simple_led_led0_config.h"

sl_simple_led_context_t simple_led0_context = {
  .port = SL_SIMPLE_LED_LED0_PORT,
  .pin = SL_SIMPLE_LED_LED0_PIN,
  .polarity = SL_SIMPLE_LED_LED0_POLARITY,
};

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,
};

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

void sl_simple_led_init_instances(void)
{
  sl_led_init(&sl_led_led0);
}
```

Ce fichier décrit précisément comment sont structurés les adresses de led0, les ports, les pins, les polarités, l'initialisation de led, les changements d'état, récupération de l'état de led, l'initialisation des instances de led.

simple_led.h

```
#define SL_SIMPLE_LED_POLARITY_ACTIVE_LOW  0U ///< LED Active polarity Low
#define SL_SIMPLE_LED_POLARITY_ACTIVE_HIGH 1U ///< LED Active polarity High

/*****************************   DATA TYPES   *********************************/

typedef uint8_t sl_led_polarity_t;    ///< LED GPIO polarities (active high/low)

/// A Simple LED instance
typedef struct {
  GPIO_Port_TypeDef port;             ///< LED port
  uint8_t           pin;              ///< LED pin
  sl_led_polarity_t polarity;         ///< Initial state of LED
} sl_simple_led_context_t;

/*****************************   PROTOTYPES   *********************************/

/* * Initialize the simple LED driver.
sl_status_t sl_simple_led_init(void *led_handle);

/* * Turn on a simple LED.
void sl_simple_led_turn_on(void *led_handle);

/* * Turn off a simple LED.
void sl_simple_led_turn_off(void *led_handle);

/* * Toggle a simple LED.
void sl_simple_led_toggle(void *led_handle);

/* * Get the current state of the simple LED.
sl_led_state_t sl_simple_led_get_state(void *led_handle);
```

Ce fichier déclare des fonctions simples de led.

QUESTION 24 : Dans app.c, à quel moment devriez-vous appeler sl_simple_led_init_instances() ? Rajoutez l'appel nécessaire.

L'appel à sl_simple_led_init_instances() se fait au début donc dans l'app_init.

```
SL_WEAK void app_init(void)
{
  /////////////////////////////////////////////////////////////////////////////
  // Put your additional application init code here!                         //
  app_log_info("%s\n",__FUNCTION__);
  sl_simple_led_init_instances();
  // This is called once during start-up.                                    //
  /////////////////////////////////////////////////////////////////////////////
}
```

QUESTION 25 : Combien y a t'il de LEDs dans l'instance? Montrez-le de deux manières différentes.

La première manière est de constater dans le fichier led.h. Donc 1 LED dans chaque instance (voir capture précédemment) puiqu'on voit le define SL_SIMPLE_COUNT à 1.
La deuxième manière est de voir que c'est spécifiquement la led0 qui est configuré et c'est tout, on a importé un fichier de configuration exprès mais pas d'autres fichiers.

QUESTION 26 : Comment accédez-vous simplement à chacune des instances?

On sait que *sl_simple_led_array[n] est un pointeur sur l'adresse de la led0 pour une instance n. Il suffit de choisir de définir n pour l'instance que l'on souhaite.

QUESTION 27 : Dans app.c, quand un client écrit dans la caractéristique Digital, assurez-vous de refléter l'état sur toutes les instances de simple LEDs:

- Write Digital Inactive → Toutes les simple LEDs disponibles sont éteintes.
- Write Digital Active→ Toutes les simple LEDs disponibles sont allumées.

Pour un Write Digital Inactive, il faut utiliser sl_simple_led_turn_off, il est utilisable sur sl_led_led0.turn_off.
Pour un Write Digital Active, il faut utiliser sl_simple_led_turn_on, il est utilisable sur sl_led_led0.turn_on.

On propose donc : Write Inactive ->

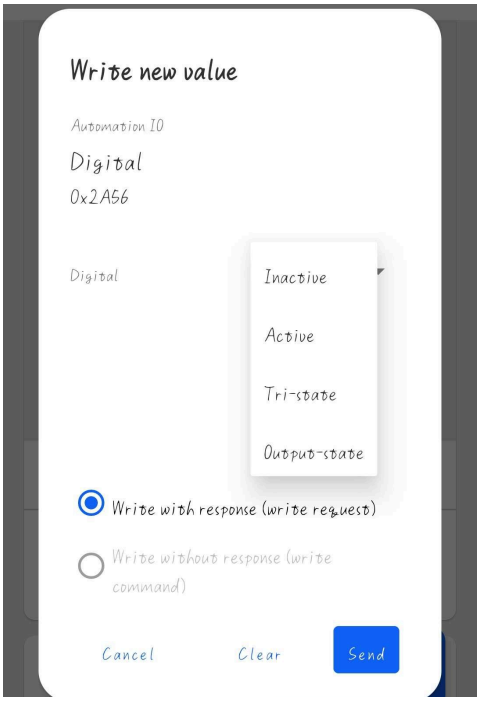
```
for (n=0; n<INSTANCE_MAX; n++) {
(*sl_simple_led_array[n]).turn_off;}
```

Write Active->

```
for (n=0; n<INSTANCE_MAX; n++) {
(*sl_simple_led_array[n]).turn_on;}
```

c. 50 nuances de Write

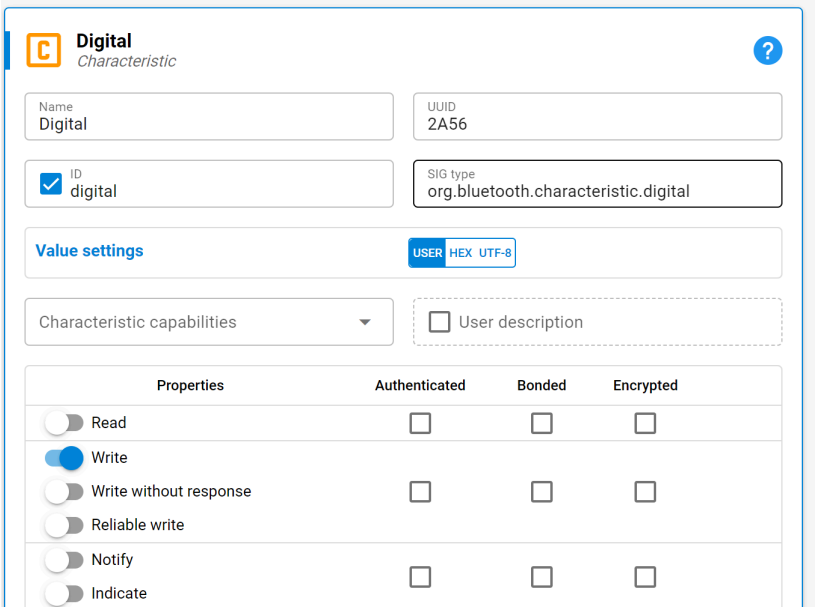
QUESTION 28 : Regardez ce que EFR Connect vous propose quand vous faites un Write dans la caractéristique Digital. Que remarquez-vous?



On peut s'attendre à trouver un état actif et un état inactif. En réalité, on trouve les caractéristiques du Digital sont Inactive, Active, Tri-state, Output-state. Et en plus de cela,

nous observons qu'il existe une variante du write que nous n'avons pas la possibilité de la sélectionner.

QUESTION 29 : Regardez de nouveau la table GATT de votre caractéristique Digital, notamment ses Properties. Qu'observez vous ? Qu'est-ce que cela vous rappelle?



Si on regarde un peu les propriétés, au niveau des lignes 2 à 4, on voit les propriétés d'écriture. On peut faire le rapprochement avec les caractéristiques Digital vues à la question précédente. Le Write correspondrait à Active, Le Write without response correspondrait à Output-State (1 aller donc sortie uniquement) et le Reliable Write correspondrait à Tri-state (plusieurs rappels)

QUESTION 30 : A votre avis, pourquoi un Write sur Digital ne marche qu'une fois puis EFR Connect se bloque et se déconnecte? Que manque t'il dans votre application?

On fait une demande de write request mais sans répondre. En effet, le Write a besoin de recevoir une sorte d'acquittement de bonne réception. Il manque donc la réponse au capteur. C'est pour cela que l'exécution ne marche qu'une seule fois et puis EFR Connect se bloque et se déconnecte.

QUESTION 31 : Confrontez votre hypothèse avec le champ sl_bt_evt_gatt_server_user_write_request_t.att_opcode. Observez les valeurs que vous rencontrez parmi celles possibles.

```
PACKSTRUCT( struct sl_bt_evt_gatt_server_user_write_request_s
{
  uint8_t    connection;     /**< Connection handle */
  uint16_t   characteristic; /**< GATT characteristic handle. This value is
                                  normally received from the gatt_characteristic
                                  event. */
  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 */
});

typedef struct sl_bt_evt_gatt_server_user_write_request_s sl_bt_evt_gatt_server_user_write_request_t;
```

On peut voir que la variable att_opcode nous intéresse le plus puisqu'elle informe sur la réception de la valeur.

```
typedef enum
{
  sl_bt_gatt_read_by_type_request      = 0x8,  /**< (0x8) Read by type request */
  sl_bt_gatt_read_by_type_response     = 0x9,  /**< (0x9) Read by type response */
  sl_bt_gatt_read_request              = 0xa,  /**< (0xa) Read request */
  sl_bt_gatt_read_response             = 0xb,  /**< (0xb) Read response */
  sl_bt_gatt_read_blob_request         = 0xc,  /**< (0xc) Read blob request */
  sl_bt_gatt_read_blob_response        = 0xd,  /**< (0xd) Read blob response */
  sl_bt_gatt_read_multiple_request     = 0xe,  /**< (0xe) Read multiple request */
  sl_bt_gatt_read_multiple_response    = 0xf,  /**< (0xf) Read multiple response */
  sl_bt_gatt_write_request             = 0x12, /**< (0x12) Write request */
  sl_bt_gatt_write_response            = 0x13, /**< (0x13) Write response */
  sl_bt_gatt_write_command             = 0x52, /**< (0x52) Write command */
  sl_bt_gatt_prepare_write_request     = 0x16, /**< (0x16) Prepare write request */
  sl_bt_gatt_prepare_write_response    = 0x17, /**< (0x17) Prepare write
                                                   response */
  sl_bt_gatt_execute_write_request     = 0x18, /**< (0x18) Execute write request */
  sl_bt_gatt_execute_write_response    = 0x19, /**< (0x19) Execute write
                                                   response */
  sl_bt_gatt_handle_value_notification = 0x1b, /**< (0x1b) Notification */
  sl_bt_gatt_handle_value_indication   = 0x1d  /**< (0x1d) Indication */
} sl_bt_gatt_att_opcode_t;
```

On peut ainsi voir que att_opcode peut renseigner si l'écriture est demandé (write request) et si elle est répondu (write response).

QUESTION 32 : Selon ce champ, quel appel devez-vous rajouter dans votre code pour que EFR Connect soit heureux et vous aussi ? Indice : la réponse est dans sl_bt_api.h

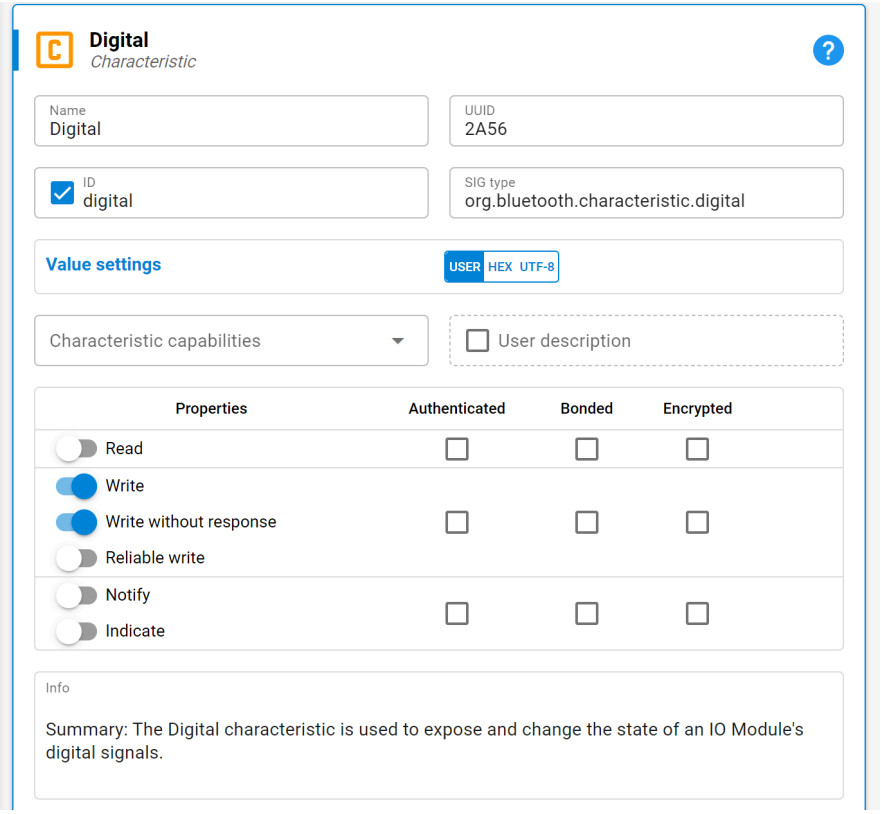
D'après ce champs, nous devons renvoyer une réponse avec la fonction sl_bt_evt_gatt_server_user_write_response().

```
case (sl_bt_evt_gatt_server_user_write_request_id):
   switch(evt->data.evt_gatt_server_user_write_request.characteristic){
      case (gattdb_digital):
        app_log_info("digital\n");
        sl_led_led0.turn_on(sl_led_led0.context); //led actif
        sl_bt_gatt_server_send_user_write_response(connection_digital,characteristic_digital,0);
        app_log_info("response write\n");
        }
```

```
[I] digital
[I] response write
```

Avec la modification apportée, nous pouvons renvoyer une réponse et donc faire fonctionner plus d'une fois la fonction sans que EFR Connect se bloque et se déconnecte.

QUESTION 33 : Dans votre GATT, activez la property juste en dessous de Write. Recompilez, reflashez. Observez maintenant ce que EFR Connect vous propose quand vous voulez écrire dans la caractéristique Digital.



EFR Connect nous propose après que nous avons activer la properties Write without response, la possibilité d'écrire en mode commande donc sans réponse.

QUESTION 34 : Dans votre code, rajoutez la condition nécessaire pour distinguer et traiter les deux options de Write adéquatement.

Par manque de temps, nous n'avons pas pu vérifier l'entièreté du bon fonctionnement de notre code, mais au vu de que nous avons pu rencontrer au cours de ce TP, nous pouvons proposer le code suivant afin de distinguer et de traiter les deux options de Write.

```
case (sl_bt_evt_gatt_server_user_write_request_id):
   switch(evt->data.evt_gatt_server_user_write_request.characteristic){
     case (gattdb_digital):
       if (evt->data.evt_gatt_server_user_write_request.att_opcode && sl_bt_gatt_write_command){
           app_log_info("digital sans\n");
            sl_led_led0.toggle(sl_led_led0.context);} //led actif
       else if (evt->data.evt_gatt_server_user_write_request.att_opcode && sl_bt_gatt_write_request){
            app_log_info("digital\n");
            sl_led_led0.toggle(sl_led_led0.context); //led actif
            sl_bt_gatt_server_send_user_write_response(connection_digital,characteristic_digital,0);
            app_log_info("response write\n");}
      default:
        break;
       }
```

Explication:
Nous avons vu que précédemment sl_bt_evt_gatt_server_user_write_request_t.att_opcode pouvais prendre certaine valeur en fonction de si notre écriture est avec ou sans réponse. Si nous avons une écriture avec réponse, nous notre valeur doit être de 0x12, sinon le cas sans réponse, nous devons avoir une valeur de 0x52.