eBook Gratuit

APPRENEZ bluetooth

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#bluetooth

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: bluetooth

It is an unofficial and free bluetooth ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official bluetooth.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Bluetooth

Remarques

Bluetooth est une norme industrielle pour la transmission de données sans fil entre appareils sur de courtes distances. Il a été défini pour la première fois dans les années 1990 par le Bluetooth Special Interest Group (SIG), dans la norme IEEE 802.15.1. La transmission de données sans connexion et orientée connexion est possible, sur un ou plusieurs périphériques, dans un Ad-hoc ou un piconet.

Le nom Bluetooth vient du roi danois Harald Blauzahn, Blauzahn signifiant «dent bleue» en anglais. La partie principale du développement initial a été réalisée par le professeur néerlandais Jaap Haartsen pour la société Ericsson. Plus tard, Intel et Nokia ont été les principaux contributeurs.

Les fréquences utilisées sont la bande ISM sans licence, comprise entre 2,402 GHz et 2,480 GHz, et peuvent donc être utilisées sans autorisation dans le monde entier. Des interférences avec les réseaux Wifi, les téléphones sans fil ou les micro-ondes, tous fonctionnant dans la même bande ISM, sont possibles.

Examples

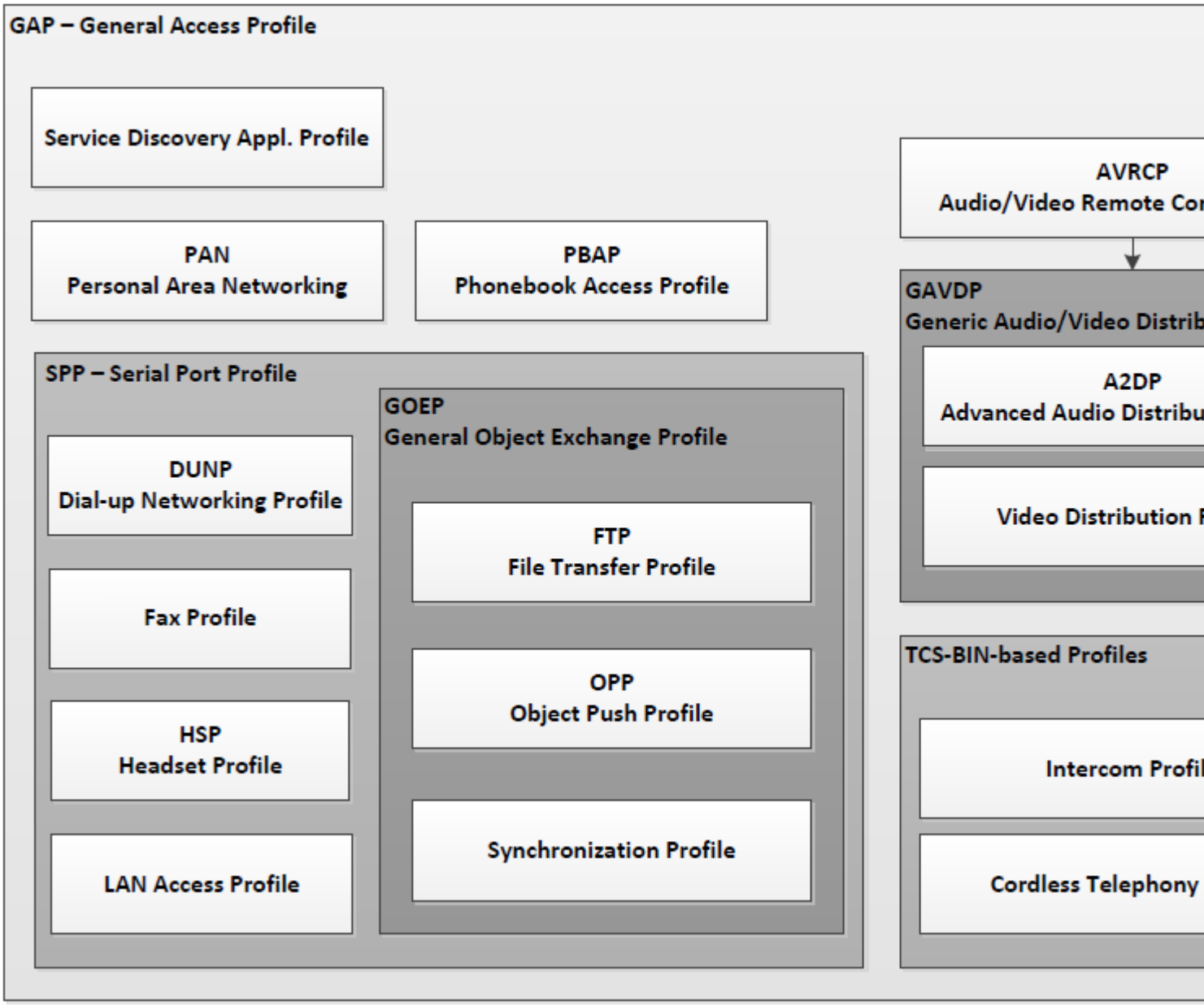
Installation ou configuration

Instructions détaillées sur la configuration ou l'installation de Bluetooth.

Les profils

La spécification Bluetooth contient plusieurs spécifications de profil. Un profil décrit comment utiliser et implémenter une fonction.

Ils peuvent dépendre les uns des autres, voici une présentation de base des dépendances de profil les plus courantes



Tous les profils peuvent être trouvés sur BT SIG , sachez que différentes versions peuvent contenir des fonctionnalités différentes. Notez également que certains des profils contiennent plusieurs catégories, qui sont parfois facultatives, alors assurez-vous que votre appareil prend en charge la catégorie en question. Voici quelques-uns des *profils de smartphone* les plus courants et leurs spécifications

A2DP - Profil de distribution audio avancée
Le profil de distribution audio avancée (A2DP) définit les protocoles et les procédures qui permettent de distribuer du contenu audio de haute qualité en mono ou en stéréo sur des canaux ACL. Un cas d'utilisation typique est la diffusion de contenu musical d'un lecteur de musique stéréo vers un casque ou des haut-parleurs. Les données audio sont compressées dans un format approprié pour une utilisation efficace de la bande passante limitée.
Dépendances: GAVDP, GAP

AVRCP - Profil de contrôle à distance audio / vidéo
Le profil de contrôle à distance audio / vidéo (AVRCP) définit les fonctions et procédures requises

pour assurer l'interopérabilité entre les périphériques Bluetooth dotés de fonctions de contrôle audio / vidéo dans les scénarios de distribution audio / vidéo. Ce profil adopte le modèle de périphérique AV / C et le format de commande pour les messages de contrôle, et ces messages sont transportés par le protocole de transport de contrôle audio / vidéo (AVCTP). Dans ce profil, le contrôleur traduit l'action de l'utilisateur détectée sur le signal de contrôle A / V, puis le transmet à un périphérique Bluetooth distant. De plus, le profil utilise des extensions spécifiques à Bluetooth pour prendre en charge le transfert de métadonnées liées au contenu à transférer entre les périphériques Bluetooth. La télécommande décrite dans ce profil est conçue spécifiquement pour le contrôle A / V.
Dépendances: GAP

HFP - Profil mains libres
Ce document définit les protocoles et les procédures qui doivent être utilisés par les dispositifs implémentant le profil mains libres. Les exemples les plus courants de ces dispositifs sont les unités mains libres embarquées utilisées avec les téléphones cellulaires ou les casques sans fil portables. Le profil définit comment deux appareils prenant en charge le profil mains libres doivent interagir les uns avec les autres sur une base point à point. Une implémentation du profil mains libres permet généralement à un casque ou à une unité mains-libres intégrée de se connecter, sans fil, à un téléphone cellulaire pour agir comme mécanisme d’entrée et de sortie audio du téléphone cellulaire et effectué sans accès au téléphone réel.
Dépendances: SPP, GAP

HSP - Profil casque
Ce profil de casque définit les protocoles et procédures à utiliser par les périphériques nécessitant une connexion audio en duplex intégral combinée à des commandes de contrôle de périphérique minimales. Les exemples les plus courants de ces appareils sont les casques d’écoute, les ordinateurs personnels, les assistants numériques personnels et les téléphones cellulaires, bien que la plupart des téléphones cellulaires préfèrent utiliser un profil plus évolué tel que le profil mains libres. Le kit oreillette peut être connecté sans fil pour agir en tant que mécanisme d'entrée et de sortie audio du périphérique, fournissant un son en duplex intégral.
Dépendances: SPP, GAP

PBAP - Profil d'accès au répertoire
Le profil d'accès au répertoire téléphonique (PBAP) définit les protocoles et les procédures à utiliser par les périphériques pour la récupération des objets du répertoire téléphonique. Il est basé sur un modèle d'interaction client-serveur où le périphérique client extrait les objets du répertoire téléphonique du périphérique serveur. Ce profil est spécialement conçu pour le cas d'utilisation mains libres (implémenté en combinaison avec le «profil mains libres» ou le «profil d'accès SIM»). Il offre de nombreuses fonctionnalités qui permettent une gestion avancée des objets de l'annuaire, selon les besoins de l'environnement automobile. En particulier, il est beaucoup plus riche que le profil Push d'objet (qui pourrait être utilisé pour pousser une entrée d'annuaire téléphonique formatée vCard d'un périphérique à un autre). Ce profil peut également être appliqué à d'autres cas d'utilisation où un périphérique client doit extraire des objets du répertoire téléphonique d'un périphérique serveur. Notez toutefois que ce profil ne permet que la consultation d'objets d'annuaire (lecture seule). Il est impossible de modifier le contenu de l'objet d'annuaire téléphonique d'origine (lecture / écriture).
Dépendances: GOEP, SPP, GAP

Lire Démarrer avec Bluetooth en ligne: https://riptutorial.com/fr/bluetooth/topic/4846/demarrer-avec-bluetooth

Chapitre 2: Commencez avec Bluetooth LE sous Windows

Remarques

Documentation

- Advertisement - Représentation d'une charge utile de publicité Bluetooth LE.
- Advertisement Publisher - Gère l'envoi des publicités Bluetooth LE.
- Annonce Watcher - Gère la surveillance des publicités Bluetooth LE.

Remarques

- Windows 10 ne peut agir qu'en mode central, il ne peut donc se connecter qu'aux périphériques prenant en charge le mode périphérique. Pour cette raison, deux périphériques Windows 10 ne peuvent pas se connecter via Bluetooth LE.
- Windows 10 doit être associé à un périphérique Bluetooth LE pour pouvoir s'y connecter.

Examples

La configuration initiale

Pour utiliser une fonctionnalité Bluetooth sur une application Universal Windows Platform, vous devez vérifier la capacité `Bluetooth` dans `Package.appxmanifest` .

1. Ouvrir `Package.appxmanifest`
2. Accédez à l'onglet `Capabilities`
3. Trouvez `Bluetooth` à gauche et cochez la case à côté

Créer une publicité Bluetooth LE

Cet exemple montre comment publier une charge utile personnalisée à partir d'un périphérique Windows 10 au premier plan. La charge utile utilise une société créée (identifiée 0xFFFE) et annonce la chaîne `Hello World` dans la publicité.

```
BluetoothLEAdvertisementPublisher publisher = new BluetoothLEAdvertisementPublisher();

// Add custom data to the advertisement
var manufacturerData = new BluetoothLEManufacturerData();
manufacturerData.CompanyId = 0xFFFE;

var writer = new DataWriter();
writer.WriteString("Hello World");

// Make sure that the buffer length can fit within an advertisement payload (~20 bytes).
// Otherwise you will get an exception.
manufacturerData.Data = writer.DetachBuffer();
```

```
// Add the manufacturer data to the advertisement publisher:
publisher.Advertisement.ManufacturerData.Add(manufacturerData);

publisher.Start();
```

> Remarque: Ceci est uniquement pour la publicité au premier plan (lorsque l'application est ouverte).

Écoutez une publicité Bluetooth LE

Écoute générale

Cet exemple montre comment écouter une publicité spécifique.

```
BluetoothLEAdvertisementWatcher watcher = new BluetoothLEAdvertisementWatcher();

// Use active listening if you want to receive Scan Response packets as well
// this will have a greater power cost.
watcher.ScanningMode = BluetoothLEScanningMode.Active;

// Register a listener, this will be called whenever the watcher sees an advertisement.
watcher.Received += OnAdvertisementReceived;

watcher.Start();
```

Filtre de publicité: écoute pour une publicité spécifique

Parfois, vous voulez écouter une publicité spécifique. Dans ce cas, écoutez une publicité contenant une charge utile avec une société constituée (identifiée 0xFFFE) et contenant la chaîne Hello World dans la publicité. Ceci peut être associé à l'exemple *Créer une annonce Bluetooth LE* pour avoir une publicité machine Windows et une autre écoute.

> Remarque: Veillez à définir ce filtre de publicité avant de démarrer votre observateur!

```
var manufacturerData = new BluetoothLEManufacturerData();
manufacturerData.CompanyId = 0xFFFE;

// Make sure that the buffer length can fit within an advertisement payload (~20 bytes).
// Otherwise you will get an exception.
var writer = new DataWriter();
writer.WriteString("Hello World");
manufacturerData.Data = writer.DetachBuffer();

watcher.AdvertisementFilter.Advertisement.ManufacturerData.Add(manufacturerData);
```

Filtre de signal: écoute des publicités proximales

Parfois, vous souhaitez uniquement déclencher votre observateur lorsque la publicité de l'appareil est à portée de main. Vous pouvez définir votre propre plage, notez simplement que les valeurs normales sont comprises entre 0 et -128.

```
// Set the in-range threshold to -70dBm. This means advertisements with RSSI >= -70dBm
```

```
// will start to be considered "in-range" (callbacks will start in this range).
watcher.SignalStrengthFilter.InRangeThresholdInDBm = -70;

// Set the out-of-range threshold to -75dBm (give some buffer). Used in conjunction
// with OutOfRangeTimeout to determine when an advertisement is no longer
// considered "in-range".
watcher.SignalStrengthFilter.OutOfRangeThresholdInDBm = -75;

// Set the out-of-range timeout to be 2 seconds. Used in conjunction with
// OutOfRangeThresholdInDBm to determine when an advertisement is no longer
// considered "in-range"
watcher.SignalStrengthFilter.OutOfRangeTimeout = TimeSpan.FromMilliseconds(2000);
```

Rappels

```
watcher.Received += OnAdvertisementReceived;
watcher.Stopped += OnAdvertisementWatcherStopped;

private async void OnAdvertisementReceived(BluetoothLEAdvertisementWatcher watcher,
BluetoothLEAdvertisementReceivedEventArgs eventArgs)
{
    // Do whatever you want with the advertisement

    // The received signal strength indicator (RSSI)
    Int16 rssi = eventArgs.RawSignalStrengthInDBm;
}


private async void OnAdvertisementWatcherStopped(BluetoothLEAdvertisementWatcher watcher,
BluetoothLEAdvertisementWatcherStoppedEventArgs eventArgs)
{
    // Watcher was stopped
}
```

Remarque: Ceci est uniquement pour l'écoute au premier plan.

Distance d'évaluation basée sur RSSI d'une publicité Bluetooth LE

Lorsque le rappel de votre Bluetooth LE Watcher est déclenché, le paramètre eventArgs inclut une valeur RSSI indiquant la force du signal reçu (la

```
private async void OnAdvertisementReceived(BluetoothLEAdvertisementWatcher watcher,
BluetoothLEAdvertisementReceivedEventArgs eventArgs)
{
    // The received signal strength indicator (RSSI)
    Int16 rssi = eventArgs.RawSignalStrengthInDBm;
}
```

Cela peut être grossièrement traduit en distance, mais ne doit pas être utilisé pour mesurer les vraies distances car chaque radio individuelle est différente. Différents facteurs environnementaux peuvent rendre la mesure difficile à évaluer (comme les murs, les caisses autour de la radio ou même l'humidité de l'air).

Une alternative à l'évaluation de la distance pure consiste à définir des "seaux". Les radios ont tendance à indiquer de 0 à -50 DBm lorsqu'elles sont très proches, de -50 à -90 lorsqu'elles sont

éloignées, et inférieures à -90 lorsqu'elles sont éloignées. L'essai et l'erreur sont les meilleurs pour déterminer ce que vous voulez que ces compartiments soient pour votre application.

Lire Commencez avec Bluetooth LE sous Windows en ligne:
https://riptutorial.com/fr/bluetooth/topic/5553/commencez-avec-bluetooth-le-sous-windows

Chapitre 3: Commencez avec Bluetooth sur le Web

Remarques

Sources:

- https://developers.google.com/web/updates/2015/07/interact-with-ble-devices-on-the-web
- https://googlechrome.github.io/samples/web-bluetooth/index.html

Examples

Lire le niveau de la batterie depuis un périphérique Bluetooth proche (readValue)

```
function onButtonClick() {

  navigator.bluetooth.requestDevice({filters: [{services: ['battery_service']}]})
  .then(device => {
    // Connecting to GATT Server...
    return device.gatt.connect();
  })
  .then(server => {
    // Getting Battery Service...
    return server.getPrimaryService('battery_service');
  })
  .then(service => {
    // Getting Battery Level Characteristic...
    return service.getCharacteristic('battery_level');
  })
  .then(characteristic => {
    // Reading Battery Level...
    return characteristic.readValue();
  })
  .then(value => {
    let batteryLevel = value.getUint8(0);
    console.log('> Battery Level is ' + batteryLevel + '%');
  })
  .catch(error => {
    console.log('Argh! ' + error);
  });
}
```

Réinitialiser l'énergie dépensée par un périphérique Bluetooth proche (writeValue)

```
function onButtonClick() {

  navigator.bluetooth.requestDevice({filters: [{services: ['heart_rate']}]})
```

```
  .then(device => {
    // Connecting to GATT Server...
    return device.gatt.connect();
  })
  .then(server => {
    // Getting Heart Rate Service...
    return server.getPrimaryService('heart_rate');
  })
  .then(service => {
    // Getting Heart Rate Control Point Characteristic...
    return service.getCharacteristic('heart_rate_control_point');
  })
  .then(characteristic => {
    // Writing 1 is the signal to reset energy expended.
    let resetEnergyExpended = new Uint8Array([1]);
    return characteristic.writeValue(resetEnergyExpended);
  })
  .then(_ => {
    console.log('> Energy expended has been reset.');
  })
  .catch(error => {
    console.log('Argh! ' + error);
  });
}
```

Lire Commencez avec Bluetooth sur le Web en ligne:
https://riptutorial.com/fr/bluetooth/topic/4936/commencez-avec-bluetooth-sur-le-web

Chapitre 4: Commencez avec la pile BLE de TI

Examples

Connexion aux appareils esclaves BLE

introduction

Les SoC de la série CC26XX de Texas Instruments (TI) sont des microcontrôleurs sans fil facilement disponibles ciblant les applications Bluetooth Low Energy (BLE). En plus des MCU, TI propose une pile de logiciels à part entière qui fournit les API nécessaires et des exemples de codes pour aider les développeurs à démarrer rapidement avec la chaîne d'outils. Cependant, pour les débutants, la question est toujours de savoir par où commencer devant une longue liste de documents de référence et de codes. Cette note vise à enregistrer les étapes nécessaires pour lancer le premier projet.

Le profil de périphérique simple est l'exemple «Hello World» de la pile BLE, où la MCU agit comme un périphérique BLE pour les hôtes en amont ou les clients de services BLE, tels que les PC et les smartphones. Les applications courantes du monde réel comprennent: casque Bluetooth, capteur de température Bluetooth, etc.

Avant de commencer, nous devons d'abord rassembler des outils logiciels et matériels de base pour la programmation et le débogage.

1. Pile BLE

   Téléchargez et installez le BLE-STACK-2-2-0 de TI sur le site officiel. Supposons qu'il soit installé à l'emplacement par défaut 'C: \ ti'.

2. IDE - il y a deux options:

- IAR Embedded Workbench pour ARM. Ceci est un outil commercial avec une période d'évaluation gratuite de 30 jours.
- Code Composer Studio (CCS) de TI. IDE officiel de TI et offre une licence gratuite. Dans cet exemple, nous utiliserons CCS V6.1.3

3. Outil de programmation matérielle

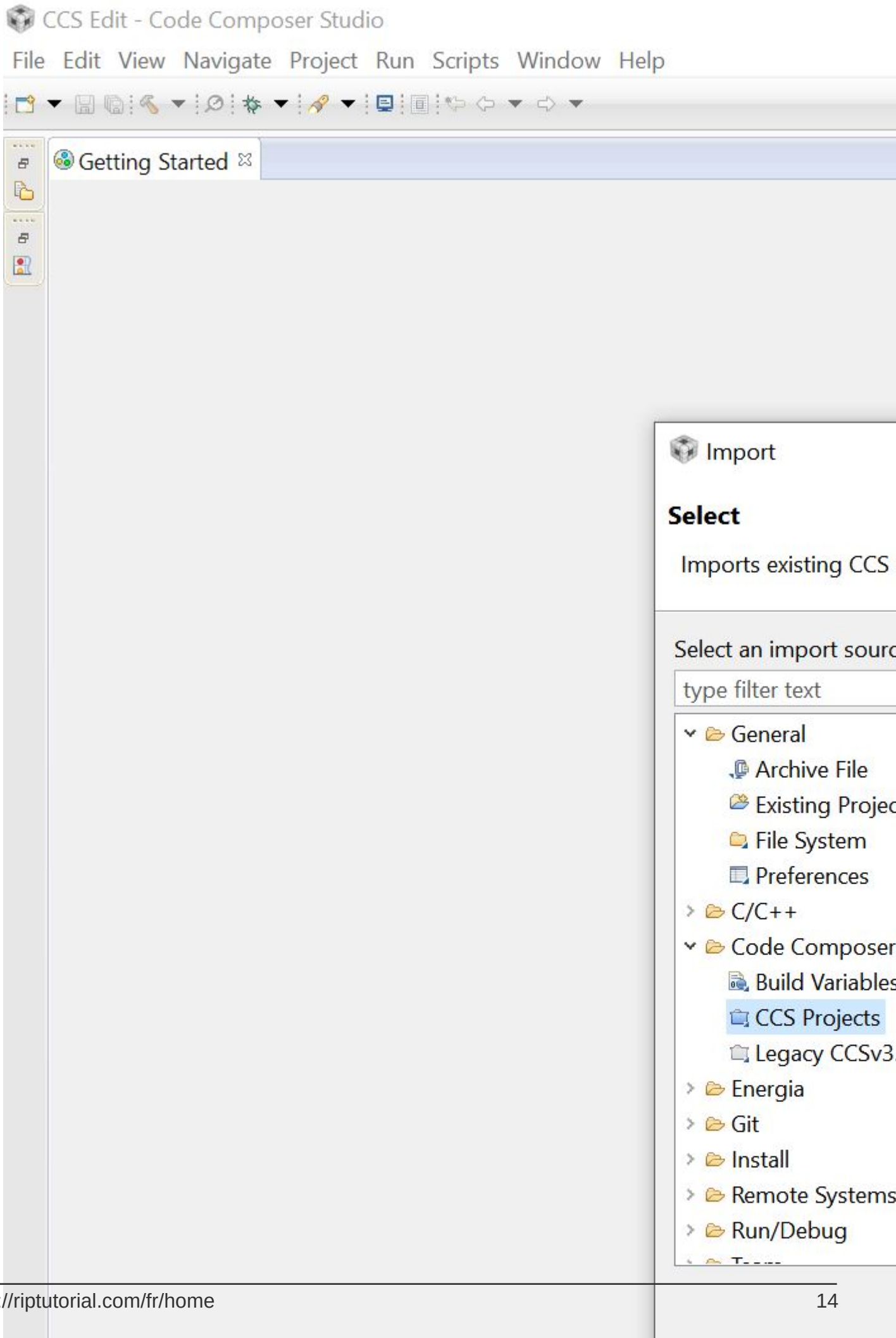
   Recommandez le périphérique JTAG à interface USB XDS100 de TI.

Importer un exemple de projet dans CCS

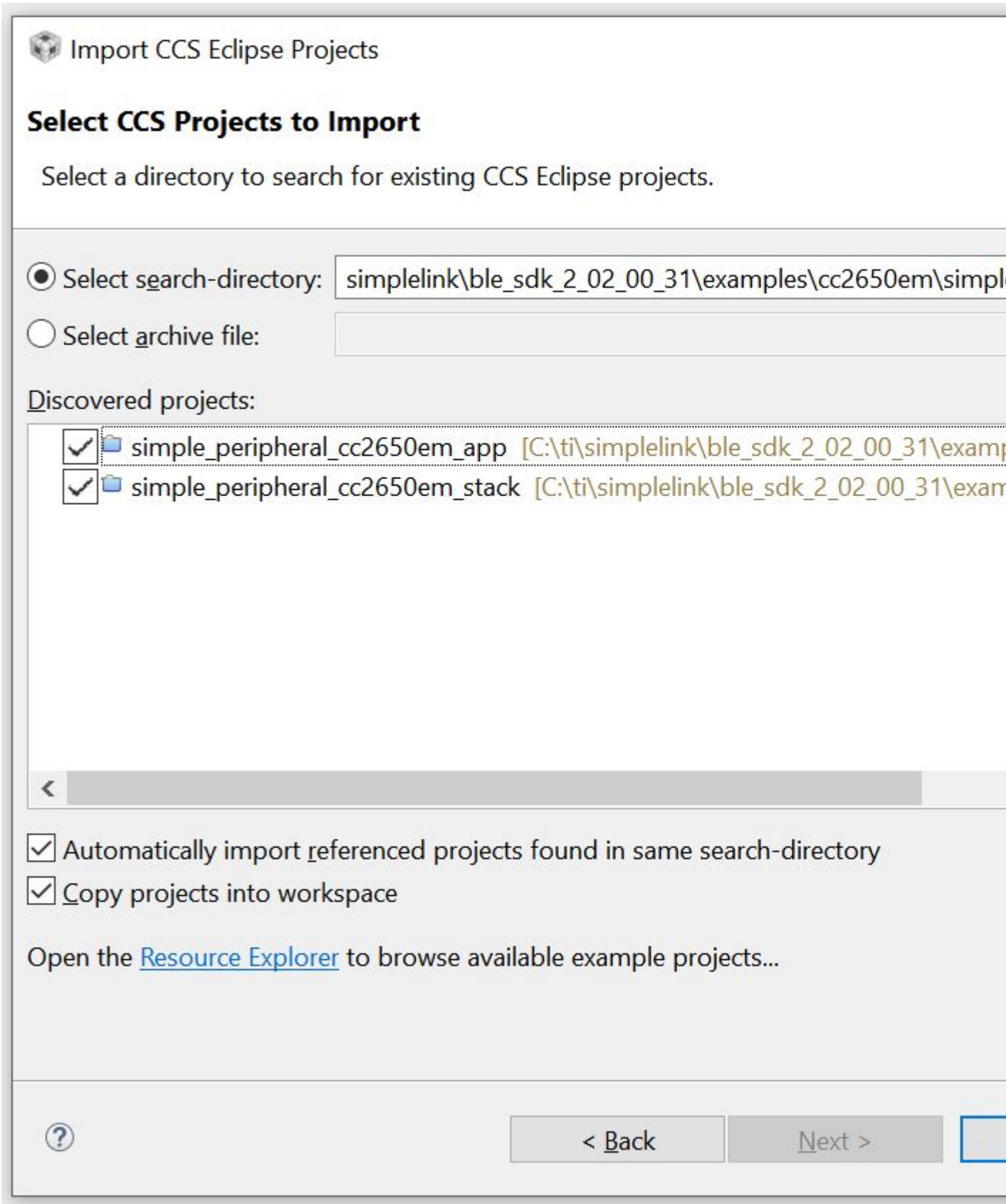
L'exemple de code Simple Peripheral Profile est fourni avec l'installation BLE-Stack. Suivez les

étapes ci-dessous pour importer cet exemple de projet dans CCS.

1. Démarrez CCS, créez un dossier d'espace de travail. Puis Fichier-> Importer. Sous "Sélectionner une source d'importation", sélectionnez l'option "Code Compose Studio -> CCS Projects" et cliquez sur "Suivant".


CCS Edit - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Getting Started
Import
Select
Imports existing CCS
Select an import sourc
type filter text
General
Archive File
Existing Projec
File System
Preferences
C/C++
Code Composer
Build Variables
CCS Projects
Legacy CCSv3
Energia
Git
Install
Remote Systems
Run/Debug

2. Accédez à 'C: \ ti \ simplelink \ ble_sdk_2_02_00_31 \ examples \ cc2650em \ simple_peripheral \ ccs'. Deux projets seront découverts. Sélectionnez tout et cochez les deux options ci-dessous. Cliquez ensuite sur "Terminer". En copiant des projets dans l'espace de travail, vous laissez le paramètre de projet d'origine inchangé pour toutes les modifications suivantes.



L'exemple de profil périphérique simple inclut deux projets:

- simple_peripheral_cc2650em_app
- simple_peripheral_cc2650em_stack

«cc2650em» est le nom de code de la carte d'évaluation cc2650 de TI. Le projet _stack comprend les codes et le binaire de BEL-Stack-2-2-0 de TI, qui gère la publicité Bluetooth, l'établissement de liaison, la synchronisation de fréquence, etc. C'est la partie du code qui est relativement stable et ne veut pas être touché par les développeurs la plupart du temps. Le projet _app est l'endroit où les développeurs implémentent leurs propres tâches et services BLE.

Construire et Télécharger

Cliquez sur les menus "Projet-> Construire tout" pour construire les deux projets. Si le compilateur signale une erreur interne lors de la liaison, désactivez l'option "compress_dwarf" pour l'éditeur de liens en:

- Faites un clic droit sur le projet et sélectionnez «Propriétés».
- Dans 'Build-> ARM Linker', cliquez sur le bouton 'Edit Flags'.
- modifiez la dernière option en '--compress_dwarf = off'.

Une fois les deux projets construits, cliquez sur "Exécuter-> Déboguer" séparément pour télécharger les images de la pile et de l'application sur le MCU.

Touchez le code

Pour pouvoir apporter des modifications agressives au code exemple, les développeurs doivent acquérir des connaissances détaillées sur la structure en couches de la pile BLE. Pour les tâches élémentaires telles que la lecture / notification de la température, nous ne pouvons nous concentrer que sur deux fichiers: PROFILES / simple_gatt_profile.c (.h) et Application / simple_peripheral.c (.h)

simple_gatt_profile.c

Toutes les applications Bluetooth offrent un certain type de service, chacune comprenant un ensemble de caractéristiques. Le profil de périphérique simple définit un service simple, avec l'UUID de 0xFFF0, composé de 5 caractéristiques. Ce service est spécifié dans simple_gatt_profile.c. Un résumé du service simple est répertorié comme suit.

prénom	Taille des données	UUID	La description	Propriété
simplePeripheralChar1	1	0xFFF1	Caractéristiques 1	Lire écrire
simplePeripheralChar2	1	0xFFF2	Caractéristiques 2	Lecture seulement
simplePeripheralChar3	1	0xFFF3	Caractéristiques	Ecrire

prénom	Taille des données	UUID	La description	Propriété
			3	seulement
simplePeripheralChar4	1	0xFFF4	Caractéristiques 4	Notifier
simplePériphériqueChar5	5	0xFFF5	Caractéristiques 5	Lecture seulement

Les cinq caractéristiques ont des propriétés différentes et servent d'exemples pour différents cas d'utilisateurs. Par exemple, la MCU peut utiliser simplePeripheralChar4 pour informer ses clients, hôtes en amont, du changement d'informations.

Pour définir un service Bluetooth, vous devez créer un tableau d'attributs.

```
/*********************************************************************
 * Profile Attributes - Table
 */

static gattAttribute_t simpleProfileAttrTbl[SERVAPP_NUM_ATTR_SUPPORTED] =
{
  // Simple Profile Service
  {
    { ATT_BT_UUID_SIZE, primaryServiceUUID }, /* type */
    GATT_PERMIT_READ,                         /* permissions */
    0,                                        /* handle */
    (uint8 *)&simpleProfileService            /* pValue */
  },

    // Characteristic 1 Declaration
    {
      { ATT_BT_UUID_SIZE, characterUUID },
      GATT_PERMIT_READ,
      0,
      &simpleProfileChar1Props
    },

      // Characteristic Value 1
      {
        { ATT_UUID_SIZE, simpleProfilechar1UUID },
        GATT_PERMIT_READ | GATT_PERMIT_WRITE,
        0,
        &simpleProfileChar1
      },

      // Characteristic 1 User Description
      {
        { ATT_BT_UUID_SIZE, charUserDescUUID },
        GATT_PERMIT_READ,
            0,
            simpleProfileChar1UserDesp
          },
        ...
    };
```

La table attributaire commence par un 'primaryServiceUUID' par défaut, qui spécifie l'UUID du service (0xFFF0 dans ce cas). Il est ensuite suivi de déclarations de toutes les caractéristiques qui constituent le service. Chaque caractéristique possède plusieurs attributs, à savoir l'autorisation d'accès, la valeur et la description de l'utilisateur, etc. Cette table est ensuite enregistrée avec la pile BLE.

```
// Register GATT attribute list and CBs with GATT Server App
status = GATTServApp_RegisterService( simpleProfileAttrTbl,
                                      GATT_NUM_ATTRS( simpleProfileAttrTbl ),
                                      GATT_MAX_ENCRYPT_KEY_SIZE,
                                      &simpleProfileCBs );
```

Lors de l'enregistrement du service, les développeurs doivent fournir trois fonctions de rappel pour «Lire», «Ecrire» et «Autorisation» des caractéristiques. Nous pouvons trouver dans l'exemple de code la liste des fonctions de rappel.

```
/*********************************************************************
 * PROFILE CALLBACKS
 */

// Simple Profile Service Callbacks
// Note: When an operation on a characteristic requires authorization and
// pfnAuthorizeAttrCB is not defined for that characteristic's service, the
// Stack will report a status of ATT_ERR_UNLIKELY to the client.  When an
// operation on a characteristic requires authorization the Stack will call
// pfnAuthorizeAttrCB to check a client's authorization prior to calling
// pfnReadAttrCB or pfnWriteAttrCB, so no checks for authorization need to be
// made within these functions.
CONST gattServiceCBs_t simpleProfileCBs =
{
  simpleProfile_ReadAttrCB,  // Read callback function pointer
  simpleProfile_WriteAttrCB, // Write callback function pointer
  NULL                       // Authorization callback function pointer
};
```

Ainsi, simpleProfile_ReadAttrCB sera appelé une fois que le client de service aura envoyé une demande de lecture via la connexion Bluetooth. De même, simpleProfile_WriteAttrCB sera appelé lorsqu'une demande d'écriture est faite. La compréhension de ces deux fonctions est la clé du succès de la personnalisation du projet.

Voici la fonction de rappel de lecture.

```
/*********************************************************************
 * @fn          simpleProfile_ReadAttrCB
 *
 * @brief       Read an attribute.
 *
 * @param       connHandle - connection message was received on
 * @param       pAttr - pointer to attribute
 * @param       pValue - pointer to data to be read
 * @param       pLen - length of data to be read
 * @param       offset - offset of the first octet to be read
 * @param       maxLen - maximum length of data to be read
 * @param       method - type of read message
 *
```

```
 * @return      SUCCESS, blePending or Failure
 */
static bStatus_t simpleProfile_ReadAttrCB(uint16_t connHandle,
                                          gattAttribute_t *pAttr,
                                          uint8_t *pValue, uint16_t *pLen,
                                          uint16_t offset, uint16_t maxLen,
                                          uint8_t method)
{
      bStatus_t status = SUCCESS;

      // If attribute permissions require authorization to read, return error
      if ( gattPermitAuthorRead( pAttr->permissions ) )
      {
        // Insufficient authorization
        return ( ATT_ERR_INSUFFICIENT_AUTHOR );
      }

      // Make sure it's not a blob operation (no attributes in the profile are long)
      if ( offset > 0 )
      {
        return ( ATT_ERR_ATTR_NOT_LONG );
      }

      uint16 uuid = 0;
      if ( pAttr->type.len == ATT_UUID_SIZE )
        // 128-bit UUID
        uuid = BUILD_UINT16( pAttr->type.uuid[12], pAttr->type.uuid[13]);
      else
        uuid = BUILD_UINT16( pAttr->type.uuid[0], pAttr->type.uuid[1]);

        switch ( uuid )
        {
          // No need for "GATT_SERVICE_UUID" or "GATT_CLIENT_CHAR_CFG_UUID" cases;
          // gattserverapp handles those reads

          // characteristics 1 and 2 have read permissions
          // characteritisc 3 does not have read permissions; therefore it is not
          //   included here
          // characteristic 4 does not have read permissions, but because it
          //   can be sent as a notification, it is included here
          case SIMPLEPROFILE_CHAR2_UUID:
              *pLen = SIMPLEPROFILE_CHAR2_LEN;
              VOID memcpy( pValue, pAttr->pValue, SIMPLEPROFILE_CHAR2_LEN );
              break;

          case SIMPLEPROFILE_CHAR1_UUID:
              *pLen = SIMPLEPROFILE_CHAR1_LEN;
              VOID memcpy( pValue, pAttr->pValue, SIMPLEPROFILE_CHAR1_LEN );
              break;

          case SIMPLEPROFILE_CHAR4_UUID:
            *pLen = SIMPLEPROFILE_CHAR4_LEN;
            VOID memcpy( pValue, pAttr->pValue, SIMPLEPROFILE_CHAR4_LEN );
            break;

          case SIMPLEPROFILE_CHAR5_UUID:
            *pLen = SIMPLEPROFILE_CHAR5_LEN;
            VOID memcpy( pValue, pAttr->pValue, SIMPLEPROFILE_CHAR5_LEN );
            break;

          default:
```

```
            // Should never get here! (characteristics 3 and 4 do not have read permissions)
            *pLen = 0;
            status = ATT_ERR_ATTR_NOT_FOUND;
            break;
        }

      return ( status );
}
```

J'ai légèrement modifié le code de sa version originale. Cette fonction prend 7 paramètres, qui sont expliqués dans les commentaires d'en-tête. La fonction commence par vérifier l'autorisation d'accès de l'attribut, par exemple si elle dispose d'une autorisation de lecture. Ensuite, il vérifie s'il s'agit d'une lecture de segment d'une requête de lecture de blob plus grande en testant la condition "if (offset> 0)". Evidemment, la fonction ne supporte pas la lecture de blob pour le moment. Ensuite, l'UUID de l'attribut demandé est extrait. Il existe deux types d'UUID: 16 bits et 128 bits. Alors que l'exemple de code définit toutes les caractéristiques utilisant des UUID 16 bits, l'UUID 128 bits est plus universel et plus couramment utilisé dans les hôtes en amont tels que les PC et les smartphones. Par conséquent, plusieurs lignes de code sont utilisées pour convertir 128 UUID en UUID 16 bits.

```
 uint16 uuid = 0;
  if ( pAttr->type.len == ATT_UUID_SIZE )
    // 128-bit UUID
    uuid = BUILD_UINT16( pAttr->type.uuid[12], pAttr->type.uuid[13]);
  else
    uuid = BUILD_UINT16( pAttr->type.uuid[0], pAttr->type.uuid[1]);
```

Enfin, après avoir reçu l'UUID, nous pouvons déterminer quel attribut est demandé. Ensuite, le travail restant du côté des développeurs consiste à copier la valeur de l'attribut demandé sur le pointeur de destination 'pValue'.

```
switch ( uuid )
    {
      case SIMPLEPROFILE_CHAR1_UUID:
          *pLen = SIMPLEPROFILE_CHAR1_LEN;
          VOID memcpy( pValue, pAttr->pValue, SIMPLEPROFILE_CHAR1_LEN );
          break;

      case SIMPLEPROFILE_CHAR2_UUID:
          *pLen = SIMPLEPROFILE_CHAR2_LEN;
          VOID memcpy( pValue, pAttr->pValue, SIMPLEPROFILE_CHAR2_LEN );
          break;

      case SIMPLEPROFILE_CHAR4_UUID:
        *pLen = SIMPLEPROFILE_CHAR4_LEN;
        VOID memcpy( pValue, pAttr->pValue, SIMPLEPROFILE_CHAR4_LEN );
        break;

      case SIMPLEPROFILE_CHAR5_UUID:
        *pLen = SIMPLEPROFILE_CHAR5_LEN;
        VOID memcpy( pValue, pAttr->pValue, SIMPLEPROFILE_CHAR5_LEN );
        break;

      default:
        *pLen = 0;
```

```
        status = ATT_ERR_ATTR_NOT_FOUND;
        break;
    }
```

La fonction de rappel d'écriture est similaire sauf qu'il existe un type d'écriture spécial avec l'UUID de GATT_CLIENT_CHAR_CFG_UUID. Il s'agit de la demande de l'hôte en amont de s'inscrire pour une notification ou une indication de caractéristiques. Appelez simplement l'API GATTServApp_ProcessCCCWriteReq pour transmettre la requête à la pile BLE.

```
case GATT_CLIENT_CHAR_CFG_UUID:
        status = GATTServApp_ProcessCCCWriteReq( connHandle, pAttr, pValue, len,
                                                 offset, GATT_CLIENT_CFG_NOTIFY |
GATT_CLIENT_CFG_INDICATE ); // allow client to request notification or indication features
        break;
```

Le côté application du code sur la MCU peut vouloir être averti de toute modification des caractéristiques autorisées en écriture. Les développeurs peuvent implémenter cette notification comme ils l'entendent. Dans l'exemple de code, la fonction de rappel est utilisée.

```
  // If a charactersitic value changed then callback function to notify application of change
  if ( (notifyApp != 0xFF ) && simpleProfile_AppCBs && simpleProfile_AppCBs-
>pfnSimpleProfileChange )
  {
    simpleProfile_AppCBs->pfnSimpleProfileChange( notifyApp );
  }
```

D'autre part, si le périphérique BLE souhaite notifier les hôtes en amont de tout changement de ses caractéristiques, il peut appeler l'API GATTServApp_ProcessCharCfg. Cette API est démontrée dans la fonction SimpleProfile_SetParameter.

```
/*********************************************************************
 * @fn      SimpleProfile_SetParameter
 *
 * @brief   Set a Simple Profile parameter.
 *
 * @param   param - Profile parameter ID
 * @param   len - length of data to write
 * @param   value - pointer to data to write.  This is dependent on
 *          the parameter ID and WILL be cast to the appropriate
 *          data type (example: data type of uint16 will be cast to
 *          uint16 pointer).
 *
 * @return  bStatus_t
 */
bStatus_t SimpleProfile_SetParameter( uint8 param, uint8 len, void *value )
{
      bStatus_t ret = SUCCESS;
      switch ( param )
      {
        case SIMPLEPROFILE_CHAR2:
          if ( len == SIMPLEPROFILE_CHAR2_LEN )
          {
            VOID memcpy( simpleProfileChar2, value, SIMPLEPROFILE_CHAR2_LEN );

          }
```

```
          else
          {
            ret = bleInvalidRange;
          }
          break;

        case SIMPLEPROFILE_CHAR3:
          if ( len == sizeof ( uint8 ) )
          {
            simpleProfileChar3 = *((uint8*)value);
          }
          else
          {
            ret = bleInvalidRange;
          }
          break;

        case SIMPLEPROFILE_CHAR1:
          if ( len == SIMPLEPROFILE_CHAR1_LEN )
          {
              VOID memcpy( simpleProfileChar1, value, SIMPLEPROFILE_CHAR1_LEN );
          }
          else
          {
            ret = bleInvalidRange;
          }
          break;

        case SIMPLEPROFILE_CHAR4:
          if ( len == SIMPLEPROFILE_CHAR4_LEN )
          {
            //simpleProfileChar4 = *((uint8*)value);
            VOID memcpy( simpleProfileChar4, value, SIMPLEPROFILE_CHAR4_LEN );
            // See if Notification has been enabled
            GATTServApp_ProcessCharCfg( simpleProfileChar4Config, simpleProfileChar4, FALSE,
                                        simpleProfileAttrTbl, GATT_NUM_ATTRS(
simpleProfileAttrTbl ),
                                        INVALID_TASK_ID, simpleProfile_ReadAttrCB );
          }
          else
          {
            ret = bleInvalidRange;
          }
          break;

        case SIMPLEPROFILE_CHAR5:
          if ( len == SIMPLEPROFILE_CHAR5_LEN )
          {
            VOID memcpy( simpleProfileChar5, value, SIMPLEPROFILE_CHAR5_LEN );
          }
          else
          {
            ret = bleInvalidRange;
          }
          break;

        default:
          ret = INVALIDPARAMETER;
          break;
      }
```

```
      return ( ret );
}
```

Ainsi, si l'application périphérique simple souhaite notifier la valeur actuelle de SIMPLEPROFILE_CHAR4 aux périphériques homologues, elle peut simplement appeler la fonction SimpleProfile_SetParameter.

En résumé, PROFILES / simple_gatt_profile.c (.h) définit le contenu du service que le périphérique BLE souhaite présenter à ses clients, ainsi que la manière d'accéder à ces caractéristiques du service.

simple_peripheral.c

La pile BLE de TI s'exécute au-dessus d'une couche OS multi-thread. Pour ajouter une charge de travail à la MCU, les développeurs doivent d'abord créer une tâche. simple_peripheral.c montre la structure de base d'une tâche personnalisée, qui comprend la création, l'initialisation et la gestion de la tâche. Pour commencer avec les tâches de base comme la lecture de la température et la notification, nous allons nous concentrer sur quelques fonctions clés ci-dessous.

Le début du fichier définit un ensemble de paramètres pouvant affecter les comportements de connexion Bluetooth.

```
// Advertising interval when device is discoverable (units of 625us, 160=100ms)
#define DEFAULT_ADVERTISING_INTERVAL          160

// Limited discoverable mode advertises for 30.72s, and then stops
// General discoverable mode advertises indefinitely
#define DEFAULT_DISCOVERABLE_MODE             GAP_ADTYPE_FLAGS_GENERAL

// Minimum connection interval (units of 1.25ms, 80=100ms) if automatic
// parameter update request is enabled
#define DEFAULT_DESIRED_MIN_CONN_INTERVAL     80

// Maximum connection interval (units of 1.25ms, 800=1000ms) if automatic
// parameter update request is enabled
#define DEFAULT_DESIRED_MAX_CONN_INTERVAL     400

// Slave latency to use if automatic parameter update request is enabled
#define DEFAULT_DESIRED_SLAVE_LATENCY         0

// Supervision timeout value (units of 10ms, 1000=10s) if automatic parameter
// update request is enabled
#define DEFAULT_DESIRED_CONN_TIMEOUT          1000

// Whether to enable automatic parameter update request when a connection is
// formed
#define DEFAULT_ENABLE_UPDATE_REQUEST         TRUE

// Connection Pause Peripheral time value (in seconds)
#define DEFAULT_CONN_PAUSE_PERIPHERAL         6

// How often to perform periodic event (in msec)
#define SBP_PERIODIC_EVT_PERIOD               1000
```

Les paramètres DEFAULT_DESIRED_MIN_CONN_INTERVAL, DEFAULT_DESIRED_MAX_CONN_INTERVAL et DEFAULT_DESIRED_SLAVE_LATENCY définissent ensemble l'intervalle de connexion d'une connexion Bluetooth, c'est-à-dire la fréquence à laquelle deux appareils échangent des informations. Un intervalle de connexion inférieur signifie un comportement plus réactif mais également une consommation d'énergie plus élevée.

Le paramètre DEFAULT_DESIRED_CONN_TIMEOUT définit la durée de réception d'une réponse homologue avant qu'une connexion soit considérée comme perdue. Le paramètre DEFAULT_ENABLE_UPDATE_REQUEST définit si le périphérique esclave est autorisé à modifier l'intervalle de connexion pendant l'exécution. Il est utile en termes d'économie d'énergie d'avoir des paramètres de connexion différents pour les phases occupées et inactives.

Le paramètre SBP_PERIODIC_EVT_PERIOD définit la période d'un événement d'horloge qui permettra à la tâche d'exécuter périodiquement un appel de fonction. C'est l'endroit idéal pour ajouter le code pour lire la température et informer les clients du service.

L'horloge périodique est lancée dans la fonction SimpleBLEPeripheral_init.

```
  // Create one-shot clocks for internal periodic events.
  Util_constructClock(&periodicClock, SimpleBLEPeripheral_clockHandler,
                      SBP_PERIODIC_EVT_PERIOD, 0, false, SBP_PERIODIC_EVT);
```

Cela va créer une horloge avec une période de SBP_PERIODIC_EVT_PERIOD. Et à la temporisation, appellera la fonction de SimpleBLEPeripheral_clockHandler avec le paramètre SBP_PERIODIC_EVT. L'événement d'horloge peut alors être déclenché par

```
Util_startClock(&periodicClock);
```

En recherchant le mot-clé Util_startClock, nous pouvons constater que cette horloge périodique est d'abord déclenchée sur l'événement GAPROLE_CONNECTED (dans la fonction SimpleBLEPeripheral_processStateChangeEvt), ce qui signifie que la tâche démarrera une routine une fois la connexion établie avec un hôte.

Lorsque l'horloge périodique arrive à expiration, sa fonction de rappel enregistrée est appelée.

```
/*********************************************************************
 * @fn      SimpleBLEPeripheral_clockHandler
 *
 * @brief   Handler function for clock timeouts.
 *
 * @param   arg - event type
 *
 * @return  None.
 */
static void SimpleBLEPeripheral_clockHandler(UArg arg)
{
  // Store the event.
  events |= arg;

  // Wake up the application.
```

```
  Semaphore_post(sem);
}
```

Cette fonction définit un indicateur dans le vecteur d'événements et active l'application à partir de la liste des tâches du système d'exploitation. Notez que nous ne faisons aucune charge de travail utilisateur spécifique dans cette fonction de rappel, car elle n'est PAS recommandée. La charge de travail de l'utilisateur implique souvent des appels aux API de la pile BLE. **Faire des appels d'API de pile BLE dans une fonction de rappel entraîne souvent des exceptions système.** Au lieu de cela, nous définissons un indicateur dans le vecteur d'événements de la tâche et attendons qu'il soit traité ultérieurement dans le contexte de l'application. Le point d'entrée de l'exemple de tâche est simpleBLEPeripheral_taskFxn ().

```
/*********************************************************************
 * @fn      SimpleBLEPeripheral_taskFxn
 *
 * @brief   Application task entry point for the Simple BLE Peripheral.
 *
 * @param   a0, a1 - not used.
 *
 * @return  None.
 */
static void SimpleBLEPeripheral_taskFxn(UArg a0, UArg a1)
{
  // Initialize application
  SimpleBLEPeripheral_init();

  // Application main loop
  for (;;)
  {
    // Waits for a signal to the semaphore associated with the calling thread.
    // Note that the semaphore associated with a thread is signaled when a
    // message is queued to the message receive queue of the thread or when
    // ICall_signal() function is called onto the semaphore.
    ICall_Errno errno = ICall_wait(ICALL_TIMEOUT_FOREVER);

    if (errno == ICALL_ERRNO_SUCCESS)
    {
      ICall_EntityID dest;
      ICall_ServiceEnum src;
      ICall_HciExtEvt *pMsg = NULL;

      if (ICall_fetchServiceMsg(&src, &dest,
                                (void **)&pMsg) == ICALL_ERRNO_SUCCESS)
      {
        uint8 safeToDealloc = TRUE;

        if ((src == ICALL_SERVICE_CLASS_BLE) && (dest == selfEntity))
        {
          ICall_Stack_Event *pEvt = (ICall_Stack_Event *)pMsg;

          // Check for BLE stack events first
          if (pEvt->signature == 0xffff)
          {
            if (pEvt->event_flag & SBP_CONN_EVT_END_EVT)
            {
              // Try to retransmit pending ATT Response (if any)
              SimpleBLEPeripheral_sendAttRsp();
            }
```

```
          }
          else
          {
            // Process inter-task message
            safeToDealloc = SimpleBLEPeripheral_processStackMsg((ICall_Hdr *)pMsg);
          }
        }

        if (pMsg && safeToDealloc)
        {
          ICall_freeMsg(pMsg);
        }
      }

      // If RTOS queue is not empty, process app message.
      while (!Queue_empty(appMsgQueue))
      {
        sbpEvt_t *pMsg = (sbpEvt_t *)Util_dequeueMsg(appMsgQueue);
        if (pMsg)
        {
          // Process message.
          SimpleBLEPeripheral_processAppMsg(pMsg);

          // Free the space from the message.
          ICall_free(pMsg);
        }
      }
    }

    if (events & SBP_PERIODIC_EVT)
    {
      events &= ~SBP_PERIODIC_EVT;

      Util_startClock(&periodicClock);

      // Perform periodic application task
      SimpleBLEPeripheral_performPeriodicTask();
    }

  }
}
```

C'est une boucle infinie qui continue d'interroger la pile de la tâche et les files d'attente de messages d'application. Il vérifie également son vecteur d'événements pour différents drapeaux. C'est là que la routine périodique est *réellement* exécutée. Lors de la découverte d'un SBP_PERIODIC_EVT, la fonction de tâche efface d'abord l'indicateur, lance immédiatement le même temporisateur et appelle la fonction de routine SimpleBLEPeripheral_performPeriodicTask ();

```
/*********************************************************************
 * @fn      SimpleBLEPeripheral_performPeriodicTask
 *
 * @brief   Perform a periodic application task. This function gets called
 *          every five seconds (SBP_PERIODIC_EVT_PERIOD). In this example,
 *          the value of the third characteristic in the SimpleGATTProfile
 *          service is retrieved from the profile, and then copied into the
 *          value of the fourth characteristic.
 *
```

```
 * @param   None.
 *
 * @return  None.
 */
static void SimpleBLEPeripheral_performPeriodicTask(void)
{
      uint8_t newValue[SIMPLEPROFILE_CHAR4_LEN];
      // user codes to do specific work like reading the temperature
      // .....
      SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR4, SIMPLEPROFILE_CHAR4_LEN,
                                 newValue);
}
```

A l'intérieur de la fonction périodique, nous exécutons notre travail très spécifique de lecture de la température, de génération de requêtes UART, etc. La pile BLE prend en charge tous les travaux de bas niveau, de la maintenance de la connexion sans fil à la transmission du message via la liaison Bluetooth. Tous les développeurs ont besoin de rassembler les données spécifiques à l'application et de les mettre à jour avec les caractéristiques correspondantes dans une table de service.

Enfin, lorsqu'une demande d'écriture est effectuée sur une caractéristique autorisée en écriture, une fonction de rappel sera évoquée.

```
static void SimpleBLEPeripheral_charValueChangeCB(uint8_t paramID)
{
  SimpleBLEPeripheral_enqueueMsg(SBP_CHAR_CHANGE_EVT, paramID);
}
```

Encore une fois, cette fonction de rappel ne met en file d'attente qu'un message d'application pour la tâche utilisateur, qui sera traitée ultérieurement dans le contexte de l'application.

```
static void SimpleBLEPeripheral_processCharValueChangeEvt(uint8_t paramID)
{
  uint8_t newValue[SIMPLEPROFILE_CHAR1_LEN];

  switch(paramID)
  {
    case SIMPLEPROFILE_CHAR1:
      SimpleProfile_GetParameter(SIMPLEPROFILE_CHAR1, &newValue[0]);
      ProcessUserCmd(newValue[0], NULL);
      break;

    case SIMPLEPROFILE_CHAR3:
      break;

    default:
      // should not reach here!
      break;
  }
}
```

Dans l'exemple ci-dessus, lorsque SIMPLEPROFILE_CHAR1 est écrit, le code utilisateur extrait d'abord la nouvelle valeur en appelant SimpleProfile_GetParameter (), puis analyse les données pour les commandes définies par l'utilisateur.

En résumé, le fichier simple_peripheral.c montre un exemple de création d'une tâche utilisateur pour des charges de travail personnalisées. Un moyen de base pour planifier la charge de travail d'une application est un événement d'horloge périodique. Les développeurs doivent uniquement traiter les informations vers / depuis les caractéristiques de la table de service, tandis que la pile BLE se charge du reste de la communication des informations de la table de service aux périphériques homologues (ou inversement) via une connexion Bluetooth.

Connecter des capteurs du monde réel

Pour que les périphériques esclaves BLE effectuent un travail utile, les GPIO du microcontrôleur sans fil sont presque toujours impliqués. Par exemple, pour lire la température d'un capteur externe, la fonctionnalité ADC des broches GPIO peut être requise. Le microcontrôleur CC2640 de TI comporte au maximum 31 GPIO, en fonction des différents types d’emballage.

Du côté du matériel, le CC2640 fournit un ensemble complet de fonctionnalités périphériques telles que ADC, UARTS, SPI, SSI, I2C, etc. Du côté logiciel, la pile BLE de TI tente d’offrir une interface de pilote Une interface de pilote uniforme peut améliorer les chances de réutilisation du code, mais elle augmente également la pente de la courbe d'apprentissage. Dans cette note, nous utilisons le contrôleur SPI comme exemple et montrons comment intégrer le pilote logiciel dans les applications utilisateur.

Flux de base du pilote SPI

Dans la pile BLE de TI, un pilote de périphérique se compose souvent de trois parties: une spécification indépendante du périphérique des API de pilote; une implémentation spécifique aux périphériques des API du pilote et un mappage des ressources matérielles.

Pour le contrôleur SPI, son implémentation de pilote implique trois fichiers:

- <ti / drivers / SPI.h> - il s'agit de la spécification API indépendante du périphérique
- <ti / drivers / spi / SPICC26XXDMA.h> - c'est l'implémentation de l'API spécifique au CC2640
- <ti / drivers / dma / UDMACC26XX.h> - il s'agit du pilote uDMA requis par le pilote SPI

(Remarque: le meilleur document pour les pilotes de périphériques de la pile BLE de TI se trouve principalement dans leurs fichiers d'en-tête, tels que SPICC26XXDMA.h dans ce cas)

Pour commencer à utiliser le contrôleur SPI, créons d'abord un fichier c personnalisé, à savoir sbp_spi.c, qui inclut les trois fichiers d'en-tête ci-dessus. L'étape suivante naturelle consiste à créer une instance du pilote et à l'initier. L'instance de pilote est encapsulée dans la structure de données - SPI_Handle. Une autre structure de données - SPI_Params est utilisée pour spécifier les paramètres clés du contrôleur SPI, tels que le débit binaire, le mode de transfert, etc.

```
#include <ti/drivers/SPI.h>
#include <ti/drivers/spi/SPICC26XXDMA.h>
#include <ti/drivers/dma/UDMACC26XX.h>

static void sbp_spiInit();
```

```
static SPI_Handle spiHandle;
static SPI_Params spiParams;

void sbp_spiInit(){
    SPI_init();
    SPI_Params_init(&spiParams);
    spiParams.mode                      = SPI_MASTER;
    spiParams.transferMode              = SPI_MODE_CALLBACK;
    spiParams.transferCallbackFxn       = sbp_spiCallback;
    spiParams.bitRate                   = 800000;
    spiParams.frameFormat               = SPI_POL0_PHA0;
    spiHandle = SPI_open(CC2650DK_7ID_SPI0, &spiParams);
}
```

L'exemple de code ci-dessus montre comment initialiser l'instance SPI_Handle. L'API SPI_init () doit être appelée en premier pour initialiser les structures de données internes. L'appel de fonction SPI_Params_init (& spiParams) définit tous les champs de la structure SPI_Params sur les valeurs par défaut. Les développeurs peuvent alors modifier les paramètres clés en fonction de leurs cas spécifiques. Par exemple, le code ci-dessus définit le contrôleur SPI pour qu'il fonctionne en mode maître avec un débit binaire de 800 kbps et utilise une méthode non bloquante pour traiter chaque transaction, de sorte que

Enfin, un appel à SPI_open () ouvre le contrôleur matériel SPI et renvoie un handle pour les transactions SPI ultérieures. Le SPI_open () prend deux arguments, le premier est l'ID du contrôleur SPI. CC2640 dispose de deux contrôleurs SPI matériels sur puce, donc ces arguments d'identification seront 0 ou 1 comme défini ci-dessous. Le deuxième argument est les paramètres souhaités pour le contrôleur SPI.

```
/*!
 *  @def    CC2650DK_7ID_SPIName
 *  @brief  Enum of SPI names on the CC2650 dev board
 */
typedef enum CC2650DK_7ID_SPIName {
    CC2650DK_7ID_SPI0 = 0,
    CC2650DK_7ID_SPI1,
    CC2650DK_7ID_SPICOUNT
} CC2650DK_7ID_SPIName;
```

Après l'ouverture réussie de SPI_Handle, les développeurs peuvent lancer immédiatement des transactions SPI. Chaque transaction SPI est décrite à l'aide de la structure de données - SPI_Transaction.

```
/*!
 *  @brief
 *  A ::SPI_Transaction data structure is used with SPI_transfer(). It indicates
 *  how many ::SPI_FrameFormat frames are sent and received from the buffers
 *  pointed to txBuf and rxBuf.
 *  The arg variable is an user-definable argument which gets passed to the
 *  ::SPI_CallbackFxn when the SPI driver is in ::SPI_MODE_CALLBACK.
 */
typedef struct SPI_Transaction {
    /* User input (write-only) fields */
    size_t     count;      /*!< Number of frames for this transaction */
```

```
    void      *txBuf;      /*!< void * to a buffer with data to be transmitted */
    void      *rxBuf;      /*!< void * to a buffer to receive data */
    void      *arg;        /*!< Argument to be passed to the callback function */

    /* User output (read-only) fields */
    SPI_Status status;     /*!< Status code set by SPI_transfer */

    /* Driver-use only fields */
} SPI_Transaction;
```

Par exemple, pour lancer une transaction d'écriture sur le bus SPI, les développeurs doivent préparer un 'txBuf' rempli de données à transmettre et définir la variable 'count' sur la longueur des octets de données à envoyer. Enfin, un appel à SPI_transfer (spiHandle, spiTrans) signale au contrôleur SPI de lancer la transaction.

```
static SPI_Transaction spiTrans;
bool sbp_spiTransfer(uint8_t len, uint8_t * txBuf, uint8_t rxBuf, uint8_t * args)
{
    spiTrans.count = len;
    spiTrans.txBuf = txBuf;
    spiTrans.rxBuf = rxBuf;
    spiTrans.arg   = args;

    return SPI_transfer(spiHandle, &spiTrans);
}
```

Étant donné que SPI est un protocole duplex qui émet et reçoit simultanément, à la fin d'une transaction d'écriture, les données de réponse correspondantes sont déjà disponibles sur le "rxBuf".

Puisque nous définissons le mode de transfert sur le mode de rappel, chaque fois qu'une transaction est terminée, la fonction de rappel enregistrée est appelée. C'est ici que nous traitons les données de réponse ou que nous initions la transaction suivante. **(Remarque: n'oubliez jamais de faire plus que les appels API nécessaires dans une fonction de rappel).**

```
void sbp_spiCallback(SPI_Handle handle, SPI_Transaction * transaction){
    uint8_t * args = (uint8_t *)transaction->arg;

    // may want to disable the interrupt first
    key = Hwi_disable();
    if(transaction->status == SPI_TRANSFER_COMPLETED){
        // do something here for successful transaction...
    }
    Hwi_restore(key);
}
```

Configuration des broches E / S

Jusqu'à présent, il semble relativement simple d'utiliser le pilote SPI. Mais attendez, comment connecter les appels de l'API logicielle aux signaux physiques SPI? Cela se fait via trois structures de données: SPICC26XXDMA_Object, SPICC26XXDMA_HWAttrsV1 et SPI_Config. Ils sont normalement instanciés à un emplacement différent comme "board.c".

```
/* SPI objects */
SPICC26XXDMA_Object spiCC26XXDMAObjects[CC2650DK_7ID_SPICOUNT];

/* SPI configuration structure, describing which pins are to be used */
const SPICC26XXDMA_HWAttrsV1 spiCC26XXDMAHWAttrs[CC2650DK_7ID_SPICOUNT] = {
    {
        .baseAddr           = SSI0_BASE,
        .intNum             = INT_SSI0_COMB,
        .intPriority        = ~0,
        .swiPriority        = 0,
        .powerMngrId        = PowerCC26XX_PERIPH_SSI0,
        .defaultTxBufValue  = 0,
        .rxChannelBitMask   = 1<<UDMA_CHAN_SSI0_RX,
        .txChannelBitMask   = 1<<UDMA_CHAN_SSI0_TX,
        .mosiPin            = ADC_MOSI_0,
        .misoPin            = ADC_MISO_0,
        .clkPin             = ADC_SCK_0,
        .csnPin             = ADC_CSN_0
    },
    {
        .baseAddr           = SSI1_BASE,
        .intNum             = INT_SSI1_COMB,
        .intPriority        = ~0,
        .swiPriority        = 0,
        .powerMngrId        = PowerCC26XX_PERIPH_SSI1,
        .defaultTxBufValue  = 0,
        .rxChannelBitMask   = 1<<UDMA_CHAN_SSI1_RX,
        .txChannelBitMask   = 1<<UDMA_CHAN_SSI1_TX,
        .mosiPin            = ADC_MOSI_1,
        .misoPin            = ADC_MISO_1,
        .clkPin             = ADC_SCK_1,
        .csnPin             = ADC_CSN_1
    }
};

/* SPI configuration structure */
const SPI_Config SPI_config[] = {
    {
         .fxnTablePtr = &SPICC26XXDMA_fxnTable,
         .object      = &spiCC26XXDMAObjects[0],
         .hwAttrs     = &spiCC26XXDMAHWAttrs[0]
    },
    {
         .fxnTablePtr = &SPICC26XXDMA_fxnTable,
         .object      = &spiCC26XXDMAObjects[1],
         .hwAttrs     = &spiCC26XXDMAHWAttrs[1]
    },
    {NULL, NULL, NULL}
};
```

Le tableau SPI_Config comporte une entrée distincte pour chaque contrôleur matériel SPI. Chaque entrée comporte trois champs: fxnTablePtr, object et hwAttrs. Le 'fxnTablePtr' est une table de points qui pointe vers les implémentations spécifiques au périphérique de l'API du pilote.

L'objet garde la trace des informations telles que l'état du pilote, le mode de transfert, la fonction de rappel pour le pilote. Cet objet est automatiquement géré par le pilote.

Le 'hwAttrs' stocke les données de cartographie des ressources matérielles réelles, par exemple les broches IO pour les signaux SPI, le numéro d'interruption matérielle, l'adresse de base du

contrôleur SPI, etc. La plupart des champs Alors que les broches IO de l'interface peuvent être librement assignées à l'utilisateur. **Remarque: les microcontrôleurs CC26XX découplent les broches IO de la fonctionnalité spécifique des périphériques que l'une des broches IO peut être affectée à une fonction périphérique.**

Bien sûr, les broches IO doivent être définies en premier dans le tableau.

```
#define ADC_CSN_1                       IOID_1
#define ADC_SCK_1                       IOID_2
#define ADC_MISO_1                      IOID_3
#define ADC_MOSI_1                      IOID_4
#define ADC_CSN_0                       IOID_5
#define ADC_SCK_0                       IOID_6
#define ADC_MISO_0                      IOID_7
#define ADC_MOSI_0                      IOID_8
```

En conséquence, après la configuration du mappage des ressources matérielles, les développeurs peuvent enfin communiquer avec des puces de capteurs externes via l'interface SPI.

Lire Commencez avec la pile BLE de TI en ligne:
https://riptutorial.com/fr/bluetooth/topic/7058/commencez-avec-la-pile-ble-de-ti

Chapitre 5: Socket L2CAP ouvert pour la communication Low Energy

Examples

En C, avec Bluez

```
int get_l2cap_connection () {
```

Tout d'abord, toutes les variables dont nous avons besoin, les explications suivront à l'endroit approprié.

```
    int ssock = 0;
    int csock = 0;
    int reuse_addr = 1;
    struct sockaddr_l2 src_addr;
    struct bt_security bt_sec;
    int result = 0;
```

Tout d'abord, nous devons créer un socket, à partir duquel nous pouvons accepter une connexion. La famille de socket est `PF_BLUETOOTH` , le type de socket est `SOCK_SEQPACKET` (nous voulons avoir un socket de type TCP, pas brut) et le protocole est le protocole Bluetooth L2CAP (`BTPROTO_L2CAP`).

```
    ssock = socket(PF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
```

Nous voulons nous assurer que ce fut un succès:

```
    if (ssock < 0) {
        perror("Opening L2CAP socket failed");
        return -1;
    }
```

Nous devons maintenant remplir la structure d'adresse source avec une adresse générique, afin que tout périphérique Bluetooth avec n'importe quelle adresse puisse se connecter à nous. L'adresse générique est définie comme `BDADDR_ANY` dans `bluetooth.h` . Pour le copier dans la structure d'adresse, nous pouvons utiliser la fonction `bacpy` . Nous devons également définir la famille d'adresses, le type d'adresse et l'identifiant du canal.

```
    memset(&src_addr, 0, sizeof(src_addr));
    bacpy(&src_addr.l2_bdaddr, BDADDR_ANY);
    src_addr.l2_family = AF_BLUETOOTH;
    src_addr.l2_bdaddr_type = BDADDR_LE_PUBLIC;
    src_addr.l2_cid = htobs(CID_ATT);
```

Définir l'option SO_REUSEADDR nous permettra d'appeler à nouveau rapidement bind si nécessaire (ceci peut être omis):

```
    setsockopt(ssock, SOL_SOCKET, SO_REUSEADDR, &reuse_addr, sizeof(reuse_addr));
```

Ensuite, nous devons lier le socket avec la structure d'adresse source que nous venons de définir. Encore une fois, nous vérifions la valeur de retour pour nous assurer que cela a fonctionné.

```
    result = bind(ssock, (struct sockaddr*) &src_addr, sizeof(src_addr));
    if (result < 0) {
        perror("Binding L2CAP socket failed");
        return -1;
    }
```

Ensuite, vous définissez le niveau de sécurité. Notez que cette étape est facultative, mais si vous définissez le niveau de sécurité sur MEDIUM, cela permettra un couplage automatique avec le périphérique (le noyau gère le couplage réel).

```
    memset(&bt_sec, 0, sizeof(bt_sec));
    bt_sec.level = BT_SECURITY_MEDIUM;
    result = setsockopt(ssock, SOL_BLUETOOTH, BT_SECURITY, &bt_sec, sizeof(bt_sec));
    if (result != 0) {
        perrorno("Setting L2CAP security level failed");
        return -1;
    }
```

Maintenant, nous pouvons dire au noyau que notre ssock est un socket passif, qui acceptera une connexion. Le second paramètre est le backlog. Si vous voulez en savoir plus, la page de manuel d'écoute contient toutes les informations dont vous avez besoin.

```
    result = listen(ssock, 10);
    if (result < 0) {
        perror("Listening on L2CAP socket failed");
        return -1;
    }
```

Maintenant, nous pouvons attendre une connexion entrante. La structure peer_addr contiendra l'adresse du périphérique connecté, une fois les retours acceptés. csock sera le descripteur de fichier du socket que nous pouvons lire / écrire, pour communiquer avec le périphérique connecté.

```
    memset(peer_addr, 0, sizeof(*peer_addr));
    socklen_t addrlen = sizeof(*peer_addr);
    csock = accept(ssock, (struct sockaddr*)peer_addr, &addrlen);
    if (csock < 0) {
        perror("Accepting connection on L2CAP socket failed");
        return -1;
    }
```

Nous pouvons imprimer l'adresse de l'appareil connecté (facultatif, bien sûr). Nous pouvons utiliser la fonction batostr pour convertir l'adresse Bluetooth en chaîne.

```
    printf("Accepted connection from %s", batostr(&peer_addr->l2_bdaddr));
```

Si nous ne voulons pas que d'autres périphériques se connectent, nous devons fermer le socket

du serveur. Faites la même chose avec csock, une fois votre communication avec l'appareil terminée.

```
    close(ssock);
    return csock;
}
```

Lire Socket L2CAP ouvert pour la communication Low Energy en ligne: https://riptutorial.com/fr/bluetooth/topic/6558/socket-l2cap-ouvert-pour-la-communication-low-energy

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Bluetooth	Community, Jon Carlstedt, Lasse Meyer
2	Commencez avec Bluetooth LE sous Windows	Carter
3	Commencez avec Bluetooth sur le Web	François Beaufort
4	Commencez avec la pile BLE de TI	RamenChef, Zefu Dai
5	Socket L2CAP ouvert pour la communication Low Energy	Lasse Meyer