FREE eBook

LEARNING bluetooth

Free unaffiliated eBook created from
Stack Overflow contributors.

#bluetooth

Table of Contents

About

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

Chapter 1: Getting started with bluetooth

Remarks

Bluetooth is an industry standard for wireless data transmission between devices over short distances. It was first defined in the 1990s by the Bluetooth Special Interest Group (SIG), in IEEE 802.15.1. Both connectionless and connection oriented data transmission is possible, to one or more device, in an Ad-hoc or piconet.

The name Bluetooth comes from the danish king Harald Blauzahn, Blauzahn meaning 'blue tooth' in English. The main part of the initial development was done by the dutch professor Jaap Haartsen for the company Ericsson. Later, Intel and Nokia were the main contributors.

The freqencies used are in the license-free ISM band, between 2.402 GHz and 2.480 GHz, and can therefore be used without a permit worldwide. Interferences with Wifi networks, wireless telephones or microwaves, all working in the same ISM band, are possible.

Examples

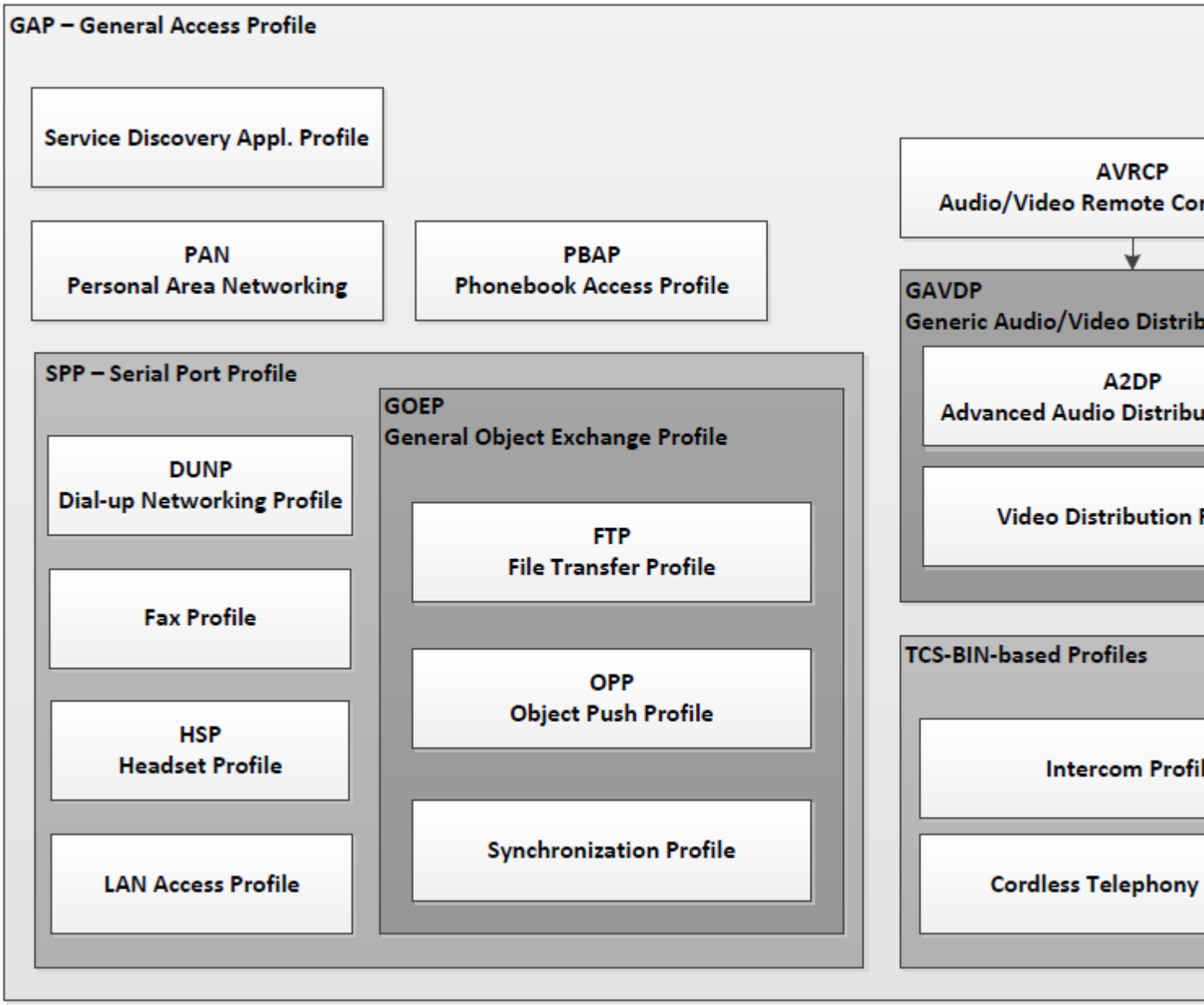
Installation or Setup

Detailed instructions on getting bluetooth set up or installed.

Profiles

The Bluetooth specification contains several profile specifications. A profile describes how to use and implement a function.

They can depend on each other, here is a basic layout of the most common profile dependencies



All profiles can be found at BT SIG, be aware that different versions might contain different functionality. Also note that some of the profiles contains several categories, these are sometimes optional, so make sure that your device supports the category in question. Here are some of the most common *smartphone Profiles* and their specifications

A2DP - Advanced Audio Distribution Profile
The Advanced Audio Distribution Profile (A2DP) defines the protocols and procedures that realize distribution of audio content of high-quality in mono or stereo on ACL channels. A typical usage case is the streaming of music content from a stereo music player to headphones or speakers. The audio data is compressed in a proper format for efficient use of the limited bandwidth.
Dependencies: GAVDP, GAP

AVRCP - Audio/Video Remote Control Profile
The Audio/Video Remote Control Profile (AVRCP) defines the features and procedures required in order to ensure interoperability between Bluetooth devices with audio/video control functions in the Audio/Video distribution scenarios. This profile adopts the AV/C device model and command

format for control messages, and those messages are transported by the Audio/Video Control Transport Protocol (AVCTP). In this profile, the controller translates the detected user action to the A/V control signal, and then transmits it to a remote Bluetooth device. In addition to this the profile uses Bluetooth specific extensions to support transfer of metadata related to content to be transferred between Bluetooth devices. The remote control described in this profile is designed specific to A/V control.
Dependencies: GAP

HFP - Hands-Free Profile
This document defines the protocols and procedures that shall be used by devices implementing the Hands-Free Profile. The most common examples of such devices are in-car Hands-Free units used together with cellular phones, or wearable wireless headsets. The profile defines how two devices supporting the Hands-Free Profile shall interact with each other on a point-to-point basis. An implementation of the Hands-Free Profile typically enables a headset, or an embedded hands-free unit to connect, wirelessly, to a cellular phone for the purposes of acting as the cellular phone's audio input and output mechanism and allowing typical telephony functions to be performed without access to the actual phone.
Dependencies: SPP, GAP

HSP - Headset Profile
This Headset profile defines the protocols and procedures that shall be used by devices requiring a full-duplex audio connection combined with minimal device control commands. The most common examples of such devices are headsets, personal computers, PDAs, and cellular phones, though most cellular phones will prefer to use a more advanced profile such as Hands-Free Profile. The headset can be wirelessly connected for the purposes of acting as the device's audio input and output mechanism, providing full duplex audio.
Dependencies: SPP, GAP

PBAP - Phonebook Access Profile
The Phone Book Access Profile (PBAP) defines the protocols and procedures that shall be used by devices for the retrieval of phone book objects. It is based on a Client-Server interaction model where the Client device pulls phone book objects from the Server device. This profile is especially tailored for the Hands-Free usage case (i.e. implemented in combination with the "Hands-Free Profile" or the "SIM Access Profile"). It provides numerous capabilities that allow for advanced handling of phone book objects, as needed in the car environment. In particular, it is much richer than the Object Push Profile (that could be used to push vCard formatted phone book entry from one device to another). This profile can also be applied to other usage cases where a Client device is to pull phone book objects from a Server device. Note however that this profile only allows for the consultation of phone book objects (read-only). It is not possible to alter the content of the original phone book object (read/write).
Dependencies: GOEP, SPP, GAP

Read Getting started with bluetooth online: https://riptutorial.com/bluetooth/topic/4846/getting-started-with-bluetooth

Chapter 2: Get Started With Bluetooth LE on Windows

Remarks

Documentation

- Advertisement - A representation of a Bluetooth LE advertisement payload.
- Advertisement Publisher - Manages the sending of Bluetooth LE advertisements.
- Advertisement Watcher - Manages the watching of Bluetooth LE advertisements.

Notes

- Windows 10 can only act in central mode, so it can only connect to devices that support peripheral mode. Due to this, two Windows 10 devices cannot connect over Bluetooth LE.
- Windows 10 must be paired with a Bluetooth LE device in order to connect to it.

Examples

Initial Setup

To use any Bluetooth functionality on a Universal Windows Platform app, you must check the `Bluetooth` capability in the `Package.appxmanifest`.

1. Open `Package.appxmanifest`
2. Go to the `Capabilities` tab
3. Find `Bluetooth` on the left and check the box next to it

Create a Bluetooth LE Advertisement

This example shows how to advertise a custom payload from a Windows 10 device in the foreground. The payload uses a made up company (identified as 0xFFFE) and advertises the string `Hello World` in the advertisement.

```
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();
```

Note: This is only for advertising in the foreground (while the app is open).

Listen for a Bluetooth LE Advertisement

General Listening

This example shows how to listen for a specific advertisement.

```
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();
```

Advertisement Filter: Listening for a Specific Advertisement

Sometimes you want to listen for a specific advertisement. In this case, listen for an advertisement containing a payload with a made up company (identified as 0xFFFE) and containing the string Hello World in the advertisement. This can be paired with the *Create a Bluetooth LE Advertisement* example to have one Windows machine advertising and another listening.

Note: Be sure to set this advertisement filter before you start your watcher!

```
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);
```

Signal Filter: Listening for Proximal Advertisements

Sometimes you only want to trigger your watcher when the device advertising has come in range. You can define your own range, just note that normal values are between 0 and -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);
```

Callbacks

```
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
}
```

Note: This is only for listening in the foreground.

Judging Distance Based on RSSI from a Bluetooth LE Advertisement

When your Bluetooth LE Watcher's callback is triggered, the eventArgs include an RSSI value telling you the received signal strength (how strong the

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

This can be roughly translated into distance, but should not be used to measure true distances as each individual radio is different. Different environmental factors can make distance difficult to gauge (such as walls, cases around the radio, or even air humidity).

An alternative to judging pure distance is to define "buckets". Radios tend to report 0 to -50 DBm when they are very close, -50 to -90 when they are a medium distance away, and below -90 when they are far away. Trial and error is best to determine what you want these buckets to be for you application.

Read Get Started With Bluetooth LE on Windows online:
https://riptutorial.com/bluetooth/topic/5553/get-started-with-bluetooth-le-on-windows

Chapter 3: Get Started with Bluetooth on the Web

Remarks

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

Read Battery Level from a nearby Bluetooth device (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);
  });
}
```

Reset energy expended from a nearby Bluetooth Device (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);
  });
}
```

Read Get Started with Bluetooth on the Web online:
https://riptutorial.com/bluetooth/topic/4936/get-started-with-bluetooth-on-the-web

Chapter 4: Get Started with TI's BLE Stack

Examples

Connecting to BLE Slave Devices

Introduction

Texas Instruments' (TI) CC26XX series SoCs are readily available wireless MCUs targeting Bluetooth Low Energy (BLE) applications. Along with the MCUs, TI offers a full-fledged software stack that provides necessary API and sample codes to help quickly get developers started with the tool chain. However, for beginners, there is always the question of where to start in front of a long list of reference document and codes. This note aims to record down the necessary steps that it takes to kick the first project going.

The Simple Peripheral Profile is the 'Hello World' example of the BLE stack, where the MCU is acting as a BLE peripheral to upstream hosts, or BLE service clients, like PC and smartphones. Common real world applications include: Bluetooth headphone, Bluetooth temperature sensor, etc.

Before start, we first need to gather basic software and hardware tools for the purpose of programming and debugging.

1. BLE stack

   Download and install TI's BLE-STACK-2-2-0 from the official website. Assume it is installed in the default location 'C:\ti'.

2. IDE - there are two options:

- IAR Embedded Workbench for ARM. This is a commercial tool with a 30 days free evaluation period.
- TI's Code Composer Studio (CCS). TI's official IDE and offers free license. In this example we will use CCS V6.1.3

3. Hardware programming tool

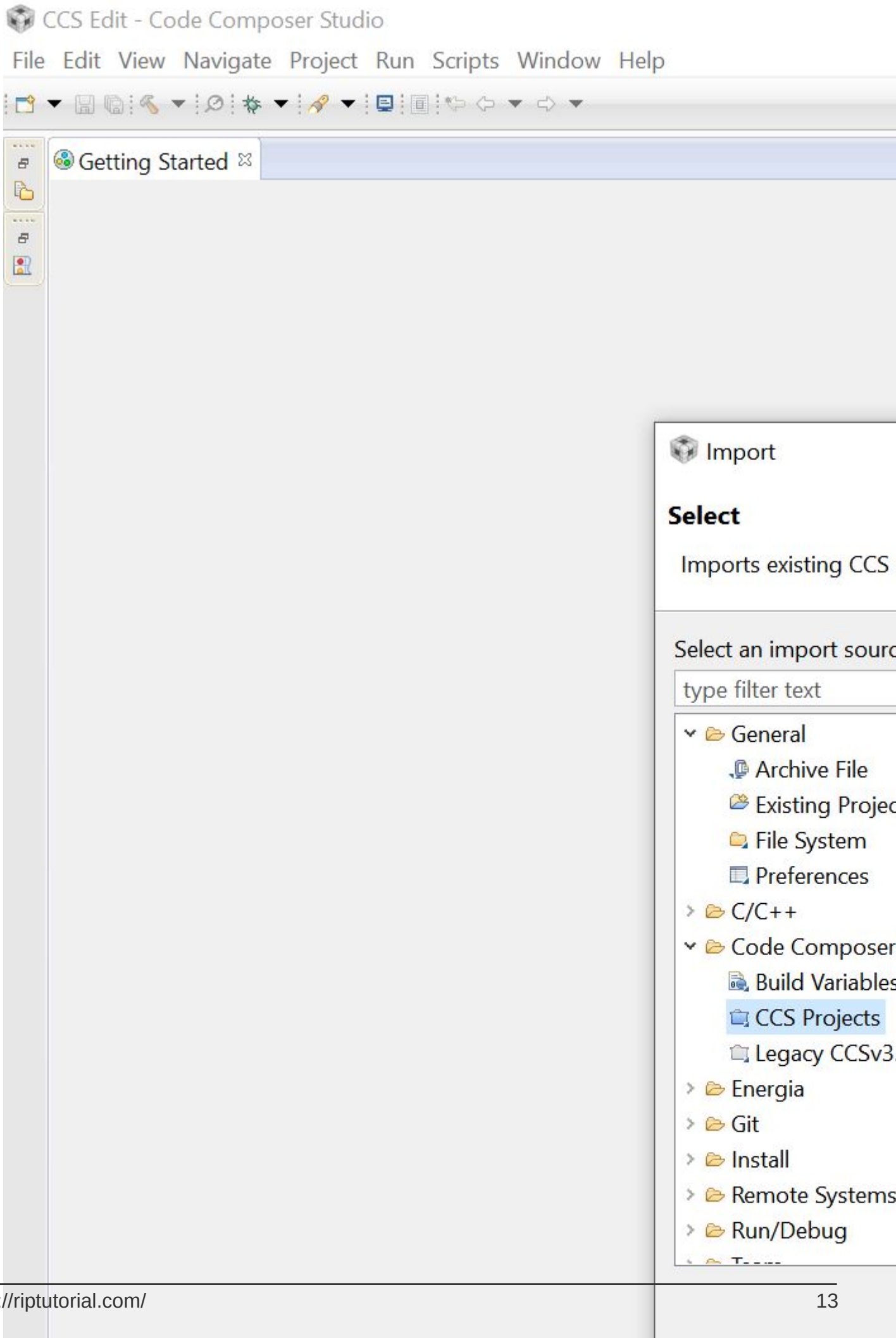
   Recommend TI's XDS100 USB-interface JTAG device.

Import Example Project in CCS

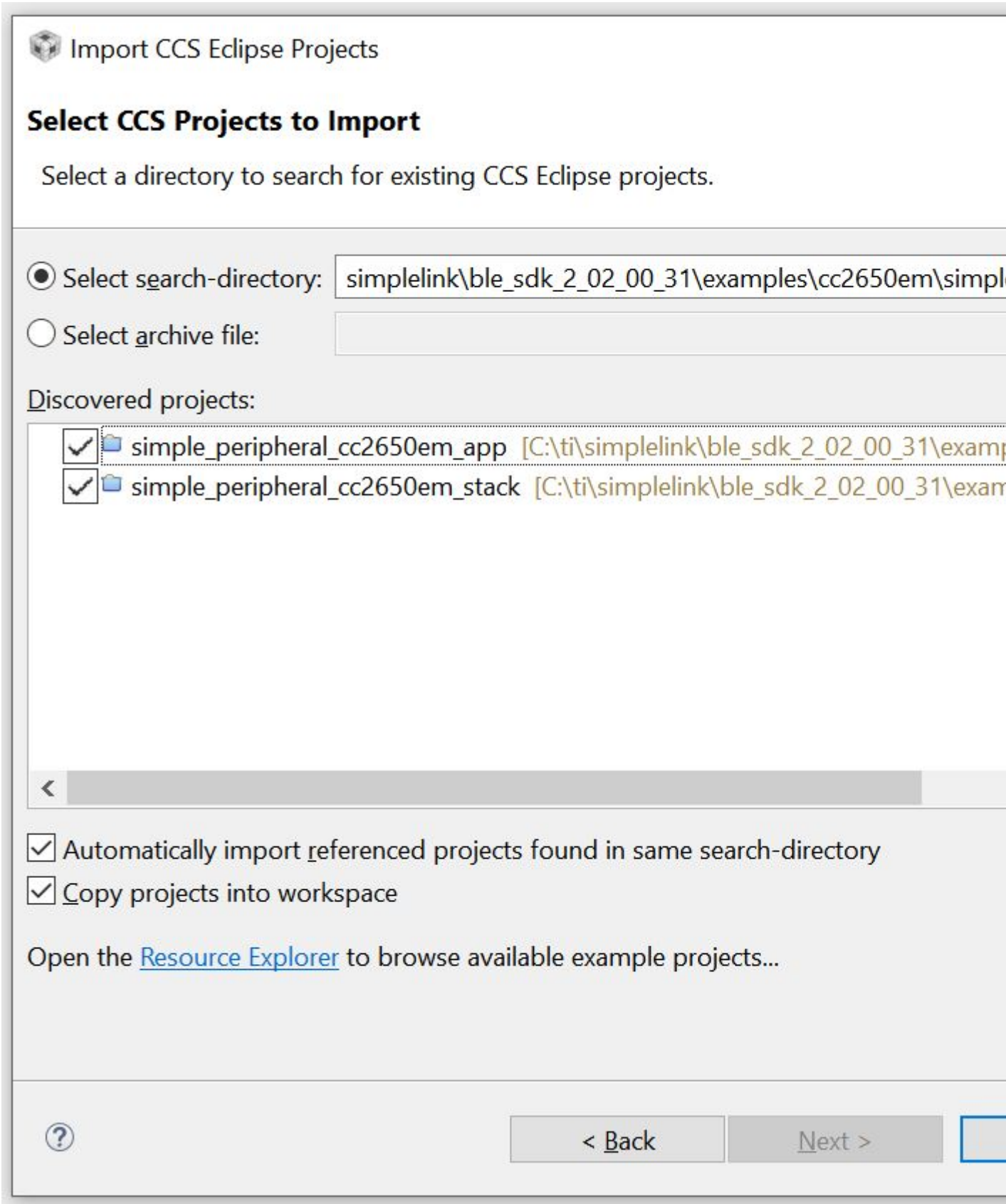
The Simple Peripheral Profile sample code comes with the BLE-Stack installation. Follow the steps below to import this example project to CCS.

1. Start CCS, create a workspace folder. Then File->Import. Under 'Select an import source',

select the 'Code Compose Studio -> CCS Projects' option and click 'Next'.


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. Browse to 'C:\ti\simplelink\ble_sdk_2_02_00_31\examples\cc2650em\simple_peripheral\ccs'. Two projects will be discovered. Select all and tick both options below. Then click 'Finish'. By copying projects into workspace, you leave the original project setting unchanged for all following modifications.



The Simple Peripheral Profile example include two projects:

- simple_peripheral_cc2650em_app
- simple_peripheral_cc2650em_stack

'cc2650em' is the code name for TI's cc2650 evaluation board. The _stack project includes the codes and binary of TI's BEL-Stack-2-2-0, which handles the Bluetooth advertising, handshaking, frequency synchronization etc.. This is the part of code that is relatively stable and don't want to be touched by developers most of the time. The _app project is where developers implement their own tasks and BLE service.

Build and Download

Click on menus 'Project->Build All' to build both projects. If the compiler report some sort of internal error on linking, try disable the 'compress_dwarf' option for the linker by:

- right click the project and select 'Propoerties'.
- in 'Build->ARM Linker', click the 'Edit Flags' button.
- modify the last option to '--compress_dwarf=off'.

After both projects are built successfully, click 'Run->debug' separately to download both the stack and app images to the MCU.

Touch the Code

To be able to make aggressive modifications to the sample code, developers have to gain detailed knowledge about the layered structure of BLE stack. For elementary tasks such as temperature reading/notification, we can focus on only two files : PROFILES/simple_gatt_profile.c(.h) and Application/simple_peripheral.c(.h)

simple_gatt_profile.c

All Bluetooth applications offer a certain type of service, each consists of a set of characteristics. The simple peripheral profile defines one simple service, with the UUID of 0xFFF0, which consists of 5 characteristics. This service is specified in simple_gatt_profile.c. A summary of the simple service is listed as follows.

Name	Data Size	UUID	Description	Property
simplePeripheralChar1	1	0xFFF1	Characteristics 1	Read & Write
simplePeripheralChar2	1	0xFFF2	Characteristics 2	Read only
simplePeripheralChar3	1	0xFFF3	Characteristics 3	Write only
simplePeripheralChar4	1	0xFFF4	Characteristics 4	Notify
simplePeripheralChar5	5	0xFFF5	Characteristics 5	Read only

The five characteristics have different properties and serve as examples for various user cases. For example, the MCU can use simplePeripheralChar4 to notify its clients, upstream hosts, about the change of information.

To define a Bluetooth service, one has to construct an Attribute Table.

```
/*********************************************************************
 * 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
          },
        ...
    };
```

The attribute table begins with a default 'primaryServiceUUID', which specifies the UUID of the service (0xFFF0 in this case). It then is followed by declarations of all characteristics that consist the service. Each characteristics has several attributes, namely access permission, value and user description, etc. This table is later registered with the BLE stack.

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

On registration of the service, developers have to provide three callback function for 'Read', 'Write' and 'Authorization' of the characteristics. We can find in the sample code the list of callback functions.

```
/*********************************************************************
 * 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
};
```

So, simpleProfile_ReadAttrCB will be called once service client sends a read request over the Bluetooth connection. Similarly, simpleProfile_WriteAttrCB will be called when a write request is made. Understanding these two functions is key to success of project customization.

Below is the read callback function.

```
/*********************************************************************
 * @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 );
}
```

I have slightly modified the code from its original version. This function takes 7 parameters, which are explained in the header comments. The function starts by checking the access permission of the attribute, e.g. whether it has read permission. Then it checks if this is a segment read of a larger blob read request by testing the condition 'if (offset > 0)'. Obviously, the function does not support blob read for now. Next, the UUID of the requested attribute is extracted. There are two

types of UUID: 16-bit and 128-bit. While the sample code defines all characteristics using 16-bit UUIDs, the 128-bit UUID is more universal and more commonly used in upstream hosts like PC and smartphones. Therefore, several lines of code are used to convert 128-bit of UUID to 16-bit UUID.

```
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]);
```

Finally, after we get the UUID, we can determine which attribute is requested. Then the remaining job at developers' side is to copy the value of requested attribute to the destination pointer '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;
    }
```

The write callback function is similar except that there is a special type of write with UUID of GATT_CLIENT_CHAR_CFG_UUID. This is upstream host's request to register for characteristics notification or indication. Simply call the API GATTServApp_ProcessCCCWriteReq to pass the request to the BLE stack.

```
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;
```

The application side of the code on the MCU may want to be notified with any change to write-permitted characteristics. Developers can implement this notification the way they like. In the sample code, callback function is used.

```
  // 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 );
  }
```

On the other hand, if the BLE peripheral wants to notify upstream hosts of any change in its characteristic, it can call the API GATTServApp_ProcessCharCfg. This API is demonstrated in the function 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 );
}
```

So if the simple peripheral application wants to notify the current value of SIMPLEPROFILE_CHAR4 to peer devices, it can simply call the SimpleProfile_SetParameter function.

In summary, PROFILES/simple_gatt_profile.c(.h) defines the content of the service that the BLE peripheral would like to present to its clients, as well as the ways that those characteristics in the service are accessed.

simple_peripheral.c

TI's BLE stack is running on top of a lite multi-threaded OS layer. To add a workload to the MCU,

developers have to create a task first. simple_peripheral.c demonstrates the basic structure of a custom task, which includes the creation, initialization and housekeeping of the task. To begin with the very basic tasks like temperature reading and notification, we will focus on a few key functions below.

The beginning of the file defines a set of parameters that can affect the Bluetooth connection behaviors.

```
// 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
```

Parameters DEFAULT_DESIRED_MIN_CONN_INTERVAL, DEFAULT_DESIRED_MAX_CONN_INTERVAL and DEFAULT_DESIRED_SLAVE_LATENCY together define the connections interval of a Bluetooth connection, which is how frequently a pair of devices exchange information. A lower connection interval means a more responsive behavior but also higher power consumption.

Parameter DEFAULT_DESIRED_CONN_TIMEOUT defines how long to receive a peer response before a connection is deemed lost. Parameter DEFAULT_ENABLE_UPDATE_REQUEST defines if the slave device is allowed to change connection interval during run-time. It is useful in terms of power saving to have different connection parameters for busy and idle phases.

The parameter SBP_PERIODIC_EVT_PERIOD defines the period of a clock event that will allow the task executes a function call periodically. This is the perfect place for us to add the code to read temperature and notify the service clients.

The periodic clock is initiated in the SimpleBLEPeripheral_init function.

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

This will create a clock with a period of SBP_PERIODIC_EVT_PERIOD. And on timeout, will call the function of SimpleBLEPeripheral_clockHandler with parameter SBP_PERIODIC_EVT. The clock event can then be triggered by

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

Searching for the keyword Util_startClock, we can find that this periodic clock is first triggered on the GAPROLE_CONNECTED event (inside the SimpleBLEPeripheral_processStateChangeEvt function), which means the task will start a periodic routine once it establishes a connection with a host.

When the periodic clock times out, its registered callback function will be called.

```
/*********************************************************************
 * @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);
}
```

This function set a flag in the events vector and activate the application from the OS task list. Notice that we do not do any specific user workload in this callback function, because it is NOT recommended. User workload often involve calls to BLE stack APIs. **Doing BLE stack API calls inside a callback functions often result in system exceptions.** Instead, we set a flag in the events vector of the task and wait for it to be processed later in the application context. The entry point for the example task is 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();
    }

  }
}
```

It is an infinite loop that keeps polling the task's stack and application message queues. It also checks its events vector for various flags. That's where the periodical routine is *actually* executed. On discovery of a SBP_PERIODIC_EVT, the task function first clears the flag, starts the same timer immediately and calls the routine function 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);
}
```

Inside the periodic function, we run our very specific job of reading temperature, generating UART requests etc.. Then we call the SimpleProfile_SetParameter() API to communicate the information to service clients through Bluetooth connection. The BLE stack takes care of all low level jobs from maintaining the wireless connection to transmitting message over the Bluetooth link. All developers need to do is to gather the application specific data and update them to corresponding characteristics in a service table.

Finally, when a write request is performed on a write-permitted characteristics, a callback function will be evoked.

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

Again, this callback function only enqueues an application message for the user task, which will be handled later in the application context.

```
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;
  }
}
```

In the above example, when the SIMPLEPROFILE_CHAR1 is written, the user code will first fetch the new value by calling SimpleProfile_GetParameter(), and then parse the data for user defined commands.

In summary, the simple_peripheral.c shows an example of how to create user task for custom workloads. A basic way to schedule application workload is by periodic clock event. Developers only need to process information to/from the characteristics in the service table while the BLE stack takes care of the rest of communicating the information from the service table to peer devices (or vice versa) through Bluetooth connection.

Connecting Real World Sensors

For BLE slave devices to do any useful work, the GPIOs of the wireless MCU are almost always involved. For instance, to read temperature from an external sensor, the ADC functionality of GPIO pins may be required. TI's CC2640 MCU features a maximum of 31 GPIOs, given different packaging types.

In the hardware side, CC2640 provides a rich set of peripheral functionalities such as ADC, UARTS, SPI, SSI, I2C etc. In the software side, TI's BLE stack tries to offer a uniform device-independent driver interface for different peripherals. A uniform driver interface may improve the chance of code re-usability, but on the other hand, it also increases the slope of the learning curve. In this note, we use the SPI controller as an example and show how to integrate the software driver into user applications.

Basic SPI Driver Flow

In TI's BLE stack, a peripheral driver often consists of three parts: a device independent specification of the driver APIs; a device specific implementation of the driver APIs and a mapping of hardware resource.

For the SPI controller, its driver implementation involves three files:

- <ti/drivers/SPI.h> -- this is the device-independent API specification
- <ti/drivers/spi/SPICC26XXDMA.h> -- this is the CC2640-specific API implementation
- <ti/drivers/dma/UDMACC26XX.h> -- this is the uDMA driver required by the SPI driver

(Note: the best document for the peripheral drivers of TI's BLE stack can mostly be found at their header files, such as SPICC26XXDMA.h in this case)

To start using the SPI controller, let's first create a custom c file, namely sbp_spi.c, that include the three header files above. The natural next step is to create an instance of the driver and initiate it. The driver instance is encapsulated in the data structure -- SPI_Handle. Another data structure -- SPI_Params is used to specify the key parameters for the SPI controller, such as bit rate, transfer mode, 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);
}
```

The above sample code exemplifies how to initialize the SPI_Handle instance. The API SPI_init() has to be called first to initialize internal data structures. The function call SPI_Params_init(&spiParams) sets all fields of SPI_Params structure to default values. Then developers can modify key parameters to suit their specific cases. For example, the above code sets the SPI controller to operate in master mode with a bit rate of 800kbps and uses a non-blocking method to process each transaction, so that when a transaction is completed the callback function sbp_spiCallback will be called.

Finally, a call to the SPI_open() opens the hardware SPI controller and return a handle for later-on SPI transactions. The SPI_open() takes two arguments, the first is the ID of the SPI controller.

CC2640 features two hardware SPI controllers on-chip, thus this ID arguments will be either 0 or 1 as defined below. The second argument is the desired parameters for the SPI controller.

```
/*!
 *  @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;
```

After successful opening of the SPI_Handle, developers can initiate SPI transactions immediately. Each SPI transaction is described using the data structure -- 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;
```

For example, to start a write transaction on the SPI bus, developers need to prepare a 'txBuf' filled with data to be transmitted and set the 'count' variable to the length of data bytes to be sent. Finally, a call to the SPI_transfer(spiHandle, spiTrans) signals the SPI controller to start the 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);
}
```

Because SPI is a duplex protocol that both transmitting and receiving happens at the same time, when a write transaction finished, its corresponding response data is already available at the

'rxBuf'.

Since we set the transfer mode to callback mode, whenever a transaction is completed, the registered callback function will be called. This is where we handle the response data or initiate the next transaction. **(Note: always remember not to do more than necessary API calls inside a callback function).**

```
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);
}
```

I/O Pin Configuration

Till now, it seems reasonably simple to use the SPI driver. But wait, how can connect the software API calls to physical SPI signals? This is done through three data structures: SPICC26XXDMA_Object, SPICC26XXDMA_HWAttrsV1 and SPI_Config. They are normally instantiated at a different location like '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}
};
```

The SPI_Config array has a separate entry for each hardware SPI controller. Each entry has three fields: fxnTablePtr, object and hwAttrs. The 'fxnTablePtr' is a point table that points to the device-specific implementations of the driver API.

The 'object' keeps track of information like driver state, transfer mode, callback function for the driver. This 'object' is automatically maintained by the driver.

The 'hwAttrs' stores the actual hardware resource mapping data, e.g. the IO pins for the SPI signals, the hardware interruption number, base address of the SPI controller etc. Most fields of the 'hwAttrs' are pre-defined and cannot be modified. Whereas the IO pins of the interface can be freely assigned based user cases. **Note: the CC26XX MCUs decouple the IO pins from specific peripheral functionality that any of the IO pins can be assigned to any peripheral function.**

Of course the actual IO pins have to be defined first in the 'board.h'.

```
#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
```

As a result, after configuration of hardware resource mapping, developers can finally communicate with external sensor chips through SPI interface.

Read Get Started with TI's BLE Stack online: https://riptutorial.com/bluetooth/topic/7058/get-started-with-ti-s-ble-stack

Chapter 5: Open L2CAP socket for Low Energy communication

Examples

In C, with Bluez

```
int get_l2cap_connection () {
```

First off, all the variables we need, explanation for will follow at the appropriate spot.

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

First, we need to create a socket, that we can accept a connection from. The socket family is `PF_BLUETOOTH`, socket type is `SOCK_SEQPACKET` (we want to have a TCP-like socket, not raw), and the protocol is the Bluetooth protocol L2CAP (`BTPROTO_L2CAP`).

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

We want to make sure the it was succesful:

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

We now have to fill the source address structure with a wildcard address, so any Bluetooth device with any address can connect to us. The wildcard address is defined as `BDADDR_ANY` in `bluetooth.h`. To copy it into the address structure, we can use the `bacpy` function. We also have to set the address family, address type and channel ID.

```
    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);
```

Setting the SO_REUSEADDR option will allow us to quickly call bind again if necessary (this can be left out):

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

Next we have to bind the socket with the source address structure we just defined. Again, we check the return value to make sure it worked.

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

Next up is setting the security level. Note that this step is optional, but setting the security level to MEDIUM will allow automatic pairing with the device (the kernel handles the actual pairing).

```
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;
}
```

Now we can tell the kernel that our ssock is a passive socket, that will accept a connection. The second parameter is the backlog. If you want to know more, the manpage of listen contains all the information you need.

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

Now we can wait for an incoming connection. The peer_addr structure will contain the address of the connected device, once accept returns. csock will be the file descriptor of the socket we can read from/write to, to communicate with the connected device.

```
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;
}
```

We can print the address of the connected device (optional, of course). We can use the batostr function to convert the Bluetooth address to a string.

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

If we don't want any other devices to connect, we should close the server socket. Do the same thing with csock, after your communication with the device is finished.

```
close(ssock);
```

```
    return csock;
}
```

Read Open L2CAP socket for Low Energy communication online:
https://riptutorial.com/bluetooth/topic/6558/open-l2cap-socket-for-low-energy-communication

Credits

S. No	Chapters	Contributors
1	Getting started with bluetooth	Community, Jon Carlstedt, Lasse Meyer
2	Get Started With Bluetooth LE on Windows	Carter
3	Get Started with Bluetooth on the Web	François Beaufort
4	Get Started with TI's BLE Stack	RamenChef, Zefu Dai
5	Open L2CAP socket for Low Energy communication	Lasse Meyer