EBook Gratuito

APPRENDIMENTO bluetooth

Free unaffiliated eBook created from
Stack Overflow contributors.

#bluetooth

Sommario

Di

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Capitolo 1: Iniziare con bluetooth

Osservazioni

Il Bluetooth è uno standard industriale per la trasmissione di dati wireless tra dispositivi a breve distanza. È stato definito per la prima volta negli anni '90 dal Bluetooth Special Interest Group (SIG), in IEEE 802.15.1. La trasmissione dei dati senza connessione e orientata alla connessione è possibile, per uno o più dispositivi, in modalità Ad-hoc o piconet.

Il nome Bluetooth deriva dal re danese Harald Blauzahn, Blauzahn che significa "dente blu" in inglese. La parte principale dello sviluppo iniziale è stata fatta dal professore olandese Jaap Haartsen per la società Ericsson. Successivamente, Intel e Nokia sono stati i principali contributori.

Le frequenze utilizzate sono nella banda ISM senza licenza, tra 2,402 GHz e 2,480 GHz, e possono quindi essere utilizzate senza autorizzazione in tutto il mondo. Sono possibili interferenze con reti Wifi, telefoni wireless o microonde, tutte funzionanti nella stessa banda ISM.

Examples

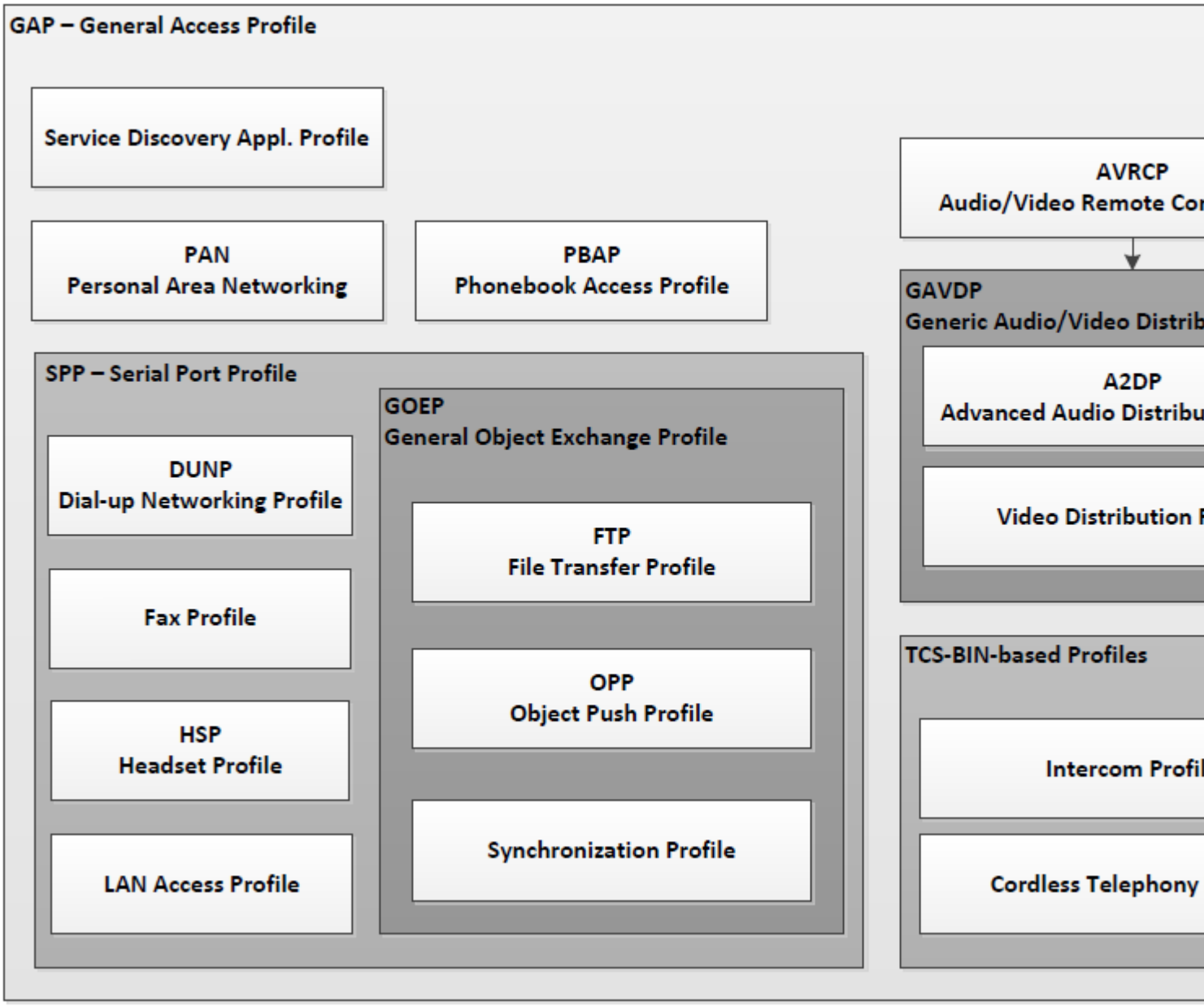
Installazione o configurazione

Istruzioni dettagliate su come impostare o installare Bluetooth.

Profili

Le specifiche Bluetooth contengono diverse specifiche del profilo. Un profilo descrive come utilizzare e implementare una funzione.

Possono dipendere l'uno dall'altro, ecco un layout di base delle dipendenze del profilo più comuni



Tutti i profili possono essere trovati presso BT SIG , essere consapevoli del fatto che diverse versioni potrebbero contenere diverse funzionalità. Si noti inoltre che alcuni profili contengono diverse categorie, a volte facoltative, pertanto assicurarsi che il dispositivo supporti la categoria in questione. Ecco alcuni dei *profili smartphone* più comuni e le loro specifiche

A2DP - Profilo di distribuzione audio avanzato
Il profilo di distribuzione audio avanzato (A2DP) definisce i protocolli e le procedure che realizzano la distribuzione di contenuti audio di alta qualità in mono o stereo su canali ACL. Un tipico caso d'uso è lo streaming di contenuti musicali da un lettore musicale stereo a cuffie o altoparlanti. I dati audio vengono compressi in un formato appropriato per l'uso efficiente della larghezza di banda limitata.
Dipendenze: GAVDP, GAP

AVRCP - Profilo di controllo remoto audio / video
Il profilo di controllo remoto audio / video (AVRCP) definisce le caratteristiche e le procedure necessarie per garantire l'interoperabilità tra i dispositivi Bluetooth con funzioni di controllo audio /

video negli scenari di distribuzione Audio / Video. Questo profilo adotta il modello di dispositivo AV / C e il formato di comando per i messaggi di controllo e tali messaggi vengono trasportati dal protocollo di trasporto audio / video (AVCTP). In questo profilo, il controller trasferisce l'azione dell'utente rilevata al segnale di controllo A / V e quindi la trasmette a un dispositivo Bluetooth remoto. Inoltre, il profilo utilizza estensioni specifiche Bluetooth per supportare il trasferimento di metadati relativi al contenuto da trasferire tra dispositivi Bluetooth. Il telecomando descritto in questo profilo è progettato specificamente per il controllo A / V.
Dipendenze: GAP

HFP - Profilo a mani libere
Questo documento definisce i protocolli e le procedure che devono essere utilizzati dai dispositivi che implementano il profilo vivavoce. Gli esempi più comuni di tali dispositivi sono le unità Hands-Free in-car utilizzate insieme ai telefoni cellulari o le cuffie wireless indossabili. Il profilo definisce in che modo due dispositivi che supportano il profilo vivavoce interagiscono tra loro su base punto-punto. Un'implementazione del profilo a mani libere consente in genere a un telefono cellulare o a un'unità vivavoce integrata di connettersi, in modalità wireless, a un telefono cellulare allo scopo di fungere da meccanismo di input e output audio del telefono cellulare e di consentire le tipiche funzioni di telefonia eseguito senza accesso al telefono reale.
Dipendenze: SPP, GAP

HSP - Profilo auricolare
Questo profilo di cuffia definisce i protocolli e le procedure che devono essere utilizzati dai dispositivi che richiedono una connessione audio full duplex combinata con comandi di controllo dispositivo minimi. Gli esempi più comuni di tali dispositivi sono cuffie, personal computer, PDA e telefoni cellulari, anche se la maggior parte dei telefoni cellulari preferirà utilizzare un profilo più avanzato come il profilo Hands-Free. L'auricolare può essere connesso in modalità wireless allo scopo di fungere da meccanismo di input e output audio del dispositivo, fornendo un audio full duplex.
Dipendenze: SPP, GAP

PBAP - Profilo di accesso alla rubrica
Il profilo di accesso alla rubrica telefonica (PBAP) definisce i protocolli e le procedure che devono essere utilizzati dai dispositivi per il recupero degli oggetti della rubrica. Si basa su un modello di interazione Client-Server in cui il dispositivo Client estrae gli oggetti della rubrica dal dispositivo Server. Questo profilo è particolarmente adatto per il caso di utilizzo Hands-Free (cioè implementato in combinazione con il "Profilo a mani libere" o il "Profilo di accesso SIM"). Fornisce numerose funzionalità che consentono una gestione avanzata degli oggetti della rubrica, in base alle necessità nell'ambiente automobilistico. In particolare, è molto più ricco di Object Push Profile (che potrebbe essere utilizzato per spingere la voce della rubrica in formato vCard da un dispositivo a un altro). Questo profilo può anche essere applicato ad altri casi di utilizzo in cui un dispositivo Client deve estrarre gli oggetti della rubrica da un dispositivo Server. Si noti tuttavia che questo profilo consente solo la consultazione degli oggetti della rubrica (sola lettura). Non è possibile modificare il contenuto dell'oggetto originale della rubrica (leggi / scrivi).
Dipendenze: GOEP, SPP, GAP

Leggi Iniziare con bluetooth online: https://riptutorial.com/it/bluetooth/topic/4846/iniziare-con-bluetooth

Capitolo 2: Aprire la presa L2CAP per la comunicazione a bassa energia

Examples

In C, con Bluez

```
int get_l2cap_connection () {
```

Prima di tutto, tutte le variabili di cui abbiamo bisogno, la spiegazione per seguirà nel punto appropriato.

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

Per prima cosa, dobbiamo creare un socket, da cui possiamo accettare una connessione. La famiglia di socket è `PF_BLUETOOTH` , il tipo di socket è `SOCK_SEQPACKET` (vogliamo avere un socket simile a TCP, non raw) e il protocollo è il protocollo Bluetooth L2CAP (`BTPROTO_L2CAP`).

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

Vogliamo assicurarci che sia stato un successo:

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

Ora dobbiamo riempire la struttura dell'indirizzo sorgente con un indirizzo jolly, in modo che qualsiasi dispositivo Bluetooth con qualsiasi indirizzo possa connetterci. L'indirizzo jolly è definito come `BDADDR_ANY` in `bluetooth.h` . Per copiarlo nella struttura dell'indirizzo, possiamo usare la funzione `bacpy` . Dobbiamo anche impostare la famiglia di indirizzi, il tipo di indirizzo e l'ID del canale.

```
    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);
```

L'impostazione dell'opzione SO_REUSEADDR ci consentirà di richiamare rapidamente il bind, se necessario (questo può essere escluso):

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

Quindi dobbiamo legare il socket con la struttura dell'indirizzo sorgente che abbiamo appena definito. Ancora una volta, controlliamo il valore di ritorno per assicurarci che funzioni.

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

Il prossimo è impostare il livello di sicurezza. Si noti che questo passaggio è facoltativo, ma l'impostazione del livello di sicurezza su MEDIUM consentirà l'accoppiamento automatico con il dispositivo (il kernel gestisce l'accoppiamento effettivo).

```
    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;
    }
```

Ora possiamo dire al kernel che il nostro ssock è un socket passivo, che accetterà una connessione. Il secondo parametro è il backlog. Se vuoi saperne di più, la manpage di listen contiene tutte le informazioni di cui hai bisogno.

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

Ora possiamo aspettare una connessione in arrivo. La struttura peer_addr conterrà l'indirizzo del dispositivo connesso, una volta accettata la restituzione. csock sarà il descrittore di file del socket da cui possiamo leggere / scrivere, per comunicare con il dispositivo connesso.

```
    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;
    }
```

Possiamo stampare l'indirizzo del dispositivo connesso (facoltativo, ovviamente). Possiamo usare la funzione batostr per convertire l'indirizzo Bluetooth in una stringa.

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

Se non vogliamo connettere altri dispositivi, dovremmo chiudere il socket del server. Fai la stessa

cosa con csock, una volta terminata la comunicazione con il dispositivo.

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

Leggi Aprire la presa L2CAP per la comunicazione a bassa energia online: https://riptutorial.com/it/bluetooth/topic/6558/aprire-la-presa-l2cap-per-la-comunicazione-a-bassa-energia

Capitolo 3: Inizia con Bluetooth LE su Windows

Osservazioni

Documentazione

- Pubblicità - Una rappresentazione di un carico utile pubblicitario di Bluetooth LE.
- Annuncio Editore - Gestisce l'invio di annunci Bluetooth LE.
- Advertisement Watcher - Gestisce la visione degli annunci Bluetooth LE.

Gli appunti

- Windows 10 può solo agire in modalità centrale, quindi può connettersi solo a dispositivi che supportano la modalità periferica. Per questo motivo, due dispositivi Windows 10 non possono connettersi tramite Bluetooth LE.
- Windows 10 deve essere associato a un dispositivo Bluetooth LE per connettersi ad esso.

Examples

Configurazione iniziale

Per utilizzare qualsiasi funzionalità Bluetooth su un'app Universal Platform Platform, è necessario verificare la funzionalità `Bluetooth` nel `Package.appxmanifest` .

1. Apri `Package.appxmanifest`
2. Vai alla scheda `Capabilities`
3. Trova `Bluetooth` sulla sinistra e seleziona la casella accanto ad essa

Creare un annuncio LE Bluetooth

Questo esempio mostra come pubblicizzare un payload personalizzato da un dispositivo Windows 10 in primo piano. Il payload utilizza una società composta (identificata come 0xFFFE) e pubblicizza la stringa `Hello World` nella pubblicità.

```
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();
```

Nota: questo è solo per la pubblicità in primo piano (mentre l'app è aperta).

Ascolta un annuncio Bluetooth LE

Ascolto generale

Questo esempio mostra come ascoltare un annuncio specifico.

```
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();
```

Filtro pubblicitario: ascolto di un annuncio specifico

A volte vuoi ascoltare per un annuncio specifico. In questo caso, ascolta un annuncio contenente un carico utile con una società costituita (identificata come 0xFFFE) e contenente la stringa Hello World nell'annuncio. Questo può essere accoppiato con l'esempio di *creazione di un annuncio LE Bluetooth* per avere una pubblicità di macchine Windows e un altro ascolto.

Nota: assicurati di impostare questo filtro pubblicitario prima di avviare l'osservatore!

```
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);
```

Filtro del segnale: ascolto per annunci prossimali

A volte vuoi solo attivare il tuo osservatore quando la pubblicità del dispositivo è arrivata nel raggio d'azione. È possibile definire il proprio intervallo, basta notare che i valori normali sono compresi tra 0 e -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);
```

callback

```
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
}
```

Nota: questo è solo per l'ascolto in primo piano.

Valutazione della distanza in base a RSSI da una pubblicità Bluetooth LE

Quando viene attivata la richiamata del Bluetooth LE Watcher, gli eventArgs includono un valore RSSI che indica la forza del segnale ricevuto (quanto è forte il

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

Questo può essere approssimativamente tradotto in distanza, ma non dovrebbe essere usato per misurare le vere distanze in quanto ogni singola radio è diversa. Diversi fattori ambientali possono rendere difficile la misurazione della distanza (ad esempio muri, custodie intorno alla radio o persino umidità dell'aria).

Un'alternativa al giudicare la pura distanza è definire "secchi". Le radio tendono a riportare da 0 a -50 dBm quando sono molto vicine, da -50 a -90 quando sono a media distanza e sotto -90

quando sono lontane. La prova e l'errore sono i migliori per determinare cosa vuoi che questi bucket siano per la tua applicazione.

Leggi Inizia con Bluetooth LE su Windows online:
https://riptutorial.com/it/bluetooth/topic/5553/inizia-con-bluetooth-le-su-windows

Capitolo 4: Inizia con Bluetooth sul Web

Osservazioni

fonti:

- 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

Leggere il livello della batteria da un dispositivo Bluetooth nelle vicinanze (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);
  });
}
```

Resetta l'energia spesa da un dispositivo Bluetooth nelle vicinanze (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);
  });
}
```

Leggi Inizia con Bluetooth sul Web online: https://riptutorial.com/it/bluetooth/topic/4936/inizia-con-bluetooth-sul-web

Capitolo 5: Inizia con lo stack BLE di TI

Examples

Connessione a dispositivi slave BLE

introduzione

I SoC CC26XX della serie Texas Instruments (TI) sono MCU wireless prontamente disponibili destinati alle applicazioni Bluetooth Low Energy (BLE). Insieme agli MCU, TI offre uno stack software completo che fornisce i codici API e di esempio necessari per aiutare rapidamente gli sviluppatori a iniziare con la catena di strumenti. Tuttavia, per i principianti, c'è sempre la domanda su dove iniziare di fronte a una lunga lista di documenti e codici di riferimento. Questa nota mira a registrare i passi necessari necessari per dare il via al primo progetto in corso.

Il Simple Peripheral Profile è l'esempio "Hello World" dello stack BLE, in cui MCU funge da periferica BLE per gli host upstream o client di servizio BLE, come PC e smartphone. Le comuni applicazioni del mondo reale includono: cuffia Bluetooth, sensore di temperatura Bluetooth, ecc.

Prima di iniziare, dobbiamo prima raccogliere strumenti software e hardware di base per la programmazione e il debug.

1. Pila BLE

   Scarica e installa TI BLE-STACK-2-2-0 dal sito ufficiale. Supponiamo che sia installato nella posizione predefinita "C: \ ti".

2. IDE - ci sono due opzioni:

- Banco da lavoro embedded IAR per ARM. Questo è uno strumento commerciale con un periodo di valutazione gratuito di 30 giorni.
- Code Composer Studio di TI (CCS). IDE ufficiale di TI e offre licenza gratuita. In questo esempio useremo CCS V6.1.3

3. Strumento di programmazione hardware

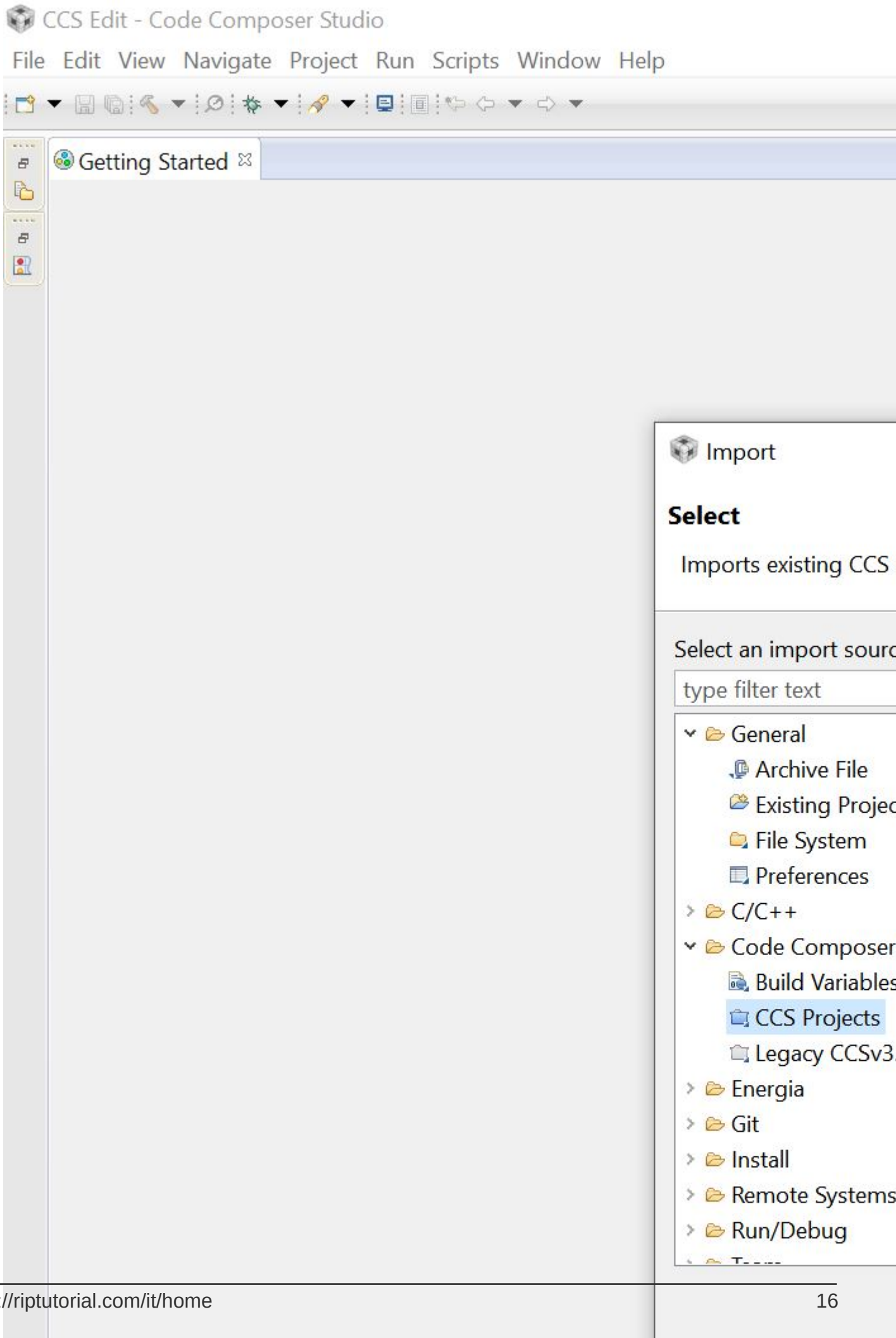
   Consiglia il dispositivo JTAG con interfaccia USB XDS100 di TI.

Importa progetto di esempio in CCS

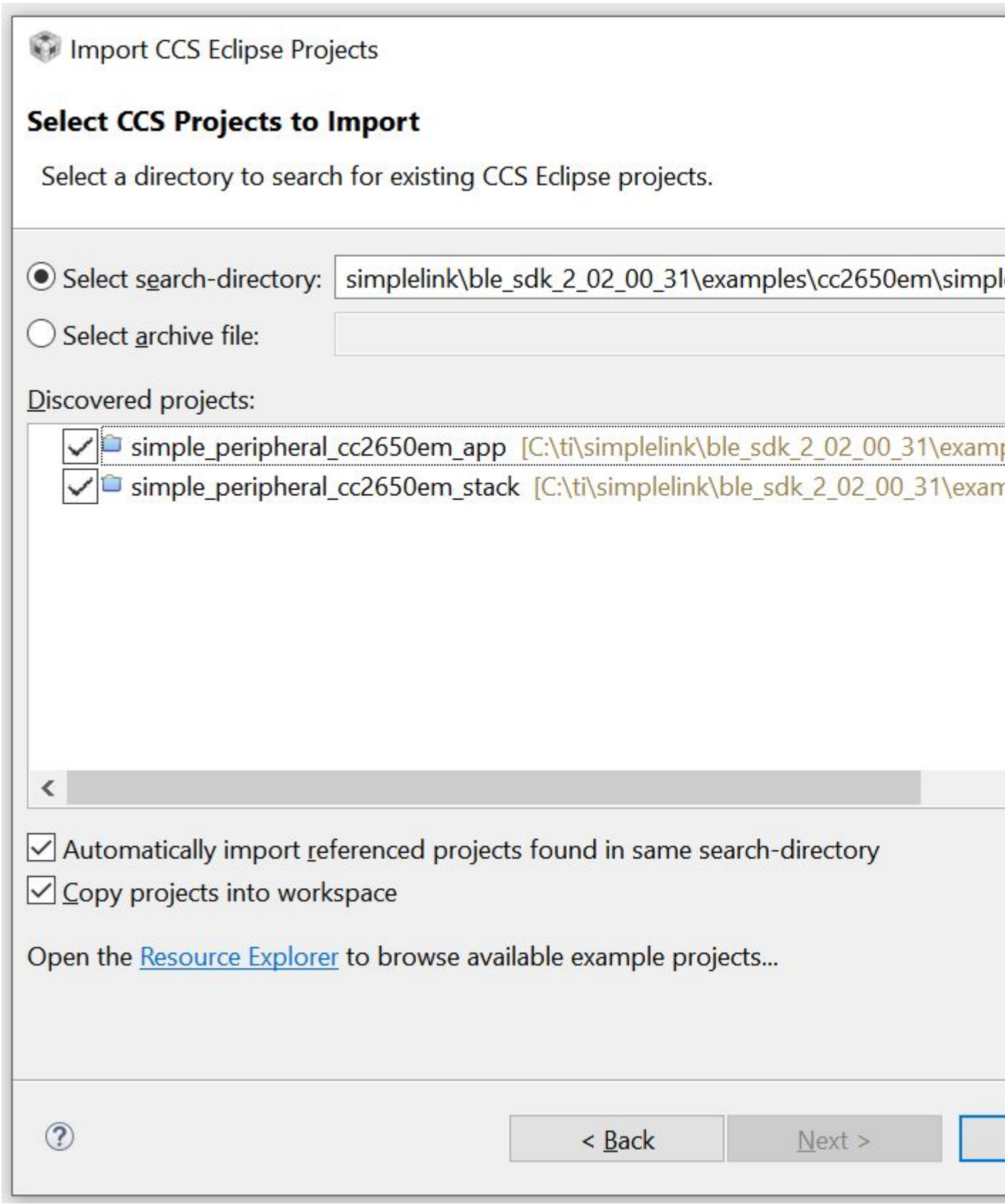
Il codice di esempio Simple Peripheral Profile viene fornito con l'installazione BLE-Stack. Seguire i passaggi seguenti per importare questo esempio di progetto in CCS.

1. Avvia CCS, crea una cartella di lavoro. Quindi File-> Importa. Sotto 'Seleziona una fonte di importazione', seleziona l'opzione 'Code Compose Studio -> Progetti CCS' e fai clic su

'Avanti'.


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. Passare a "C: \ ti \ simplelink \ ble_sdk_2_02_00_31 \ examples \ cc2650em \ simple_peripheral \ ccs". Due progetti saranno scoperti. Seleziona tutto e seleziona entrambe le opzioni di seguito. Quindi fare clic su "Fine". Copiando i progetti nello spazio di lavoro, si lascia invariata l'impostazione originale del progetto per tutte le successive modifiche.



L'esempio di Simple Peripheral Profile include due progetti:

- simple_peripheral_cc2650em_app
- simple_peripheral_cc2650em_stack

'cc2650em' è il nome in codice per la scheda di valutazione cc2650 di TI. Il progetto _stack include i codici e il binario di BEL-Stack-2-2-0 di TI, che gestisce la pubblicità Bluetooth, l'handshake, la sincronizzazione di frequenza, ecc. Questa è la parte del codice che è relativamente stabile e non vuole essere toccato dagli sviluppatori il più delle volte. Il progetto _app è il luogo in cui gli sviluppatori implementano i propri compiti e il servizio BLE.

Costruisci e scarica

Fai clic sui menu "Progetto-> Crea tutto" per creare entrambi i progetti. Se il compilatore segnala una sorta di errore interno sul collegamento, prova a disabilitare l'opzione 'compress_dwarf' per il linker:

- fare clic con il tasto destro del mouse sul progetto e selezionare "Propoerties".
- in 'Build-> ARM Linker', fai clic sul pulsante 'Modifica flag'.
- modifica l'ultima opzione a '--compress_dwarf = off'.

Dopo che entrambi i progetti sono stati creati correttamente, fai clic su "Esegui-> esegui il debug" separatamente per scaricare sia le immagini dello stack che quelle delle app sull'MCU.

Tocca il codice

Per essere in grado di apportare modifiche aggressive al codice di esempio, gli sviluppatori devono acquisire conoscenze dettagliate sulla struttura a livelli dello stack BLE. Per compiti elementari come la lettura / notifica della temperatura, possiamo concentrarci solo su due file: PROFILI / simple_gatt_profile.c (.h) e Application / simple_peripheral.c (.h)

simple_gatt_profile.c

Tutte le applicazioni Bluetooth offrono un determinato tipo di servizio, ciascuna costituita da un insieme di caratteristiche. Il semplice profilo periferico definisce un servizio semplice, con l'UUID di 0xFFF0, che consiste di 5 caratteristiche. Questo servizio è specificato in simple_gatt_profile.c. Un riepilogo del servizio semplice è elencato come segue.

Nome	Dimensione dei dati	UUID	Descrizione	Proprietà
simplePeripheralChar1	1	0xFFF1	Caratteristiche 1	Leggere scrivere
simplePeripheralChar2	1	0xFFF2	Caratteristiche 2	Sola lettura
simplePeripheralChar3	1	0xFFF3	Caratteristiche 3	Scrivi solo

Nome	Dimensione dei dati	UUID	Descrizione	Proprietà
simplePeripheralChar4	1	0xFFF4	Caratteristiche 4	Notificare
simplePeripheralChar5	5	0xFFF5	Caratteristiche 5	Sola lettura

Le cinque caratteristiche hanno proprietà diverse e servono come esempi per vari casi di utenti. Ad esempio, MCU può utilizzare simplePeripheralChar4 per notificare ai suoi clienti, host a monte, la modifica delle informazioni.

Per definire un servizio Bluetooth, si deve costruire una tabella degli attributi.

```
/*********************************************************************
 * 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 tabella degli attributi inizia con un 'primaryServiceUUID' predefinito, che specifica l'UUID del servizio (0xFFF0 in questo caso). Poi è seguito da dichiarazioni di tutte le caratteristiche che

consistono nel servizio. Ogni caratteristica ha diversi attributi, vale a dire il permesso di accesso, il valore e la descrizione dell'utente, ecc. Questa tabella viene successivamente registrata con lo stack 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 );
```

Al momento della registrazione del servizio, gli sviluppatori devono fornire tre funzioni di callback per "Leggi", "Scrivi" e "Autorizzazione" delle caratteristiche. Possiamo trovare nel codice di esempio l'elenco delle funzioni di callback.

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

Quindi, simpleProfile_ReadAttrCB verrà chiamato una volta che il client di servizio invia una richiesta di lettura tramite la connessione Bluetooth. Allo stesso modo, simpleProfile_WriteAttrCB verrà chiamato quando viene effettuata una richiesta di scrittura. Comprendere queste due funzioni è la chiave per il successo della personalizzazione del progetto.

Di seguito è riportata la funzione di callback di lettura.

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

Ho leggermente modificato il codice dalla sua versione originale. Questa funzione richiede 7 parametri, che sono spiegati nei commenti dell'intestazione. La funzione inizia controllando il permesso di accesso dell'attributo, ad esempio se ha il permesso di lettura. Quindi controlla se questa è una lettura di segmento di una richiesta di lettura del blob più grande testando la condizione 'if (offset> 0)'. Ovviamente, la funzione non supporta blob read per ora. Successivamente, viene estratto l'UUID dell'attributo richiesto. Esistono due tipi di UUID: 16 bit e 128 bit. Mentre il codice di esempio definisce tutte le caratteristiche utilizzando UUID a 16 bit, l'UUID a 128 bit è più universale e più comunemente utilizzato negli host upstream come PC e smartphone. Pertanto, vengono utilizzate diverse righe di codice per convertire l'UUID a 128 bit in UUID a 16 bit.

```
 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]);
```

Infine, dopo aver ottenuto l'UUID, possiamo determinare quale attributo è richiesto. Quindi il lavoro rimanente sul lato degli sviluppatori consiste nel copiare il valore dell'attributo richiesto nel puntatore di destinazione "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 funzione di callback di scrittura è simile tranne che esiste un tipo speciale di scrittura con UUID di GATT_CLIENT_CHAR_CFG_UUID. Questa è la richiesta dell'host upstream di registrarsi per la notifica o l'indicazione delle caratteristiche. Basta chiamare l'API GATTServApp_ProcessCCCWriteReq per passare la richiesta allo stack 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;
```

Il lato applicativo del codice sull'MCU potrebbe voler essere avvisato con qualsiasi modifica alle caratteristiche consentite dalla scrittura. Gli sviluppatori possono implementare questa notifica nel modo che preferiscono. Nel codice di esempio, viene utilizzata la funzione di callback.

```
  // 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'altra parte, se la periferica BLE vuole notificare agli host upstream qualsiasi cambiamento nelle sue caratteristiche, può chiamare l'API GATTServApp_ProcessCharCfg. Questa API è dimostrata nella funzione 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 );
}
```

Quindi, se la semplice applicazione periferica vuole notificare il valore corrente di SIMPLEPROFILE_CHAR4 ai dispositivi peer, può semplicemente chiamare la funzione SimpleProfile_SetParameter.

In sintesi, PROFILES / simple_gatt_profile.c (.h) definisce il contenuto del servizio che la periferica BLE vorrebbe presentare ai propri client, nonché i modi in cui si accede a quelle caratteristiche del servizio.

simple_peripheral.c

Lo stack BLE di TI è in esecuzione su un layer OS multi-thread lite. Per aggiungere un carico di lavoro alla MCU, gli sviluppatori devono prima creare un'attività. simple_peripheral.c mostra la struttura di base di un'attività personalizzata, che include la creazione, l'inizializzazione e la gestione delle attività. Per iniziare con le attività di base come la lettura della temperatura e la notifica, ci concentreremo su alcune funzioni chiave di seguito.

L'inizio del file definisce una serie di parametri che possono influenzare i comportamenti di connessione 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
```

I parametri DEFAULT_DESIRED_MIN_CONN_INTERVAL, DEFAULT_DESIRED_MAX_CONN_INTERVAL e DEFAULT_DESIRED_SLAVE_LATENCY definiscono insieme l'intervallo di connessioni di una connessione Bluetooth, che è la frequenza

con cui una coppia di dispositivi scambia informazioni. Un intervallo di connessione più basso significa un comportamento più reattivo ma anche un maggiore consumo di energia.

Il parametro DEFAULT_DESIRED_CONN_TIMEOUT definisce la durata della ricezione di una risposta peer prima che una connessione venga considerata persa. Il parametro DEFAULT_ENABLE_UPDATE_REQUEST definisce se il dispositivo slave può modificare l'intervallo di connessione durante l'esecuzione. È utile in termini di risparmio energetico per avere parametri di connessione diversi per le fasi di occupato e inattivo.

Il parametro SBP_PERIODIC_EVT_PERIOD definisce il periodo di un evento di clock che consentirà al task di eseguire periodicamente una chiamata di funzione. Questo è il posto perfetto per noi per aggiungere il codice per leggere la temperatura e informare i clienti del servizio.

L'orologio periodico viene avviato nella funzione SimpleBLEPeripheral_init.

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

Ciò creerà un orologio con un periodo di SBP_PERIODIC_EVT_PERIOD. E al timeout, chiamerà la funzione di SimpleBLEPeripheral_clockHandler con il parametro SBP_PERIODIC_EVT. L'evento di clock può quindi essere attivato da

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

Cercando la parola chiave Util_startClock, possiamo scoprire che questo orologio periodico viene innescato per la prima volta sull'evento GAPROLE_CONNECTED (all'interno della funzione SimpleBLEPeripheral_processStateChangeEvt), il che significa che l'attività avvierà una routine periodica una volta stabilita una connessione con un host.

Quando l'orologio periodico scade, viene richiamata la sua funzione di callback registrata.

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

Questa funzione imposta un flag nel vettore degli eventi e attiva l'applicazione dall'elenco delle attività del SO. Si noti che non eseguiamo alcun carico di lavoro specifico dell'utente in questa

funzione di callback, perché NON è raccomandato. Il carico di lavoro degli utenti spesso comporta chiamate alle API dello stack BLE. **Effettuare chiamate API stack BLE all'interno di una funzione di callback spesso genera eccezioni di sistema.** Invece, impostiamo un flag nel vettore degli eventi dell'attività e aspettiamo che venga elaborato successivamente nel contesto dell'applicazione. Il punto di ingresso per l'attività di esempio è 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();
    }

  }
}
```

È un ciclo infinito che continua a interrogare lo stack delle attività e le code dei messaggi delle applicazioni. Controlla anche il vettore degli eventi per varie bandiere. È qui che la routine periodica viene *effettivamente* eseguita. Alla scoperta di un SBP_PERIODIC_EVT, la funzione compito cancella prima il flag, avvia immediatamente lo stesso timer e chiama la funzione di 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);
}
```

All'interno della funzione periodica, eseguiamo il nostro lavoro specifico di lettura della temperatura, generando richieste UART ecc. Quindi chiamiamo l'API SimpleProfile_SetParameter () per comunicare le informazioni ai client di servizio tramite la connessione Bluetooth. Lo stack BLE si occupa di tutti i lavori di basso livello dal mantenimento della connessione wireless alla trasmissione di messaggi tramite il collegamento Bluetooth. Tutti gli sviluppatori devono fare è raccogliere i dati specifici dell'applicazione e aggiornarli alle caratteristiche corrispondenti in una tabella di servizio.

Infine, quando una richiesta di scrittura viene eseguita su caratteristiche permesse per la scrittura, verrà richiamata una funzione di callback.

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

Ancora una volta, questa funzione di callback accoda solo un messaggio dell'applicazione per l'attività dell'utente, che verrà gestito successivamente nel contesto dell'applicazione.

```
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;
  }
}
```

Nell'esempio precedente, quando viene scritto SIMPLEPROFILE_CHAR1, il codice utente preleverà prima il nuovo valore chiamando SimpleProfile_GetParameter (), quindi analizzerà i dati per i comandi definiti dall'utente.

In breve, simple_peripheral.c mostra un esempio di come creare attività utente per carichi di lavoro personalizzati. Un modo base per pianificare il carico di lavoro dell'applicazione è l'evento di orologio periodico. Gli sviluppatori devono solo elaborare le informazioni da / per le caratteristiche nella tabella dei servizi mentre lo stack BLE si occupa del resto della comunicazione delle informazioni dalla tabella dei servizi ai dispositivi peer (o viceversa) tramite la

connessione Bluetooth.

Collegamento dei sensori del mondo reale

Perché i dispositivi slave BLE possano svolgere qualsiasi lavoro utile, i GPIO della MCU wireless sono quasi sempre coinvolti. Ad esempio, per leggere la temperatura da un sensore esterno, potrebbe essere necessaria la funzionalità ADC dei pin GPIO. L'MCU CC2640 di TI presenta un massimo di 31 GPIO, a seconda del tipo di imballaggio.

Dal punto di vista hardware, CC2640 offre un ricco set di funzionalità periferiche come ADC, UARTS, SPI, SSI, I2C, ecc. Nel lato software, lo stack BLE di TI cerca di offrire un'interfaccia driver indipendente dalla periferica uniforme per diverse periferiche. Un'interfaccia driver uniforme può migliorare la possibilità di riusabilità del codice, ma d'altra parte aumenta anche la pendenza della curva di apprendimento. In questa nota, utilizziamo il controller SPI come esempio e mostriamo come integrare il driver del software in applicazioni utente.

Flusso base del driver SPI

Nello stack BLE di TI, un driver periferico è spesso costituito da tre parti: una specifica indipendente dal dispositivo delle API del driver; un'implementazione specifica del dispositivo delle API del driver e una mappatura della risorsa hardware.

Per il controller SPI, l'implementazione del driver prevede tre file:

- <ti / drivers / SPI.h>: questa è la specifica dell'API indipendente dalla periferica
- <ti / drivers / spi / SPICC26XXDMA.h> - questa è l'implementazione dell'API specifica per CC2640
- <ti / drivers / dma / UDMACC26XX.h> - questo è il driver uDMA richiesto dal driver SPI

(Nota: il documento migliore per i driver periferici dello stack BLE di TI può essere trovato principalmente nei loro file di intestazione, come SPICC26XXDMA.h in questo caso)

Per iniziare a utilizzare il controller SPI, creiamo innanzitutto un file c personalizzato, ovvero sbp_spi.c, che include i tre file di intestazione sopra. Il naturale passo successivo è creare un'istanza del driver e avviarlo. L'istanza del driver è incapsulata nella struttura dati - SPI_Handle. Un'altra struttura dati: SPI_Params viene utilizzata per specificare i parametri chiave per il controller SPI, come bit rate, modalità di trasferimento, ecc.

```
#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);
}
```

Il codice di esempio sopra esemplificativo illustra come inizializzare l'istanza SPI_Handle. L'API SPI_init () deve essere chiamata prima per inizializzare le strutture di dati interne. La funzione chiama SPI_Params_init (& spiParams) imposta tutti i campi della struttura SPI_Params sui valori predefiniti. Quindi gli sviluppatori possono modificare i parametri chiave per soddisfare i loro casi specifici. Ad esempio, il codice precedente imposta il controller SPI per operare in modalità master con una velocità in bit di 800kbps e utilizza un metodo non bloccante per elaborare ogni transazione, in modo che quando una transazione viene completata, la funzione di callback sbp_spiCallback venga chiamata.

Infine, una chiamata a SPI_open () apre il controller SPI hardware e restituisce un handle per le transazioni SPI successive. SPI_open () accetta due argomenti, il primo è l'ID del controller SPI. CC2640 dispone di due controller SPI hardware on-chip, quindi gli argomenti ID saranno 0 o 1 come definito di seguito. Il secondo argomento è i parametri desiderati per il controller 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;
```

Dopo aver aperto con successo SPI_Handle, gli sviluppatori possono avviare immediatamente transazioni SPI. Ogni transazione SPI è descritta utilizzando la struttura dati - 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;
```

Ad esempio, per avviare una transazione di scrittura sul bus SPI, gli sviluppatori devono preparare un 'txBuf' pieno di dati da trasmettere e impostare la variabile 'conteggio' sulla lunghezza dei byte di dati da inviare. Infine, una chiamata a SPI_transfer (spiHandle, spiTrans) segnala al controller SPI di avviare la transazione.

```
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);
}
```

Poiché SPI è un protocollo duplex che trasmette e riceve allo stesso tempo, quando una transazione di scrittura è terminata, i relativi dati di risposta sono già disponibili su "rxBuf".

Poiché impostiamo la modalità di trasferimento sulla modalità di richiamata, ogni volta che una transazione viene completata, viene richiamata la funzione di richiamata registrata. Qui è dove gestiamo i dati di risposta o iniziamo la transazione successiva. **(Nota: ricorda sempre di non fare più delle necessarie chiamate API all'interno di una funzione di callback).**

```
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);
}
```

Configurazione pin I / O

Fino ad ora, sembra abbastanza semplice usare il driver SPI. Ma aspetta, come è possibile connettere le chiamate API del software ai segnali SPI fisici? Questo viene fatto attraverso tre strutture dati: SPICC26XXDMA_Object, SPICC26XXDMA_HWAttrsV1 e SPI_Config. Normalmente vengono istanziati in una posizione diversa come "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}
};
```

L'array SPI_Config ha una voce separata per ogni controller SPI hardware. Ogni voce ha tre campi: fxnTablePtr, object e hwAttrs. 'FxnTablePtr' è una tabella di punti che punta alle implementazioni specifiche del dispositivo dell'API del driver.

L '"oggetto" tiene traccia delle informazioni come stato del driver, modalità di trasferimento, funzione di callback per il conducente. Questo "oggetto" viene gestito automaticamente dal conducente.

'HwAttrs' memorizza i dati di mappatura delle risorse hardware effettivi, ad es. I pin IO per i segnali SPI, il numero di interruzione hardware, l'indirizzo base del controller SPI ecc. La maggior parte dei campi 'hwAttrs' sono predefiniti e non possono essere modificati. Mentre i pin IO dell'interfaccia possono essere liberamente assegnati ai casi utente. **Nota: gli MCU CC26XX scollegano i pin IO da specifiche funzionalità periferiche che uno qualsiasi dei pin IO può essere assegnato a qualsiasi funzione periferica.**

Ovviamente i pin IO effettivi devono essere definiti prima nel "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
```

Di conseguenza, dopo la configurazione della mappatura delle risorse hardware, gli sviluppatori possono finalmente comunicare con i chip dei sensori esterni tramite l'interfaccia SPI.

Leggi Inizia con lo stack BLE di TI online: https://riptutorial.com/it/bluetooth/topic/7058/inizia-con-lo-stack-ble-di-ti

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con bluetooth	Community, Jon Carlstedt, Lasse Meyer
2	Aprire la presa L2CAP per la comunicazione a bassa energia	Lasse Meyer
3	Inizia con Bluetooth LE su Windows	Carter
4	Inizia con Bluetooth sul Web	François Beaufort
5	Inizia con lo stack BLE di TI	RamenChef, Zefu Dai