UG10184

Bluetooth Low Energy Application Developer’s Guide

Rev. 1.0 — 26 November 2024 **User guide**

Document information

Information	Content
Keywords	UG10184, NXP Bluetooth Low Energy Host Stack, Application Programming Interface (API), BLE host stack APIs, initialization, Generic Access Profile (GAP) Layer, Generic Attribute Profile (GATT) Layer
Abstract	This document explains how to integrate the NXP Bluetooth Low Energy Host Stack in an application. It also describes the most commonly used APIs and provides code examples. These examples are applicable to NXP hardware platforms using KW45, KW47, MCXW71, MCXW72, and K32W1 family of devices.

1 Introduction

This document explains how to integrate the NXP Bluetooth Low Energy Host Stack in an application and provides detailed explanation of the most commonly used APIs and code examples.

- Section 1 "Introduction": This section outlines the document structure.
- Section 2 "Prerequisites": The document sets out the prerequisites.
- Section 3 "Bluetooth LE Host Stack Initialization and APIs": This section describes the Bluetooth Low Energy Host Stack initialization. It also presents the APIs categorized according to the layer and by application role.
- Section 4 "Generic Access Profile (GAP) Layer": The Generic Access Profile (GAP) layer is divided into two sections according to the GAP role of the device: Central and Peripheral. The basic setup of two such devices is explained with code examples, such as how to prepare the devices for connections, how to connect them together, and pairing and bonding processes.
- Section 5 "Generic Attribute Profile (GATT) Layer": This section describes the Generic Attribute Profile (GATT) layer and introduces the APIs required for data transfer between the two connected devices. This section is divided into two subsections according to the GATT role of the device: Client and Server.
- Section 6 "GATT database application interface": The document further describes the usage of the GATT database APIs that allow the application to manipulate data stored in the GATT Server database.
- Section 7 "Creating GATT database": This section describes a user-friendly method to build a GATT database statically. The method involves the use of a predefined set of macros that the application can include to build the database at application compile time.
- Section 8 "Creating a Custom Profile": This section contains instructions on how to build a custom profile.
- Section 9 "Application Structure": The section describes the structure of the typical application.
- Section 10 "Low-Power Management": This section describes low-power management and how an application can use the low-power modes of the hardware and software.
- Section 11 "Over the Air Programming (OTAP)": This section describes the Over The Air Programming (OTAP) capabilities that the Host Stack offers via a dedicated Service/Profile. The section also describes how to use the OTAP capabilities in an application and also contains a detailed description of the SDK components involved in the OTAP process.
- Section 12 "Creating a Bluetooth LE application when the Host Stack runs on another processor": This section describes how to build a Bluetooth Low Energy application when the Host Stack is running on a separate processor.
- Section 13 "References": This section lists the documents that can be referred to for more information.
- Section 14 "Acronyms and abbreviations": This section lists the acronyms used in this document.

2 Prerequisites

The Bluetooth Low Energy Host Stack library contains several external references that the application must define to enable full functionality.

Attention: *Application developers must ensure to define these references to prevent linkage errors when trying to build the application binary.*

2.1 Task and event queues

The task queues are declared in the *ble_host_tasks.h* as follows:

```
/*! App to Host message queue for the Host Task */
    extern messaging_t   gApp2Host_TaskQueue;
/*! HCI to Host message queue for the Host Task */
    extern messaging_t   gHci2Host_TaskQueue;
/*! Event for the Host Task Queue */
    extern OSA_EVENT_HANDLE_DEFINE(gHost_TaskEvent);
```

See Section 3.1 "Initialization" for more details about the RTOS tasks required by the Bluetooth LE Host Stack.

2.2 GATT database

The application must define and populate the database according to its requirements and constraints either statically, at application compile time, or dynamically.

Regardless of how the application creates the GATT database, the following two external references from *gatt_database.h* must be defined:

```
/*! The number of attributes in the GATT Database. */
extern uint16_t gGattDbAttributeCount_c;
/*! Reference to the GATT database */
extern gattDbAttribute_t* gattDatabase;
```

The attribute template is defined as shown here:

```
typedef struct {
    uint16_t handle ;
/*!< Attribute handle - cannot be 0x0000; attribute handles need not be
 consecutive, but must be strictly increasing. */
    uint16_t permissions ;
/*!< Attribute permissions as defined by ATT. */
    uint32_t uuid ;
/*!< The UUID should be read according to the gattDbAttribute_t.uuidType member:
 for 2-byte and 4-byte UUIDs, this contains the value of the UUID; for 16-byte
 UUIDs, this is a pointer to the allocated 16-byte array containing the UUID. */
    uint8_t * pValue ;
/*!< Pointer to allocated value array. */
    uint16_t valueLength ;
/*!< Size of the value array. */
    uint16_t uuidType : 2;
/*!< Identifies the length of the UUID; the 2-bit values are interpreted
 according to the bleUuidType_t enumeration. */
    uint16_t maxVariableValueLength : 10;
/*!< Maximum length of the attribute value array; if this is set to 0, then the
 attribute's length (valueLength) is fixed and cannot be changed. */
```

```
} gattDbAttribute_t ;
```

2.3 Non-Volatile Memory (NVM) access

The Bluetooth LE Host Stack implements an internal module responsible for managing device information. This module relies on accessing a Non-Volatile Memory module for storing and loading bonded devices data.

The application developers determine the NVM access mechanism through the definition of three functions and one variable. The functions must first pre-process the information and then perform standard NVM operations (erase, write, read). The declarations are as follows:

```
 bleResult_t App_NvmErase
(
    uint8_t mEntryIdx
);
bleResult_t App_NvmRead
(
    uint8_t  mEntryIdx,
    void*    pBondHeader,
    void*    pBondDataDynamic,
    void*    pBondDataStatic,
    void*    pBondDataLegacy,
    void*    pBondDataDeviceInfo,
    void*    pBondDataDescriptor,
    uint8_t  mDescriptorIndex
);
bleResult_t App_NvmWrite
(
    uint8_t  mEntryIdx,
    void*    pBondHeader,
    void*    pBondDataDynamic,
    void*    pBondDataStatic,
    void*    pBondDataLegacy,
    void*    pBondDataDeviceInfo,
    void*    pBondDataDescriptor,
    uint8_t  mDescriptorIndex
);
```

The device information is divided into several components to ensure that even software wear leveling mechanisms can be used optimally. The components sizes are fixed (defined in *ble_constants.h*) and have the following meaning:

API pointer to bond component	Component size (*ble_constants.h*)	Description
pBondHeader: points to a bleBondIdentity HeaderBlob_t element	gBleBondIdentityHeaderSize_c	Bonding information which is sufficient to identify a bonded device.
pBondDataDynamic: points to a bleBond DataDynamicBlob_t element	gBleBondDataDynamicSize_c	Bonding information that might change frequently.
pBondDataStatic: points to a bleBond DataStaticBlob_t element	gBleBondDataStaticSize_c	Bonding information that is unlikely to change frequently.
pBondDataLegacy: points to a bleBond DataLegacyBlob_t element	gBleBondDataLegacySize_c	Stores legacy pairing and Connection Signature Resolving Key (CSRK) bond information.

API pointer to bond component	Component size (*ble_constants.h*)	Description
pBondDataDeviceInfo: points to a ble BondDataDeviceInfoBlob_t element	gBleBondDataDeviceInfoSize_c	Additional bonding information that can be accessed using the host stack API.
pBondDataDescriptor: points to a ble BondDataDescriptorBlob_t element	gBleBondDataDescriptorSize_c	Bonding information used to store one Client Characteristic Configuration Descriptor (CCCD).

The Bluetooth LE Host Stack handles the format of the bonding information. Therefore, application developers need not to take care of this aspect.

Each bonding data slot must contain one bonding header blob, one dynamic data blob, one static data blob, one data legacy blob, one device information blob, and an array of descriptor blobs equal to *gcGapMaximumSavedCccds_c.*

Note: *The application must define the gcGapMaximumSavedCccds_c. macro according to its requirement. The default value can be found in the* `ble_constants.h` *file.)*

A slot is uniquely identified by the *mEntryIdx* parameter.

A descriptor is uniquely identified by the pair *mEntryIdx - mDescriptorIndex*.

If one or more pointers passed as parameters are NULL, the read from or write to the corresponding blob of the bonding slot must be ignored. The erase function must clear the entire bonding data slot specified by the entry index.

Note:

When Advanced Secure Mode is chosen (`gAppSecureMode_d` is defined as `1` in `app_preinclude.h`), two additional application NVM functions are defined to handle local keys encrypted blob storage. Their declaration is:

```
bleResult_t App_NvmWriteLocalKeys
(
    uint8_t mEntryIdx,
    void* pLocalKey
)
bleResult_t App_NvmReadLocalKeys
(
    uint8_t mEntryIdx,
    void* pLocalKey
)
```

The functions write/read a structure of type `bleLocalKeysBlob_t` into/from NVM using a dedicated data set. The parameter `mEntryIdx` can be `0` (local IRK is handled) or `1` (local CSRK is handled).

The format of the local keys blob nor about the generation and storage of the local keys is automatically handled in **BLE Connection Manager** (*BleConnManager_GenericEvent*). Therefore the application developer need not manage this aspect.

The current implementation of the aforementioned functions uses either the framework NVM module or a RAM buffer. Additional details about the NVM configuration and functionality can be found in the *Connectivity Framework Reference Manual*. See Section 13 "References".

To enable the NVM mechanism, ensure the following points:

- `gAppUseNvm_d` (in *app_preinclude.h*) is set to `1` and
- `gUseNVMLink_d` is set to `1` in the linker options of the toolchain.

Note:

- *If `gAppUseNvm_d` is set to `0`, then all bonding data is stored in the RAM and is accessible until reset or power cycle.*
- *If `gAppUseNvm_d` is set to `1`, the default NVM module configurations are applied in the app_preinclude.h file.*

3 Bluetooth LE Host Stack Initialization and APIs

3.1 Initialization

The application developer is required to configure the Host Task as part of the Host Stack requirement. The task is the context for running all the Host layers (GAP, GATT, ATT, L2CAP, SM, GATTDB)

The prototype of the task function is located in the *ble_host_tasks.h* file:

```
void Host_TaskHandler(void * args);
```

It should be called with *NULL* as an argument in the task code from the application.

Application developers are required to define task events and queues as explained in RTOS Task Queues and Events.

If the Controller software runs on the same chip as the Host, the Controller task always has a higher priority than the Host task. The priority value of the Host Task can be configured through the *gHost_TaskPriority_c define* (by default set in *ble_host_task_config.h*). Note that changing this value can have a significant impact on the Bluetooth Low Energy stack.

3.2 Main function to initialize the Bluetooth LE Host Stack

Figure 1 provides an overview of Bluetooth Low Energy Host Stack. When using the existing application common files, the startup task uses *BluetoothLEHost_AppInit()*, which is defined in *app_conn.h*. The function initializes all components related to the Bluetooth Low Energy application. It has the following prototype:

```
void BluetoothLEHost_AppInit(void);
```

The *BluetoothLEHost_AppInit()* function must be implemented by each application. It should register its generic event callback using *BluetoothLEHost_SetGenericCallback()* and initialize the Bluetooth LE Host Stack layer by calling *BluetoothLEHost_Init()*. The prototype for the *BluetoothLEHost_Init()* function is found in *app_conn.h* and is implemented in *app_conn.c*.

```
void BluetoothLEHost_Init
(
    appBluetoothLEInitCompleteCallback_t pCallback
);
```

BluetoothLEHost_Init takes as parameter a function to be called at the end of Bluetooth LE Host Stack initialization. In this callback, the application can register its callbacks with the Host layer, allocate timers, start services, and perform similar tasks. The callback should have a prototype as follows:

```
static void BluetoothLEHost_Initialized(void);
```

BluetoothLEHost_Init() is responsible for initializing the Host.

Initialize the Bluetooth LE Host Stack after platform setup is complete and all RTOS tasks have been started. The function that should be called for this purpose is located in the *ble_general.h* file and has the following prototype:

```
bleResult_t Ble_HostInitialize
(
    gapGenericCallback_t            genericCallback,
```

```
    hciHostToControllerInterface_t  hostToControllerInterface
);
```

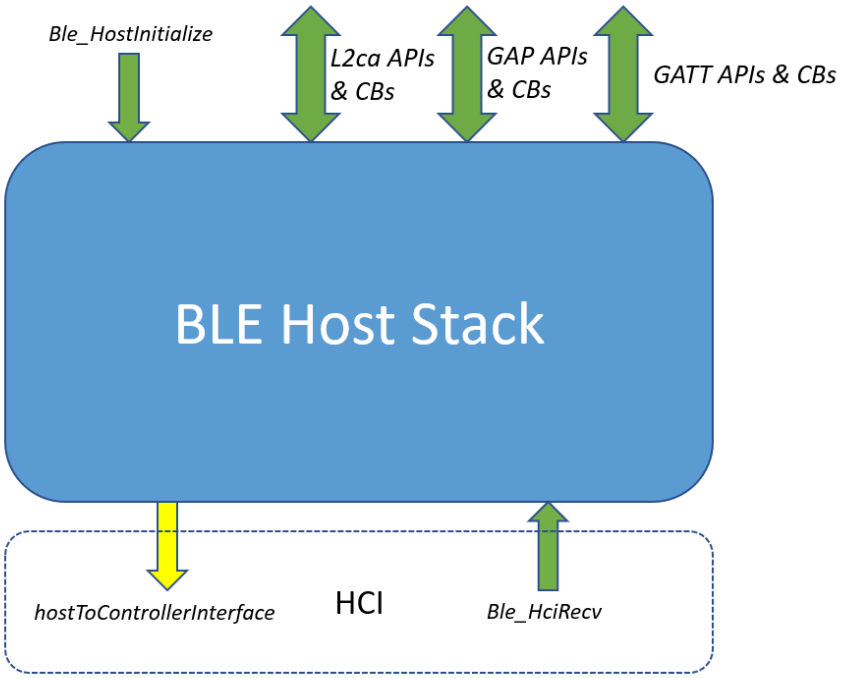

Figure 1. Bluetooth Low Energy Host Stack overview

3.3 HCI entry and exit points

The HCI entry point of the Host Stack is the second function located in the *ble_general.h* file:

```
void Ble_HciRecv
(
    hciPacketType_t packetType,
    void* pHciPacket,
    uint16_t packetSize
);
```

This is the function that the application must call to insert an HCI message into the Host.

An equivalent exists, to be used in ISR context:

```
bleResult_t Ble_HciRecvFromIsr
(
    hciPacketType_t     packetType,
    void*               pHciPacket,
    uint16_t            packetSize
);
```

Therefore, the *Ble_HciRecv* function and the *hostToControllerInterface* parameter of the *Ble_HostInitialize* function represent the two points that need to be connected to the LE Controller (see Figure 1), either directly (if the Controller software runs on the same chip as the Host) or through a physical interface (for example, UART).

3.4 Bluetooth LE Host Stack libraries and API availability

All the APIs referenced in this document are available in the Central and Peripheral libraries. The support for Bluetooth 5.3 optional features such as Advertising, Advertising Extensions, GATT Caching, and EATT are provided in separate host libraries. They are distributed in a similar process as the legacy ones using GAP/ GATT role support, which is described as follows.

For example, below are listed the full-featured libraries with complete support for both Central and Peripheral APIs, at GAP level.

- *lib_ble_OPT_host_cm33_iar.a* (for IAR projects)
- *lib_ble_OPT_host_cm33_gcc.a* (for MCUX projects)

These libraries include optional features implemented by the Bluetooth LE Host. For applications that need to use only the mandatory 5.3 Bluetooth LE and below features, the *lib_ble_host_cm33_iar.a* or *lib_ble_host_cm33_gcc.a* libraries can be used instead.

However, some applications may be targeted to memory-constrained devices and do not need the full support. In the interest of reducing code size and RAM utilization, optimized libraries are provided:

- *lib_ble_host_peripheral_cm33_iar.a/ lib_ble_host_peripheral_cm33_gcc.a* and
- *lib_ble_OPT_host_peripheral_cm33_iar.a / lib_ble_OPT_host_peripheral_cm33_gcc.a*.
  - Support only APIs for the GAP Peripheral and GAP Broadcaster roles
  - Support only APIs for the GATT Server role
- *lib_ble_host_central_cm33_iar.a* and *lib_ble_OPT_host_central_cm33_iar.a*
- *lib_ble_host_central_cm33_gcc.a* and *lib_ble_OPT_host_central_cm33_gcc.a*
  - Support only APIs for the GAP Central and GAP Observer roles
  - Support only APIs for the GATT Client role

If one attempts to use an API that is not supported (for instance, calling *Gap_Connect* with the *lib_ble_host_peripheral_cm33_iar.a* and *lib_ble_host_peripheral_cm33_gcc.a*), then the API returns the *gBleFeatureNotSupported_c* error code.

Similarly, if the API for OPT is used with a host library that does not have support for optional features, then *gBleFeatureNotSupported_c* is returned. For instance, calling *Gap_SetExtAdvertising* parameters with the *lib_ble_host_peripheral_cm33_iar.a* and *lib_ble_host_peripheral_cm33_gcc.a*) returns the exit code *gBleFeatureNotSupported_c*.

Note: *See the Bluetooth Low Energy Host Stack API Reference Manual for explicit information regarding API support. Each function documentation contains this information in the Remarks section.*

3.5 Synchronous and asynchronous functions

The vast majority of the GAP and GATT APIs are executed **asynchronously**. Calling these functions generates a message and places it in the Host Task message queue.

Therefore, the actual result of these APIs is signaled in **events** triggered by specific callbacks installed by the application. See the *Bluetooth Low Energy Host Stack API Reference Manual* for specific information about the events that are triggered by each API.

However, there are a few APIs which are executed immediately (**synchronously**). This is explicitly mentioned in the *Bluetooth Low Energy Host Stack API Reference Manual* in the *Remarks* section of each function documentation.

If nothing is mentioned, then the API is asynchronous.

3.6 Radio TX Power level

The controller interface includes APIs that can be used to set the Radio TX Power to a different level than the default one.

The power level can be set differently for advertising and connection channels by calling the function *Controller_SetTxPowerLevelDbm()* with the channel parameter set to *gAdvTxChannel_c* or *gConnTxChannel_c*.

4 Generic Access Profile (GAP) Layer

The GAP layer manages connections, security, and bonded devices.

The GAP layer APIs are built on top of the Host-Controller Interface (HCI), the Security Manager Protocol (SMP), and the Device database.

GAP defines four possible roles that a Bluetooth Low Energy device may have in a Bluetooth Low Energy system:

- **Central**
  - Scans for advertisers (Peripherals and Broadcasters)
  - Initiates connection to Peripherals; Central at Link Layer (LL) level
  - Usually acts as a GATT Client, but can also contain a GATT Database itself
- **Peripheral**
  - Advertises and accepts connection requests from Central devices; LL Peripheral
  - Usually contains a GATT Database and acts as a GATT Server, but may also be a Client
- **Observer**
  - Scans for advertisers, but does not initiate connections; Transmit is optional
- **Broadcaster**
  - Advertises, but does not accept connection requests from Central devices; Receive is optional

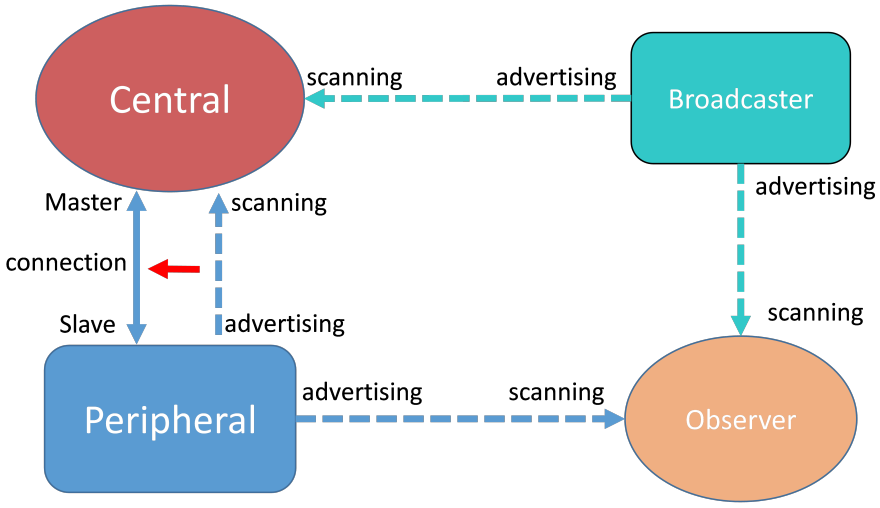

Figure 2. GAP topology

Figure 2 illustrates the generic GAP topology.

4.1 Peripheral setup

The Peripheral starts advertising and waits for scan and connection requests from other Central devices.

4.1.1 Advertising

Before starting advertising, the advertising parameters should be configured. Otherwise, the following defaults are used.

```
#define gGapDefaultAdvertisingParameters_d \
{ \
   /* minInterval */     gGapAdvertisingIntervalDefault_c, \
   /* maxInterval */     gGapAdvertisingIntervalDefault_c, \
   /* advertisingType */ gConnectableUndirectedAdv_c, \
   /* addressType */     gBleAddrTypePublic_c, \
   /* peerAddressType */  gBleAddrTypePublic_c, \
   /* peerAddress */     {0U, 0U, 0U, 0U, 0U, 0U}, \
   /* channelMap */

 (gapAdvertisingChannelMapFlags_t)gGapAdvertisingChannelMapDefault_c,  \
   /* filterPolicy */    gProcessAll_c \
}
```

To set different advertising parameters, a *gapAdvertisingParameters_t* structure should be allocated and initialized with defaults. Then, the necessary fields may be modified.

After that, the following function should be called:

```
bleResult_t Gap_SetAdvertisingParameters
(
    const gapAdvertisingParameters_t *     pAdvertisingParameters
);
```

The application should listen to the *gAdvertisingParametersSetupComplete_c* generic event.

Next, the advertising data should be configured and, if the advertising type supports active scanning, the scan response data should also be configured. If either of these is not configured, they are defaulted to empty data.

The function used to configure the advertising and/or scan response data is shown here:

```
bleResult_t Gap_SetAdvertisingData
(
  const gapAdvertisingData_t *     pAdvertisingData,
  const gapScanResponseData_t *    pScanResponseData
);
```

Either of the two pointers may be *NULL*, in which case they are ignored (the corresponding data is left as it was previously configured, or empty if it has never been set), but not both at the same time.

The application should listen to the *gAdvertisingDataSetupComplete_c* generic event.

After all the necessary setup is done, advertising may be started with this function:

```
bleResult_t Gap_StartAdvertising
(
    gapAdvertisingCallback_t advertisingCallback,
    gapConnectionCallback_t connectionCallback
);
```

The advertising callback is used to receive advertising events (advertising state changed or advertising command failed), while the connection callback is only used if a connection is established during advertising.

The connection callback is the same as the callback used by the Central when calling the *Gap_Connect* function.

When using the common application structure, the application can use the following API defined in *app_conn.h*:

```
bleResult_t BluetoothLEHost_StartAdvertising
(
    appAdvertisingParams_t   *pAdvParams,
    gapAdvertisingCallback_t pfAdvertisingCallback,
    gapConnectionCallback_t  pfConnectionCallback
);
```

The API goes through the steps of setting the advertising data and parameters. Events from the Host task are treated in the *App_AdvertiserHandler()* function, implemented in *app_advertiser.c*. To set the advertising parameters and data *BluetoothLEHost_StartAdvertising* requires a parameter of the following type:

```
typedef struct appAdvertisingParams_tag { gapAdvertisingParameters_t
 *pGapAdvParams; /*!< Pointer to the GAP advertising parameters */ const
 gapAdvertisingData_t *pGapAdvData; /*!< Pointer to the GAP advertising data
 */ const gapScanResponseData_t *pScanResponseData; /*!< Pointer to the scan
 response data */ } appAdvertisingParams_t;
```

If a Central initiates a connection to this Peripheral, the *gConnEvtConnected_c* connection event is triggered.

To stop advertising while the Peripheral has not yet received any connection requests, use this function:

```
bleResult_t Gap_StopAdvertising (void);
```

This function should not be called after the Peripheral enters a connection, as the advertising automatically stops in this case.

4.1.2 Pairing and bonding (peripheral)

After a connection has been established to a Central, the Peripheral's role regarding security is a passive one. It is the responsibility of the Central device to start the pairing process. In case, the devices have already bonded in the past, the Central encrypts the link using the shared LTK.

The Peripheral sends error responses (at ATT level) with proper error code if the Central attempts to access sensitive data without authenticating. Examples of such error responses are: Insufficient Authentication, Insufficient Encryption, Insufficient Authorization, and so on. Therefore, it indicates to the Central that it needs to perform security procedures.

All security checks are performed internally by the GAP module and the security error responses are sent automatically. All the application developer needs to do is register the security requirements.

First, when building the GATT Database (see Section 7 "Creating GATT database"), the sensitive attributes should have the security built into their access permissions (for example, read-only / read with authentication / write with authentication / write with authorization, and so on.).

Second, if the GATT Database requires additional security besides that already specified in attribute permissions (for example, certain services require higher security in certain situations), the following function must be called:

```
bleResult_t Gap_RegisterDeviceSecurityRequirements
(
```

```
    const gapDeviceSecurityRequirements_t * pSecurity
);
```

The parameter is a pointer to a structure which contains a "device security setting" and service-specific security settings. All these security requirements are pointers to `gapSecurityRequirements_t` structures. The pointers that are to be ignored should be set to `NULL`.

Although the Peripheral does not initiate any kind of security procedure, it can inform the Central about its security requirements. This is usually done immediately after the connection to avoid exchanging useless packets for requests that might be denied because of insufficient security.

The informing is performed through the Peripheral Security Request packet at SMP level. To use it, the following GAP API is provided:

```
bleResult_t Gap_SendPeripheralSecurityRequest
(
    deviceId_t                    deviceId,
    const gapPairingParameters_t* pPairingParameters
);
```

The `gapPairingParameters_t` structure includes two important fields. The withBonding field indicates to the Central whether this Peripheral can bond and the securityModeAndLevel field informs about the required security mode and level that the Central should pair for. See Section 4.2.3 "Pairing and bonding (Central)" for an explanation about security modes and levels, as defined by the GAP module.

This request expects no reply, nor any immediate action from the Central. The Central may easily choose to ignore the Peripheral Security Request.

If the two devices have bonded in the past, the Central proceeds directly to encrypting the link.
If the bond was not made using LE Secure Connections, the Peripheral expects to receive a `gConnEvtLongTermKeyRequest_c` connection event. If the bond was made using LE Secure Connections, the Host provides the LTK automatically to the LE Controller.

When the devices have been previously pairing without using LE Secure Connections, along with the Peripheral's LTK, the EDIV (2 bytes) and RAND (8 bytes) values were also sent (their meaning is defined by the SMP). Therefore, before providing the key to the Controller, the application should check that the two values match with those received in the `gConnEvtLongTermKeyRequest_c` event. If they do, the application should reply with:

```
bleResult_t Gap_ProvideLongTermKey
(
    deviceId_t          deviceId,
    const uint8_t       aLtk,
    uint8_t             ltkSize
);
```

The LTK size cannot exceed the maximum value of 16.

If the EDIV and RAND values do not match, or if the Peripheral does not recognize the bond, it can reject the encryption request with:

```
bleResult_t Gap_DenyLongTermKey
(
    deviceId_t deviceId
);
```

If LE SC Pairing was used then the LTK is generated internally by the Bluetooth LE Host Stack and it is not requested from the application during post-bonding link encryption. In this scenario, the application is only notified of the link encryption through the `gConnEvtEncryptionChanged_c` connection event.

If the devices are not bonded, the Peripheral should expect to receive the `gConnEvtPairingRequest_c`, indicating that the Central has initiated pairing.

If the application agrees with the pairing parameters (see Section 4.2.3 "Pairing and bonding (Central)" for detailed explanations), it can reply with:

```
bleResult_t Gap_AcceptPairingRequest
(
    deviceId_t                          deviceId,
    const gapPairingParameters_t *      pPairingParameters
);
```

This time, the Peripheral sends its own pairing parameters, as defined by the SMP.

After sending this response, the application should expect to receive the same pairing events as the Central (see Section 4.2.3 "Pairing and bonding (Central)"), with one exception: the `gConnEvtPasskeyRequest_c` event is not called if the application sets the Passkey (PIN) for pairing before the connection by calling the API:

```
bleResult_t Gap_SetLocalPasskey
(
    uint32_t passkey
);
```

This is done because, usually, the Peripheral has a static secret PIN that it distributes only to trusted devices. If, for any reason, the Peripheral must dynamically change the PIN, it can call the aforementioned function every time it wants to, before the pairing starts (for example, right before sending the pairing response with `Gap_AcceptPairingRequest`).

If the Peripheral application never calls `Gap_SetLocalPasskey`, then the `gConnEvtPasskeyRequest_c` event is sent to the application as usual.

The Peripheral can use the following API to reject the pairing process:

```
bleResult_t Gap_RejectPairing
(
deviceId_t                          deviceId,
gapAuthenticationRejectReason_t     reason
);
```

The `reason` should indicate why the application rejects the pairing. The value `gLinkEncryptionFailed_c` is reserved for the `gConnEvtAuthenticationRejected_c` connection event to indicate the link encryption failure rather than pairing failures. Therefore, it is not meant as a pairing reject reason.

The `Gap_RejectPairing` function may be called not only after the Pairing Request was received, but also during the pairing process. For example, when handling pairing events or asynchronously, if for any reason the Peripheral decides to abort the pairing, this function can be called. This also holds true for the Central. Figure 3 illustrates the Peripheral pairing flow and lists the main APIs and events. `Gap_RejectPairing` can be called on any pairing event.

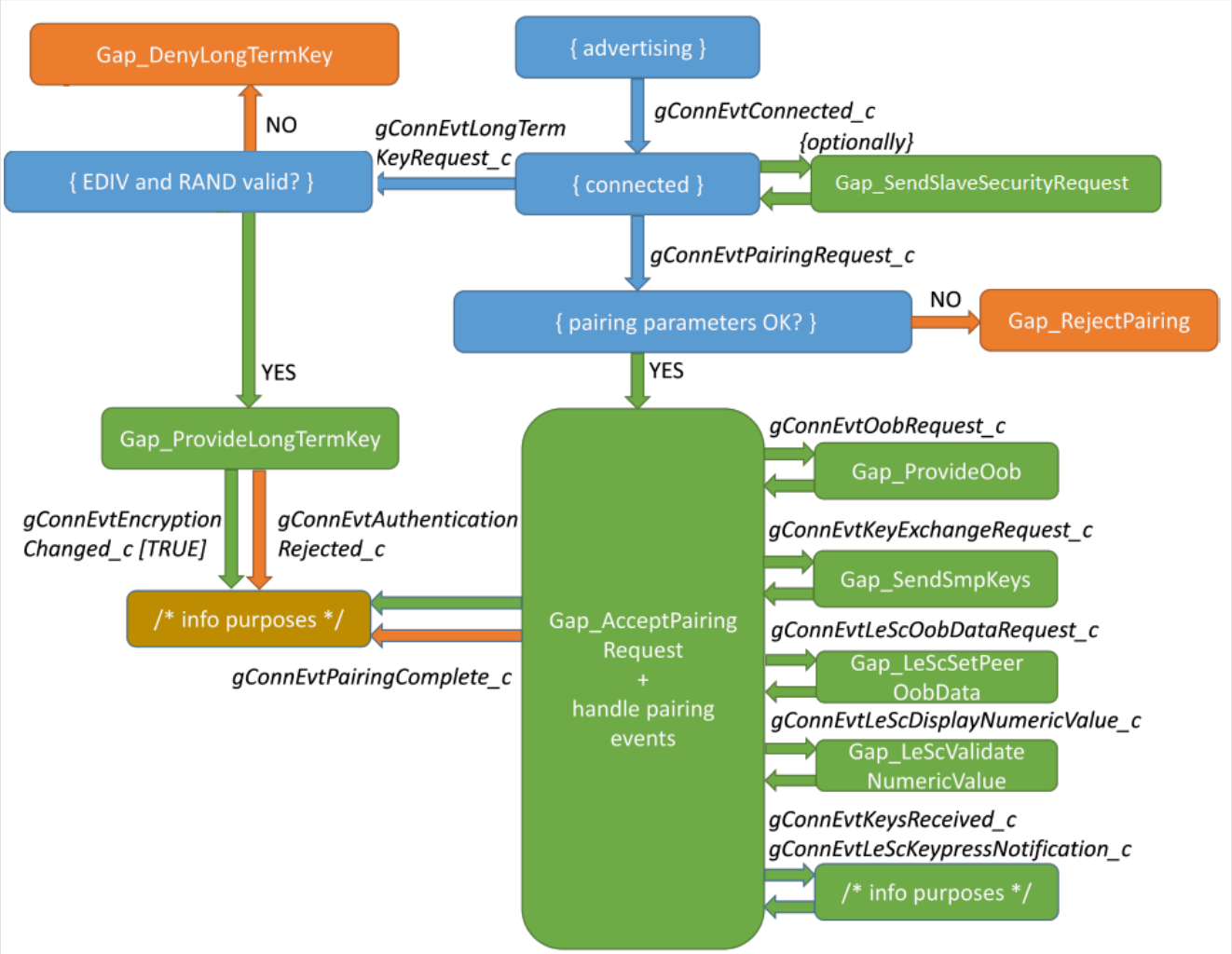

Figure 3. Peripheral pairing flow – APIs and events

For both the Central and the Peripheral, bonding is performed internally and is not the application's concern. The `gConnEvtPairingComplete_c` event parameters inform the application if bonding has occurred.

4.2 Central setup

Usually, a Central must start scanning to find Peripherals. When the Central has scanned a Peripheral it wants to connect to, it stops scanning and initiates a connection to that Peripheral. After the connection has been established, it may start pairing, if the Peripheral requires it, or directly encrypt the link, if the two devices have already bonded in the past.

4.2.1 Scanning

The most basic setup for a Central device begins with scanning, which is performed by the following function from *gap_interface.h*:

```
bleResult_t Gap_StartScanning
(
  const gapScanningParameters_t* pScanningParameters,
  gapScanningCallback_t scanningCallback,
  gapFilterDuplicates_t enableFilterDuplicates,
  uint16_t duration,
  uint16_t period
);
```

If the *pScanningParameters* pointer is NULL, the currently set parameters are used. If no parameters have been set after a device power-up, the standard default values are used:

```
#define gGapDefaultScanningParameters_d \
{ \
    /* type */              gGapScanTypePassive_c, \
    /* interval */          gGapScanIntervalDefault_d, \
    /* window */            gGapScanWindowDefault_d, \
    /* ownAddressType */    gBleAddrTypePublic_c, \
    /* filterPolicy */      gScanAll_c \
    /* scanning PHY */      gLePhylMFlag_c\
}
```

The easiest way to define non-default scanning parameters is to initialize a *gapScanningParameters_t* structure with the above default and change only the required fields.

For example, to perform active scanning and only scan for devices in the Filter Accept List, the following code can be used:

```
gapScanningParameters_t scanningParameters = gGapDefaultScanningParameters_d;
scanningParameters.type = gGapScanTypeActive_c;
scanningParameters.filterPolicy = gScanWithFilterAcceptList_c;
Gap_StartScanning(&scanningParamters, scanningCallback, enableFilterDuplicates,
 duration, period);
```

When using the common application structure, the application can use the following API defined in *app_conn.h*:

```
bleResult_t BluetoothLEHost_StartScanning
(
    appScanningParams_t   *pAppScanParams,
    gapScanningCallback_t pfCallback
);
```

The API uses the *appScanningParams_t* structures, which is defined as follows:

```
typedef struct appScanningParams_tag
```

```
{
    gapScanningParameters_t *pHostScanParams;         /*!< Pointer to host scan
 structure */
    gapFilterDuplicates_t enableDuplicateFiltering;   /*!< Duplicate filtering
 mode */
    uint16_t duration;                                /*!< scan duration  */
    uint16_t period;                                  /*!< scan period  */
} appScanningParams_t;
```

The *scanningCallback* is triggered by the GAP layer to signal events related to scanning.

The most important event is the *gDeviceScanned_c* event, which is triggered each time an advertising device is scanned. This event data contains information about the advertiser:

```
typedef struct
{
    bleAddressType_t                 addressType ;
    bleDeviceAddress_t               aAddress ;
    int8_t                           rssi ;
    uint8_t                          dataLength ;
    uint8_t*                         data ;
    bleAdvertisingReportEventType_t  advEventType ;
    bool_t                           directRpaUsed;
    bleDeviceAddress_t               directRpa;
    bool_t                           advertisingAddressResolved;
} gapScannedDevice_t;
```

If this information signals a known Peripheral that the Central wants to connect to, the latter must stop scanning and connect to the Peripheral.

To stop scanning, call this function:

```
bleResult_t Gap_StopScanning (void);
```

By default, the GAP layer is configured to report all scanned devices to the application using the *gDeviceScanned_c* event type. However, some use cases might require to perform specific GAP Discovery Procedures. In such use cases the advertising reports might require the filtering of Flags AD value from the advertising data. Other use cases require the Bluetooth LE Host Stack to automatically initiate a connection when a specific device has been scanned.

To enable filtering based on the Flags AD value or to set device addresses for automatic connections, the following function must be called before the scanning is started:

```
bleResult_t Gap_SetScanMode
(
    gapScanMode_t           scanMode,
    gapAutoConnectParams_t* pAutoConnectParams,
    gapConnectionCallback_t connCallback
);
```

The default value for the scan mode is *gDefaultScan_c,* which reports all packets regardless of their content and does not perform any automatic connection.

To enable Limited Discovery, the *gLimitedDiscovery_c* value must be used, while the *gGeneralDiscovery_c* value activates General Discovery.

To enable automatic connection when specific devices are scanned, the *gAutoConnect_c* value must be set, in which case the *pAutoConnectParams* parameter must point to the structure that holds the target device addresses and the connection parameters to be used by the Host for these devices.

If `scanMode` is set to *gAutoConnect_c*, connCallback must be set and is triggered by GAP to send the events related to the connection.

4.2.2 Initiating and closing a connection

To connect to a scanned Peripheral, extract its address and address type from the *gDeviceScanned_c* event data, stop scanning, and call the following function:

```
bleResult_t Gap_Connect
(
const gapConnectionRequestParameters_t * pParameters,
gapConnectionCallback_t connCallback
);
```

When using the common application structure, the application can also use the following API defined in *app_conn.h*:

```
bleResult_t BluetoothLEHost_Connect
(
    gapConnectionRequestParameters_t*   pParameters,
    gapConnectionCallback_t             connCallback
);
```

An easy way to create the connection parameter structure is to initialize it with the defaults, then change only the necessary fields. The default structure is defined as shown here:

```
#define gGapDefaultConnectionRequestParameters_d \
{ \
    /* scanInterval */       gGapScanIntervalDefault_d, \
    /* scanWindow */         gGapScanWindowDefault_d, \
    /* filterPolicy */       gUseDeviceAddress_c, \
    /* ownAddressType */     gBleAddrTypePublic_c, \
    /* peerAddressType */    gBleAddrTypePublic_c, \
    /* peerAddress */        { 0, 0, 0, 0, 0, 0 }, \
    /* connIntervalMin */    gGapDefaultMinConnectionInterval_d, \
    /* connIntervalMax */    gGapDefaultMaxConnectionInterval_d, \
    /* connLatency */        gGapDefaultConnectionLatency_d, \
    /* supervisionTimeout */ gGapDefaultSupervisionTimeout_d, \
    /* connEventLengthMin */ gGapConnEventLengthMin_d, \
    /* connEventLengthMax */ gGapConnEventLengthMax_d \
    /* initiatingPHYs     */ gLePhylMFlag_c \
}
```

In the following example, Central scans for a specific Heart Rate Sensor with a known address. When it finds it, it immediately connects to it.

```
static void BleApp_ScanningCallback
(
    gapScanningEvent_t *pScanningEvent
)
{
    switch (pScanningEvent->eventType)
    {
        case gDeviceScanned_c:
        {
            if (BleApp_CheckScanEvent(&pScanningEvent->eventData.scannedDevice))
            {
```

```
                gConnReqParams.peerAddressType = pScanningEvent-
>eventData.scannedDevice.addressType;
                FLib_MemCpy(gConnReqParams.peerAddress,
                        pScanningEvent->eventData.scannedDevice.aAddress,
                        sizeof(bleDeviceAddress_t));
                (void)Gap_StopScanning();
#if gAppUsePrivacy_d
                gConnReqParams.usePeerIdentityAddress =  pScanningEvent-
>eventData.scannedDevice.advertisingAddressResolved;
#endif
                (void)BluetoothLEHost_Connect(&gConnReqParams,
 BleApp_ConnectionCallback);
            }
        }
        break;
}
```

The *connCallback* is triggered by GAP to send all events related to the active connection. It has the following prototype:

```
typedef void (* gapConnectionCallback_t )
(
    deviceId_t              deviceId,
    gapConnectionEvent_t * pConnectionEvent
);
```

The very first event that should be listened inside this callback is the *gConnEvtConnected_c* event. If the application decides to drop the connection establishment before this event is generated, it should call the following macro:

```
#define Gap_CancelInitiatingConnection()\
    Gap_Disconnect(gCancelOngoingInitiatingConnection_d)
```

This is useful, for instance, when the application chooses to use an expiration timer for the connection request.

Upon receiving the *gConnEvtConnected_c* event, the application may proceed to extract the necessary parameters from the event data (*pConnectionEvent->event.connectedEvent*). The most important parameter to be saved is the *deviceId*.

The *deviceId* is a unique 8-bit, unsigned integer, used to identify an active connection for subsequent GAP and GATT API calls. All functions related to a certain connection require a *deviceId* parameter. For example, to disconnect, call this function:

```
bleResult_t Gap_Disconnect
(
    deviceId_t deviceId
);
```

4.2.3 Pairing and bonding (Central)

After the user has connected to a Peripheral, use the following function to check whether this device has bonded in the past:

```
bleResult_t Gap_CheckIfBonded
(
    deviceId_t   deviceId,
    bool_t *     pOutIsBonded
```

```
    uint8_t*    pOutNvmIndex
);
```

If it has, link encryption can be requested with:

```
bleResult_t Gap_EncryptLink
(
    deviceId_t     deviceId
);
```

If the link encryption is successful, the *gConnEvtEncryptionChanged_c* connection event is triggered. Otherwise, a *gConnEvtAuthenticationRejected_c* event is received with the *rejectReason* event data parameter set to *gLinkEncryptionFailed_c*.

On the other hand, if this is a new device (not bonded), pairing may be started as shown here:

```
bleResult_t Gap_Pair
(
    deviceId_t                          deviceId,
    const gapPairingParameters_t *      pPairingParameters
);
```

The pairing parameters are shown here:

```
typedef struct gapPairingParameters_tag {
    bool_t                          withBonding ;
    gapSecurityModeAndLevel_t       securityModeAndLevel ;
    uint8_t                         maxEncryptionKeySize ;
    gapIoCapabilities_t             localIoCapabilities ;
    bool_t                          oobAvailable ;
    gapSmpKeyFlags_t                centralKeys ;
    gapSmpKeyFlags_t                peripheralKeys ;
    bool_t                          leSecureConnectionSupported ;
    bool_t                          useKeypressNotifications ;
} gapPairingParameters_t;
```

The names of the parameters are self-explanatory. The *withBonding* flag should be set to *TRUE* if the Central must/wants to bond.

When Advanced Secure Mode is enabled, (*gAppSecureMode_d* id defined as *1* in *app_preinclude.h*), the security mode and level for pairing is automatically enforced as Mode 1 Level 4, and LE Secure Connection Supported is automatically enforced TRUE. Legacy pairing is not supported in this mode.

For the Security Mode and Level, the GAP layer defines them as follows:

- Security Mode 1 Level 1 stands for no security requirements.
- Except for Level 1 (which is only used with Mode 1), Security Mode 1 requires encryption, while Security Mode 2 requires data signing.
- Mode 1 Level 2 and Mode 2 Level 1 do not require authentication (in other words, they allow Just Works pairing, which has no MITM protection). Mode 1 Level 3 and Mode 2 Level 2 require authentication (must pair with PIN or OOB data, which provide MITM protection).
- Starting with Bluetooth specification 4.2, OOB pairing offers MITM protection only in certain conditions. The application must inform the stack if the OOB data exchange capabilities offer MITM protection via a dedicated API.
- Security Mode 1 Level 4 is reserved for authenticated pairing (with MITM protection) using a LE Secure Connections pairing method.

- If a pairing method is used but it does not offer MITM protection, then the pairing parameters must use Security Mode 1 level 2. If the requested pairing parameters are incompatible (for example, Security Mode 1 Level 4 without LE Secure Connections enabled), a `gBleInvalidParameter_c` status is returned by the security API functions: `Gap_SetDefaultPairingParameters`, `Gap_SendPeripheralSecurityRequest`, `Gap_Pair` and `Gap_AcceptPairingRequest`.

Table 1. GAP Security Modes and Levels

—	No security	No MITM protection	Legacy MITM protection	LE secure connections with MITM protection
Mode 1 (encryption) distributed LTK (EDIV +RAND) or generated LTK	**Level 1** no security	**Level 2** unauthenticated encryption	**Level 3** authenticated encryption	**Level 4** LE SC authenticated encryption
Mode 2 (data signing) distributed CSRK	—	**Level 1** unauthenticated data signing	**Level 2** authenticated data signing	—

The *centralKeys* should have the flags set for all the keys that are available in the application. The IRK is mandatory if the Central is using a Private Resolvable Address, while the CSRK is necessary if the Central wants to use data signing. The LTK is provided by the Peripheral and should only be included if the Central intends on becoming a Peripheral in future reconnections (GAP role change).

The *peripheralKeys* should follow the same guidelines. The LTK is mandatory if encryption is to be performed, while the peer's IRK should be requested if the Peripheral is using Private Resolvable Addresses.

See Table 2 for detailed guidelines regarding key distribution.

The first three rows are both guidelines for Pairing Parameters (*centralKeys* and *peripheralKeys*) and for distribution of keys with *Gap_SendSmpKeys*.

If LE Secure Connections Pairing is performed (Bluetooth Low Energy 4.2 and above), then the LTK is generated internally, so the corresponding bits in the key distribution fields from the pairing parameters are ignored by the devices.

The Identity Address is distributed if the IRK is also distributed (its flag has been set in the Pairing Parameters). Therefore, it can be "asked" only by asking for IRK (it does not have a separate flag in a *gapSmpKeyFlags_t* structure). Therefore, it is N/A.

The negotiation of the distributed keys is as follows:

- In the SMP Pairing Request (started by *Gap_Pair*), the Central sets the flags for the keys it wants to distribute (*centralKeys*) and receive (*peripheralKeys*).

Table 2. Key Distribution guidelines

	CENTRAL		PERIPHERAL	
	Central keys	**Peripheral keys**	**Peripheral keys**	**Central keys**
Long Term Key (LTK) +EDIV +RAND	If it wants to be a peripheral in a future reconnection	If it wants encryption	If it wants encryption	If it wants to become a central in a future reconnection
Identity Resolving Key (IRK)	If it uses or intends to use private resolvable addresses	If a peripheral is using a private resolvable address	If it uses or intends to use private resolvable addresses	If a central is using a private resolvable address
Connection Signature Resolving Key (CSRK)	If it wants to sign data as GATT Client	If it wants the peripheral to sign data as GATT Client	If it wants to sign data as GATT Client	If it wants the Central to sign data as GATT Client
Identity address	If it distributes the IRK	N/A	If it distributes the IRK	N/A

- The Peripheral examines the two distributions and must send an SMP Pairing Response (started by the *Gap_AcceptPairingRequest*) after performing any changes it deems necessary. The Peripheral is only allowed to set to 0 some flags that are set to 1 by the Central, but not the other way around. For example, it cannot request/distribute keys that were not offered/requested by the Central. If the Peripheral is adverse to the Central's distributions, it can reject the pairing by using the *Gap_RejectPairing* function.
- The Central examines the updated distributions from the Pairing Response. If it is adverse to the changes made by the Peripheral, it can reject the pairing (*Gap_RejectPairing*). Otherwise, the pairing continues and, during the key distribution phase (the *gConnEvtKeyExchangeRequest_c* event) only the final negotiated keys are included in the key structure sent with *Gap_SendSmpKeys*.
- For LE Secure Connections (both devices set the SC bit in the AuthReq field of the Pairing Request and Pairing Response packets), the LTK is not distributed. It is generated and the corresponding bit in the Initiator Key Distribution and Responder Key Distribution fields of the Pairing Response packet are set to 0.

If LE Secure Connections Pairing (Bluetooth LE 4.2 and above) is used, and OOB data needs to be exchanged, the application must obtain the local LE SC OOB Data from the Bluetooth LE Host Stack by calling the *Gap_LeScGetLocalOobData* function. The data is contained by the generic *gLeScLocalOobData_c* event.

The local LE SC OOB Data is refreshed in the following situations:

- The *Gap_LeScRegeneratePublicKey* function is called (the *gLeScPublicKeyRegenerated_c* generic event is also generated as a result of this API).
- The device is reset (which also causes the Public Key to be regenerated).

If the pairing continues, the following connection events may occur:

- **Request events**
  - *gConnEvtPasskeyRequest_c*: a PIN is required for pairing; the application must respond with the *Gap_EnterPasskey(deviceId, passkey).*
  - *gConnEvtOobRequest_c*: if the pairing started with the *oobAvailable* set to *TRUE* by both sides; the application must respond with the *Gap_ProvideOob(deviceId, oob).*
  - *gConnEvtKeyExchangeRequest_c*: the pairing has reached the key exchange phase; the application must respond with the *Gap_SendSmpKeys(deviceId, smpKeys).*
  - *gConnEvtLeScOobDataRequest_c*: the stack requests the LE SC OOB Data received from the peer (r, Cr and Addr); the application must respond with *Gap_LeScSetPeerOobData(deviceId, leScOobData).*
  - *gConnEvtLeScDisplayNumericValue_c*: the stack requests the display and confirmation of the LE SC Numeric Comparison Value; the application must respond with *Gap_LeScValidateNumericValue(deviceId, ncvValidated).*
- **Informational events**
  - *gConnEvtKeysReceived_c*: the key exchange phase is complete; keys are automatically saved in the internal device database and are also provided to the application for immediate inspection; application does not have to save the keys in NVM storage because this is done internally if *withBonding* was set to TRUE by both sides.
  - *gConnEvtAuthenticationRejected_c*: the peer device rejected the pairing; the *rejectReason* parameter of the event data indicates the reason that the Peripheral does not agree with the pairing parameters (it cannot be *gLinkEncryptionFailed_c* because that reason is reserved for the link encryption failure).
  - *gConnEvtPairingComplete_c*: the pairing process is complete, either successfully, or an error may have occurred during the SMP packet exchanges; note that this is different from the *gConnEvtKeyExchangeRequest_c* event; the latter signals that the pairing was rejected by the peer, while the former is used for failures due to the SMP packet exchanges.
  - *gConnEvtLeScKeypressNotification_c*: the stack informs the application that a remote SMP Keypress Notification has been received during Passkey Entry Pairing Method.

After the link encryption or pairing is completed successfully, the Central may immediately start exchanging data using the GATT APIs.

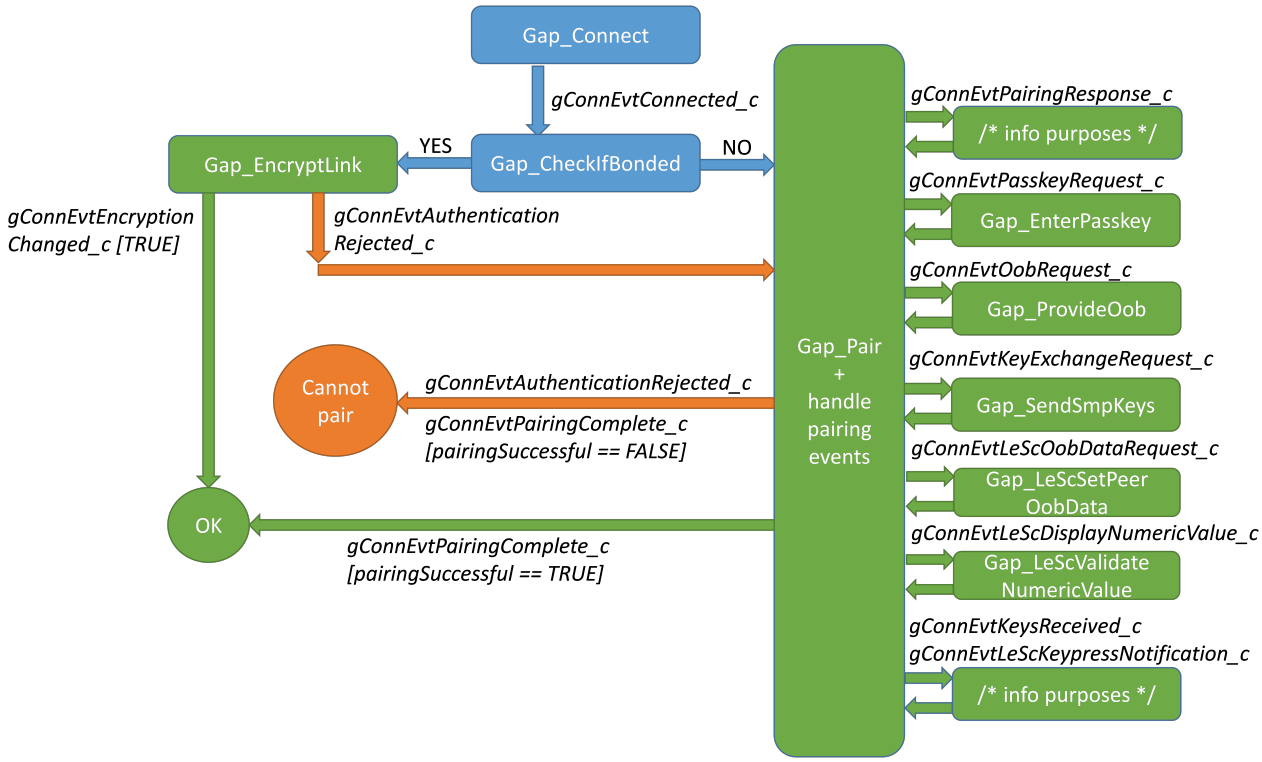

Figure 4. Central pairing flow – APIs and events. Gap_RejectPairing may be called on any pairing event

4.3 LE data packet length extension

This new feature extends the maximum data channel payload length from 27 to 251 octets.

The length management is done automatically by the link layer immediately after the connection is established. The stack passes the default values for maximum transmission number of payload octets and maximum packet transmission time that the application configures at compilation time in *ble_config.h*:

```
#ifndef gBleDefaultTxOctets_c
#define gBleDefaultTxOctets_c        0x00FB
#endif

#ifndef gBleDefaultTxTime_c
#define gBleDefaultTxTime_c          0x0848
#endif
```

The device can update the data length anytime, while in connection. The function that triggers this mechanism is the following:

```
bleResult_t Gap_UpdateLeDataLength
(
    deviceId_t         deviceId,
    uint16_t    txOctets,
    uint16_t    txTime
);
```

After the procedure executes, a *gConnEvtLeDataLengthChanged_c* connection event is triggered with the maximum values for number of payload octets and time to transmit and receive a link layer data channel PDU. The event is send event if the remote device initiates the procedure. This procedure is shown in Figure 5.

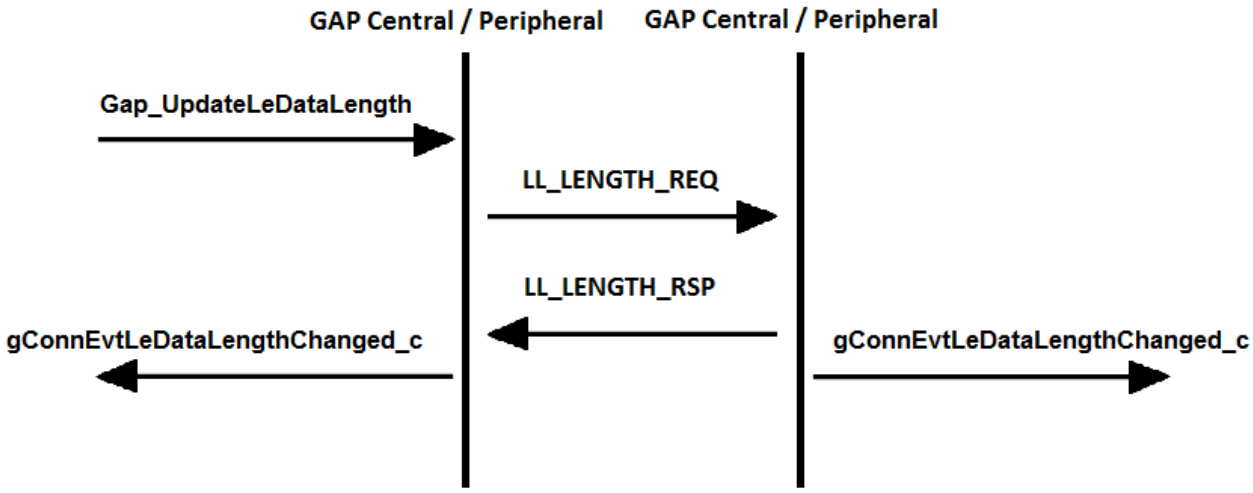

Figure 5. Data Length Update Procedure

4.4 Privacy feature

4.4.1 Introduction

Starting with Bluetooth 4.2, Privacy can be enabled either in the Host or in the Controller:

- **Host Privacy** consists of two use cases that are described in detail in the following sections. These are:
  - Random address generation - Periodically regenerating a random address (Resolvable or Non-Resolvable Private Address) inside the Host and then applying it into the Controller.
  - Random address resolution - Trying to resolve incoming RPAs using the IRKs stored in the bonded devices list. The address resolution is performed when a connection is established with a device or for the autoconnect scan. The advertising packets that have an RPA are not resolved automatically due to the high MCU processing that is required.
    The random address resolution is performed by default by the Host whenever the Controller is not able to resolve an RPA. The Host performs random address generation only when Host Privacy is requested to be enabled. During random address generation, the advertising and scan operations, if active, are stopped and restarted. If errors occur during this process and the scan or advertising cannot be started, the application is notified through the corresponding event (`gAdvertisingStateChanged_c`, `gExtAdvertisingStateChanged_c` or `gScanStateChanged_c`)
- **Controller Privacy**, introduced by Bluetooth 4.2, consists of writing the local IRK in the Controller, together with all known peer IRKs, and letting the Controller perform hardware, fully automatic RPA generation and resolution. The Controller uses a Resolving List to store these entries. The size of the list is platform dependent and determined by `gMaxResolvingListSize_c`. For RPA resolution, the entries that do not fit in this list are processed by the Host to be resolved using the IRKs from Bonded Devices list.

Either Host Privacy or Controller Privacy can be enabled at any time. Trying to enable one while the other is in progress generates a *gBleInvalidState_c* error. The same error is returned when trying to enable the same privacy type twice, or when trying to disable privacy when it is not enabled.

The recommended way of using Privacy is the Controller Privacy. However, enabling Controller Privacy requires at least a pair of local IRK and peer IRK, so this can only be enabled only after a pairing is performed with a peer and the IRKs are exchanged during the Key Distribution phase. When a device starts, if Privacy is required, the workflow is the following:

1. Enable Host Privacy using the local IRK.
2. Connect to a peer and perform pairing and bonding to exchange IRKs.
3. Disable Host Privacy.
4. Enable Controller Privacy using the local IRK and the peer IRK and peer identity address.

After enabling Host Privacy or Controller Privacy, the application must wait for the `gHostPrivacyStateChanged_c` or `gControllerPrivacyStateChanged_c` generic event and verify that privacy has been successfully enabled. Only then it is safe to proceed with setting advertising parameters (via the `Gap_SetAdvertisingParameters` or `Gap_SetExtAdvertisingParameters` APIs) or starting scanning (via the `Gap_StartScanning` API). Failure to do so could result in unwanted behavior, such as the device advertising or scanning with a public address.

4.4.1.1 Resolvable private addresses

A Resolvable Private Address (RPA) is a random address generated using an Identity Resolving Key (IRK). This address appears completely random to an outside observer, so a device may periodically regenerate its RPA to maintain privacy, as there is no correlation between any two different RPAs generated using the same IRK.

On the other hand, an IRK can also be used to resolve an RPA, in other words, to check if this RPA has been generated with this IRK. This process is called "resolving the identity of a device". Whoever has the IRK of a device can always try to resolve its identity against an RPA.

For example, assume device A frequently changes its RPA using IRKA. At some point, A bonds with B. A must give B a way to recognize it in a subsequent connection when it (A) has a different address. To achieve this purpose, A distributes the IRKA during the Key Distribution phase of the pairing process. B stores the IRKA it received from A.

Later, B connects to a device X that uses RPAX. This address appears completely random, but B can try to resolve RPAX using IRKA. If the resolving operation is successful, it means that IRKA was used to generate RPAX, and since IRKA belongs to device A, it means that X is A. So B was able to recognize the identity of device X, but nobody else can do that since they do not have IRKA.

4.4.1.2 Non-resolvable private addresses

A Non-Resolvable Private Address (NRPA) is a completely random address that has no generation pattern and therefore cannot be resolved by a peer.

A device that uses an NRPA that is changed frequently is impossible to track because each new address appears to belong to a new device.

4.4.1.3 Multiple identity resolving keys

If a device bonds with multiple peers, all of which are using RPAs, it needs to store the IRK of each in order to be able to recognize them later (see previous section).

This means that whenever the device connects to a peer that uses an unknown RPA, it needs to try and resolve the RPA with each of the stored IRKs. If the number of IRKs is large, then this introduces a lot of computation.

Performing all these resolving operations in the Host can be costly. It is much more efficient to take advantage of hardware acceleration and enable the Controller Privacy.

4.4.2 Host privacy

To enable or disable Host Privacy, the following API may be used:

```
bleResult_t Gap_EnableHostPrivacy
(
    bool_t              enable,
    const uint8_t *     aIrk
);
```

When *enable* is set to TRUE, the *aIrk* parameter defines which type of Private Address to generate. If *aIrk* is NULL, then a new NRPA is generated periodically and written into the Controller. Otherwise, an IRK is copied internally from the *aIrk* address and it is used to periodically generate a new RPA.

The lifetime of the Private Address (NRPA or RPA) is a number of seconds contained by the *gGapHostPrivacyTimeout* external constant, which is defined in the *ble_config.c* source file. The default value for this is 900 (15 minutes).

When Host Privacy is enabled, the Host ignores the ownAddressType value for the advertising, scanning or connect parameters. It will always use the random address type in order to use the RPA configured in the Controller in the packets sent over the air.

As mentioned in the Introduction section, call this API for random address generation. For random address resolution there is no need to do so, it is performed by default against the bonded devices list.

4.4.3 Controller privacy

To enable or disable Controller Privacy, the following API may be used:

```
bleResult_t Gap_EnableControllerPrivacy
(
    bool_t                                  enable,
    const uint8_t *                         aOwnIrk,
    uint8_t                                 peerIdCount,
    const gapIdentityInformation_t*         aPeerIdentities
);
```

When *enable* is set to TRUE, *aOwnIrk* parameter shall not be NULL, *peerIdCount* shall not be zero or greater than *gMaxResolvingListSize_c*, and *aPeerIdentities* shall not be NULL.

The IRK defined by *aOwnIrk* is used by the Controller to periodically generate a new Resolvable Private Address (RPA). The lifetime of the RPA is a number of seconds contained by the *gGapControllerPrivacyTimeout* external constant, which is defined in the *ble_config.c* source file. The default value for this is 900 (15 minutes).

The *aPeerIdentities* is an array of identity information for each bonded device. The identity information contains the device's identity address (public or random static address) and the device's IRK. This array can be obtained from the Host with the *Gap_GetBondedDevicesIdentityInformation* API.

Enabling Controller Privacy involves a quick sequence of commands to the Controller. When the sequence is complete, the *gControllerPrivacyStateChanged_c* generic event is triggered.

4.4.3.1 Privacy mode

In Bluetooth LE 5.0, the privacy mode has been introduced as an optional feature and is part of the GAP identity structure together with the address and address type. There are two modes: Network Privacy Mode (default) and Device Privacy Mode. These are valid only for Controller Privacy.

A device in network privacy mode only accepts packets from peers using private addresses.

A device in device privacy mode also accepts packets from peers using identity addresses, even if the peer had previously distributed the IRK. Private addresses are also accepted.

The privacy mode of a device is stored in NVM together with the IRK with a default value of Network. If the application wants to change this value it can extract the peer identities, modify the privacy mode from network to device and then enable Controller Privacy with the value.

To change the privacy mode of a device and make the change persistent, the user must call the following API:

```
bleResult_t Gap_SetPrivacyMode
(
    uint8_t nvmIndex,
    blePrivacyMode_t privacyMode
);
```

4.4.3.2 Scanning and initiating

When a Central device is scanning while Controller Privacy is enabled, the Controller actively tries to resolve any RPA contained in the Advertising Address field of advertising packets. If any match is found against the peer IRK list, then the *advertisingAddressResolved* parameter from the scanned device structure is set to TRUE.

In this case, the *addressType* and *aAddress* fields no longer contain the actual Advertising Address as seen over the air, but instead they contain the identity address of the device whose IRK was able to resolve the Advertising Address. In order to connect to this device, these fields shall be used to complete the *peerAddressType* and *peerAddress* fields of the connection request parameter structure, and the *usePeerIdentityAddress* field shall be set to TRUE.

If *advertisingAddressResolved* is equal to FALSE, then the advertiser is using a Public or Random Static Address, an NRPA, or a RPA that could not be resolved. Therefore, the connection to this device is initiated as if Controller Privacy was not enabled, by setting *usePeerIdentityAddress* to FALSE.

4.4.3.3 Advertising

When a Peripheral starts advertising while Controller Privacy is enabled, the *ownAddressType* field of the advertising parameter structure is unused. Instead, the Controller always generates an RPA and advertises with it as Advertising Address.

If directed advertising is used, the Host only allows advertising to a device in the resolving list in order to be able to generate RPAs.

4.4.3.4 Connected

When a device connects while Controller Privacy is enabled, the *gConnEvtConnected_c* connection event parameter structure contains more relevant fields than without Controller Privacy.

The *peerRpaResolved* field equals TRUE if the peer was using an RPA that was resolved using an IRK from the list. In that case, the *peerAddressType* and *peerAddress* fields contain the identity address of the resolved device, and the actual RPA used to create the connection (the RPA that a Central used when initiating the connection, or the RPA that the Peripheral advertised with) is contained by the *peerRpa* field.

The *localRpaUsed* field equals TRUE if the local Controller was automatically generating an RPA when the connection was created, and the actual RPA is contained by the *localRpa* field.

4.5 Setting PHY mode in a connection

In Bluetooth LE 5.0, the user is able to change the PHY mode in a connection through the Link Layer PHY Update Procedure and choose between default 1 Mbit/s, 2 Mbit/s high data rate or the coded S2 or S8 PHYs with 500 Kbps or 125 Kbps for longer range.

To set the PHY, the user can call:

```
bleResult_t Gap_LeSetPhy
(
    bool_t        defaultMode,
    deviceId_t    deviceId,
    uint8_t       allPhys,
    uint8_t       txPhys,
    uint8_t       rxPhys,
    uint16_t      phyOptions
);
```

There are two modes to use this API:

1. If defaultMode is set to TRUE, the user can call this function without being in a connection, i.e. provide a device ID. The PHY option is used by the Link Layer in the PHY response when a connection is created and the peer device initiates the PHY Update Procedure. The application should listen for gLePhyEvent_c with the gPhySetDefaultComplete_c sub event type for the confirmation of the operation.
2. If defaultMode is set to FALSE, the user must also provide a valid device ID. The Host asks the Link Layer to initiate the PHY Update Procedure with the peer device using the provided parameters.

The application should listen for gLePhyEvent_c with the gPhyUpdateComplete_c sub event type for the confirmation of the update procedure to have ended. The result of the operation populates in the txPhy and rxPhy of the event. The result is from the negotiation of the local parameters and the peer PHY preferences.

To read the current PHY on a connection, call the following API:

```
bleResult_t Gap_LeReadPhy
(
    deviceId_t deviceId
);
```

The application should listen for gLePhyEvent_c with the gPhyRead_c sub event type for the confirmation of the operation. The txPhy and rxPhy indicate the current modes used in the connection.

4.6 Data management of bonded devices

The Host handles the management of the bonding data without requiring application intervention. The application must provide the NVM write, read, and erase functions presented in Section 2.3 "Non-Volatile Memory (NVM) access ". The Host creates bonds if bonding is required after the pairing.

The bonded data structure is presented below, together with the GAP APIs that access it, for most APIs require a connection to be established with the device in the bonded list, the others can be accessed any time using the NVM index.

1. Bond Header – identity address and address type that uniquely identify a device together with the IRK and privacy mode.
   - `Gap_GetBondedDevicesIdentityInformation` – for all bonds
2. Bond Data Dynamic - security counters for signed operations – managed by the stack
3. Bond Data Static – LTK, CSRK, Rand, EDIV, security information for read and write authorizations
   - `Gap_SaveKeys` – NVM index

- `Gap_LoadKeys` – NVM index
- `Gap_LoadEncryptionInformation` - deviceId
- `Gap_Authorize – deviceId` - GATT Server only

4. Bond Data Legacy - Legacy pair information and CSRK
   - `Gap_LoadEncryptionInformation` - deviceId
5. Bond Data Device Info - custom peer information (service discovery data) and device name
   - `Gap_SaveCustomPeerInformation` - deviceId
   - `Gap_LoadCustomPeerInformation` - deviceId
   - `Gap_SaveDeviceName` - deviceId
   - `Gap_GetBondedDeviceName` – NVM index
6. Bond Data Descriptor List - configuration of indications and notifications for CCCD handles – GATT Server only
   - `Gap_CheckNotificationStatus` - deviceId
   - `Gap_CheckIndicationStatus` - deviceId

However, there may be some cases when an application wants to manage this data to read data from a bonded device created by the Host, create a bond obtained out-of-band or update an existing bond. For this use case, two GAP APIs and a GAP event have been added.

1. **Load the Keys of a bonded device**.
   The user can call the following function to read the keys exchanged during pairing and stored by the Bluetooth LE Host Stack in the bond area when the pairing is complete.
   The application is informed of the NVM index through the `gBondCreatedEvent_c` sent by the stack immediately after the bond creation. The application is responsible for passing the memory in the `pOutKeys OUT` parameter to fill in the keys, if any of the keys are set to NULL, the stack does not fill that information. The `pOutKeyFlags OUT` parameter indicates to the application which of the keys were stored by the stack as not all of them may have been distributed during pairing.
   The `pOutLeSc` indicates if Bluetooth LE 4.2 LE Secure Connections Pairing was used, while the `pOutAuth` indicates if the peer device is authenticated for MITM protection. All these OUT parameters are recommended to be retrieved from the bond and added if later passed as input parameters for the save keys API.
   This function executes synchronously.

```
bleResult_t Gap_LoadKeys
(
    uint8_t             nvmIndex,
    gapSmpKeys_t*       pOutKeys,
    gapSmpKeyFlags_t*   pOutKeyFlags,
    bool_t*             pOutLeSc,
    bool_t*              pOutAuth);
);
```

   The `gapSmpKeys_t` is the structure used during the key distribution phase, as well as in the `gConnEvtKeysReceived_c` event and is as follows. The difference is that the Bluetooth LE device address cannot be set to NULL neither when loading a bond or when creating one as it identifies the bonded device together with the NVM index.

Table 3. 'gapSmpKeys_t' structure

Event Data	Data type	Data Description
`cLtkSize`	uint8_t	Encryption Key Size filled by the stack. If `aLtk` is NULL, this is ignored. In Advanced Secure Mode, this should be the size of the LTK encrypted blob of 40 bytes.
`aLtk`	uint8_t*	Long Term (Encryption) Key

Table 3. 'gapSmpKeys_t' structure*...continued*

Event Data	Data type	Data Description
		or LTK encrypted blob if Advanced Secure Mode is enabled. NULL if LTK is not distributed, else size is given by `cLtkSize` .
`aIrk`	uint8_t*	Identity Resolving Key. NULL if `aIrk` is not distributed.
`aCsrk`	uint8_t*	Connection Signature Resolving Key. NULL if `aCsrk` is not distributed.
`cRandSize`	uint8_t	Size of RAND filled by the stack; usually equal to `gcSmpMaxRandSize_c`. If `aLtk` is NULL, this is ignored.
`aRand`	uint8_t*	RAND value used to identify the LTK. If `aLtk` is NULL, this is ignored.
`ediv`	uint16_t	EDIV value used to identify the LTK. If `aLtk` is NULL, this is ignored.
`addressType`	`bleAddress Type_t`	Public or Random address.
`aAddress`	uint8_t*	Device Address. It cannot be NULL.

The structure for the GAP SMP Key Flags is the following:

Table 4. GAP SMP Key Flags

Flag Type	Description
`gNoKeys_c`	No key is available.
`gLtk_c`	Long-Term Key is available.
`gIrk_c`	Identity Resolving Key is available.
`gCsrk_c`	Connection Signature Resolving Key is available.

2. Save the Keys to create a bond or update an existing bonded device.
The user can call the following function to create a bond on a device based on information obtained Out of Band. For instance, one can use the output of `Gap_LoadKeys` from the previous section. This can be useful in transferring a bond created by the stack after a pairing procedure or if the application wants to manipulate bonding data. The behavior of the stack remains the same, if the bonding is required after a pairing, the stack stores the bonding information if possible. In this case, the NVM index is passed to the application through `gBondCreatedEvent_c`.
This function executes asynchronously, as the stack can create a bond during the execution. The application should listen for the previous mentioned event `gBondCreatedEvent_c`. The result of the function call is passed synchronously. However, if an asynchronous error has occurred during the actual save, it is passed to the application through the `gInternalError_c` event with a `gSaveKeys_c` error source.
The stack creates a bond if the NVM index is free or update the keys from an NVM index if it stores a valid entry.
The address from the GAP SMP Keys structure must not be NULL. If other members of the structure are NULL, they are ignored.
LE SC flag indicates if Bluetooth LE 4.2 Secure Connections was used during pairing and Auth specifies if the peer is authenticated for MITM protection.

```
bleResult_t Gap_SaveKeys
(
    uint8_t         nvmIndex,
    gapSmpKeys_t*   pKeys,
    bool_t          leSc,
    bool_t          auth
```

```
);
```

3. Bond created event.
   A GAP event is added to the Bluetooth LE Generic Callback to inform the application of the NVM index whenever the stack creates a bond or when a `Gap_SaveKeys` request succeeds. The event is also generated if the NVM index was a valid occupied entry and only some of the keys in the bonded information have been updated.
   The NVM index is then used in the GAP APIs to save or load information from the bond.

Event Data	Data type	Data Description
nvmIndex	uint8_t	NVM index for the new created bond
addressType	bleAddressType_t	Public or Random (static) address of the bond
address	bleDeviceAddress_t	Address of the bond

4.6.1 Application removal of bonded devices data

The application can remove a bonded device from NVM. The bonded device cannot be deleted if it is in an active connection. The application can remove one or all bonds by calling the following synchronous GAP APIs:

- **`Gap_RemoveBond`**`(uint8_t nvmIndex)` – `nvmIndex` can be obtained via the `Gap_CheckIfBonded` API.
- **`Gap_RemoveAllBonds()`** - no connections should be active otherwise the call fails.

Removing a bonded device does not affect the controller address resolution state nor the contents of either the Controller Filter Accept List or the Controller Resolving List. If Controller Privacy is enabled, it remains so until it is disabled or the device is reset.

In a scenario where the user wants to remove a bonded device and all its effects on device behavior (Controller Filter Accept List, Controller Resolving List), the following operations should be executed:

- **`Gap_ClearFilterAcceptList`** `or` **`Gap_RemoveDeviceFromFilterAcceptList`**
  - Clear Controller Filter Accept List or clear a device from Filter Accept List.
- **`Gap_RemoveAllBonds`** `or` **`Gap_RemoveBond`**
  - All bonded devices are removed or one bonded device is removed from NVM.
- **`BleConnManager_DisablePrivacy`**
  - Controller Privacy is disabled, Controller Resolving List is cleared and address resolution is disabled. The device should not be advertising or scanning, otherwise this call fails.
- **`BleConnManager_EnablePrivacy`**
  - Called after the `gControllerPrivacyStateChanged_c` event is received, confirming Controller Privacy has been disabled. If not all bonds have been deleted, Controller Privacy is reenabled. In the absence of bonds, Host Privacy is enabled.

4.7 Controller enhanced notifications

This section describes how the application can configure and monitor the notifications generated by the Bluetooth Controller when advertising, scan, or connection events occur. This feature is proprietary to NXP that is available on selected Controllers.

The user can choose between two options:

1. Enable notifications from the GAP layer and monitor GAP events in the GAP Generic Callback. The controller issues HCI vendor-specific events processed by the Bluetooth LE Host and presented to the application in the GAP Generic Callback.
2. Enable notifications from the Controller interface and monitor controller events in a user-defined Application Callback.

3. Combination of the above two options: configure feature at GAP layer and install an Application Callback through the Controller interface. After setting the callback, the HCI vendor-specific events are not issued and implicitly the GAP events. Instead, the user receives the notifications in the installed callback until setting the callback to `NULL` again if it wants to revert to GAP events.

- **GAP configuration:**
  The user should call the following function to enable various events from the mask or use Event `None` to disable the feature. The `Device ID` is valid only for connection events.

```
bleResult_t Gap_ControllerEnhancedNotification
(    uint16_t eventType,
    deviceId_t deviceId
);
```

The event type is a bitmask having the following options:

Table 5. Event types and their description

Event Type	Event Description
gNotifEventNone_c	No enhanced notification event enabled
gNotifConnEventOver_c	Connection event over
gNotifConnRxPdu_c	Connection RX PDU
gNotifAdvEventOver_c	Advertising event over
gNotifAdvTx_c	Advertising ADV transmitted
gNotifAdvScanReqRx_c	Advertising SCAN REQ RX
gNotifAdvConnReqRx_c	Advertising CONN REQ RX
gNotifScanEventOver_c	Scanning event over
gNotifScanAdvPktRx_c	Scanning ADV PKT RX
gNotifScanRspRx_c	Scanning SCAN RSP RX
gNotifScanReqTx_c	Scanning SCAN REQ TX
gNotifConnCreated_c	Connection created
gNotifChannelMatrix_c	Enable channel status monitoring
gNotifPhyReq_c	Phy Req Pdu ack received
gNotifConnChannelMapUpdate_c	Channel map update
gNotifConnInd_c	Connect indication
gNotifPhyUpdateInd_c	Phy update indication

After enabling events, the user should wait for a `gControllerNotificationEvent_c` GAP Generic Event in the GAP Generic Callback. The first event received should have the event type set to `gNotifEventNone_c` with a status of `success` confirming the selected event mask has been enabled. The same event types apply for both the GAP command and the GAP event. The structure for the Controller Notification event is the following:

Table 6. Controller Notification Event structure

Event Data	Data type	Data Description
eventType	bleNotificationEvent_t	Enhanced notification event type
deviceId	deviceId_t	Device id of the peer, valid for connection events
rssi	int8_t	RSSI, valid for RX event types

Table 6. Controller Notification Event structure*...continued*

Event Data	Data type	Data Description
channel	uint8_t	Channel, valid for connection event over or Rx/Tx events
ce_counter	uint16_t	Connection event counter, valid for connection events only
status	bleResult_t	Status of the request to select which events to be enabled/disabled
timestamp	uint16_t	Timestamp in 625 µs slots, valid for Conn RX event and Conn Created event
adv_handle	uint8_t	Advertising Handle, valid for advertising events, if multiple ADV sets supported

- **Controller configuration:**
  The user should call the following function to enable various events from the mask or use Event None to disable the feature. The same event types apply as the GAP layer types. The connection handle is valid only for connection events.

```
bleResult_t Controller_ConfigureEnhancedNotification
(
    uint16_t eventType,
    uint16_t conn_handle
);
```

The event monitoring is done in a user-installed callback by calling:

```
bleResult_t Controller_RegisterEnhancedEventCallback
(
    bleCtrlNotificationCallback_t notificationCallback
);
```

Where the types are the following:

```
typedef struct bleCtrlNotificationEvent_tag
{
    uint16_t event_type; /*! bleNotificationEventType_t */
    uint16_t conn_handle;
    uint8_t  rssi;
    uint8_t  channel_index;
    uint16_t conn_ev_counter;
    uint16_t timestamp;
    uint8_t  adv_handle;
} bleCtrlNotificationEvent_t;
typedef void (*bleCtrlNotificationCallback_t)
(
    bleCtrlNotificationEvent_t    *pNotificationEvent
);
```

The event structure is nearly identical as the GAP one, except there is no status as the function call executes synchronously.

Table 7. '`bleCtrlNotificationEvent_tag`' Event structure

Event Data	Data type	Data Description
event_type	bleNotificationEvent_t	Enhanced notification event type
conn_handle	uint16_t	Connection handle of the peer, valid for connection events
rssi	int8_t	RSSI, valid for RX event types
channel_index	uint8_t	Channel, valid for connection event over or Rx/Tx events
conn_ev_counter	uint16_t	Connection event counter, valid for connection events only

Table 7. 'bleCtrlNotificationEvent_tag' Event structure ...continued

Event Data	Data type	Data Description
timestamp	uint16_t	Timestamp in 625 µs slots, valid for Conn RX event and Conn Created event
adv_handle	uint8_t	Advertising Handle, valid for advertising events, if multiple ADV sets supported

4.8 Extended advertising

Starting with Bluetooth 5, the advertising channels are separated in primary advertising channels and secondary advertising channels:

1. Primary advertising channels
   - Use 3 legacy advertising channels 37, 38, and 39.
   - Can use either legacy 1M PHY or new LE Coded PHY.
   - PHY payload can vary from 6 to 37 bytes.
   - Packets on these channels are part of the advertising events.
2. Secondary advertising channels
   - Use 37 channels, with the same channel index as the data channels.
   - Can use any LE PHY, but the same PHY during an Extended Advertising Event.
   - PHY payload can vary from 0 to 255 bytes.
   - Auxiliary packets on these channels are part of the Extended Advertising Event that begins at the same time with the advertising event on primary channel and ends with the last packet on the secondary channel.

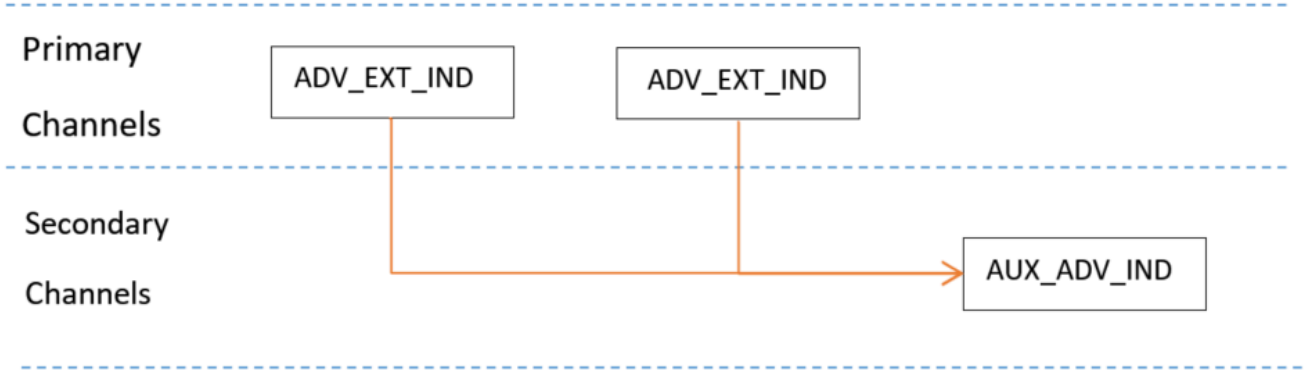

Figure 6. Extended advertising

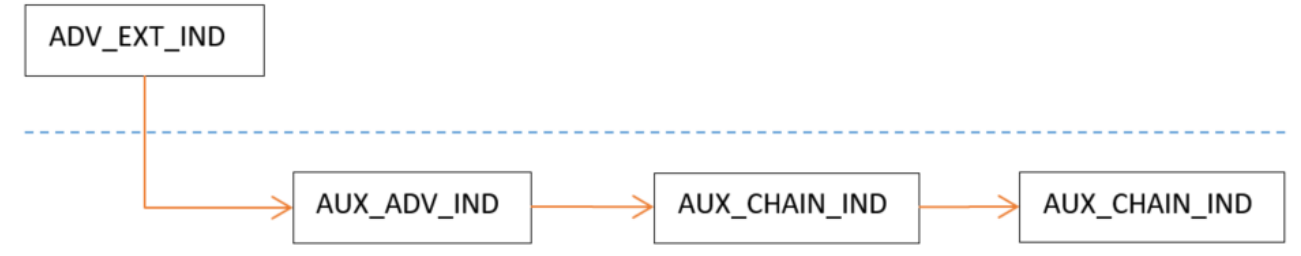

Figure 7. Extended advertising – Multiple chains

An advertising data set is represented by advertising PDUs belonging together in an advertising event. Each set has different advertising parameters: PDU type, advertising interval, and PHY mode. The advertising data sets are identified by the Advertising SID (Set ID) field from the ADI – Advertising Data Info. Advertising data or

Scan response data can be changed for each adverting data set and the random value of DID (Data ID) field is updated to differentiate between them.

Refer to Figure 6 and Figure 7.

4.8.1 Peripheral setup

This section describes the extended advertising GAP API. The application should not use both the extended and legacy API (described in section 4.2.1). If this requirement cannot be met, the application should at least wait for the generated events in the Advertising Callback prior to using the other API. That is, it is advisable to call legacy functions only after the event pertaining to an extended API is received, and vice versa. This GAP constraint can be considered an extension of the HCI constraint from the Bluetooth 5 specification: "A Host should not issue legacy commands to a Controller that supports the LE Feature (Extended Advertising)".

The application configures extended advertising by going through the following states:

1. Set the extended advertising parameters by calling:

```
bleResult_t Gap_SetExtAdvertisingParameters
(
gapExtAdvertisingParameters_t* pAdvertisingParameters
);
```

It may use the default set of parameters *gGapDefaultExtAdvertisingParameters_d*. The application should wait for a *gExtAdvertisingParametersSetupComplete_c event* in the Generic Callback. Only one advertising set can be configured at a time. Comparing with the legacy *Gap_SetAdvertisingParameters* command, the new set of parameters is as follows.

Table 8. New extended advertising parameters

Parameter	Description
SID	Value of the Advertising SID subfield in the ADI field of the PDU.
handle	Used to identify an advertising set. Possible values are 0x00 or 0x01 since the current implementation supports two advertising sets.
extAdvProperties	BIT0 - Connectable advertising BIT1 - Scannable advertising BIT2 - Directed advertising BIT3 - High Duty Cycle Directed Connectable advertising (≤3.75 ms Advertising Interval) BIT4 - Use legacy advertising PDUs BIT5 - Omit advertiser's address from all PDUs ("anonymous advertising") BIT6 - Include TxPower in the extended header of the advertising PDU. If legacy advertising PDU types are being used (BIT4 = 1), permitted properties values are presented in the next table. If the advertising set already contains data, the type shall be one that supports advertising data and the amount of data shall not exceed 31 octets. If extended advertising PDU types are being used (BIT4 =0), then the advertisement shall not be both connectable and scannable. While high duty cycle directed connectable advertising (≤ 3.75 ms advertising interval) shall not be used (BIT3 = 0).
txPower	Maximum power level at which the advertising packets are to be transmitted, the Controller can choose any power level <= txPower. Value 127 to be used if Host has no preference.
primaryPHY	PHY for ADV_EXT_IND: LE 1 M or LE Coded

Table 8. New extended advertising parameters*...continued*

Parameter	Description
secondaryPHY	PHY for AUX_ADV_IND and periodic advertising: LE 1 M, LE 2 M or LE Coded. Ignored for legacy advertising
secondaryAdvMaxSkip	Maximum advertising events that the Controller can skip before sending the AUX_ADV_IND packets on the secondary advertising channel. Higher values may result in lower power consumption. Ignored for legacy advertising
enableScanReqNotification	Whether to enable notifications when scanning PDUs (SCAN_REQ, AUX_SCAN_REQ) are received. If enabled, the application is notified upon scan requests by gExtScanNotification_c events in the Advertising Callback

When using LE Coded PHY for advertising, the default coding scheme chosen by link layer is S=8 (125 kb/s data rate). To change the default coding scheme, the user has two options:

- At compile time by defining *mLongRangeAdvCodingScheme_c*, or
- At run time by calling the API *Controller_ConfigureAdvCodingScheme()*.

In both cases, the value of the define or the parameter of the API has to be an appropriate value for primary and secondary PHYs as defined by the enumeration *advCodingScheme_tag* found in *controller_interface.h*.

EventType	PDU Type	Advertising Event Properties
Connectable and scannable undirected	ADV_IND	00010011b
Connectable directed (low duty cycle)	ADV_DIRECT_IND	00010101b
Connectable directed (high duty cycle)	ADV_DIRECT_IND	00011101b
Scannable undirected	ADV_SCAN_IND	00010010b
Non-connectable and Nonscannable undirected	ADV_NONCONN_IND	00010000b

2. Set the advertising data and/or scan response data by calling:

```
bleResult_t Gap_SetExtAdvertisingData
(
uint8_t handle,
gapAdvertisingData_t* pAdvertisingData,
gapScanResponseData_t* pScanResponseData
);
```

Either of the pAdvertisingData or pScanResponseData parameters can be NULL, but not both. For extended advertising (BIT4 = 0) only one must be different than NULL – the scannable advertising bit (BIT1) indicates whether pAdvertisingData (BIT1 = 0) orpScanResponseData (BIT1 = 1) is accepted. The total amount of Advertising Data shall not exceed 1650 bytes. Application should wait for a *gExtAdvertisingDataSetupComplete_c* event in the Generic Callback.

3. Enable extended advertising by calling:

```
bleResult_t Gap_StartExtAdvertising
(
gapAdvertisingCallback_t advertisingCallback,
gapConnectionCallback_t connectionCallback,
uint8_t handle,
uint16_t duration,
uint8_t maxExtAdvEvents
);
```

When using the common application structure, the application can use the following API defined in *app_conn.h*:

```
bleResult_t BluetoothLEHost_StartExtAdvertising
(
    appExtAdvertisingParams_t *pExtAdvParams,
    gapAdvertisingCallback_t  pfAdvertisingCallback,
    gapConnectionCallback_t   pfConnectionCallback
);
```

The API goes through the steps of setting the advertising data and parameters. Events from the Host task are treated in the *App_AdvertiserHandler()* function, implemented in *app_advertiser.c*. To set the extended advertising parameters and data *BluetoothLEHost_StartExtAdvertising* a parameter of the following type:

```
typedef struct appExtAdvertisingParams_tag
{
    gapExtAdvertisingParameters_t *pGapExtAdvParams;
    gapAdvertisingData_t *pGapAdvData;
    gapScanResponseData_t *pScanResponseData;
    uint8_t                    handle;
    uint16_t                   duration;
    uint8_t                    maxExtAdvEvents;
} appExtAdvertisingParams_t;
```

Advertising may be enabled for each previously configured advertising set, identified by the handle parameter. If duration is set to 0, advertising continues until the Host disables it, otherwise advertising is only enabled for this period (multiple of 10 ms). *maxExtAdvEvents* represent the maximum number of extended advertising events the Controller shall attempt to send prior to terminating the extended advertising, ignored if set to 0. Application should wait for a *gExtAdvertisingStateChanged_c* or a *gAdvertisingCommandFailed_c* event in the Advertising Callback.

4. Disable advertising by calling:

```
bleResult_t Gap_StopExtAdvertising
(
uint8_t handle
);
```

Application should wait for a *gExtAdvertisingStateChanged_c* or a *gAdvertisingCommandFailed_c* event in the Advertising Callback.

5. Remove the advertising set by calling:

```
bleResult_t Gap_RemoveAdvSet
(
uint8_t handle
);
```

Application should wait for a *gExtAdvertisingSetRemoveComplete_c* event in the Generic Callback.

4.8.2 Central setup

The application configures the extended scanning by going through the following states:

1. Start scanning by calling:

```
bleResult_t Gap_StartScanning
(
const gapScanningParameters_t* pScanningParameters,
gapScanningCallback_t scanningCallback,
gapFilterDuplicates_t enableFilterDuplicates,
uint16_t duration,
uint16_t period
```

```
)
```

When using the common application structure, the application can use the following API defined in *app_conn.h*:

```
bleResult_t BluetoothLEHost_StartScanning
(
    appScanningParams_t   *pAppScanParams,
    gapScanningCallback_t pfCallback
);
```

The API starts scanning using the given parameters, which must have the following structure:

```
typedef struct appScanningParams_tag
{
    gapScanningParameters_t *pHostScanParams;         /*!< Pointer to host
 scan structure */
    gapFilterDuplicates_t enableDuplicateFiltering;        /*!< Duplicate
 filtering mode */
    uint16_t duration;
    /*!< scan duration  */
    uint16_t period;
      /*!< scan period  */
} appScanningParams_t;
```

Application may use the default set of parameters *gGapDefaultExtScanningParameters_d*. If the *pScanningParameters* pointer is NULL, the latest set of parameters are used. The *scanningPHYs* parameter indicates the PHYs on which the advertising packets should be received on the primary advertising channel. As a result, permitted values for the parameter are 0x01 (scan LE 1M), 0x04 (scan LE Coded) and 0x05 (scan both LE 1M and LE Coded). There are no strict timing rules for scanning, yet if both PHYs are enabled for scanning, the scan interval value must be large enough to accommodate two scan windows (interval >= 2 * window).
If the advertiser uses legacy advertising PDUs, the device may actively scan by sending a SCAN_REQ PDU to the advertiser on the LE 1M primary advertising channel (no secondary channel in legacy advertising). Respectively, if the advertiser uses extended advertising PDUs, the active scan operation takes place on the secondary advertising channel. After the device receives a scannable ADV_EXT_IND PDU on the primary advertising channel (PHY LE 1M or Coded), it starts listening for the AUX_ADV_IND PDU on the secondary advertising channel (PHY 1M, 2M or Coded). Once received, the device sends an AUX_SCAN_REQ to the advertiser. Next, an AUX_SCAN_RSP PDU should be received, containing the scan response data. Application should wait for a *gScanStateChanged_c* or a *gScanCommandFailed_c*in the Scanning Callback.

2. Collect information by waiting for *gDeviceScanned_c* (legacy advertising PDUs) or *gExtDeviceScanned_c* (extended advertising PDUs) event in the Scanning Callback. The *gExtDeviceScanned_c* event contains additional information pertaining to the extended received PDU, such as: primary PHY, secondary PHY, advertising SID, interval of the periodic advertising if enabled in the set.
   When using the common application structure, the application can use the following API defined in *app_conn.h*, to search the contents from *pData* in an advertising element:

```
bool_t BluetoothLEHost_MatchDataInAdvElementList
(
    gapAdStructure_t *pElement,
    void             *pData,
    uint8_t          iDataLen
);
```

3. Stop scanning by calling the function below:

```
bleResult_t Gap_StopScanning(void);
```

Application should wait for a *gScanStateChanged_c* or a *gScanCommandFailed_c* in the Scanning Callback.

4. Connect to a device by calling the function below:

```
bleResult_t Gap_Connect
(
const gapConnectionRequestParameters_t* pParameters,
gapConnectionCallback_t connCallback
);
```

When using the common application structure, the following API can be used:

```
bleResult_t BluetoothLEHost_Connect
(
    gapConnectionRequestParameters_t*   pParameters,
    gapConnectionCallback_t             connCallback
);
```

The *initiatingPHYs* parameter indicates the PHYs on which the advertising packets should be received on the primary advertising channel and the PHYs for which connection parameters have been specified. The parameter is a bitmask of PHYs: BIT0 = LE 1M, BIT1 = LE 2M and BIT2 = LE Coded. The Host may enable one or more initiating PHYs, but it must at least set one bit for a PHY allowed for scanning on the primary advertising channel, i.e., BIT0 for LE 1M PHY or BIT2 for LE Coded PHY.
If the advertiser uses legacy advertising PDUs, the device may connect by sending a CONNECT_IND PDU to the advertiser on the LE 1M primary advertising channel (no secondary channel in legacy advertising). On the other hand, if the advertiser uses extended advertising PDUs, the extended connect operation takes place on the secondary advertising channel. After the device receives a connectable ADV_EXT_IND PDU on the primary advertising channel (PHY LE 1M or Coded), it starts listening for the connectable AUX_ADV_IND PDU on the secondary advertising channel (PHY 1M, 2M or Coded). Once received, the device sends an AUX_CONNECT_REQ to the advertiser. Next, if AUX_CONNECT_RSP PDU is received, the device enters the Connection State in the Central role on the secondary advertising channel PHY.
Application should wait for a *gConnEvtConnected_c* event in the Connection Callback. If the channel selection algorithm #2 is used for this connection, then a *gConnEvtChanSelectionAlgorithm2_c* event is also generated.
After the connection is successfully established, the application may choose to read the connection PHY by calling the *Gap_LeReadPhy* API. It may also opt to change the PHY of the connection by triggering a PHY Update Procedure using the *Gap_LeSetPhy API*. However, the Controller might not be able to perform the change if, in case the peer does not support the new requested PHY.

4.9 Periodic Advertising

Periodic channels are used for periodic broadcast between unconnected devices. A periodic channel is represented by a channel map and a set of hopping and timing parameters.

The set of channels is represented by the 37 data channels. A packet sent by an advertiser can also have a payload of up to 255 bytes and it can be sent on any LE PHY. Figure 8 and Figure 9.

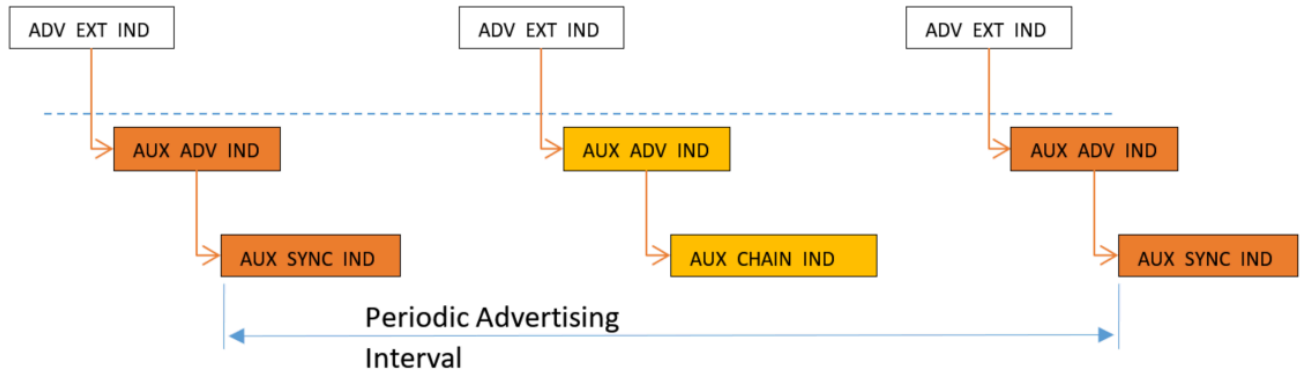

Figure 8. Extended Advertising and Periodic Advertising combined

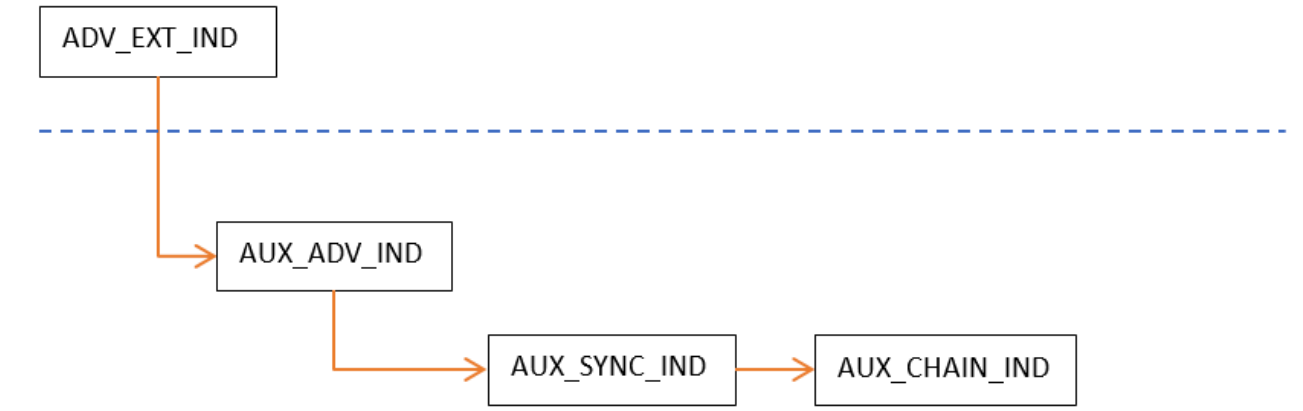

Figure 9. Periodic Advertising – Multiple chains

4.9.1 Peripheral Setup

1. First set the extended advertising parameters using Gap_SetExtAdvertisingParameters. The extended advertising type must be set to non-connectable and non-scannable.
2. Set the periodic advertising parameters using the same handle as in the previous command.

```
bleResult_t Gap_SetPeriodicAdvParameters
(
    gapPeriodicAdvParameters_t*    pAdvertisingParameters
);
```

Wait for a *gPeriodicAdvParamSetupComplete_*cevent in the generic callback.

3. Next, set the periodic advertising data by calling:

```
bleResult_t Gap_SetPeriodicAdvertisingData
(
    uint8_t                       handle,
    gapAdvertisingData_t*  pAdvertisingData,
    bool_t bUpdateDID
);
```

pAdvertisingData cannot be NULL. If periodic advertising data must be empty, set *cNumAdStructures* to 0. Wait for a *gPeriodicAdvDataSetupComplete_c* event in the generic callback.

4. Start extended advertising using Gap_StartExtAdvertising.
5. Last, enable Periodic Advertising. Periodic advertising starts only after extended advertising is started.

```
bleResult_t Gap_StartPeriodicAdvertising
(
    uint8_t handle,
    bool_t  bIncludeADI
);
```

Wait for a *gPeriodicAdvertisingStateChanged_c* event in the advertising callback.

4.9.2 Central Setup

The application may decide to listen to periodic advertising by going through the following states:

1. [Optional] Add a known periodic advertiser to the periodic advertiser list held in the Controller by calling:

```
bleResult_t Gap_UpdatePeriodicAdvList
(
    gapPeriodicAdvListOperation_t operation,
    bleAddressType_t              addrType,
    uint8_t*                      pAddr,
    uint8_t                       SID
);
```

Wait for the *gPeriodicAdvListUpdateComplete_c* event in the Generic Callback.

2. Synchronize with a periodic advertiser by calling:

```
bleResult_t Gap_PeriodicAdvCreateSync
(
    gapPeriodicAdvSyncReq_t*   pReq,
);
```

pReq parameter *filterPolicy* can be set to *gUseCommandParameters_c* to synchronize with the given peer, or to *gUsePeriodicAdvList_c* to start synchronizing with all the devices in the previously populated periodic advertiser list.
Wait for the *gPeriodicAdvSyncEstablished_c* event and check the status. If scanning is not enabled at the time this command is sent, synchronization occurs after scanning is started. Synchronization remains pending until *gPeriodicAdvSyncEstablished_c* event is received. If synchronization was successful, the *syncHandle* is returned in this event.

3. Terminate the synchronization with the periodic advertiser by calling:

```
bleResult_t Gap_PeriodicAdvTerminateSync
(
    uint16_t syncHandle
);
```

To cancel a pending synchronization, the application should call *Gap_PeriodicAdvTerminateSync* with *syncHandle* set to the reserved value *gBlePeriodicAdvOngoingSyncCancelHandle* and wait for *gPeriodicAdvCreateSyncCancelled_c* event.
Otherwise, to terminate an already established sync with an advertiser, use the *syncHandle* value from the *gPeriodicAdvSyncEstablished_c* event and wait for a *gPeriodicAdvSyncTerminated_c* event.

4.10 Periodic Advertising with Responses (PAwR)

This section describes the Central and Peripheral setup for Periodic Advertising with Responses (PAwR).

4.10.1 Central Setup

1. Start scanning using `Gap_StartScanning`. Wait for `gPeriodicDeviceScannedV2_c` events in the scanning callback.
2. Synchronize with a periodic advertiser by calling `Gap_PeriodicAdvCreateSync`. Wait for the `gPeriodicAdvSyncEstablished_c` event in the scanning callback. When PAwR is involved, this event includes additional information such as number of subevents, subevent interval, response slot delay and spacing,
3. Synchronize to a PAwR subevent by calling `Gap_SetPeriodicSyncSubevent`. This API instructs the Controller to sync with a subset of the subevents within a PAwR train identified by syncHandle (obtained after synchronizing with the PAwR train in the previous step).

   ```
   bleResult_t Gap_SetPeriodicSyncSubevent ( uint16_t syncHandle, const
    gapPeriodicSyncSubeventParameters_t* pParams );
   ```

   Wait for the `gPeriodicSyncSubeventComplete_c` event.
4. Use `Gap_SetPeriodicAdvResponseData` to set data in the AD format which would be sent as a Periodic Advertising Response to the broadcaster.

   ```
   bleResult_t Gap_SetPeriodicAdvResponseData ( uint16_t syncHandle, const
    gapPeriodicAdvertisingResponseData_t* pData );
   ```
5. Optionally, the periodic advertiser may initiate a connection. If no connection callback was set on the scanner via APIs such as `Gap_Connect` or `Gap_StartAdvertising/Gap_StartExtAdvertising`, one must be explicitly set. This is achieved by calling `BluetoothLEHost_SetConnectionCallback` (defined in `app_conn.h`), which in turn calls `Gap_SetConnectionCallback`.

   ```
   void Gap_SetConnectionCallback ( gapConnectionCallback_t
    pfConnectionCallback );
   ```

4.10.2 Peripheral Setup

1. First set the extended advertising parameters using `Gap_SetExtAdvertisingParameters`. The extended advertising type must be set to non-connectable and non-scannable.

2. Set the periodic advertising parameters with the same handle as in the previous command. Use the `Gap_SetPeriodicAdvParametersV2` command. Compared to `Gap_SetPeriodicAdvParameters`, this command also configures parameters relevant to PAwR, such as the number of subevents and response slots as well as timing information.

```
bleResult_t Gap_SetPeriodicAdvParametersV2
   (gapPeriodicAdvParametersV2_t* pAdvertisingParameters);
```

Wait for a `gPeriodicAdvParamSetupComplete_c` event in the generic callback.

3. Start extended advertising using `Gap_StartExtAdvertising`.

4. Start periodic advertising using `Gap_StartPeriodicAdvertising`.

5. Wait for `gPerAdvSubeventDataRequest_c` events. These events are used by the Controller to indicate that it is ready to transmit one or more subevents and it is requesting the advertising data for these subevents.

Upon receiving an event, use `Gap_SetPeriodicAdvSubeventData` to set the advertising data for specific subevents.

```
bleResult_t Gap_SetPeriodicAdvSubeventData
      (uint8_t advHandle, const gapPeriodicAdvertisingSubeventData_t* pData);
```

Wait for the `gPeriodicAdvSetSubeventDataComplete_c` event in the generic callback.

6. Wait for `gPerAdvResponse_c` events. These events contain responses sent by devices who are synchronized to the periodic advertising. They include data in the AD format.

7. Optionally, PAwR allows the advertising device to initiate a connection to one of the synchronized scanners. The connection can be initiated by calling `Gap_ConnectFromPawr`.

```
bleResult_t Gap_ConnectFromPawr
      (const gapConnectionFromPawrParameters_t* pParameters,
 gapConnectionCallback_t connCallback);
```

4.11 Encrypted Advertising Data

This section describes the Central and Peripheral setup for encrypted advertising data.

4.11.1 Central Setup

Use the `Gap_DecryptAdvertisingData` API to decrypt the contents of "Encrypted Advertising Data" (0x31) AD types included in scanned data.

```
bleResult_t Gap_DecryptAdvertisingData
    (uint8_t *pData, uint16_t dataLength, const uint8_t *pKey, const uint8_t
 *pIV, uint8_t *pOutput)
```

4.11.2 Peripheral Setup

Use the `Gap_EncryptAdvertisingData` API to obtain the encrypted advertising data, which can then be placed inside the "Encrypted Advertising Data" (0x31) AD type.

```
bleResult_t Gap_EncryptAdvertisingData
  ( const gapAdvertisingData_t *pAdvertisingData, const uint8_t *pKey, const
 uint8_t *pIV, uint8_t *pOutput )
```

4.12 L2CAP credit-based channels

The L2CAP layer, which is responsible for protocol multiplexing, segmentation, and reassembly operations, allows devices to communicate via connection-oriented channels. These channels use credit-based flow control, in which a device grants each peer a number of credits which the peer can use to send packets. The number of credits is decremented with every sent packet. A device can grant more credits to its peers over the duration of the connection.

Unlike the fixed L2CAP CIDs used by protocols such as ATT and SMP, credit-based channels use dynamically allocated CIDS (in the 0x0040-0xFFFF range). The CIDs are automatically allocated by the Bluetooth LE Host Stack.

The Bluetooth LE Host Stack supports both the Credit-based Flow Control Mode and the Enhanced Credit-based Flow Control Mode. In the Enhanced Credit-based Flow Control Mode, devices can open up to five channels in a single connect request/response exchange. Additionally, these channels can be later reconfigured with new MTU and MPS values. In the Credit-based Flow Control Mode, reconfiguration is not possible.

The first thing an application must do is register the control and data callbacks:

```
bleResult_t   L2ca_RegisterLeCbCallbacks
(
    l2caLeCbDataCallback_t     pCallback,
    l2caLeCbControlCallback_t  pCtrlCallback
);
```

The control callback receives events related to channel management such as connection, disconnection, received credits, reconfiguration, and so on.

The data callback receives the data which is being exchanged on the channel.

To use L2CAP credit-based channels, the application must register a PSM. The PSM is analogous to a TCP/UDP port. It is an identifier used to determine the upper layer protocol which is making use of the L2CAP channel. The dynamic PSM range is 0x0080-0x00FF. The number of PSMs supported by an application can be

configured at compile time via the gL2caMaxLePsmSupported_c define. The following API must be called in order to register a PSM:

```
bleResult_t  L2ca_RegisterLePsm
(
   uint16_t     lePsm,
   uint16_t     lePsmMtu
);
```

The MTU configured via this API is used by every channel opened under the PSM, if the Credit-based Flow Control Mode is used. The minimum MTU is 23 and the maximum MTU is 65535.

When the Enhanced Credit-based Flow Control Mode is used, the MTU is specified at each connection request. In this mode, the minimum MTU is 64 and the maximum MTU is 65535.

The local MPS is not configurable by the application. It is set automatically by the Host Stack based on Controller capabilities. Usually, it will be 247.

A previously registered PSM can be deregistered:

```
bleResult_t L2ca_DeregisterLePsm
(
  uint16_t     lePsm
);
```

The number of credit-based channels that can be opened is configurable by the application via the gL2caMaxLeCbChannels_c define. This is the total number for all peers. To open a channel, the following API must be called:

```
bleResult_t L2ca_ConnectLePsm
(
   uint16_t     lePsm,
   deviceId_t   deviceId,
   uint16_t     initialCredits
);
```

To open up to five channels using Enhanced Credit-based Flow Control Mode, use this API:

```
bleResult_t  L2ca_EnhancedConnectLePsm
(
  uint16_t     lePsm,
  deviceId_t   deviceId,
  uint16_t     mtu,
  uint16_t     initialCredits,
  uint16_t     initialCredits,
  uint16_t     *aCids
);
```

The connect APIs must be called by both the initiator and the responder (upon receiving the gL2ca_LePsmConnectRequest_c or gL2ca_LePsmEnhancedConnectRequest_c events in the application).

If the responder does not wish to accept the connection request, it can use the following APIs:

```
 bleResult_t  L2ca_CancelConnection
    (
      uint16_t      lePsm,
      deviceId_t    deviceId,
```

```
     l2caLeCbConnectionRequestResult_t refuseReason
    );
 bleResult_t L2ca_EnhancedCancelConnection
  (
    uint16_t lePsm,
    deviceId_t deviceId,
    l2caLeCbConnectionRequestResult_t refuseReason,
    uint8_t noOfChannels,
    uint16_t *aCids
    );
```

When a channel has been successfully established, the `gL2ca_LePsmConnectionComplete_c` or `gL2ca_LePsmEnhancedConnectionComplete_c` events are received in the application.

To send data on a channel:

```
 bleResult_t L2ca_SendLeCbData
(
   deviceId_t      deviceId,
   uint16_t        channelId,
   const uint8_t*  pPacket,
   uint16_t        packetLength
);
```

The Host Stack keeps track of the credits granted to peers for each channel and decrements them accordingly. When a peer's credit count reaches zero, the application is notified through the `gL2ca_NoPeerCredits_c` event and it can decide to send more credits to the peer for that channel:

```
 bleResult_t L2ca_SendLeCredit
 (
  deviceId_t     deviceId,
  uint16_t       channelId,
  uint16_t       credits
 );
```

The application can also choose to be notified when the number of credits allocated to a peer for a certain channel is nearing `0`, by setting the `gL2caLowPeerCreditsThreshold_c` macro to a non-zero value. When this limit is reached, the `gL2ca_LowPeerCredits_c` event is received and the application can choose to send more credits.

Similarly, when a device receives credits from a peer, the application is notified through the `gL2ca_LocalCreditsNotification_c` event. When the local device has used its last credit, it receives the same `gL2ca_LocalCreditsNotification_c` event with the `localCredits` field set to `0`. The packet that could not be sent due to exhausting the credits remains queued in the Host Stack and it is sent automatically when the local device receives credits from the peer.

To improve application flow control, two notification-type events are implemented by the Host Stack:

- `gL2ca_ChannelStatusChannelBusy_c`
- `gL2ca_ChannelStatusChannelIdle_c`

When the application sends a packet using `L2ca_SendLeCbData`, it receives a `gL2ca_ChannelStatusChannelBusy_c` event in the L2CAP control callback when the Host Stack begins sending the packet. When the Host Stack has sent the packet, a `gL2ca_ChannelStatusChannelIdle_c` event is received. The application can choose to use this event as a signal that it is safe to send the next packet.

To disconnect a channel:

```
bleResult_t L2ca_DisconnectLeCbChannel
(
    deviceId_t  deviceId,
    uint16_t    channelId
);
```

As mentioned previously, channels which use the Enhanced Credit-based Flow Control Mode can be reconfigured. This is achieved via the API:

```
bleResult_t L2ca_EnhancedChannelReconfigure
(
    deviceId_t     deviceId,
    uint16_t       newMtu,
    uint16_t       newMps,
    uint8_t        noOfChannels,
    uint16_t       *aCids
);
```

The reconfiguration request is automatically accepted by the Host Stack if parameters are valid (as per the Bluetooth Core Spec v5.3, MTU cannot be lowered and MPS cannot be lowered for more than one channel). On the responder, the `gL2ca_EnhancedReconfigureRequest_c` is received by the application in case of a successful reconfiguration, informing it of the new channel parameters. On the initiator, the `gL2ca_EnhancedReconfigureResponse_c` is received, informing the application about the received response or a timeout.

4.13 Enhanced ATT

The Enhanced ATT protocol allows concurrent transactions to be handled by the stack. The sequential transaction rule still exists when EATT is used, but its scope is now defined as being per instance of the Enhanced ATT Bearer. EATT transactions might execute in parallel if they are supported by distinct L2CAP channels, which use the Enhanced Credit Based Flow Control Mode (that is, distinct Enhanced ATT Bearers).

When using an Enhanced ATT Bearer, ATT MTU and L2CAP MTU are independently configurable and may be reconfigured during a connection. An increase to the MTU is allowed but reducing its size is not. Allowing MTU to be increased without needing to reestablish the connection has an advantage. It eliminates the risk of a second application using the stack, being unable to continue, due to the previously negotiated MTU being too small.

Enhanced ATT bearers are identified through Bearer Ids. Enhanced ATT Bearer Ids are assigned internally and have a valid range between 1 and 251. The Unenhanced ATT bearer is always available for a connected peer device and has the *BearerId 0*.

4.13.1 EATT Credits management

Credits for the L2CAP channels used by Enhanced ATT bearers may be managed internally if the `autoCreditsMgmt` parameter is set to TRUE in the **`Gap_EattConnectionRequest`** or **`Gap_EattConnectionAccept`** function call. Otherwise, the application is responsible for credits management.

If the application chooses to manage the credits of the L2CAP channels used as Enhanced ATT bearers, it should use the following function to send credits for a specified bearer to a peer device:

```
bleResult_t Gap_EattSendCredits
(
    deviceId_t  deviceId,
```

```
    bearerId_t  bearerId,
    uint16_t    credits
);
```

If the local credits or peer credits of the L2CAP channel used by an Enhanced ATT bearer reaches `0`, a `gConnEvtEattBearerStatusNotification_c` connection event is updated with a status value of `gEnhancedBearerSuspendedNoLocalCredits_c`, or `gEnhancedBearerNoPeerCredits_c` respectively.

4.13.2 EATT Connection establishment

In order to take advantage of the Enhanced ATT features, first a number of Enhanced Bearers should be opened for a connected peer device. For this, the function below may be used to create up to five Enhanced ATT bearers at a time:

```
bleResult_t Gap_EattConnectionRequest
(
   deviceId_t  deviceId,
   uint16_t    mtu,
   uint8_t     cBearers,
   uint16_t    initialCredits,
   bool_t      autoCreditsMgmt
);
```

The `mtu` parameter specifies the MTU for all the bearers to be established.

The `cBearers` parameter is used to specify the number of Enhanced ATT bearers to be opened, and should have a value between 1 and 5. The `initialCredits` parameter specifies the initial number of credits of the L2CAP credit based channels used as Enhanced ATT bearers.

The `autoCreditsMgmt` parameter is used to tell the Bluetooth LE Host Stack if it should manage L2CAP channel credits automatically. If set to TRUE the Bluetooth LE Host Stack automatically sends credits to a peer device when exhausted in chunks of `initialCredits`.

For example, to establish two Enhanced ATT bearers with a peer device the application may call the `Gap_EattConnectionRequest` as shown below:

```
bleResult_t result = Gap_EattConnectionRequest(peerDeviceId,
                        64U,
                         2U,
                         3U,
                         TRUE);
if (gBleSuccess_c != result)
{
    /* Treat error */
}
```

If an EATT Connection Request is received from a peer device it would be signaled through the `gConnEvtEattConnectionRequest_c` connection event of type `gapEattConnectionRequest_t` sent to the connection callback. The application should handle this event by calling `Gap_EattConnectionAccept`. The example below shows how an application may accept an incoming EATT Connection Request with the same MTU as requested by the peer device.

```
case gConnEvtEattConnectionRequest_c:
{
    gapEattConnectionRequest_t *pEattConnectionReq = &pConnectionEvent-
>eventData.eattConnectionRequest;

    bleResult_t result = Gap_EattConnectionAccept(peerDeviceId,
                                                  TRUE,
                                                  pEattConnectionReq->mtu,
```

```
                                                3U,
                                                TRUE);

    if (gBleSuccess_c != result)
    {
        /* Treat error */
    }
}
break;
```

In case the `localMtu` specified when accepting a connection differs from the MTU requested by the peer device, the minimum of the two would become the MTU of the Enhanced Bearers.

After the **`Gap_EattConnectionRequest`** or **`Gap_EattConnectionAccept`** is called, for the result the application should wait for the `gConnEvtEattConnectionComplete_c` connection event of type `gapEattConnectionComplete_t` shown below:

```
typedef struct {
    l2caLeCbConnectionRequestResult_t           status;
    uint16_t                                    mtu;
    uint8_t                                     cBearers;
    bearerId_t                                  aBearerIds[gGapEattMaxBearers];
} gapEattConnectionComplete_t;
```

If successful, the `aBearerIds` array contains the bearer ids, for the Enhanced ATT bearers established. These ids may be used with the GATT Enhanced APIs in order to trigger GATT procedures over Enhanced ATT bearers.

4.13.3 EATT Bearer reconfiguration

One of the advantages of Enhanced ATT bearers over the Unenhanced ATT bearer is the ability to increase the MTU multiple times. To reconfigure the MTU and/or MPS of existing Enhanced ATT bearers, the **`Gap_EattReconfigureRequest`** should be used. If a `mps` value of `0` is given, the maximum available MPS value for that channel is used.

For example, in order to reconfigure the MTU of two bearers from 64 to 128 the application may call **`Gap_EattReconfigureRequest`** as shown below:

```
bleResult_t result = gBleSuccess_c;
bearerId_t aBearerIds[2] = {1U, 2U};
result = Gap_EattReconfigureRequest(peerDeviceId,
                                    128U,
                                    0U,
                                    2U,
                                    aBearerIds);
if (gBleSuccess_c != result)
{
    /* Treat error */
}
```

The application should monitor the `gConnEvtEattChannelReconfigureResponse_c` connection event of type `gapEattReconfigureResponse_t` for the result.

The procedure triggered by **`Gap_EattReconfigureRequest`** updates only the local MTU. The `ATT_MTU` for Enhanced ATT bearers is the minimum of the MTU values of the two devices.

4.13.4 EATT Bearer disconnection

Individual Enhanced ATT bearers can be disconnected by calling the `Gap_EattDisconnect` API as shown below:

```
bleResult_t result = gBleSuccess_c;
result = Gap_EattDisconnect(peerDeviceId, bearerId);
if (gBleSuccess_c != result)
{
 /* Treat error */
}
```

The application should look for a connection event of type `gEnhancedBearerDisconnected_c` in the connection callback.

5 Generic Attribute Profile (GATT) Layer

The GATT layer contains the APIs for discovering services and characteristics and transferring data between devices and is built on top of the Attribute Protocol (ATT).

The Attribute Protocol (ATT) transfers data between Bluetooth Low Energy devices on a dedicated L2CAP channel (channel ID 0x04).

As soon as a connection is established between devices, the GATT APIs are readily available. No initialization is required because the L2CAP channel is automatically created.

To identify the GATT peer instance, the same *deviceId* value from the GAP layer (obtained in the *gConnEvtConnected_c*connection event) is used.

There are two GATT roles that define the two devices exchanging data over ATT:

- GATT Server – the device that contains a GATT Database, which is a collection of services and characteristics exposing meaningful data. Usually, the Server responds to *requests* and *commands* sent by the Client. However, it can be configured to send data on its own through *notifications* and *indications*.
- GATT Client – the "active" device that usually sends *requests* and *commands* to the Server to *discover* Services and Characteristics on the Server's Database and to exchange data.

There is no fixed rule deciding which device is the Client and which one is the Server. Any device may initiate a request at any moment. Therefore, it temporarily acts as a Client, at which the peer device may respond, provided it has the Server support and a GATT Database.

Often, a GAP Central acts as a GATT Client to discover Services and Characteristics and obtain data from the GAP Peripheral, which usually has a GATT database. Many standard Bluetooth Low Energy profiles assume that the Peripheral has a database and must act as a Server. However, this is by no means a general rule.

5.1 Client APIs

A Client can configure the ATT MTU, discover Services and Characteristics, and initiate data exchanges.

All the functions have the same first parameter: a *deviceId* which identifies the connected device whose GATT Server is targeted in the GATT procedure. This is necessary because a Client may be connected to multiple Servers at the same time.

First, however, the application must install the necessary callbacks.

5.1.1 Installing client callbacks

There are three callbacks that the Client application must install.

5.1.1.1 Client procedure callback

All the procedures initiated by a Client are asynchronous. They rely on exchanging ATT packets over the air.

To be informed of the procedure completion, the application must install a callback with the following signature:

```
typedef void (* gattClientProcedureCallback_t )
(
    deviceId_t              deviceId,
    gattProcedureType_t     procedureType,
    gattProcedureResult_t   procedureResult,
    bleResult_t             error
);
```

For EATT, the following signature should be used:

```
typedef void (*gattClientEnhancedProcedureCallback_t)
(
   deviceId_t deviceId,
   bearerId_t bearerId,
   gattProcedureType_t procedureType,
   gattProcedureResult_t procedureResult,
   bleResult_t error
);
```

To install this callback, the following function must be called:

```
bleResult_t GattClient_RegisterProcedureCallback
(
    gattClientProcedureCallback_t callback
);
```

The EATT procedure callback should be installed using the following API:

```
bleResult_t GattClient_RegisterEnhancedProcedureCallback
(
 gattClientEnhancedProcedureCallback_t callback
);
```

The *procedureType* parameter can be used to identify the procedure that was started and has reached completion. Only one procedure would be active at a given moment. Trying to start another procedure while a procedure is already in progress returns the error *gGattAnotherProcedureInProgress_c*.

The *procedureResult* parameter indicates whether the procedure completes successfully or an error occurs. In the latter case, the *error* parameter contains the error code.

```
void gatt ClientProcedureCallback
(
    deviceId_t              deviceId,
    gattProcedureType_t     procedureType,
    gattProcedureResult_t   procedureResult,
    bleResult_t             error
)
{
    switch (procedureType)
    {
        /* ... */
    }
}
GattClient_RegisterProcedureCallback(gattClientProcedureCallback);
```

5.1.1.2 Notification and indication callbacks

When the Client receives a notification from the Server, it triggers a callback with the following prototype:

```
typedef void (* gattClientNotificationCallback_t )
(
    deviceId_t     deviceId,
    uint16_t       characteristicValueHandle,
    uint8_t *      aValue,
    uint16_t       valueLength
```

```
);
```

The *deviceId* identifies the Server connection (for multiple connections at the same time). The *characteristicValueHandle* is the attribute handle of the Characteristic Value declaration in the GATT Database. The Client must have discovered it previously to be able recognize it.

For EATT, the following signature should be used:

```
typedef void (*gattClientEnhancedNotificationCallback_t)
 ( deviceId_t deviceId,
   bearerId_t bearerId,
   uint16_t characteristicValueHandle,
   uint8_t* aValue,
   uint16_t valueLength );
```

The callback must be installed with:

```
bleResult_t GattClient_RegisterNotificationCallback
(
    gattClientNotificationCallback_t callback
);
```

Very similar definitions exist for indications.

The EATT notification callback should be installed using the following API:

```
bleResult_t GattClient_RegisterEnhancedNotificationCallback
(
 gattClientEnhancedNotificationCallback_t callback
)
```

When receiving a notification or indication, the Client uses the *characteristicValueHandle* to identify which Characteristic was notified. The Client must be aware of the possible Characteristic Value handles that can be notified/indicated at any time, because it has previously activated them by writing its CCCD (see Section 5.1.5 "Reading and writing characteristic descriptors").

When the Client receives a multiple value notification from the Server, it triggers a callback with the following prototype:

```
typedef void (*gattClientMultipleValueNotificationCallback_t)
(
    deviceId_t     deviceId,
    /*!< Device ID identifying the active connection. */
    uint8_t* aHandleLenValue,
    /*!< The array of handle, value length, value tuples. */
    uint16_t totalLength
    /*!< Value array size. */
);
```

The callback must be installed with:

```
bleResult_t GattClient_RegisterMultipleValueNotificationCallback
(
    gattClientMultipleValueNotificationCallback_t callback
);
```

When using EATT, the following callback prototype and registration APIs should be used:

```
typedef void (*gattClientEnhancedMultipleValueNotificationCallback_t)
( deviceId_t deviceId,
    /*!< Device ID identifying the active connection. */
  bearerId_t bearerId,
   /*!< Bearer ID identifing the Enhanced ATT bearer used. */
 uint8_t* aHandleLenValue,
   /*!< The array of handle, value length, value tuples. */
uint16_t totalLength     /*!< Value array size. */
 );
```

5.1.2 MTU exchange

A radio packet sent over the Bluetooth Low Energy contains a maximum of 27 bytes of data for the L2CAP layer. The L2CAP header is 4 bytes long, including the Channel ID. Therefore, all layers above L2CAP, including ATT and GATT, can only send 23 bytes of data in a radio packet (as per *Bluetooth 4.1 Specification for Bluetooth Low Energy*). This specification also sets the default length of an ATT packet (also called *ATT_MTU*) to 23. The ATT packet length is set to this value to maintain a logical mapping between radio packets and ATT packets.

Note: *This number is fixed and cannot be increased in Bluetooth Low Energy 4.1.*

Therefore, any ATT request fits in a single radio packet. If the layer above ATT wishes to send more than 23 bytes of data, the data must be fragmented into smaller packets and multiple ATT requests issued.

Despite this setting, the ATT protocol allows devices to increase the ATT_MTU, only if both can support it. Increasing the ATT_MTU has only one effect: the application does not have to fragment long data. However, it can send more than 23 bytes in a single transaction. The fragmentation is moved on to the L2CAP layer. Over the air though, there would still be more than one radio packet sent.

If the GATT Client supports a larger than default MTU, it must start an MTU exchange as soon as it connects to any server. During the MTU exchange, both devices would send their maximum MTU to the other, and the minimum of the two is chosen as the new MTU.

Consider an example where the Client supports a maximum ATT_MTU of 250, and the server supports a maximum value of 120 for the same attribute. For this case, after MTU exchange, both devices must set the new ATT_MTU value equal to 120.

To initiate the MTU exchange, call the following function from *gatt_client_interface.h*:

```
bleResult_t result = GattClient_ExchangeMtu(deviceId, mtu);
if (gBleSuccess_c != result)
{
    /* Treat error */
}
```

When having the role of a GATT Client, the value of the maximum supported *ATT_MTU* of the local device is given as a parameter to the `GattClient_ExchangeMtu` API. On the GATT Server side, the application configures this value via the `gcGattServerMtu_c` variable that exists in the file `ble_globals.c`. The minimum of these two values is chosen as the new MTU for the connection.

When the exchange is complete, the *gGattProcExchangeMtu_c* procedure type triggers the Client callback.

```
void gattClientProcedureCallback
(
    deviceId_t deviceId,
    gattProcedureType_t procedureType,
```

```
    gattProcedureResult_t procedureResult,
    bleResult_t error
)
{
    switch (procedureType)
    {
        /* ... */
        casegGattProcExchangeMtu_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                /* To obtain the new MTU */
                uint16_t newMtu;
                bleResult_t result = Gatt_GetMtu(deviceId, &newMtu);
                if (gBleSuccess_c == result)
                {
                    /* Use the value of the new MTU */
                    (void) newMtu;
                }
            }
            else
            {
                /* Handle error */
            }
            break;
        /* ... */
    }
}
```

Note: *The Exchange MTU procedure is only available for the unenhanced/legacy bearer. For procedures sent on enhanced bearers, the upper layer must use provided* ***L2CAP APIs*** *to create dedicated L2CAP channels. Each channel has its own MTU value specified upon creation, which can also be reconfigured later.*

5.1.3 Service and characteristic discovery

There are multiple APIs that can be used for Discovery. The application may use any of them, according to its necessities.

All of the following APIs have an enhanced counterpart of the form *GattClient_Enhanced[procedure]*. A *bearerId* parameter was added to specify on which bearer the transaction should take place. A value of *0* for the bearer Id identifies the Unenhanced ATT bearer. Values higher than *0* are used to identify the Enhanced ATT bearer used for the ATT procedure.

5.1.3.1 Discover all primary services

The following API can be used to discover all the Primary Services in a Server's database:

```
bleResult_t GattClient_DiscoverAllPrimaryServices
(
    deviceId_t          deviceId,
    gattService_t *     aOutPrimaryServices,
    uint8_t             maxServiceCount,
    uint8_t *           pOutDiscoveredCount
);
```

The *aOutPrimaryServices* parameter must point to an allocated array of services. The size of the array must be equal to the value of the *maxServiceCount* parameter, which is passed to make sure the GATT module does not attempt to write past the end of the array if more Services are discovered than expected.

The *pOutDiscoveredCount* parameter must point to a static variable because the GATT module uses it to write the number of Services discovered at the end of the procedure. This number is less than or equal to the *maxServiceCount*.

If there is equality, it is possible that the Server contains more than *maxServiceCount* Services, but they could not be discovered as a result of the array size limitation. It is the application developer's responsibility to allocate a large enough number according to the expected contents of the Server's database.

In the following example, the application expects to find no more than 10 Services on the Server.

```
#define mcMaxPrimaryServices_c 10
static gattService_t primaryServices[mcMaxPrimaryServices_c];
uint8_t mcPrimaryServices;
bleResult_t result = GattClient_DiscoverAllPrimaryServices
(
    deviceId,
    primaryServices,
    mcMaxPrimaryServices_c,
    &mcPrimaryServices
);
if (gBleSuccess_c != result)
{
    /* Treat error */
}
```

The operation triggers the Client Procedure Callback when complete. The application may read the number of discovered services and each service's handle range and UUID.

```
void gattClientProcedureCallback
(
    deviceId_t              deviceId,
    gattProcedureType_t     procedureType,
    gattProcedureResult_t   procedureResult,
    bleResult_t             error
)
{
    switch (procedureType)
    {
        /* ... */
        case gGattProcDiscoverAllPrimaryServices_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                /* Read number of discovered services */
                PRINT( mcPrimaryServices );
                /* Read each service's handle range and UUID */
                for (int j = 0; j < mcPrimaryServices; j++)
                {
                    PRINT( primaryServices[j]. startHandle );
                    PRINT( primaryServices[j]. endHandle );
                    PRINT( primaryServices[j]. uuidType );
                    PRINT( primaryServices[j]. uuid );
                }
            }
            else
            {
                /* Handle error */
                PRINT( error );
            }
            break;
```

```
        /* ... */
    }
}
```

5.1.3.2 Discover primary services by UUID

To discover only Primary Services of a known type (Service UUID), the following API can be used:

```
bleResult_t GattClient_DiscoverPrimaryServicesByUuid
(
    deviceId_t          deviceId,
    bleUuidType_t       uuidType,
    const bleUuid_t *   pUuid,
    gattService_t *     aOutPrimaryServices,
    uint8_t             maxServiceCount,
    uint8_t *           pOutDiscoveredCount
);
```

The procedure is very similar to the one described in Section 5.1.3.1 "Discover all primary services". The only difference is this time we are filtering the search according to a Service UUID described by two extra parameters: *pUuid* and *uuidType*.

This procedure is useful when the Client is only interested in a specific type of Services. Usually, it is performed on Servers that are known to contain a certain Service, which is specific to a certain profile. Therefore, most of the times the search is expected to find a single Service of the given type. As a result, only one structure is usually allocated.

For example, when two devices implement the Heart Rate (HR) Profile, an HR Collector connects to an HR Sensor and may only be interested in discovering the Heart Rate Service (HRS) to work with its Characteristics. The following code example shows how to achieve this. Standard values for Service and Characteristic UUIDs, as defined by the Bluetooth SIG, are located in the *ble_sig_defines.h* file.

```
static gattService_t heartRateService;
static uint8_t mcHrs;
bleResult_t result = GattClient_DiscoverPrimaryServicesByUuid
(
    deviceId,
    gBleUuidType16_c,             /* Service UUID type */
    gBleSig_HeartRateService_d,   /* Service UUID */
    &heartRateService,            /* Only one HRS is expected to be found */
    1,
    &mcHrs
   /* Will be equal to 1 at the end of the procedure
    if the HRS is found, 0 otherwise */

);
if (gBleSuccess_c != result)
{
    /* Treat error */
}
```

In the Client Procedure Callback, the application should check if any Service with the given UUID was found and read its handle range (also perhaps proceed with Characteristic Discovery within that service range).

```
void gattClientProcedureCallback
(
    deviceId_t deviceId,
```

```
    gattProcedureType_t procedureType,
    gattProcedureResult_t procedureResult,
    bleResult_t error
)
{
    switch (procedureType)
    {
        /* ... */
        case gGattProcDiscoverPrimaryServicesByUuid_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                if (1 == mcHrs)
                {
                    /* HRS found, read the handle range */
                    PRINT( heartRateService. startHandle );
                    PRINT( heartRateService. endHandle );
                }
                else
                {
                    /* HRS not found! */
                }
            }
            else
            {
                /* Handle error */
                PRINT( error );
            }
            break;
        /* ... */
    }
}
```

5.1.3.3 Discover included services

Section 5.1.3.1 "Discover all primary services" shows how to discover Primary Services. However, a Server may also contain Secondary Services, which are not meant to be used standalone and are usually included in the Primary Services. The inclusion means that all the Secondary Service's Characteristics may be used by the profile that requires the Primary Service.

Therefore, after a Primary Service has been discovered, the following procedure may be used to discover services (usually Secondary Services) included in it:

```
bleResult_t GattClient_FindIncludedServices
(
    deviceId_t          deviceId,
    gattService_t *     pIoService,
    uint8_t             maxServiceCount
);
```

The service structure that *pIoService* points to must have the *aIncludedServices* field linked to an allocated array of services, of size *maxServiceCount*, chosen according to the expected number of included services to be found. This is the application's choice, usually following profile specifications.

Also, the service's range must be set (the *startHandle* and *endHandle* fields), which may have already been done by the previous Service Discovery procedure (as described in Section 5.1.3.1 "Discover all primary services" and Section 5.1.3.2 "Discover primary services by UUID").

The number of discovered included services is written by the GATT module in the *cNumIncludedServices* field of the structure from *ploService*. Obviously, a maximum of *maxServiceCount* included services is discovered.

The following example assumes the Heart Rate Service was discovered using the code provided in Section 5.1.3.2 "Discover primary services by UUID".

```
/* Finding services included in the Heart Rate Primary Service */
gattService_t * pPrimaryService = &heartRateService;
#define mxMaxIncludedServices_c 3
static gattService_t includedServices[mxMaxIncludedServices_c];
/* Linking the array */
pPrimaryService-> aIncludedServices = includedServices;
bleResult_t result = GattClient_FindIncludedServices
(
    deviceId,
    pPrimaryService,
    mxMaxIncludedServices_c
);
if (gBleSuccess_c != result)
{
    /* Treat error */
}
```

When the Client Procedure Callback is triggered, if any included services are found, the application can read their handle range and their UUIDs.

```
void gattClientProcedureCallback
(
    deviceId_t deviceId,
    gattProcedureType_t procedureType,
    gattProcedureResult_t procedureResult,
    bleResult_t error
)
{
    switch (procedureType)
    {
        /* ... */
        case gGattProcFindIncludedServices_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                /* Read included services data */
                PRINT( pPrimaryService-> cNumIncludedServices );
                for (int j = 0; j < pPrimaryService-> cNumIncludedServices ; j+
+)
                {
                    PRINT( pPrimaryService-> aIncludedServices [j].
 startHandle );
                    PRINT( pPrimaryService-> aIncludedServices [j]. endHandle );
                    PRINT( pPrimaryService-> aIncludedServices [j]. uuidType );
                    PRINT( pPrimaryService-> aIncludedServices [j]. uuid );
                }
            }
            else
            {
                /* Handle error */
                PRINT( error );
            }
            break;
        /* ... */
```

```
    }
}
```

5.1.3.4 Discover all characteristics of a service

The main API for Characteristic Discovery has the following prototype:

```
bleResult_t GattClient_DiscoverAllCharacteristicsOfService
(
    deviceId_t          deviceId,
    gattService_t *     pIoService,
    uint8_t             maxCharacteristicCount
);
```

All required information is contained in the service structure pointed to by *pIoService*, most importantly being the service range (*startHandle* and *endHandle*) which is usually already filled out by a Service Discovery procedure. If not, they need to be written manually.

Also, the service structure's *aCharacteristics* field must be linked to an allocated characteristic array.

The following example discovers all Characteristics contained in the Heart Rate Service discovered in Section Section 5.1.3.2 "Discover primary services by UUID".

```
gattService_t* pService = &heartRateService
#define mcMaxCharacteristics_c 10
static gattCharacteristic_t hrsCharacteristics[mcMaxCharacteristics_c];
pService->aCharacteristics = hrsCharacteristics;
bleResult_t result = GattClient_DiscoverAllCharacteristicsOfService
(
    deviceId,
    pService,
    mcMaxCharacteristics_c
);
```

The Client Procedure Callback is triggered when the procedure completes.

```
void gattClientProcedureCallback
(
    deviceId_t                  deviceId,
    gattProcedureType_t         procedureType,
    gattProcedureResult_t       procedureResult,
    bleResult_t                 error
)
{
    switch (procedureType)
    {
      /* ... */
      casegGattProcDiscoverAllCharacteristics_c:
          if (gGattProcSuccess_c == procedureResult)
          {
            /* Read number of discovered Characteristics */
              PRINT(pService-> cNumCharacteristics );
            /* Read discovered Characteristics data */
              for ( uint8_t j = 0; j < pService-> cNumCharacteristics ; j++)
                {
            /* Characteristic UUID is found inside the value field
               to avoid duplication */
                    PRINT(pService-> aCharacteristics [j]. value . uuidType );
                    PRINT(pService-> aCharacteristics [j]. value . uuid );
            /* Characteristic Properties indicating the supported operations:
```

```
                    * - Read
                    * - Write
                    * - Write Without Response
                    * - Notify
                    * - Indicate
            */
                    PRINT(pService-> aCharacteristics [j]. properties );
           /* Characteristic Value Handle is used to identify the
              Characteristic in future operations */
                    PRINT(pService-> aCharacteristics [j]. value . handle );
                }
            }
            else
            {
                /* Handle error */
                PRINT( error );
            }
            break;
        /* ... */
    }
}
```

5.1.3.5 Discover characteristics by UUID

This procedure is useful when the Client intends to discover a specific Characteristic in a specific Service. The API allows for multiple Characteristics of the same type to be discovered, but most often it is used when a single Characteristic of the given type is expected to be found.

Continuing the example from Section 5.1.3.2 "Discover primary services by UUID", assume the Client wants to discover the Heart Rate Control Point Characteristic inside the Heart Rate Service, as shown in the following code.

```
gattService_t * pService = &heartRateService;
static gattCharacteristic_t hrcpCharacteristic;
static uint8_t mcHrcpChar;
bleResult_t result = GattClient_DiscoverCharacteristicOfServiceByUuid
(
    deviceId,
    gBleUuidType16_c,
    gBleSig_HrControlPoint_d,
    pService,
    &hrcpCharacteristic,
    1,
    &mcHrcpChar
);
```

This API can be used as in the previous examples, following a Service Discovery procedure. However, the user may want to perform a Characteristic search with UUID over the entire database, skipping the Service Discovery entirely. To do so, a dummy service structure must be defined and its range must be set to maximum, as shown in the following example:

```
gatt Service_t dummyService;
dummyService. startHandle = 0x0001;
dummyService. endHandle = 0xFFFF;
static gattCharacteristic_t hrcpCharacteristic;
static uint8_t mcHrcpChar;
bleResult_t result = GattClient_DiscoverCharacteristicOfServiceByUuid
(
    deviceId,
    gBleUuidType16_c,
```

```
    gBleSig_HrControlPoint_d,
    &dummyService,
    &hrcpCharacteristic,
    1,
    &mcHrcpChar
);
```

In either case, the value of the *mcHrcpChar* variable should be checked in the procedure callback.

```
void gattClientProcedureCallback
(
    deviceId_t              deviceId,
    gattProcedureType_t     procedureType,
    gattProcedureResult_t   procedureResult,
    bleResult_t             error
)
{
    switch (procedureType)
    {
        /* ... */
        casegGattProcDiscoverCharacteristicByUuid_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                if (1 == mcHrcpChar)
                {
                    /* HRCP found, read discovered data */
                    PRINT(hrcpCharacteristic. properties );
                    PRINT(hrcpCharacteristic. value . handle );
                }
                else
                {
                    /* HRCP not found! */
                }
            }
            else
            {
                /* Handle error */
                PRINT(error);
            }
            break;
        /* ... */
    }
}
```

5.1.3.6 Discover characteristic descriptors

To discover all descriptors of a Characteristic, the following API is provided:

```
bleResult_t GattClient_DiscoverAllCharacteristicDescriptors
(
    deviceId_t                  deviceId,
    gattCharacteristic_t *      pIoCharacteristic,
    uint16_t                    endingHandle,
    uint8_t                     maxDescriptorCount
);
```

The *ploCharacteristic* pointer must point to a Characteristic structure with the *value.handle* field set (either by a discovery operation or by the application) and the *aDescriptors* field pointed to an allocated array of Descriptor structures.

The *endingHandle* should be set to the handle of the next Characteristic or Service declaration in the database to indicate when the search for descriptors must stop. The GATT Client module uses ATT Find Information Requests to discover the descriptors, and it does so until it discovers a Characteristic or Service declaration or until *endingHandle* is reached. Thus, by providing a correct ending handle, the search for descriptors is optimized and the number of packets sent over the air is reduced.

If, however, the application does not know where the next declaration lies and cannot provide this optimization hint, the *endingHandle* should be set to *0xFFFF*.

Continuing the example from Section 5.1.3.5 "Discover characteristics by UUID", the following code assumes that the Heart Rate Control Point Characteristic has no more than 5 descriptors and performs Descriptor Discovery.

```
#define mcMaxDescriptors_c 5
static gattAttribute_t aDescriptors[mcMaxDescriptors_c];
hrcpCharacteristic. aDescriptors = aDescriptors;
bleResult_t result = GattClient_DiscoverAllCharacteristicDescriptors
(
    deviceId,
    &hrcpCharacteristic,
    0xFFFF, /* We do not know where the next Characterstic Service begins */
    mcMaxDescriptors_c
);
if (gBleSuccess_c != result)
{
    /* Handle error */
}
```

The Client Procedure Callback is triggered at the end of the procedure.

```
void gattClientProcedureCallback
(
    deviceId_t              deviceId,
    gattProcedureType_t     procedureType,
    gattProcedureResult_t   procedureResult,
    bleResult_t             error
)
{
    switch (procedureType)
    {
        /* ... */
        casegGattProcDiscoverAllCharacteristicDescriptors_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                /* Read number of discovered descriptors */
                PRINT(hrcpCharacteristic. cNumDescriptors );
                /* Read descriptor data */
                for ( uint8_t j = 0; j < hrcpCharacteristic. cNumDescriptors ; j
++)
                {
                    PRINT(hrcpCharacteristic. aDescriptors [j]. handle );
                    PRINT(hrcpCharacteristic. aDescriptors [j]. uuidType );
                    PRINT(hrcpCharacteristic. aDescriptors [j]. uuid );
                }
            }
```

```
            else
            {
                /* Handle error */
                PRINT(error);
            }
            break;
        /* ... */
    }
}
```

5.1.4 Reading and writing characteristics

All the APIs described in the following sections have an enhanced counterpart of the form *GattClient_Enhanced[procedure]*. A *bearer id* parameter was added to specify on which bearer the transaction should take place. A value of *0* for the bearer id identifies the Unenhanced ATT bearer. Values higher than *0* are used to identify the Enhanced ATT bearer used for the ATT procedure.

5.1.4.1 Characteristic value read procedure

The main API for reading a Characteristic Value is shown here:

```
bleResult_t GattClient_ReadCharacteristicValue
(
    deviceId_t                  deviceId,
    gattCharacteristic_t *      pIoCharacteristic,
    uint16_t                    maxReadBytes
);
```

This procedure assumes that the application knows the Characteristic Value Handle, usually from a previous Characteristic Discovery procedure. Therefore, the *value.handle* field of the structure pointed by *pIoCharacteristic* must be completed.

Also, the application must allocate a large enough array of bytes where the received value (from the ATT packet exchange) is written. The *maxReadBytes* parameter is set to the size of this allocated array.

The GATT Client module takes care of long characteristics, whose values have a greater length than can fit in a single ATT packet, by issuing repeated ATT Read Blob Requests when needed.

The following examples assume that the application knows the Characteristic Value Handle and that the value length is variable, but limited to 50 bytes.

```
gattCharacteristic_t myCharacteristic;
myCharacteristic. value . handle = 0x10AB;
#define mcMaxValueLength_c 50
static uint8_t aValue[mcMaxValueLength_c];
myCharacteristic. value . paValue = aValue;
bleResult_t result = GattClient_ReadCharacteristicValue
(
    deviceId,
    &myCharacteristic,
    mcMaxValueLength_c
);
if (gBleSuccess_c != result)
{
    /* Handle error */
}
```

Regardless of the value length, the Client Procedure Callback is triggered when the reading is complete. The received value length is also filled in the *value* structure.

```
void gattClientProcedureCallback
(
    deviceId_t              deviceId,
    gattProcedureType_t     procedureType,
    gattProcedureResult_t   procedureResult,
    bleResult_t             error
)
{
    switch (procedureType)
    {
        /* ... */
        casegGattProcReadCharacteristicValue_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                /* Read value length */
                PRINT(myCharacteristic. value . valueLength );
                /* Read data */
                for ( uint16_t j = 0; j < myCharacteristic. value .
 valueLength ; j++)
                {
                    PRINT(myCharacteristic. value . paValue [j]);
                }
            }
            else
            {
                /* Handle error */
                PRINT(error);
            }
            break;
        /* ... */
    }
}
```

5.1.4.2 Characteristic read by UUID procedure

This API for this procedure is shown here:

```
bleResult_t GattClient_ReadUsingCharacteristicUuid
(
    deviceId_t                deviceId,
    bleUuidType_t             uuidType,
    const bleUuid_t*          pUuid,
    const gattHandleRange_t* pHandleRange,
    uint8_t*                  aOutBuffer,
    uint16_t                  maxReadBytes,
    uint16_t*                 pOutActualReadBytes
);
```

This provides support for an important optimization, which involves reading a Characteristic Value without performing any Service or Characteristic Discovery.

For example, the following is the process to write an application that connects to any Server and wants to read the device name.

The device name is contained in the Device Name Characteristic from the GAP Service. Therefore, the necessary steps involve discovering all primary services, identifying the GAP Service by its UUID, discovering all Characteristics of the GAP Service and identifying the Device Name Characteristic (alternatively, discovering Characteristic by UUID inside GAP Service), and, finally, reading the device name by using the Characteristic Read Procedure.

Instead, the Characteristic Read by UUID Procedure allows reading a Characteristic with a specified UUID, assuming one exists on the Server, without knowing the Characteristic Value Handle.

The described example is implemented as follows:

```
#define mcMaxValueLength_c
/* First byte is for handle-value pair length. Next 2 bytes are the handle */
static uint8_t aValue[1 + 2 + mcMaxValueLength_c];
static uint16_t deviceNameLength;
bleUuid_t uuid = {
        .uuid16 = gBleSig_GapDeviceName_d
};
bleResult_t result = GattClient_ReadUsingCharacteristicUuid
(
    deviceId,
    gBleUuidType16_c,
    &uuid,
    &pHandleRange,
    aValue,
    1 + 2 + mcMaxValueLength_c,
    deviceNameLength
);
if (gBleSuccess_c != result)
{
    /* Handle error */
}
```

The Client Procedure Callback is triggered when the reading is complete. Because only one air packet is exchanged during this procedure, it can only be used as a quick reading of Characteristic Values with length no greater than *ATT_MTU – 1*.

```
void gattClientProcedureCallback
(
    deviceId_t                  deviceId,
    gattProcedureType_t         procedureType,
    gattProcedureResult_t       procedureResult,
    bleResult_t                 error
)
{
    switch (procedureType)
    {
        /* ... */
        case gGattProcReadUsingCharacteristicUuid_c:
            if (gGattProcSuccess_c == procedureResult)
            {
               /* Read handle-value pair length */
                PRINT(aValue[0]);
                deviceNameLength -= 1;
              /* Read characteristic value handle */
                PRINT(aValue[1] | (aValue[2] << 8));
                deviceNameLength -= 2;
                /* Read value length */
                PRINT(deviceNameLength);
```

```
                /* Read data */
                for ( uint8_t j = 0; j < deviceNameLength; j++)
                {
                    PRINT(aValue[3 + j]);
                }
            }
            else
            {
                /* Handle error */
                PRINT(error);
            }
            break;
        /* ... */
    }
}
```

5.1.4.3 Characteristic read multiple procedure

The API for this procedure is shown here:

```
bleResult_t GattClient_ReadMultipleCharacteristicValues
(
    deviceId_t                  deviceId,
    uint8_t                     cNumCharacteristics,
    gattCharacteristic_t *      aIoCharacteristics
);
```

This procedure also allows an optimization for a specific situation, which occurs when multiple Characteristics, whose values are of known, fixed-length, can be all read in one single ATT transaction (usually one single over-the-air packet).

The application must know the value handle and value length of each Characteristic. It must also write the *value.handle* and *value.maxValueLength* with the aforementioned values, respectively, and then link the *value.paValue* field with an allocated array of size *maxValueLength*.

The following example involves reading three characteristics in a single packet.

```
#define mcNumCharacteristics_c 3
#define mcChar1Length_c 4
#define mcChar2Length_c 5
#define mcChar3Length_c 6
static uint8_t aValue1[mcChar1Length_c];
static uint8_t aValue2[mcChar2Length_c];
static uint8_t aValue3[mcChar3Length_c];
static gattCharacteristic_t myChars[mcNumCharacteristics_c];
myChars[0]. value . handle = 0x0015;
myChars[1]. value . handle = 0x0025;
myChars[2]. value . handle = 0x0035;
myChars[0]. value . maxValueLength = mcChar1Length_c;
myChars[1]. value . maxValueLength = mcChar2Length_c;
myChars[2]. value . maxValueLength = mcChar3Length_c;
myChars[0]. value . paValue = aValue1;
myChars[1]. value . paValue = aValue2;
myChars[2]. value . paValue = aValue3;
bleResult_t result = GattClient_ReadMultipleCharacteristicValues
(
    deviceId,
    mcNumCharacteristics_c,
```

```
    myChars
);
if (gBleSuccess_c != result)
{
    /* Handle error */
}
```

When the Client Procedure Callback is triggered, if no error occurs, each Characteristic's value length should be equal to the requested lengths.

```
void gattClientProcedureCallback
(
    deviceId_t              deviceId,
    gattProcedureType_t p   rocedureType,
    gattProcedureResult_t   procedureResult,
    bleResult_t             error
)
{
    switch (procedureType)
    {
        /* ... */
        casegGattProcReadMultipleCharacteristicValues_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                for ( uint8_t i = 0; i < mcNumCharacteristics_c; i++)
                {
                    /* Read value length */
                    PRINT(myChars[i]. value . valueLength );
                    /* Read data */
                    for ( uint8_t j = 0; j < myChars[i]. value . valueLength ; j
++)
                    {
                        PRINT(myChars[i]. value . paValue [j]);
                    }
                }
            }
            else
            {
                /* Handle error */
                PRINT(error);
            }
            break;
        /* ... */
    }
}
```

If the server does not know the length of the characteristic values, then the Read Multiple Variable Characteristic Values procedure can be used. This sub-procedure is used to read multiple characteristic values of variable length from a server when the client knows the characteristic value handles. The response returns the characteristic values and their corresponding lengths in the Length Value Tuple List parameter.

```
bleResult_t GattClient_ReadMultipleVariableCharacteristicValues
(
  deviceId_t             deviceId,
  uint8_t                cNumCharacteristics,
  gattCharacteristic_t*  pIoCharacteristics
);
```

The following example involves reading three characteristics of variable length in a single packet.

```
#define mcNumCharacteristics_c 3
#define mcCharLengthMax_c 10
static uint8_t aValue1[mcCharLengthMax _c];
static uint8_t aValue2[mcCharLengthMax _c];
static uint8_t aValue3[mcCharLengthMax _c];
static gattCharacteristic_t myChars[mcNumCharacteristics_c];
myChars[0].value .handle = 0x0015;
myChars[1].value .handle = 0x0025;
myChars[2].value .handle = 0x0035;
myChars[0].value .paValue = aValue1;
myChars[1].value .paValue = aValue2;
myChars[2].value .paValue = aValue3;
bleResult_t result = GattClient_ReadMultipleVariableCharacteristicValues
(
    deviceId,
    mcNumCharacteristics_c,
    pIoCharacteristics
);
if (gBleSuccess_c != result)
{
    /* Handle error */
}
```

The result of this procedure is sent to the application via the GATT procedure callback. The response includes the characteristic value together with a handle, length pair corresponding to each characteristic.

```
static void BleApp_GattClientCallback
(
    deviceId_t              serverDeviceId,
    gattProcedureType_t     procedureType,
    gattProcedureResult_t   procedureResult,
    bleResult_t             error
)
{
    switch (procedureResult)
    {
        /* ... */
        case gGattProcReadMultipleVarLengthCharValues_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                for (uint8_t i = 0; i < mcNumCharacteristics_c; i++)
                {
                    /* Print characteristic handle and length */
                    PRINT(myChars[i].value.handle);
                    PRINT(myChars[i].value.valueLength);
                    for (uint8_t j = 0; j < myChars[i].value.maxValueLength; j++)
                    {
                        /* Print characteristic value */
                        PRINT(myChars[i].value.paValue[j]);
                    }
                }
            }
            else
            {
                /* Handle error */
            }
            break;
}
```

5.1.4.4 Characteristic write procedure

There is a general API that may be used for writing Characteristic Values:

```
bleResult_t GattClient_WriteCharacteristicValue
(
    deviceId_t                          deviceId,
    const gattCharacteristic_t *        pCharacteristic,
    uint16_t                            valueLength,
    const uint8_t *                     aValue,
    bool_t                              withoutResponse,
    bool_t                              signedWrite,
    bool_t                              doReliableLongCharWrites,
    const uint8_t *                     aCsrk
);
```

It has many parameters to support different combinations of Characteristic Write Procedures.

The structure pointed to by the *pCharacteristic* is only used for the *value.handle* field which indicates the Characteristic Value Handle. The value to be written is contained in the *aValue* array of size *valueLength*.

The *withoutResponse* parameter can be set to *TRUE* if the application wishes to perform a Write Without Response Procedure, which translates into an ATT Write Command. If this value is selected, the *signedWrite* parameter indicates whether data should be signed (Signed Write Procedure over ATT Signed Write Command), in which case the *aCsrk* parameters must not be NULL and contains the CSRK to sign the data with. Otherwise, both *signedWrite* and *aCsrk* are ignored.

Finally, *doReliableLongCharWrites* should be sent to *TRUE* if the application is writing a long Characteristic Value (one that requires multiple air packets due to *ATT_MTU* limitations) and wants the Server to confirm each part of the attribute that is sent over the air.

To simplify the application code, the following macros are defined:

```
#define GattClient_SimpleCharacteristicWrite(deviceId, pChar, valueLength,
 aValue) \
    GattClient_WriteCharacteristicValue\
        (deviceId, pChar, valueLength, aValue, FALSE, FALSE, FALSE, NULL)
```

This is the simplest usage for writing a Characteristic. It sends an ATT Write Request if the value length does not exceed the maximum space for an over-the-air packet (*ATT_MTU – 3*). Otherwise, it sends ATT Prepare Write Requests with parts of the attribute, without checking the ATT Prepare Write Response data for consistency, and in the end an ATT Execute Write Request.

```
#define GattClient_CharacteristicWriteWithoutResponse(deviceId, pChar,
 valueLength, aValue) \
    GattClient_WriteCharacteristicValue\
        (deviceId, pChar, valueLength, aValue, TRUE, FALSE, FALSE, NULL)
```

This usage sends an ATT Write Command. Long Characteristic values are not allowed here and trigger a *gBleInvalidParameter_c* error.

```
#define GattClient_CharacteristicSignedWrite(deviceId, pChar, valueLength,
 aValue, aCsrk) \
    GattClient_WriteCharacteristicValue\
        (deviceId, pChar, valueLength, aValue, TRUE, TRUE, FALSE, aCsrk)
```

This usage sends an ATT Signed Write Command. The CSRK used to sign data must be provided.

This is a short example to write a 3-byte long Characteristic Value.

```
gattCharacteristic_t myChar;
myChar. value . handle = 0x00A0; /* Or maybe it was previously discovered? */
#define mcValueLength_c 3
uint8_t aValue[mcValueLength_c] = { 0x01, 0x02, 0x03 };
bleResult_t result = GattClient_SimpleCharacteristicWrite
(
    deviceId,
    &myChar,
    mcValueLength_c,
    aValue
);
if (gBleSuccess_c != result)
{
    /* Handle error */
}
```

The Client Procedure Callback is triggered when writing is complete.

```
void gattClientProcedureCallback
(
    deviceId_t                  deviceId,
    gattProcedureType_t         procedureType,
    gattProcedureResult_t       procedureResult,
    bleResult_t                 error
)
{
    switch (procedureType)
    {
        /* ... */
        casegGattProcWriteCharacteristicValue_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                /* Continue */
            }
            else
            {
                /* Handle error */
                PRINT(error);
            }
            break;
        /* ... */
    }
}
```

5.1.5 Reading and writing characteristic descriptors

Two APIs are provided for these procedures which are very similar to Characteristic Read and Write.

The only difference is that the handle of the attribute to be read/written is provided through a pointer to an *gattAttribute_t* structure (same type as the *gattCharacteristic_t.value* field).

All of the following APIs have an enhanced counterpart of the form *GattClient_Enhanced[procedure]*. A *bearerId* parameter was added to specify on which bearer the transaction should take place. A value of *0* for the bearer

Id identifies the Unenhanced ATT bearer. Values higher than *0* are used to identify the Enhanced ATT bearer used for the ATT procedure.

```
bleResult_t GattClient_ReadCharacteristicDescriptor
(
    deviceId_t              deviceId,
    gattAttribute_t *       pIoDescriptor,
    uint16_t                maxReadBytes
);
```

The *pIoDescriptor->handle* is required (it may have been discovered previously by *GattClient_DiscoverAll CharacteristicDescriptors*). The GATT module fills the value that was read in the fields *pIoDescriptor->aValue* (must be linked to an allocated array) and *pIoDescriptor->valueLength* (size of the array).

Writing a descriptor is also performed similarly with this function:

```
bleResult_t GattClient_WriteCharacteristicDescriptor
(
    deviceId_t              deviceId,
    gattAttribute_t *       pDescriptor,
    uint16_t                valueLength,
    uint8_t *               aValue
);
```

Only the *pDescriptor->handle* must be filled before calling the function.

One of the most frequently written descriptors is the Client Characteristic Configuration Descriptor (CCCD). It has a well-defined UUID (*gBleSig_CCCD_d*) and a 2-byte long value that can be written to enable/disable notifications and/or indications.

In the following example, a Characteristic's descriptors are discovered and its CCCD written to activate notifications.

```
static gattCharacteristic_t myChar;
myChar. value . handle = 0x00A0; /* Or maybe it was previously discovered? */
#define mcMaxDescriptors_c 5
static gattAttribute_t aDescriptors[mcMaxDescriptors_c];
myChar. aDescriptors = aDescriptors;
/* ... */
{
    bleResult_t result = GattClient_DiscoverAllCharacteristicDescriptors
    (
        deviceId,
        &myChar,
        0xFFFF,
        mcMaxDescriptors_c
    );
    if (gBleSuccess_c != result)
    {
        /* Handle error */
    }
}
/* ... */
void gattClientProcedureCallback
(
    deviceId_t                  deviceId,
    gattProcedureType_t         procedureType,
    gattProcedureResult_t       procedureResult,
    bleResult_t                 error
```

```
)
{
    switch (procedureType)
    {
      /* ... */
      case gGattProcDiscoverAllCharacteristicDescriptors_c:
         if (gGattProcSuccess_c == procedureResult)
          {
           /* Find CCCD */
            for ( uint8_t j = 0; j < myChar. cNumDescriptors ; j++)
               {
                if (aDescriptors[j].uuidType && gBleSig_CCCD_d
 ==myChar.aDescriptors[j].uuid.uuid16) )
                 {
                    uint8_t cccdValue[2];
                    packTwoByteValue(gCccdNotification_c, cccdValue);
                    bleResult_t result =
 GattClient_WriteCharacteristicDescriptor
                     (
                       deviceId,
                       &myChar. aDescriptors [j],
                       2,
                       cccdValue
                     );
                    if (gBleSuccess_c != result)
                       {
                        /* Handle error */
                       }
                      break;
                     }
                }
            }
            else
            {
                /* Handle error */
                PRINT(error);
            }
            break;
        case gGattProcWriteCharacteristicDescriptor_c:
            if (gGattProcSuccess_c == procedureResult)
            {
                /* Notification successfully activated */
            }
            else
            {
                /* Handle error */
                PRINT(error);
            }
        /* ... */
    }
}
```

5.1.6 Resetting procedures

To cancel an ongoing Client Procedure, the following API can be called:

```
bleResult_t GattClient_ResetProcedure (void);
```

It resets the internal state of the GATT Client and new procedure may be started at any time.

5.2 Server APIs

Once the GATT Database has been created and the required security settings have been registered with *Gap_RegisterDeviceSecurityRequirements*, all ATT Requests and Commands and attribute access security checks are handled internally by the GATT Server module.

Besides this automatic functionality, the application may use GATT Server APIs to send Notifications and Indication and, optionally, to intercept Clients' attempts to write certain attributes.

5.2.1 Server callback

The first GATT Server call is the installation of the Server Callback, which has the following prototype:

```
typedef void (* gattServerCallback_t )
(
    deviceId_t              deviceId,    /*!< Device ID identifying the active
 connection. */
    gattServerEvent_t *     pServerEvent /*!< Server event. */
);
```

For EATT, the following signature should be used:

```
typedef void (*gattServerEnhancedCallback_t) ( deviceId_t deviceId, bearerId_t
 bearerId, gattServerEvent_t* pServerEvent );
```

The callback can be installed with:

```
bleResult_t GattServer_RegisterCallback
(
    gattServerCallback_t callback
);
```

The EATT server callback should be installed using the following API:

```
bleResult_t GattServer_RegisterEnhancedCallback
(
     gattServerEnhancedCallback_t callback
);
```

The first member of the *gattServerEvent_t* structure is the *eventType*, an enumeration type with the following possible values:

- *gEvtMtuChanged_c:* Signals that the Client-initiated MTU Exchange Procedure has completed successfully and the *ATT_MTU* has been increased. The event data contains the new value of the *ATT_MTU*. Is it possible that the application flow depends on the value of the *ATT_MTU*, for example, there may be specific optimizations for different *ATT_MTU* ranges. This event is not triggered if the *ATT_MTU* was not changed during the procedure.
- *gEvtHandleValueConfirmation_c:* A Confirmation was received from the Client after an Indication was sent by the Server.
- *gEvtAttributeWritten_c, gEvtAttributeWrittenWithoutResponse_c:* See Section 5.2.3 "Attribute write notifications".
- *gEvtCharacteristicCccdWritten_c:* The Client has written a CCCD. The application should save the CCCD value for bonded devices with *Gap_SaveCccd*.

- *gEvtError_c:* An error occurred during a Server-initiated procedure.
- *gEvtLongCharacteristicWritten_c*: A long characteristic was written.
- *gEvtInvalidPduReceived_c*: An invalid PDU was received from Client. Application decides if disconnection is required.
- *gEvtAttributeRead_c*: An attribute registered with `GattServer_RegisterHandlesForReadNotifications` is being read.

5.2.2 Sending notifications and indications

The APIs provided for these Server-initiated operations are very similar.

All of the following APIs have an enhanced counterpart of the form *GattServer_Enhanced[procedure]*. A *bearerId* parameter was added to specify on which bearer the transaction should take place. A value of *0* for the bearerId identifies the Unenhanced ATT bearer. Values higher than *0* are used to identify the Enhanced ATT bearer used for the ATT procedure.

```
bleResult_t GattServer_SendNotification
(
    deviceId_t      deviceId,
    uint16_t        handle
);
bleResult_t GattServer_SendIndication
(
    deviceId_t      deviceId,
    uint16_t        handle
);
```

Only the attribute handle needs to be provided to these functions. The attribute value is automatically retrieved from the GATT Database.

Note: *It is the application developer's responsibility to check if the Client designated by the deviceId has previously activated Notifications/Indications by writing the corresponding CCCD value. To do that, the following GAP APIs should be used:*

```
bleResult_t Gap_CheckNotificationStatus
(
    deviceId_t      deviceId,
    uint16_t        handle,
    bool_t *        pOutIsActive
);
bleResult_t Gap_CheckIndicationStatus
(
    deviceId_t      deviceId,
    uint16_t        handle,
    bool_t *        pOutIsActive
    );
```

Note: *It is necessary to use these two functions with the Gap_SaveCccd only for bonded devices, because the data is saved in NVM and reloaded at reconnection. For devices that do not bond, the application may also use its own bookkeeping mechanism.*

There is an important difference between sending **Notifications and Indications**:

- The latter can only be sent one at a time. In addition, the application must wait for the Client Confirmation (signaled by the *gEvtHandleValueConfirmation_c* Server event, or by a *gEvtError_c* event with *gGattClientConfirmationTimeout_c* error code) before sending a new Indication. Otherwise, a *gEvtError_c* event with *gGattIndicationAlreadyInProgress_c* error code is triggered.

- The Notifications can be sent consecutively.

5.2.3 Attribute write notifications

When the GATT Client reads and writes values from/into the Server's GATT Database, it uses ATT Requests.

The GATT Server module implementation manages these requests and, according to the database security settings and the Client's security status (authenticated, authorized, and so on), automatically sends the ATT Responses without notifying the application.

There are however some situations where the application needs to be informed of ATT packet exchanges. For example, a lot of standard profiles define, for certain Services, some, so-called, Control-Point Characteristics. These are Characteristics whose values are only of immediate significance to the application. Writing these Characteristics usually triggers specific actions.

For example, consider a fictitious Smart Lamp. It has Bluetooth Low Energy connectivity in the Peripheral role and it contains a small GATT Database with a Lamp Service (among other Services). The Lamp Service contains two Characteristics: the Lamp State Characteristic (LSC) and the Lamp Action Characteristic (LAC).

LSC is a "normal" Characteristic with Read and Write properties. Its value is either 0, lamp off, or 1, lamp on). Writing the value sets the lamp in the desired state. Reading it provides its current state, which is only useful when passing the information remotely.

The LAC has only one property, which is Write Without Response. The user can use the Write Without Response procedure to write only the value 0x01 (all other values are invalid). Whenever the user writes 0x01 in LAC, the lamp switches its state.

The LAC is a good example of a Control-Point Characteristic for these reasons:

- Writing a certain value (in this case 0x01) triggers an action on the lamp.
- The value the user writes has immediate significance only ("0x01 switches the lamp") and is never used again in the future. For this reason, it does not need to be stored in the database.

Obviously, whenever a Control-Point Characteristic is written, the application must be notified to trigger some application-specific action.

The GATT Server allows the application to register a set of attribute handles as "write-notifiable", in other words, the application wants to receive an event each time any of these attributes is written by the peer Client.

All Control-Point Characteristics in the GATT Database must have their Value handle registered. In fact, the application may register any other handle for write notifications for its own purposes with the following API:

```
bleResult_t GattServer_RegisterHandlesForWriteNotifications
(
    uint8_t         handleCount,
    const uint16_t *      aAttributeHandles
);
```

The *handleCount* is the size of the *aAttributeHandles* array and it cannot exceed *gcGattMaxHandleCountFor WriteNotifications_c*.

After an attribute handle has been registered with this function, whenever the Client attempts to write its value, the GATT Server Callback is triggered with one of the following event types:

- *gEvtAttributeWritten_c* is triggered when the attribute is written with a Write procedure (ATT Write Request). In this instance, the application has to decide whether the written value is valid and whether it must be written in the database, and, if so, the application must write the value with the *GattDb_WriteAttribute*, see Chapter

6. At this point, the GATT Server module does not automatically send the ATT Write Response over the air. Instead, it waits for the application to call this function:

```
bleResult_t GattServer_SendAttributeWrittenStatus
(
    deviceId_t      deviceId,
    uint16_t        attributeHandle,
    uint8_t         status
);
```

This API also has an enhanced counterpart, which adds the *bearerId* parameter.

The value of the *status* parameter is interpreted as an ATT Error Code. It must be equal to the *gAttErrCodeNoError_c* (0x00) if the value is valid and it is successfully processed by the application. Otherwise, it must be equal to a profile-specific error code (in interval 0xE0-0xFF) or an application-specific error code (in interval 0x80-0x9F).

- *gEvtAttributeWrittenWithoutResponse_c* is triggered when the attribute is written with a Write Without Response procedure (ATT Write Command). Because this procedure expects no response, the application may process it and, if necessary, write it in the database. Regardless of whether the value is valid or not, no response is needed from the application.
- *gEvtLongCharacteristicWritten_c* is triggered when the Client has completed writing a Long Characteristic value; the event data includes the handle of the Characteristic Value attribute and a pointer to its value in the database.

Attributes can also be registered for read notifications using the followng API:

```
bleResult_t GattServer_RegisterHandlesForReadNotifications
(
 uint8_t handleCount,
 const uint16_t* aAttributeHandles
);
```

To unregister one or more handles from the list for either write or read, the following APIs can be used:

```
bleResult_t GattServer_UnregisterHandlesForWriteNotifications
(
 uint8_t handleCount,
 const uint16_t* aAttributeHandles
);

bleResult_t GattServer_UnregisterHandlesForReadNotifications
(
 uint8_t handleCount,
 const uint16_t* aAttributeHandles
);
```

6 GATT database application interface

For over-the-air packet exchanges between a Client and a Server, the GATT Server module automatically retrieves data from the GATT database and responds to all ATT Requests from the peer Client, provided it passes the security checks. This ensures that the Server application does not have to perform any kind of searches over the database.

However, the application must have access to the database to write meaningful data into its characteristics. For example, a temperature sensor must periodically write the temperature, which is measured by an external thermometer, into the Temperature Characteristic.

For these kinds of situations, a few APIs are provided in the *gatt_db_app_interface.h* file.

Note: *All functions provided by this interface are executed synchronously. The result of the operation is saved in the return value and it generates no event.*

6.1 Writing and reading attributes

These are the two functions to perform basic attribute operations from the application:

```
bleResult_t GattDb_WriteAttribute
(
    uint16_t           handle,
    uint16_t           valueLength,
    const uint8_t *    aValue
);
```

The value length must be valid, as defined when the database is created. Otherwise, a *gGattInvalidValueLength_c* error is returned.

Also, if the database is created statically, as explained in Section 7 "Creating GATT database", the *handle* may be referenced through the enumeration member with a friendly name defined in the *gatt_db.h*.

```
bleResult_t GattDb_ReadAttribute
(
    uint16_t     handle,
    uint16_t     maxBytes,
    uint8_t *    aOutValue,
    uint16_t *   pOutValueLength
);
```

The *aOutValue* array must be allocated with the size equal to *maxBytes*.

6.2 Finding attribute handles

Although the application should be fully aware of the contents of the GATT Database, in certain situations it might be useful to perform some dynamic searches of certain attribute handles.

To find the handle value for a Service for which only the UUID is know the following API can be used:

```
bleResult_t GattDb_FindServiceHandle
(
 uint16_t startHandle,
 bleUuidType_t serviceUuidType,
 const bleUuid_t* pServiceUuid,
 uint16_t* pOutServiceHandle
);
```

To find a specific Characteristic Value Handle in a Service whose declaration handle is known, the following API is provided:

```
bleResult_t GattDb_FindCharValueHandleInService
(
    uint16_t               serviceHandle,
    bleUuidType_t          characteristicUuidType,
    const bleUuid_t *      pCharacteristicUuid,
    uint16_t *             pOutCharValueHandle
);
```

If the return value is *gBleSuccess_c*, the handle is written at *pOutCharValueHandle*. If the *serviceHandle* is invalid or not a valid Service declaration, the *gBleGattDbInvalidHandle_c* is returned. Otherwise, the search is performed starting with the *serviceHandle+1*. If no Characteristic of the given UUID is found, the function returns the *gBleGattDbCharacteristicNotFound_c* value.

To find a Characteristic Descriptor of a given type in a Characteristic, when the Characteristic Value Handle is known, the following API is provided:

```
bleResult_t GattDb_FindDescriptorHandleForCharValueHandle
(
    uint16_t           charValueHandle,
    bleUuidType_t      descriptorUuidType,
    const bleUuid_t *  pDescriptorUuid,
    uint16_t *         pOutDescriptorHandle
);
```

Similarly, the function returns *gBleGattDbInvalidHandle_c* is the handle is invalid. Otherwise, it starts searching from the *charValueHandle+1*. Then, *gBleGattDbDescriptorNotFound_c* is returned if no Descriptor of the specified type is found. Otherwise, its attribute handle is written at the *pOutDescriptorHandle* and the function returns *gBleSuccess_c*.

One of the most commonly used Characteristic Descriptors is the Client Configuration Characteristic Descriptor (CCCD), which has the UUID equal to *gBleSig_CCCD_d*. For this specific type, a special API is used as a shortcut:

```
bleResult_t GattDb_FindCccdHandleForCharValueHandle
(
    uint16_t     charValueHandle,
    uint16_t *   pOutCccdHandle
);
```

7 Creating GATT database

The GATT Database contains several *GATT Services* where each Service must contain at least one *GATT Characteristic*.

The Attribute Database contains a collection of *attributes*. Each attribute has four fields:

- The *attribute handle* – a 2-byte database index, which starts from 0x0001 and increases with each new attribute, not necessarily consecutive; maximum value is 0xFFFF.
- The *attribute type* or *UUID* – a 2-byte or 16-byte UUID.
- The *attribute permissions* – 1 byte containing access flags; this defines whether the attribute's value can be read or written and the security requirements for each operation type
- The *attribute value* – an array of maximum 512 bytes.

The ATT does not interpret the UUIDs and values contained in the database. It only deals with data transfer based on the attributes' handles.

The GATT gives meaning to the attributes based on their UUIDs and groups them into Characteristics and Services.

There are two possible ways of defining the GATT database:

- At compile-time (statically) or
- At runtime (dynamically)

7.1 Creating static GATT database

To define a GATT Database at compile-time, several macros are provided by the GATT_DB API. These macros expand in many different ways at compilation, generating the corresponding *Attribute Database* on which the Attribute Protocol (ATT) may operate.

This is the default way of defining the database.

The GATT Database definition is written in two files that are required to be added to the application project together with all macro expansion files:

- ***gatt_db.h*** - contains the actual declaration of Services and Characteristics.
- ***gat_uuid128.h*** – contains the declaration of Custom UUIDs (16-byte wide); these UUIDs are given a user-friendly name that is used in *gatt_db.h* file instead of the entire 16-byte sequence.

7.1.1 Declaring custom 128-bit UUIDs

All Custom 128-bit UUIDs are declared in the required file *gatt_uuid128.h*.

Each line in this file contains a single UUID declaration. The declaration uses the following macro:

- *UUID128 (name, byte1, byte2, …, byte16)*

The *name* parameter is the user-friendly handle that references this UUID in the *gatt_db.h* file.

The 16 bytes are written in the *LSB-first* order each one using the 0xZZ format.

Note: *On some occasions, it is desired to reuse an 128-bit UUID declared in gatt_uuid128.h. The 16 byte array is available through its friendly name and be accessed by including gatt_db_handles.h in the application. It is strongly advised to use it only in read-only operations. For example:*

```
(gatt_uuid128.h)
```

```
UID128(uuid_service_wireless_uart, 0xE0, 0x1C, 0x4B, 0x5E, 0x1E, 0xEB, 0xA1,
 0x5C, 0xEE, 0xF4, 0x5E, 0xBA, 0x00, 0x01, 0xFF, 0x01)
(app.c)
#include "gatt_db_handles.h"
........
/* Start Service Discovery*/
BleServDisc_FindService(peerDeviceId, gBleUuidType128_c, (bleUuid_t*)
 &uuid_service_wireless_uart);
```

7.1.2 Declaring a service

There are two types of Services:

- *Primary Services*
- *Secondary Services* - these are only to be included by other Primary or Secondary Services

The Service declaration attribute has one of these UUIDs, as defined by the Bluetooth SIG:

- 0x2800 a.k.a. <<*Primary Service*>> - for a Primary Service declaration
- 0x2801 a.k.a. <<*Secondary Service*>> - for a Secondary Service declaration

The Service declaration attribute permissions are read-only and no authentication required. The Service declaration attribute value contains the *Service UUID*. The *Service Range* starts from the Service declaration and ends at the next service declaration. All the Characteristics declared within the Service Range are considered to belong to that Service. For a more comprehensive list of SIG defied UUID values, check `ble_sig_defines.h`.

7.1.2.1 Service declaration macros

The following macros are to be used for declaring a Service:

- *PRIMARY_SERVICE (name, uuid16)*
  - Most often used.
  - The *name* parameter is common to all macros; it is a universal, user-friendly identifier for the generated attribute.
  - The *uuid16* is a 2-byte SIG-defined UUID, written in 0x*ZZZZ* format.
- *PRIMARY_SERVICE_UUID32 (name, uuid32)*
  - This macro is used for a 4-byte, SIG-defined UUID, written in 0x*ZZZZZZZZ* format.
- *PRIMARY_SERVICE_UUID128 (name, uuid128)*
  - The *uuid128* is the friendly name given to the custom UUID in the *gatt_uuid128.h* file.
- *SECONDARY _SERVICE (name, uuid16)*
- *SECONDARY_SERVICE_UUID32 (name, uuid32)*
- *SECONDARY _SERVICE_UUID128 (name, uuid128)*
  - All three are similar to Primary Service declarations.

7.1.2.2 Include declaration macros

Secondary Services are meant to be included by other Services, usually by Primary Services. Primary Services may also be included by other Primary Services. The inclusion is done using the Include declaration macro:

- *INCLUDE (service_name)*
  - The *service_name* parameter is the friendly name used to declare the Secondary Service.

  - This macro is used only for Secondary Services with a SIG-defined, 2-byte, Service UUID.
- *INCLUDE_CUSTOM (service_name)*
  - This macro is used for Secondary Services that have either a 4-byte UUID or a 16-byte UUID.

The effect of the service inclusion is that the *including* Service is considered to contain all the Characteristics of the *included* Service.

7.1.3 Declaring a characteristic

A Characteristic must only be declared inside a Service. It belongs to the most recently declared Service, so the GATT Database must always begin with a Service declaration.

The Characteristic declaration attribute has the following UUID, as defined by the Bluetooth SIG:

- 0x2803 a.k.a. <<*Characteristic*>>

The Characteristic declaration attribute value contains:

- the *Characteristic UUID*
- the *Characteristic Value* 's declaration handle
- the *Characteristic Properties* – Read, Write, Notify, and so on. (1 byte of flags)

The *Characteristic Range* starts from the Characteristic declaration and ends before a new Characteristic or a Service declaration.

After the Characteristic declaration these follow:

- A *Characteristic Value* declaration (mandatory; immediately after the Characteristic declaration).
- Zero or more *Characteristic Descriptor* declarations.

7.1.3.1 Characteristic declaration macros

The following macros are used to declare Characteristics:

- *CHARACTERISTIC (name, uuid16, properties)*
- *CHARACTERISTIC_UUID32 (name, uuid32, properties)*
- *CHARACTERISTIC _UUID128 (name, uuid128, properties)*

See Service declaration for *uuidXXX* parameter explanation.

The *properties* parameter is a bit mask. The flags are defined in the *gattCharacteristicPropertiesBitFields_t*.

7.1.3.2 Declaring characteristic values

The Characteristic Value declaration immediately follows the Characteristic declaration and uses one of the following macros:

- *VALUE (name, uuid16, permissions, valueLength, valueByte1, valueByte2, …)*
- *VALUE_UUID32 (name, uuid32, permissions, valueLength, valueByte1, valueByte2, …)*
- *VALUE _UUID128(name, uuid128, permissions, valueLength, valueByte1, valueByte2, …)*
  - See Section 7.1.2 "Declaring a service" for description of the *uuidXXX* parameter.
  - The *permissions* parameter is a bit mask, whose flags are defined in *gattAttributePermissionsBitFields_t* .
  - The *valueLength* is the number of bytes to be allocated for the Characteristic Value. After this parameter, exactly [*valueLength*] bytes follow in 0xZZ format, representing the initial value of this Characteristic.

These macros are used to declare Characteristic Values of *fixed lengths*.

Some Characteristics have *variable length values*. For those, the following macros are used:

- *VALUE_VARLEN (name, uuid16, permissions, maximumValueLength, initialValueLength, valueByte1, valueByte2, …)*
- *VALUE_UUID32_VARLEN (name, uuid32, permissions, maximumValueLength, initialValueLength, valueByte1, valueByte2, …)*
- *VALUE_UUID128_VARLEN (name, uuid128, permissions, maximumValueLength, initialValueLength, valueByte1, valueByte2, …)*
  - The number of bytes allocated for this Characteristic Value is *maximumValueLength*.
  - The number of *valueByteXXX* parameters shall be equal to *initialValueLength*.

Obviously, *initialValueLength* is, at most, equal to *maximumValueLength*.

7.1.3.3 Declaring characteristic descriptors

Characteristic's Descriptors are declared after the Characteristic Value declaration and before the next Characteristic declaration.

The macros used to declare Characteristic Descriptors are very similar to those used to declare fixed-length Characteristic Values:

- *DESCRIPTOR (name, uuid16, permissions, descriptorValueLength, descriptorValueByte1, descriptorValueByte2, …)*
- *DESCRIPTOR_UUID32 (name, uuid32, permissions, descriptorValueLength, descriptorValueByte1, descriptorValueByte2, …)*
- *DESCRIPTOR_UUID128(name, uuid128, permissions, descriptorValueLength, descriptorValueByte1, descriptorValueByte2, …)*

A special Characteristic Descriptor that is used very often is the *Client Characteristic Configuration Descriptor (CCCD)*. This is the descriptor where clients write some of the bits to activate Server notifications and/or indications. It has a reserved, 2-byte, SIG-defined UUID (0x2902), and its attribute value consists of only 1 byte (out of which 2 bits are used for configuration, the other 6 are reserved).

Because the CCCD appears very often in Characteristic definitions for standard Bluetooth Low Energy profiles, a special macro is used for CCCD declaration:

- *CCCD (name)*

This simple macro is basically equivalent to the following Descriptor declaration:

```
DESCRIPTOR (name,
   0x2902,
   (gGattAttPermAccessReadable_c
   | gGattAttPermAccessWritable_c),
   2, 0x00, 0x00)
```

7.1.4 Static GATT database definition examples

The GAP Service must be present on any GATT Database. It has the Service UUID equal to 0x1800, *<<GAP Service>>*, and it contains three read-only Characteristics, no authentication required: *Device Name, Appearance,* and *Peripheral Preferred Connection Parameters*. These also have well defined UUIDs in the SIG documents.

Most of the demos also include the optional GATT Security Levels characteristic, which defines the highest security requirements of the GATT server when operating in a LE connection.

The definition for this Service is shown here:

```
PRIMARY_SERVICE(service_gap, 0x1800)
    CHARACTERISTIC(char_device_name, 0x2A00, (gGattCharPropRead_c) )
            VALUE(value_device_name, 0x2A00, (gGattAttPermAccessReadable_c),  6,
 "Sensor")
    CHARACTERISTIC(char_appearance, 0x2A01, (gGattCharPropRead_c) )
            VALUE(value_appearance, 0x2A01, (gGattAttPermAccessReadable_c), 2,
 0xC2, 0x03)
    CHARACTERISTIC(char_ppcp, 0x2A04, (gGattCharPropRead_c) )
            VALUE(value_ppcp, 0x2A04, (gGattAttPermAccessReadable_c), 8, 0x0A,
 0x00, 0x10, 0x00, 0x64, 0x00, 0xE2, 0x04)
    CHARACTERISTIC(char_security_levels, gBleSig_GattSecurityLevels_d,
 (gGattCharPropRead_c) )
            VALUE(value_security_levels, gBleSig_GattSecurityLevels_d,
 (gPermissionFlagReadable_c), 2, 0x01, 0x01)
```

Another often encountered Service is the Scan Parameters Service:

```
PRIMARY_SERVICE(service_scan_parameters, 0x1813)
    CHARACTERISTIC(char_scan_interval_window, 0x2A4F,
 (gGattCharPropWriteWithoutRsp_c) )
        VALUE(value_scan_interval_window, 0x2A4F,
 (gGattAttPermAccessWritable),  4, 0x00, 0x00, 0x00, 0x00)
    CHARACTERISTIC(char_scan_refresh, 0x2A31, (gGattCharPropRead_c |
 gGattCharPropNotify_c) )
        VALUE(value_scan_refresh, 0x2A31, (gGattAttPermAccessReadable_c), 1,
 0x00) CCCD(cccd_scan_refresh)
```

Note: *All "user-friendly" names given in declarations are statically defined as enum members, numerically equal to the attribute handle of the declaration. This means that one of those names can be used in code wherever an attribute handle is required as a parameter of a function if gatt_db_handles.h is included in the application source file. For example, to write the value of the Scan Refresh Characteristic from the application-level code, use these instructions:*

```
#include "gatt_db_handles.h"
...
uint8_t scan_refresh_value = 0x12;
GattDb_WriteAttribute(char_scan_refresh, 1, &scan_refresh_value);
```

For static database declarations, the 'attribute handle' is equal to the line number in the `gatt_fb.h` file, where the attribute is defined.

7.2 Creating a GATT database dynamically

To define a GATT Database at runtime, the *gGattDbDynamic_d* macro must be defined in *app_preinclude.h* with the value equal to 1.

Then, the application must use the APIs provided by the *gatt_db_dynamic.h* interface to add and remove Services and Characteristics as needed.

See Section 7.1 "Creating static GATT database" for a detailed description of Service and Characteristic parameters.

7.2.1 Memory considerations

The GATT Dynamic database module internally manages the memory allocation for the database. If the `gMemManagerLightExtendHeapAreaUsage` define is set to `1` in the desired application, the whole available heap is used. In such as case, the user does not have to allocate space for the dynamic database. If this is not done, the user only needs to make sure that the `MinimalHeapSize_c` define is set to a high enough value considering all attributes and attribute values they want to add to the database, as well as other memory requirements the application might have.

Internally, two buffers are used by the dynamic database module: an attribute buffer and a value buffer. The attribute buffer size increases with the addition of each attribute to the database. The value buffer size increases depending on the UUID type and value lengths required by the application. The two buffers start with a minimum size and are reallocated whenever new requests to add entries are received and there is not enough available memory left. If the user removes these entries from the database, the memory reserved for those entries is not freed, but shifted, leaving room for new entries. Thus, an add operation after a remove operation might not necessarily reallocate the buffer if the new entries fit. The two buffers used by the Dynamic database module will not be available to the application until the user releases the database.

7.2.2 Initialization and release

Before anything can be added to the database, it must be initialized with an empty collection of attributes.

The *GattDbDynamic_Init()* API is automatically called by the *GattDb_Init()* implementation provided in the *gatt_database.c* source file. Application-specific code does not need to call this API again, unless at some point it destroys the database with *GattDb_ReleaseDatabase()*.

7.2.3 Adding services

The APIs that can be used to add Services are self-explanatory:

- *GattDbDynamic_AddPrimaryServiceDeclaration*
  - The Service UUID is specified as parameter.
    Memory requirements: one entry in the attribute buffer and UUID size in value buffer.
- *GattDbDynamic_AddSecondaryServiceDeclaration*
  - The Service UUID is specified as parameter.
    Memory requirements: one entry in the attribute buffer and UUID size in value buffer.
- *GattDbDynamic_AddIncludeDeclaration*
  - The Service UUID and handle range are specified as parameters.
    Memory requirements: one entry in the attribute buffer and 6 bytes in value buffer.

The functions have an optional out parameter *pOutHandle*. If its value is not NULL, the execution writes a 16-bit value in the pointed location representing the attribute handle of the added declaration. The application can use this handle as parameter in few *GattDbApp* APIs or in the Service removal functions.

At least one Service must be added before any Characteristic.

7.2.4 Adding characteristics and descriptors

The APIs for adding Characteristics and Descriptors are enumerated below:

- *GattDbDynamic_AddCharacteristicDeclarationAndValue*
  - The Characteristic UUID, properties, access permissions, and initial value are specified as parameters.
- *GattDbDynamic_AddCharacteristicDeclarationWithUniqueValue*
  - Multiple calls to this API allocate a unique 512-byte value buffer as an optimization for application that deal with large value buffers that do not always need to be stored separately.
- *GattDbDynamic_AddCharacteristicDescriptor*
  - The Descriptor UUID, access permissions and initial value are specified as parameters.
- *GattDbDynamic_AddCccd*
  - Shortcut for a CCCD.

Characteristics and descriptors are automatically added at the end of the database. Thus, a service declaration should be followed by all desired characteristic and descriptor definitions before adding a new service to the database.

7.2.5 Removing services and characteristics

To remove a Service or a Characteristic, the following APIs may be used, both of which only require the declaration handle as parameter:

- *GattDbDynamic_RemoveService*
- *GattDbDynamic_RemoveCharacteristic*

7.3 Gatt caching

7.3.1 Service change feature

The **service changed** feature applies to GATT servers and supports the service changed characteristic, dynamic databases, and handle value indications. The GATT clients that require to be notified for structural modifications on the database should write the CCCD of the Service Changed Characteristic on the server. The value of the Service Changed characteristic is represented by 2 handle values for the handle range affected by the modifications.

The changes that trigger a server database modification are represented by the following API calls:

- *GattDbDynamic_AddPrimaryServiceDeclaration*
- *GattDbDynamic_AddSecondaryServiceDeclaration*
- *GattDbDynamic_AddIncludeDeclaration*
- *GattDbDynamic_AddCharacteristicDeclarationAndValue*
- *GattDbDynamic_AddCharDescriptor*
- *GattDbDynamic_AddCccd*
- *GattDbDynamic_RemoveService*
- *GattDbDynamic_RemoveCharacteristic*

Those GATT server API calls update two internal handles to memorize the minimum and maximum range affected by the change.

After the GATT server database update is done, the application must call the `GattDbDynamic_EndDatabaseUpdate()` API. After this, a Service Changed indication is internally sent to each connected peer that has enabled these indications. The indication contains the handle range affected by the change.

. For bonded devices with whom the server is not currently in an active connection, the changes are buffered on the server and the peers are notified upon reconnection.

7.3.2 Robust caching

Robust caching is a feature where the server sends an error response to the client if the server does not consider the client to be aware of a database structural change. A server supporting robust caching provides the Client Supported Features, Database Hash, and Service Changed characteristics. To indicate support for robust caching, clients should write the robust caching bit (bit 0) of the Client Supported Features characteristic on the server. An example of the GAP service definition for a server with robust caching support is the following:

- *PRIMARY_SERVICE(service_gatt, gBleSig_GenericAttributeProfile_d)*
- *CHARACTERISTIC(char_service_changed, gBleSig_GattServiceChanged_d, (gGattCharPropIndicate_c))*
- *VALUE(value_service_changed, gBleSig_GattServiceChanged_d, (gPermissionNone_c), 4, 0x01, 0x00, 0xFF, 0xFF)*
- *CCCD(cccd_service_changed)*
- *CHARACTERISTIC(char_client_supported_features, gBleSig_GattClientSupportedFeatures_d, (gGattCharPropRead_c | gGattCharPropWrite_c))*
- *VALUE(value_client_supported_features, gBleSig_GattClientSupportedFeatures_d, (gPermissionFlagReadable_c | gPermissionFlagWritable_c), 8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00)*
- *CHARACTERISTIC(char_database_hash, gBleSig_GattDatabaseHash_d, (gGattCharPropRead_c))*
- *VALUE(value_database_hash, gBleSig_GattDatabaseHash_d, (gPermissionFlagReadable_c), 16, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00)*

The Client Supported Features characteristic should be declared as an array. The array size is equal to the maximum number of connections, which can be active at one time, so that each possible peer can have its own value. The Server Supported Features characteristic value is automatically written by the Host to indicate robust caching support when appropriate. The Database Hash characteristic value is a 128-bit unsigned integer number where the computed hash value is written.

In order to enable the Robust Caching feature, the *gGattCaching_d* define should be enabled in the file `app_preinclude.h`. In order to enable automatic Host support for Robust Caching, the *gGattAutomaticRobustCachingSupport_d* define should also be enabled. This includes writing the Client Supported Features characteristic, writing the CCCD of the Service Changed characteristic, and reading the Database Hash after reconnecting with a previously bonded peer to check for new changes. If this define is not enabled, then it is up to the application to read and write all necessary characteristic for Robust Caching support.

The client state is kept on the server using the following enum:

```
typedef enum
{
gGattClientChangeUnaware_c = 0x00U, /*!< Gatt client state */
gGattClientStateChangePending_c = 0x01U, /*!< Gatt client state */
gGattClientChangeAware_c = 0x02U, /*!< Gatt client state */
} gattCachingClientState_c;
```

The initial state of a client without a trusted relationship is change-aware. The state of a client with a trusted relationship remains unchanged from the previous connection. However, in cases where the database has been updated since the last connection, the initial state is change-unaware. When a database update occurs, all connected clients become change unaware.

If a change-unaware client sends an ATT command, the server ignores it. For ATT requests received from a change-unaware client, the server sends an error response with the error code set to *gAttErrCodeDatabaseOutOfSync_c*. The server should also not send indications and notifications to change

unaware clients, except for the Service Changed indication. The state of a client is verified by the GATT server before executing each command, request or sending any notifications or indications.

The following PDU types are an exception to this rule and do not generate an `gAttErrCodeDatabaseOutOfSync_c` error code:

- ATT_FIND_INFORMATION_REQ
- ATT_FIND_BY_TYPE_VALUE_REQ
- ATT_READ_BY_GROUP_TYPE_REQ
- ATT_EXECUTE_WRTIE

For a change unaware client to become change aware again, one of the following must happen:

- The client receives and confirms a Service Changed Indication.
- The server, upon receiving a request from a change unaware client, sends the client a response with the error code set to `Database Out Of Sync` and then the server receives another ATT request from the client.
- The change unaware client reads the Database Hash characteristic and then the server receives another ATT request from the client.

The function *GattDb_ComputeDatabaseHash()* is used by the server to compute the hash value and save its value in the database. The computation is done when a read request for the database hash characteristic is first received from a peer GATT client for dynamic databases.

For static databases, hash computation is disabled by default. If you have a static database and want to compute the database hash, then declare the following define to `TRUE` in `app_preinclude.h`: *gGattDbComputeHash_d*. By doing this, the hash value is computed during the host initialization. The value is written directly to the database as characteristic and it can be viewed in the memory, as see in the image below. Since static databases do not change in structure over time, this value remains constant, so it can be saved separately and written manually to memory if needed. See Figure 10.

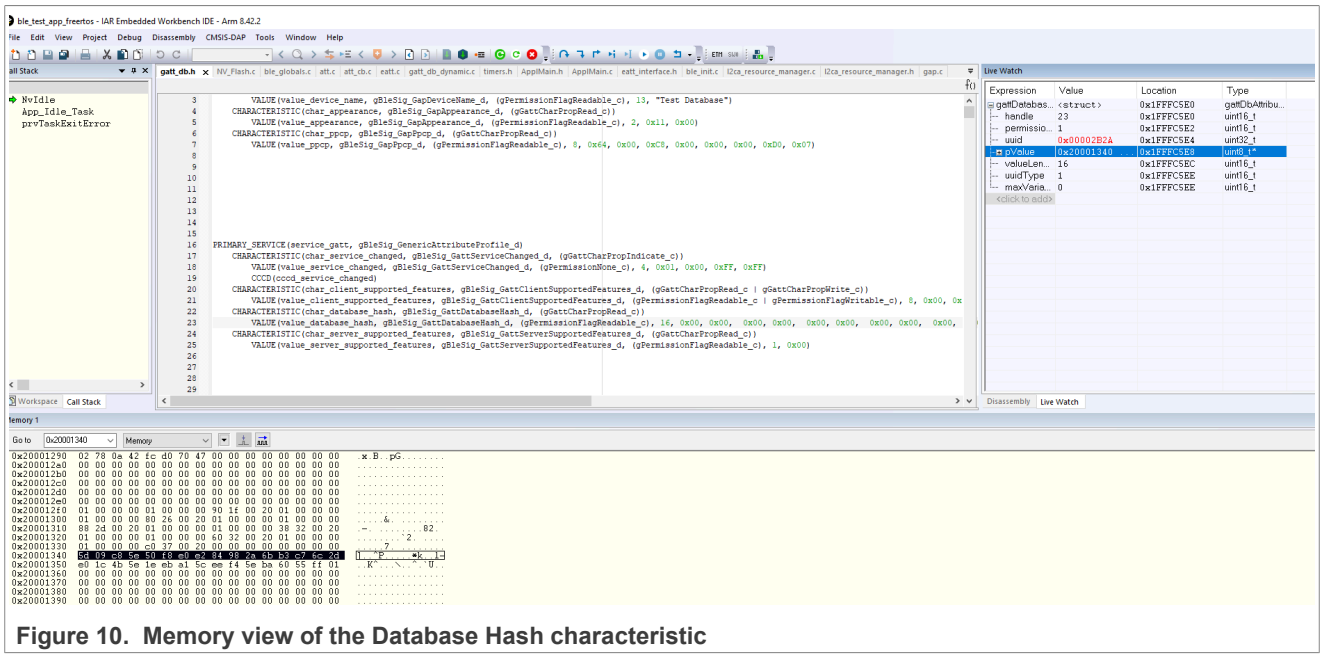

Figure 10. Memory view of the Database Hash characteristic

On the client side, the *GattClient_GetDatabaseHash()* function is used to read the hash value from a peer GATT server. If the *gGattAutomaticRobustCachingSupport_c* define is enabled, then the following steps are executed automatically:

- Writing the Client Supported Features characteristic value to indicate robust caching support – set BIT0 to `1`.
- Write the CCCD of the Service Changed characteristic.
- Read the initial value of the Database Hash characteristic and store it locally.

Otherwise, just the read request for the Database Hash characteristic is sent to the peer.

The following arrays and variables are used for the implementations of the robust caching and service changed features (declared in `ble_globals.c`):

```
/* Service changed indication buffer */
    gattHandleRange_t gServiceChangedIndicationStorage[gMaxBondedDevices_c];

/* client saved values for service changed characteristic and CCCD handles */
    uint16_t mActiveServiceChangedCharHandle[gAppMaxConnections_c] = {gGattDbInvalidHandle_d};
    uint16_t mServiceChangedCharHandle[gMaxBondedDevices_c] = {gGattDbInvalidHandle_d};
    uint16_t mActiveServiceChangedCCCDHandle[gAppMaxConnections_c] = {gGattDbInvalidHandle_d};

/* server values for its own service changed characteristic and CCCD handles */
    uint16_t mServerServiceChangedCharHandle;
    uint16_t mServerServiceChangedCCCDHandle;

/* client state information for bonded and active clients */
    gattCachingClientState_c gGattClientState[gMaxBondedDevices_c] = {gGattClientChangeAware_c};
    gattCachingClientState_c gGattActiveClientState[gAppMaxConnections_c] =
 {gGattClientChangeAware_c};

/* Database hash values - the client needs a hash value for each possible peer */
    uint8_t mGattActiveServerDatabaseHash[gGattDatabaseHashSize_c * gAppMaxConnections_c] = {0};
    uint8_t mGattServerDatabaseHash[gGattDatabaseHashSize_c * gMaxBondedDevices_c] = {0};

/* client supported features handles for active gatt servers */
    uint16_t gGattActiveClientSupportedFeaturesHandles[gAppMaxConnections_c] =
 {gGattDbInvalidHandle_d};

/* client supported features information for bonded gatt clients */
    uint8_t gGattClientSupportedFeatures[gMaxBondedDevices_c] = {0U};

/* index of the database hash characteristic in the database */
    uint32_t mServerDatabaseHashIndex = gGattDbInvalidHandleIndex_d;

/* index of the client supported features characteristic in the database */
    uint32_t mServerClientSupportedFeatureIndex = gGattDbInvalidHandleIndex_d;
```

It is up to the application to save a local copy of the information from the server's database and to initiate service discovery only on the first connection or when it is informed of a change by the peer using Service Changed and Robust Caching.

If the *gGattAutomaticRobustCachingSupport_c* define is not set, it is up to the application to check the Server Supported Features characteristic value on the peer, to write the Client Supported Features characteristic value and to write the Service Changed CCCD.

Two new GATT procedures are introduced as part of the Robust Caching feature. Both should be treated according to the application needs in the GATT procedure callback of the application.

- *gGattProcSignalServiceDiscoveryComplete_c* – informs the application that the service discovery procedure has finished after reading the Database Hash value using the read using characteristic UUID procedure. The application procedure callback should call *BleServDisc_Finished()* on this event when robust caching is supported.
- *gGattProcUpdateDatabaseCopy_c* – informs the application that its database copy is no longer up to date and service discovery should be reperformed. Used when the client received an error response with the

gAttErrCodeDatabaseOutOfSync_c opcode, when the local database hash value is found to be out of sync with the one on the server, or when a service changed indication is received from the server.

If service discovery is performed using our ble_service_discovery module, then the application should wait for the *gDiscoveryFinished_c* event before initiating its own GATT procedures. The application should also make sure to not initiate a second GATT procedure which requires a response from the peer before receiving a response to the first request it made.

8 Creating a Custom Profile

This chapter describes how the user can create customizable functionality over the Bluetooth Low Energy Host Stack by defining profiles and services. The Temperature Profile, used by the Temperature Sensor and Collector applications, is used as a reference to explain the steps of building custom functionality.

8.1 Defining custom UUIDs

The first step when defining a new service included in a profile is to define the custom 128-bit UUID for the service and the included characteristics. These values are defined in *gatt_uuid128.h*, which is located in the application folder. For example, the Temperature Profile uses the following UUID for the service:

```
/* Temperature */
UUID128(uuid_service_temperature, 0xfb ,0x34 ,0x9b ,0x5f ,0x80 ,0x00
,0x00 ,0x80 ,0x00 ,0x10 ,0x00 ,0x02 ,0x00 ,0xfe ,0x00 ,0x00)
```

The definition of the services and characteristics are made in *gattdb.h*, as explained in Section 7 "Creating GATT database". For more details on how to structure the database, see Section 9 "Application Structure".

8.2 Creating service functionality

All defined services in the SDK have a common template which helps the application to act accordingly.

The service locally stores the device identification for the connected client. This value is changed on subscription and non-subscription events.

```
/*! Temperature Service - Subscribed Client*/
static deviceId_t mTms_SubscribedClientId;
```

The service is initialized and changed by the application through a service configuration structure. It usually contains the service handle, initialization values for the service (for example, the initial temperature for the Temperature Service) and in some cases user-specific structures that can store saved measurements (for example, the Blood Pressure Service). Below is an example for the custom Temperature Service:

```
/*! Temperature Service - Configuration */
typedef struct tmsConfig_tag
{
    uint16_t serviceHandle ;
    int16_t initialTemperature ;
} tmsConfig_t ;
```

The initialization of the service is made by calling the start procedure. The function requires as input a pointer to the service configuration structure. This function is usually called when the application is initialized. It resets the static device identification for the subscribed client and initializes both dynamic and static characteristic values. An example for the Temperature Service (TMS) is shown below:

```
bleResult_t Tms_Start ( tmsConfig_t *pServiceConfig)
{
    mTms_SubscribedClientId = gInvalidDeviceId_c;
    return Tms_RecordTemperatureMeasurement (pServiceConfig-> serviceHandle ,
                                                pServiceConfig->
 initialTemperature );
}
```

The service subscription is triggered when a device connects to the server. It requires the peer device identification as an input parameter to update the local variable. On disconnect, the unsubscribe function is called to reset the device identification. For the Temperature Service:

```
bleResult_t Tms_Subscribe ( deviceId_t deviceId)
{
    mTms_SubscribedClientId = deviceId;
    return gBleSuccess_c;
}
bleResult_t Tms_Unsubscribe (void)
{
    mTms_SubscribedClientId = gInvalidDeviceId_c;
    return gBleSuccess_c;
}
```

Depending on the complexity of the service, the API implements additional functions. For the Temperature Service, there is only a temperature characteristic that is notifiable by the server. The API implements the record measurement function which saves the new measured value in the GATT database and send the notification to the client device if possible. The function needs the service handle and the new temperature value as input parameters:

```
bleResult_t Tms_RecordTemperatureMeasurement ( uint16_t serviceHandle, int16_t
 temperature)
{
    uint16_t handle;
    bleResult_t result;
    bleUuid_t uuid = Uuid16(gBleSig_Temperature_d);
    /* Get handle of Temperature characteristic */
    result = GattDb_FindCharValueHandleInService(serviceHandle,
        gBleUuidType16_c, &uuid, &handle);
    if (result != gBleSuccess_c)
        return result;
    /* Update characteristic value */
    result = GattDb_WriteAttribute(handle, sizeof( uint16_t ), ( uint8_t
 *)&temperature);
    if (result != gBleSuccess_c)
        return result;
    Hts_SendTemperatureMeasurementNotification(handle);
    return gBleSuccess_c;
}
```

To accommodate some use cases where the service is reset, the stop function is called. The reset also implies a service unsubscribe. Below is an example for the Temperature Service:

```
bleResult_t Tms_Stop ( tmsConfig_t *pServiceConfig)
{
    return Tms_Unsubscribe();
}
```

8.3 GATT client interactions

The client side of the service, which includes the service discovery, notification configuration, attribute reads and others are left to be handled by the application. The application calls the GATT client APIs and reacts accordingly. The only exception for this rule is that the service interface declares the client configuration structure. This structure usually contains the service handle and the handles of all the characteristic values and

descriptors discovered. Additionally it can contain values that the client can use to interact with the server. For the Temperature Service client, the structure is as follows:

```
/*! Temperature Client - Configuration */
typedef struct tmcConfig_tag
{
    uint16_t                hService;
    uint16_t                hTemperature ;
    uint16_t                hTempCccd ;
    uint16_t                hTempDesc ;
    gattDbCharPresFormat_t  tempFormat ;
} tmcConfig_t ;
```

9 Application Structure

This chapter describes the organization of the Bluetooth Low Energy demo applications that can be found in the SDK. By familiarizing with the application structure, the user is able to quickly adapt its design to an existing demo or create a new application.

The Temperature Sensor application is used as a reference to showcase the architecture.

9.1 Folder structure

The Figure 11 shows the application folder structure.

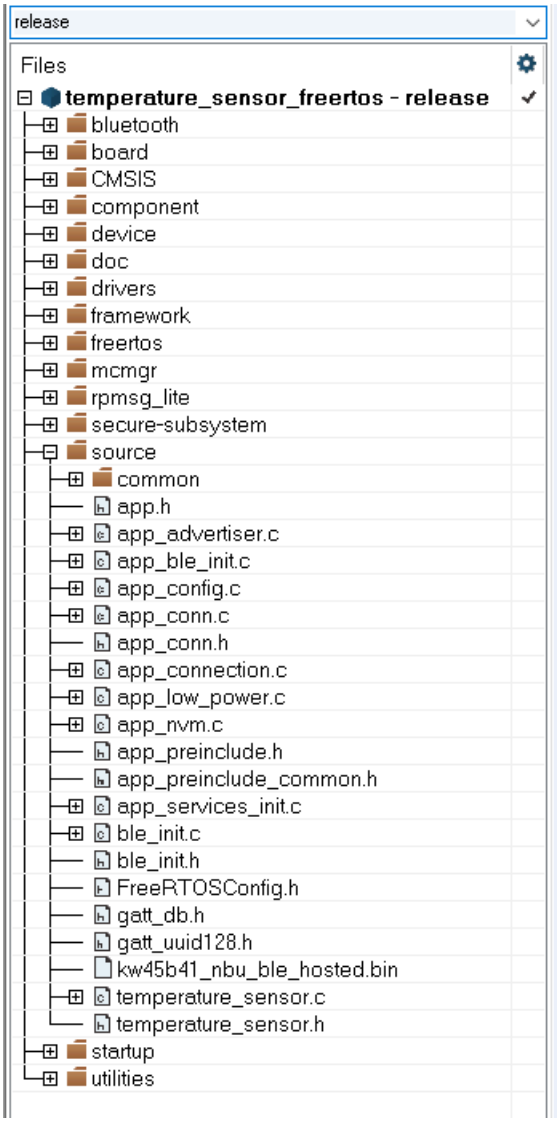

Figure 11. Application Folder structure in workspace

The *app* folder follows a specific structure which is recommended for any application developed using the Bluetooth Low Energy Host Stack:

- The *common* group contains the application framework shared by all profiles and demo applications:
  - Bluetooth Low Energy Connection Manager

  - Bluetooth Service Discovery Manager
  - Bluetooth Low Energy Stack and Task Initialization and Configuration
  - GATT Database
- The *source* group contains code specific to the Temperature Sensor application
  In the following examples, *app.c* is used as a placeholder for the main application source file. In the case of Temperature Sensor, it is *temperature_sensor.c*.
  The source group also contains files which aid with the handling of Host events and framework related functionality. These are: *app_con.c*/*app_con.h*, *app_advertiser.c*, *app_connection.c*, *app_scanner.c*, *app_nvm.c*, and *app_lowpower.c*. These files do not allow the application to implement its own state machines, unless application-specific functionality is required. For example, the application is restricted from setting advertising parameters and starting advertising, unless certain application-specific functionalities, such as UI related updates are required.

The *bluetooth* folder/group contains:

- The *controller/interface*, *host/interface*, and *host/config*. These are public interfaces and configuration files for the Controller and the Host. For the Host, functionality is included in the library located in the *host/lib* subfolder. The folder is not shown in the IAR project structure, but added into the toolchain linker settings under the library category.
- *profiles* contains profile-specific code; it is used by each demo application of standard profiles.

The framework and component folders/groups contain framework components used by the demo applications. For additional information, see the *Connectivity Framework Reference Manual*.

The *freertos* folder contains sources for the supported operating system.

9.2 Application main framework

The Application Main module contains common code used by all the applications, such as:

- The Main Task
- Messaging framework between the Bluetooth LE Host Stack Task and the Application Task

9.2.1 Start task

The Start Task (*start_task*) is the first task created by the operating system and is also the one that initializes the rest of the system. It initializes framework components (Memory Manager, Timers Manager etc.) and it calls *BluetoothLEHost_AppInit* from *app.c*, which is used to initialize the Bluetooth LE Host Stack as well as peripheral drivers specific to the implemented application.

The function calls *BluetoothLEHost_HandleMessages*, which represents the Application Task and is used to process events and messages from the Host Stack.

The stack size and priority of the main task are defined in *fsl_os_abstraction_config.h*:

```
#ifndef gMainThreadStackSize_c
#define gMainThreadStackSize_c 1024
#endif
#ifndef gMainThreadPriority_c
#define gMainThreadPriority_c 7
#endif
```

9.2.2 Application messaging

The module contains a wrapper that is used to create messages for events generated by the Bluetooth LE Host Stack in the Host Task context. The wrapper also sends them to be processed by the application in the context of the Application Task.

For example, connection events generated by the Host are received by *App_ConnectionCallback*. The function creates a message, places it in the Host to Application queue and signals the Application with *gAppEvtMsgFromHostStack_c*. The Application Task de-queues the message and calls *App_HandleHostMessageInput*, which calls the corresponding callback implemented the application-specific code (*app.c*), in this example: *BleApp_ConnectionCallback*.

It is strongly recommended that the application developer uses the *app.c* module to add custom code on this type of callbacks.

9.3 Bluetooth LE Connection Manager

The connection manager is a helper module that contains common application configurations and interactions with the Bluetooth LE Host Stack. It implements the following events and methods:

- Host Stack GAP Generic Event
- Host Stack Connection Event on both GAP Peripheral and GAP Central configuration
- Host Stack configuration for GAP Peripheral or GAP Central

9.3.1 GAP generic event

The GAP Generic Event is triggered by the Bluetooth LE Host Stack and sent to the application via the generic callback. Before any application-specific interactions, the Connection Manager callback is called to handle common application events, such as device address storage.

```
void BleApp_GenericCallback ( gapGenericEvent_t * pGenericEvent)
{
    /* Call Bluetooth Low Energy Conn Manager */
    BleConnManager_GenericEvent(pGenericEvent);
    switch (pGenericEvent-> eventType )
    {
        ...
    }
}
```

In the BleConnManager_GenericEvent function, the local keys are generated.

- The local LTK, IRK, and CSRK as well as EDIV and RAND are obtained hashing over the board's UID and stored in RAM as plain-text every time the *gInitializationComplete_c* event is received.
- In Advanced Secure mode, local IRK and CSRK are generated using the EdgeLock Secure Enclave and stored into a dedicated NVM data set as ELKE blobs (40 bytes blob encrypted using unique die key) on the first `gInitializationComplete_c` event received.

The NBU Decryption key for IRK is generated and distributed to the NBU over the private key bus and the EIRK blob (16 bytes blob which can be decrypted only by NBU hardware using NBU Decryption key for IRK) is generated from the IRK ELKE blob and stored in RAM to be used for controller privacy on every `gInitializationComplete_c` event received. For the host privacy the ELKE IRK blob is used instead. For details, refer to the section "Advanced security capabilities".

9.3.2 GAP configuration

The GAP Central or Peripheral Configuration is used to create common configurations (such as setting the public address, registering the security requirements, adding the addresses of bonded devices in the Controller Filter Accept List), which can be customized by the application afterwards. It is called inside the *BluetoothLEHost_Initialized callback* function, before any application-specific configuration, as shown in the example code below.

```
static void BluetoothLEHost_Initialized()
{
    /* Set common GAP configuration */
    BleConnManager_GapCommonConfig();
    ...
}
```

9.3.3 GAP connection event

The GAP Connection Event is triggered by the Host Stack and sent to the application via the connection callback. Before any application-specific interactions, the Connection Manager callback is called to handle common application events, such as device connect, disconnect or pairing-related requests. It is called inside the registered connection such as shown below:

```
static void BleApp_ConnectionCallback ( deviceId_t peerDeviceId,
 gapConnectionEvent_t * pConnectionEvent)
{
    /* Connection Manager to handle Host Stack interactions */
    BleConnManager_GapPeripheralEvent(peerDeviceId, pConnectionEvent);
    switch (pConnectionEvent-> eventType )
    {
        ...
    }
}
```

It is strongly recommended that the application developer uses the `app.c` module to add custom code.

9.3.4 Privacy

To enable or disable Privacy, the following APIs may be used:

```
bleResult_t
 BleConnManager_EnablePrivacy(void);
```

```
bleResult_t
 BleConnManager_DisablePrivacy(void);
```

The function BleConnManager_EnablePrivacy calls *BleConnManager_ManagePrivacyInternal* after checking if the privacy is enabled.

```
 static bleResult_t
 BleConnManager_ManagePrivacyInternal
 (bool_t bCheckNewBond);
```

If the privacy feature is supported (*gAppUsePrivacy_d = 1*), the Connection Manager activates Controller Privacy or Host Privacy depending on the board capabilities.

The *bCheckNewBond* is a boolean that tells the Manager whether it should check or not if a bond between the devices already exists.

In order to update the identity information after a bond is added or removed privacy should be disabled and enabled. For pairing with bonding this is done automatically in *ble_conn_manager*. In case the application adds or removes a bond through the GAP API, it should also disable and enable privacy.

9.4 GATT database

The *gatt_db* contains a set of header files grouped in the *macros* subfolder. These macros are used for static code generation for the GATT Database by expanding the contents of the *gatt_db.h* file in different ways. Section 7 "Creating GATT database" explains how to write the *gatt_db.h* file using user-friendly macros that define the GATT database.

At application compile time, the *gatt_database.c* file is populated with enumerations, structures, and initialization code used to allocate and properly populate the GATT database. In this way, the *gattDatabase*array and the *gGattDbAttributeCount_c* variable (see Section 2.2 "GATT database") are created and properly initialized.

Note: *Do not modify any of the files contained in the gatt_db folder and its subfolder.*

To complete the GATT database initialization, this demo application includes the required `gatt_db.h` and `gatt_uuid128.h` files in its specific application folder, along with other profile-specific configuration and code files.

9.5 RTOS specifics

9.5.1 Operating system selection

The SDK offers different projects for each supported operating system (FreeRTOS OS) and for BareMetal configuration. To switch between systems, the user needs to switch the workspace.

The RTOS source code is found in the SDK package and is linked in the workspace in the *freertos* virtual folder, as shown in Figure 12:

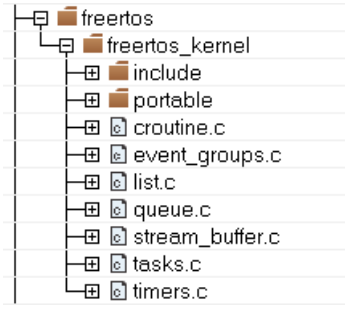

Figure 12. Location of FreeRTOS source code in workspace

9.5.2 Bluetooth LE Host task configuration

Application developers are provided with two files for RTOS task initialization:

- *ble_host_task_config.h*, and *ble_host_tasks.c* for the Host.

Reusing these files is recommended because they perform all the necessary RTOS-related work. The application developer must only modify the macros from **_config.h* files whenever tasks need a bigger stack size or different priority settings. The new values should be overridden in the *app_preinclude.h* file.

9.6 Board configuration

The configuration files for the supported boards can be found in the *board* folder, as shown in Figure 13. The files contain clock and pin configurations that are used by the drivers. The user can customize the board files by modifying the configuration of the pins and clock source according to his design.

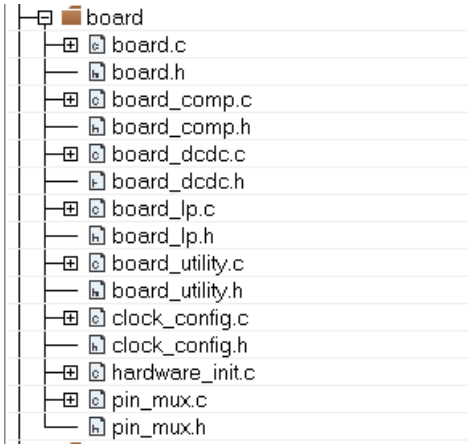

Figure 13. Board configuration files

9.7 Bluetooth Low Energy initialization

The *ble_init.h* and *ble_init.c* files contain the declaration and the implementation of the following function:

```
bleResult_t Ble_Initialize
(
    gapGenericCallback_t gapGenericCallback
)
{
#if defined(gUseHciTransportDownward_d) && gUseHciTransportDownward_d
    /* HCI Transport Init */
    if (gHciSuccess_c != Hcit_Init(Ble_HciRecvFromIsr))
    {
        return gHciTransportError_c;
    }
#if defined(KW45B41Z83_SERIES) || \
    defined(KW45B41Z82_SERIES) || \
    defined(K32W1480_SERIES)
    /*
     * Set BD Address in Controller. Must be done after HCI init
     * and before Host init.
     */
      Ble_SetBDAddr();
#endif /* KW45B41Z83_SERIES */
    /* Check for available memory storage */
    if (!Ble_CheckMemoryStorage())
    {
        return gBleOutOfMemory_c;
    }
    /* Bluetooth Low Energy Host Tasks Init */
```

```
    if (KOSA_StatusSuccess != Ble_HostTaskInit())
    {
        return gBleOsError_c;
    }
    /* Bluetooth Low Energy Host Stack Init */
    return Ble_HostInitialize(gapGenericCallback, Hcit_SendPacket);
#elif defined(gUseHciTransportUpward_d) && gUseHciTransportUpward_d
#else /* gUseHciTransportUpward_d */
#endif /* gUseHciTransportUpward_d */
}
```

Note: *This function should be used by your application because it correctly performs all the necessary Bluetooth Low Energy initialization.*

Step-by-step analysis is provided below:

1. First, the HCI interface is initialized by calling *Hcit_Init*. This initializes communication between the Host and the Controller.
2. After setting the BD address into the Controller (Ble_SetBDAddr) and performing memory validation checks (*Ble_CheckMemoryStorage*), the Host task is initialized by calling *Ble_HostTaskInit*.
3. Finally the *Ble_HostInitialize* function initializes the Host with the transport packet transmit function used as the *hciHostToControllerInterface_t parameter*.

9.8 Bluetooth Low Energy Host Stack configuration

The Bluetooth LE Host Stack libraries are found in the `middleware/wireless/bluetooth/host/lib` folder. The user should add the best matching library for its use case to the linker options of its project.

For example, the temperature sensor uses the Peripheral Host Stack library, as shown in Figure 14:

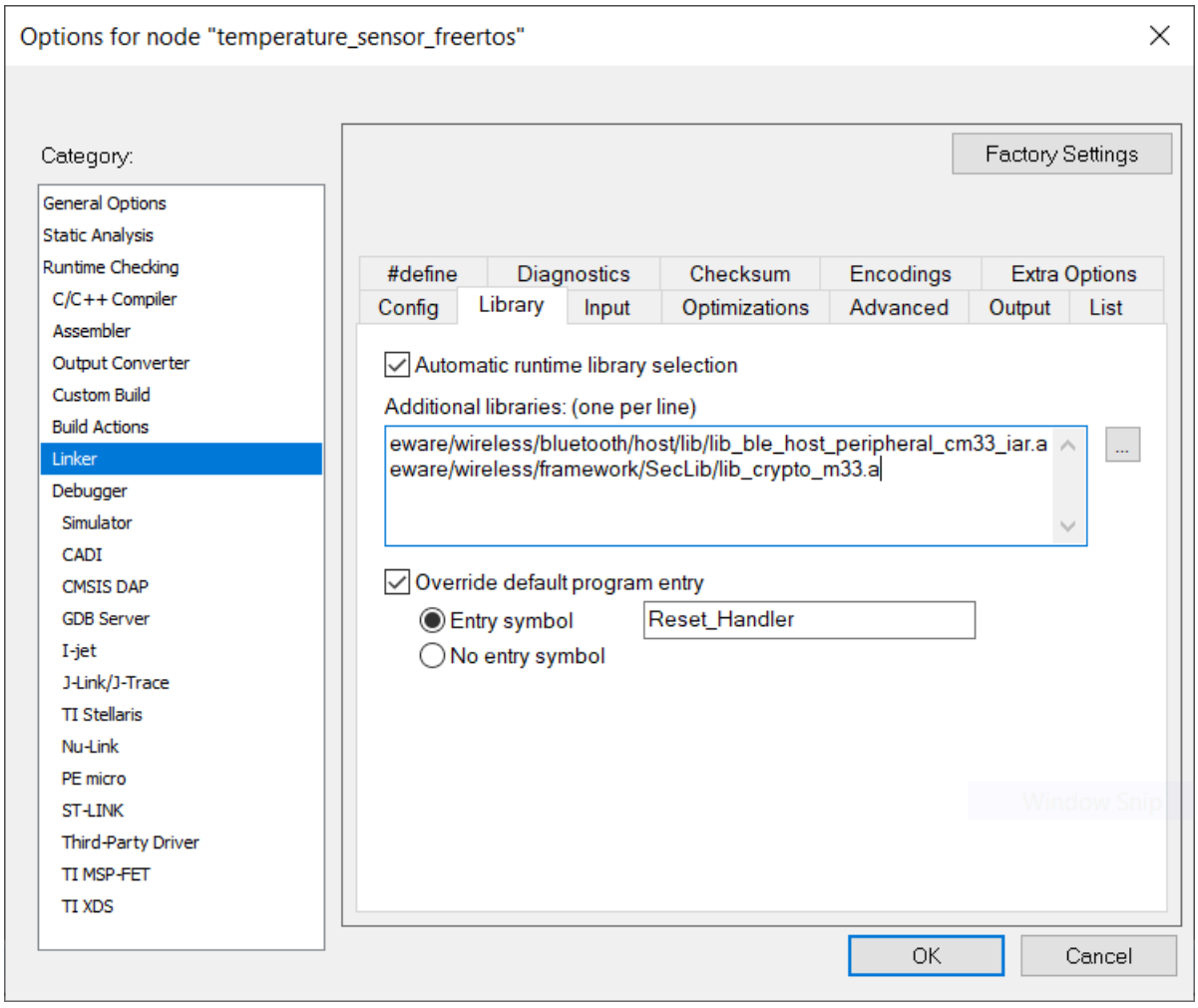

Figure 14. Linker configuration for Temperature Sensor

9.9 Profile configuration

The implemented profiles and services are located in *middleware/wireless/bluetooth/profiles* folder. The application links every service source file and interface it needs to implement the profile. For example, for the Temperature Sensor the tree looks as shown Figure 15:

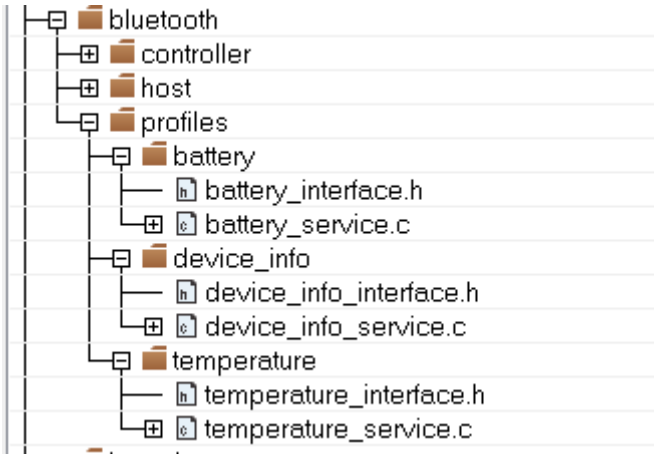

Figure 15. Temperature Sensor profile-related files

The Temperature Profile implements the custom Temperature service, the Battery, and Device Information services.

9.9.1 Application code

The application folder contains the following modules:

- *app.c* and *app.h*. This module stores the application-specific functionality (APIs for specific triggers, handling of peripherals, callbacks from the stack, handling of low power, and so on).

Before initializing the Bluetooth LE Host Stack, the start task calls *BluetoothLEHost_AppInit*. This function initializes application specific functionality before initializing the Bluetooth LE Host Stack by calling *BluetoothLEHost_Init*.

After the stack is initialized, the *BluetoothLEHost_Initialized* callback is called. The function contains configurations made to the Bluetooth LE Host Stack after the initialization. This includes registering callbacks, setting security for services, starting services, allocating timers, adding devices to the Filter Accept List, and so on. For example, the Temperature Sensor configures the following:

```
static void BluetoothLEHost_Initialized(void)
{
  /* Common GAP configuration */
    BleConnManager_GapCommonConfig();

  /* Register for callbacks*/
    (void)App_RegisterGattServerCallback(BleApp_GattServerCallback);

    mAdvState.advOn = FALSE;

  /* Start services */
    SENSORS_TriggerTemperatureMeasurement();
    (void)SENSORS_RefreshTemperatureValue();
    /* Multiply temperature value by 10. SENSORS_GetTemperature() reports
 temperature
    value in tenths of degrees Celsius. Temperature characteristic value is
 degrees
    Celsius with a resolution of 0.01 degrees Celsius (GATT Specification
    Supplement v6). */
    tmsServiceConfig.initialTemperature = (int16_t)(10 *
 SENSORS_GetTemperature());
    (void)Tms_Start(&tmsServiceConfig);
```

```
    basServiceConfig.batteryLevel = SENSORS_GetBatteryLevel();
    (void)Bas_Start(&basServiceConfig);
    (void)Dis_Start(&disServiceConfig);

  /* Allocate application timer */
    (void)TM_Open(appTimerId);

    AppPrintString("\r\nTemperature sensor -> Press switch to start advertising.
\r\n");
}
```

To start the application functionality, `BleApp_Start()` function is called. This function usually contains code to start advertising for sensor nodes or scanning for central devices. In the example of the Temperature Sensor, the function is the following:

```
static void BleApp_Start(void)
{
    Led1On();

    if (mPeerDeviceId == gInvalidDeviceId_c)
    {
      /* Device is not connected and not advertising */
        if (!mAdvState.advOn)
        {
           /* Set advertising parameters, advertising to start on
 gAdvertisingParametersSetupComplete_c */
           BleApp_Advertise();
        }
    }
    else
    {
        /* Device is connected, send temperature value */
        BleApp_SendTemperature();
    }
}
```

- *app_config.c.* This file contains data structures that are used to configure the stack.

This includes advertising data, scanning data, connection parameters, advertising parameters, SMP keys, security requirements, and so on.

- *app_preinclude.h*.

This header file contains macros to override the default configuration of any module in the application. It is added as a preinclude file in the preprocessor command line in IAR, as shown in Figure 16:

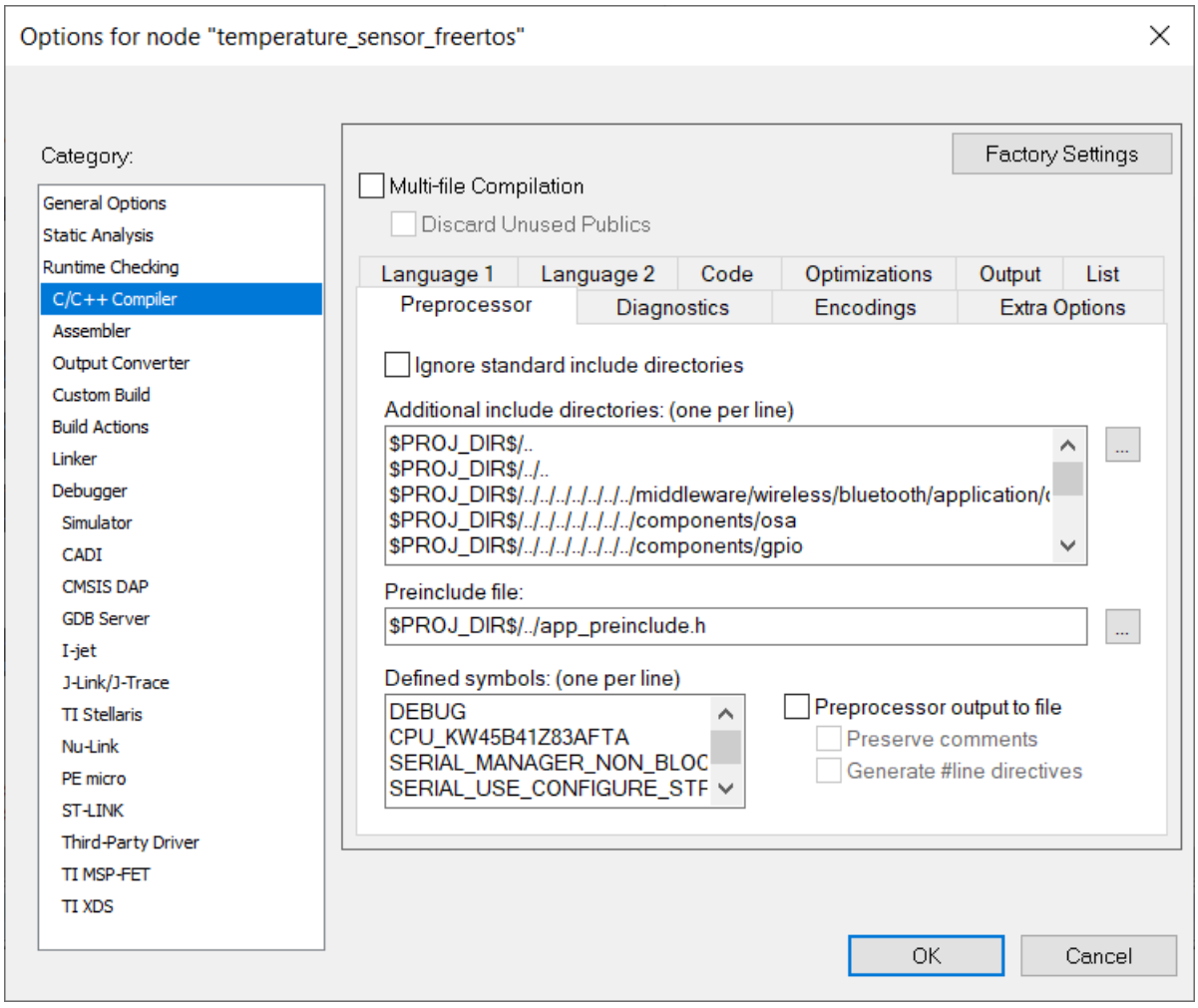

Figure 16. Preinclude file

- *gatt_db.h* and *gatt_uuid128.h*. The two header files contain the definition of the GATT database and the custom UUIDs used by the application. See Section 7 "Creating GATT database" for more information.

9.10 Multiple connections

Applications can be configured to support multiple connections. To allow multiple connections, the *gAppMaxConnections_c* must be set to a value up to the maximum number of connections (*this value is chip-specific*). Refer to the chip documentation for the supported number of connections.

The application can save information about the peer devices it connects to according to the value of *gAppMaxConnections_c*. The Bluetooth Low Energy profile associated with the application use case must be instantiated to support the use of its functionality for each peer device. When handling multiple connections, the applications can behave as either the GAP central, GAP peripheral, or both at the same time. It is up to the application code to decide whether to start the advertising or scanning before creating the next connection. The supported combinations enable a device to connect as a peripheral to multiple centrals, as a central to multiple peripherals, or for it to be a central for some peers and a peripheral to others. The demo applications provide this functionality as an example of exercising multiple connection support. In such applications, the GAP role can be changed from central to peripheral and the information is saved for each peer device.

9.11 Bluetooth address generation

BD_ADDR is represented by 48 bits that uniquely identify a device and consist of a 24-bit OUI (Organizationally Unique Identifier) and a 24-bit random part that varies between devices. There are multiple options of storing and using the BD_ADDR. Depending on the chip, it may be read from a device specific register (if supported), from the global hardware parameters stored in the flash, or generated randomly based on the processor-unique identifier. The demo applications provide a combination of the last two options. The *Ble_SetBDAddr* function is called during the initialization process, after initializing HCI and before initializing the Bluetooth LE Host Stack. The global hardware parameters are read from the flash. If a useful value is found, it is used as the BD address. If the found value is all 0xFFs, an address is generated by concatenating the OUI configured at compile time with three randomly generated bytes. The result is stored in the hardware parameters for future use and then set into the Controller. The Bluetooth LE Host Stack uses little-endian format to represent all addresses, in compliance with the Bluetooth Core Specification.

9.12 Repeated attempts

Applications can be configured to enable protection against repeated Pairing Requests/Peripheral Security Requests coming from the same device. This is to prevent an intruder from repeating the pairing process with a large number of different keys in order to extract information about the local device's private key. If this feature is enabled, after a pairing procedure fails, another attempt to pair coming from the same device is allowed only after a specific time period has passed. For each failure, the waiting period doubles up until a maximum period.

The following *app_preinclude.h* macros support this feature:

- `gRepeatedAttempts_d`
  - Set to 1 to enable the feature. By default, it is disabled (0).
- `gRepeatedAttemptsNoOfDevices_c`
  - Number of remote devices to keep track of. By default, the value is 4.
  - If a new device needs to be added and the list is full, one of the oldest entries will be replaced.
- `gRepeatedAttemptsTimeoutMin_c`
  - Minimum waiting period in seconds – default 10.
- `gRepeatedAttemptsTimeoutMax_c`
  - Maximum waiting period in seconds – default 640. The waiting period doubles after each failed pairing with the same device.

9.13 Advanced Secure Mode (kw45_k32w)

This section describes the advanced security capabilities of the Bluetooth LE Host Stack which are available on the KW45/K32W1 platform via the EdgeLock Secure Enclave (ELKE).

The security capabilities are enabled at application, Host and Controller level by setting Advanced Secure Mode to active. To do this, the user must set the *gAppSecureMode_d* macro to *1* in the project's *app_preinclude.h* file. This macro is defined by default as *0* in *app_preinclude_common.h*:

```
#if (gAppSecureMode_d == 1U)
#define gSecLibSssUseEncryptedKeys_d     (1U)
#define gHostSecureMode_d                (1U)
#else
#define gHostSecureMode_d                (0U)
#endif
```

At application level, when Advanced Secure Mode is enabled, the security mode and level for pairing is automatically enforced as Mode1 Level 4, ensuring LE Secure Connections pairing. Legacy pairing is not supported in this mode.

When enabled, the main benefit of Advanced Secure Mode is the secured storage and handling of Bluetooth LE security keys. The EdgeLock Secure Enclave is capable of generating, importing, and exporting security keys as plain text or as encrypted blobs. All encrypted blobs are created by the EdgeLock Secure Enclave using a die unique key, which makes them impossible to decrypt by devices other than the one that created them. The Bluetooth LE security keys are managed in Advanced Secure Mode as follows:

- IRK
  - The peer IRKs received after pairing and bonding are no longer stored into NVM as plaintext, but as encrypted blobs 40 bytes in length (or ELKE blobs). They can still be retrieved and converted to plaintext.
  - The local IRK is no longer generated using the default method of hashing over the board's UID at every startup. It is instead generated once using the EdgeLock Secure Enclave and stored into a new NVM dataset as an ELKE blob.
  - Local and peer IRKs are no longer transmitted through HCI in plaintext but as EIRK (Encrypted IRK) blobs, 16 bytes in length, which can be decrypted by the Controller.
- LTK
  - The LTK is no longer stored into NVM as plaintext, but as an ELKE blob. Furthermore, the plaintext of the LTK is never available to the Host/application. Generating the LTK via the ECDH process and generating the Session Key for individual connections is done via the EdgeLock Secure Enclave and custom vendor HCI messages which are transparent to the application.
- CSRK
  - The local CSRK is no longer generated using the default method of hashing over the board's UID at every startup. It is instead generated once using the EdgeLock Secure Enclave and stored into a new NVM dataset as an ELKE blob.

At startup, Advanced Secure Mode for the Controller is enabled dynamically by calling:

```
#if (defined(gAppSecureMode_d) && (gAppSecureMode_d > 0U))
    (void) PLATFORM_EnableBleSecureKeyManagement();
#endif
```

This call can be found in BluetoothLEHost_Init, as part of the initialization sequence.

10 Low-Power Management

10.1 System considerations

The KW45/K32W1 has a dual-core architecture and has two separated power domains:

- The main domain for the Cortex M33
- The Radio domain which comprises the Cortex M3 core and the NBU (Narrow Band Unit).

The two power domains can go into or exit independently different low-power modes, namely Wait for instruction (WFI), Deep-sleep mode, Power-down mode, and Deep Power-down mode.

In Wait For Interrupt (WFI) mode, the CPU core is powered ON but is in an idle mode with the clock turned OFF.

In Deep Sleep mode, the fast clock is turned off, and the CPU along with the main power domain are placed into a retention state, with the voltage being scaled down to support state retention only. As no high frequency clock runs in this mode, the voltage applied on the power domain can be reduced to reduce leakage on the hardware logic. This reduces the overall power consumption in the Deep Sleep mode. When waking up from Deep Sleep mode, the core voltage is increased back to nominal voltage, the fast clock (FRO) is turned back on, and the peripheral in this domain can be reused as normal.

In Power-down mode, both the clock, and power are shut off to the CPU and the main peripheral domain. SRAM is retained, but register values are lost. The SDK power manager handles the restore of the processor registers and dependencies such as interrupt controller and similar ones transparently from the application.

In Deep Power-down mode the SRAM is not retained. This is the lowest power mode available. It is exited through the reset sequence.

10.2 When/how to enter low power

To enable low power at application level, the `gAppLowpowerEnabled_d` define should be set to `1` in `app_preinclude.h` file.

The system should enter low power when the entire system is idle, and all software layers agree on that. The device enters low power by calling the `PWR_EnterLowPower` function.

For FreeRTOS applications, the low-power entry point is placed in the FreeRTOS idle task, which has the lowest priority in the system. From that task, the `vPortSuppressTicksAndSleep` function is called, which at its turn, calls the `PWR_EnterLowPower` to enter low power.

For the BareMetal examples, the application low power entry point is placed in the main function and is called when there are no messages to be processed by other tasks.

The wake-up sources that can be configured for the application are UART or button. Note that Low-power timer wake-up source and wake-up from the Radio domain are directly enabled from the Connectivity framework.

Each software layer/entity running on the system can prevent it from entering low power by calling *PWR_LowPowerEnterCritical* function. The system stays awake until all software layers that called *PWR_LowPowerEnterCritical* call back *PWR_LowPowerExitCritical* and the system reaches the low-power entry point.

When going to low power, the SDK Power Manager selects the best low-power mode that fits all the constraints.

The default low-power mode for each application is Deep-sleep mode. Users can change the behavior by setting a new low-power constraint for the application.

For example, if the low power constraint set from the application is Deep-sleep mode, and no other constraint is set, the **SDK Power Manager** selects Deep-sleep the next time the device enters low power. However, there

might be a case when a WFI constraint is set (*PWR_SetLowPowerModeConstraint (PWR_WFI)*) by another component, such as the SecLib module that operates Hardware encryption. In such cases, the **SDK Power Manager** selects this WFI mode until the constraint is released by the SecLib module (*PWR_ReleaseLowPower ModeConstraint(PWR_WFI)*).

If it is required to change the mode from Deep-sleep to Power-down mode, the deep sleep constraint (*PWR_ ReleaseLowPowerModeConstraint(PWR_DeepSleep)*) must be released and the power-down constraint (*PWR_SetLowPowerModeConstraints(PWR_PowerDown)*) must be set. The **SDK Power Manager** selects the Power-down mode when the device enters low power.

11 Over the Air Programming (OTAP)

This chapter contains a detailed description of the Over The Air Programming capabilities of the Bluetooth Low Energy Host Stack enabled by dedicated GATT Service/Profile and of the support modules needed for OTA programming.

The image transfer is done using a dedicated protocol which is designed to run on both the Bluetooth Low Energy transport and serial transport.

The container for the upgrade image is an image file which has a predefined format which is described in detail. The image file format is independent of the protocol but must contain information specific to the image upgrade infrastructure on an OTAP Client device. Detailed information on how to build an image file starting from a generic format executable generated by an embedded cross-compiling toolchain is shown.

The demo applications implement a typical scenario where a new image is sent from a PC via serial interface to a Bluetooth Low Energy OTAP Server and then over the air to an OTAP Client which is the target of the upgrade image. There are 3 applications involved in the OTAP demo: 1 PC application which builds the image file and serves it to the embedded OTAP Server and 2 embedded applications (OTAP Server and OTAP Client). This chapter contains enough information for building Bluetooth Low Energy OTAP applications which implement different image upgrade scenarios specific to other use cases.

11.1 General functionality

A Bluetooth Low Energy OTAP system consists of an OTAP Server and an OTAP Client which exchange an image file over the air using the infrastructure provided by Bluetooth Low Energy (GAP, GATT, SM) via a custom GATT Service and GATT Profile. Additionally, a third application may be used to serve an image to the embedded OTAP Server.

The OTAP Server runs on the GATT Client via the Bluetooth Low Energy OTAP Profile and the OTAP Client runs on the GATT Server via the Bluetooth Low Energy OTAP Service. For the moment the OTAP Server runs on the GAP Central and the OTAP Client runs on the GAP Peripheral.

The Figure 17 shows a typical image upgrade scenario.

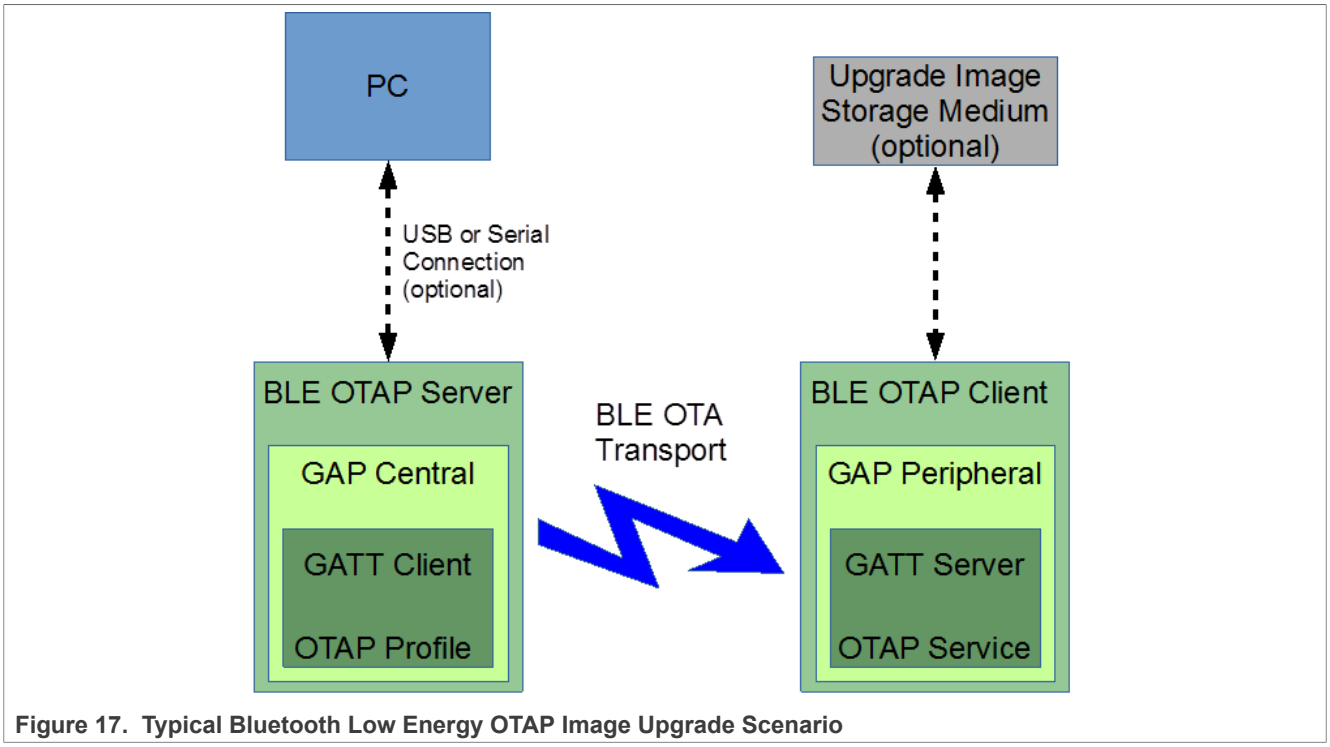

Figure 17. Typical Bluetooth Low Energy OTAP Image Upgrade Scenario

11.2 Bluetooth Low Energy OTAP service-profile

The Bluetooth Low Energy OTAP Service is implemented using the GATT Server which runs on the OTAP Client (GAP Peripheral).

The Bluetooth LE OTAP Service does not require any other Bluetooth LE services because it is a custom service it has a 128-bit UUID. The service has 2 custom characteristics which also have 128-bit UUIDs.

The service must be included in the GATT database of the GATT Server as described in Section 7 "Creating GATT database" of this document.

11.2.1 OTAP service and characteristics

The OTAP Service has a custom 128-bit UUID which is shown below. The UUID is based on a base 128-bit UUID used for Bluetooth LE custom services and characteristics. These are shown in the tables below.

Table 9. Base Bluetooth LE 128-bit UUID

Service	128-bit UUID
Base Bluetooth Low Energy	00000000 -ba5e-f4ee-5ca1-eb1e5e4b1ce0

The OTAP Service custom 128-bit UUID is built using the base UUID by replacing the most significant 4 bytes which are 0 with a value specific to the OTAP Service which is 01FF5550 in hexadecimal format.

Table 10. Bluetooth LE Service UUID

Service	UUID (128-bit)
Bluetooth LE OTAP Service	01ff5550 -ba5e-f4ee-5ca1-eb1e5e4b1ce0

The Bluetooth LE OTAP Service Characteristics UUIDs are built the same as the Bluetooth LE OTAP Service UUID starting from the base 128-bit UUID but using other values for the most significant 4 bytes.

Table 11. Bluetooth LE OTAP Service Characteristics

Characteristic	UUID (128-bit)	Properties	Descriptors
Bluetooth LE OTAP Control Point	01ff5551 -ba5e-f4ee-5ca1-eb1e5e4b1ce0	Write, Indicate	CCC
Bluetooth LE OTAP Data	01ff5552 -ba5e-f4ee-5ca1-eb1e5e4b1ce0	Write Without Response	-

Both characteristics are implemented as variable length characteristics.

The Bluetooth LE OTAP Control Point Characteristic is used for exchanging OTAP commands between the OTAP Server and the OTAP Client. The OTAP Client sends commands to the OTAP Server using ATT Notifications for this characteristic and the OTAP Server sends commands to the OTAP Client by making ATT Write Requests to this characteristic. Both ATT Writes and ATT Notifications are acknowledged operations via ATT Write Responses and ATT Confirmations.

The Bluetooth LE OTAP Data characteristic is used by the OTAP Server to send parts of the OTAP image file to the OTAP Client when the ATT transfer method is chosen by the application. The ATT Write Commands (GATT Write Without Response operation) is not an acknowledged operation.

The Bluetooth LE OTAP service and characteristics 128-bit UUIDs are defined in the *gatt_uuid128.h* just as shown below.

```
UUID128(uuid_service_otap, 0xE0, 0x1C, 0x4B, 0x5E, 0x1E, 0xEB, 0xA1, 0x5C, 0xEE, 0xF4, 0x5E, 0xBA,
 0x50, 0x55, 0xFF, 0x01)
UUID128(uuid_char_otap_control_point, 0xE0, 0x1C, 0x4B, 0x5E, 0x1E, 0xEB, 0xA1, 0x5C, 0xEE, 0xF4,
 0x5E, 0xBA, 0x51, 0x55, 0xFF, 0x01)
UUID128(uuid_char_otap_data, 0xE0, 0x1C, 0x4B, 0x5E, 0x1E, 0xEB, 0xA1, 0x5C, 0xEE, 0xF4, 0x5E, 0xBA,
 0x52, 0x55, 0xFF, 0x01)
```

The service is included into the GATT database of the device. It is declared in the *gatt_db.h* file as shown below.

```
PRIMARY_SERVICE_UUID128(service_otap, uuid_service_otap)
CHARACTERISTIC_UUID128(char_otap_control_point, uuid_char_otap_control_point, (gGattCharPropWrite_c
 | gGattCharPropIndicate_c))
VALUE_UUID128_VARLEN(value_otap_control_point, uuid_char_otap_control_point,
 (gPermissionFlagWritable_c), 16, 16, 0x00)
CCCD(cccd_otap_control_point)
CHARACTERISTIC_UUID128(char_otap_data, uuid_char_otap_data, (gGattCharPropWriteWithoutRsp_c))
VALUE_UUID128_VARLEN(value_otap_data, uuid_char_otap_data, (gPermissionFlagWritable_c), gAttMaxMtu_c
 - 3, gAttMaxMtu_c - 3, 0x00)
```

The Bluetooth LE OTAP Control Point characteristic should be large enough for the longest command which can be exchanged between the OTAP Server and The OTAP Client.

The Bluetooth LE OTAP Data characteristic should be large enough for the longest data chunk command the OTAP Client expects from the OTAP Server to be sent via ATT. The maximum length of the OTAP Data Characteristic value is ATT_MTU- 3. 1 byte is used for the ATT OpCode and 2 bytes are used for the Attribute Handle when performing a Write Without Response, the only operation permitted for this characteristic value.

11.2.2 OTAP server and OTAP client interactions

The OTAP Server application scans for devices advertising the OTAP Service. When it finds one it connects to that device and notifies it of the available image files or waits for requests regarding available image files. The behavior is specific to each application which needs the OTAP functionality. The Bluetooth LE OTAP Protocol described below details how to do this.

After an OTAP Server (GAP Central, GATT Client) connects to an OTAP Client (GAP Peripheral, GATT Server) it scans the device database and identifies the handles of the OTAP Control Point and OTAP Data characteristics and their descriptors. Then it writes the CCC Descriptor of the OTAP Control point to allow the

OTAP Client to send it commands via ATT Indications. It can send commands to the OTAP Client by using ATT Write Commands to the OTAP Control Point characteristic.

After the connection is established, if the OTAP Client wants to use the L2CAP CoC transfer method it must register a L2CAP PSM with the OTAP Server.

The OTAP Client only starts any image information request or image transfer request procedures only after the OTAP Server writes the OTAP Control Point CCCD to ensure there is bidirectional communication between the devices.

11.3 Bluetooth LE OTAP protocol

The protocol consists of a set of commands (messages) which allow the OTAP Client to request or be notified about the available images on an OTAP Server and to request the transfer of parts of images from the OTAP Server.

All commands with the exception of the image data transfer commands are exchanged through the OTAP Control Point characteristic of the OTAP Service. The data transfer commands are sent only from the OTAP Server to the OTAP Client either via the OTAP Data characteristic of the OTAP Service or via a dedicated Credit Based Channel assigned to a L2CAP PSM.

11.3.1 Protocol design considerations

The OTAP Client is a GAP Peripheral device, and therefore has limited resources. This is why the OTAP Protocol was designed in such a way that it is at the discretion of the OTAP Client if, when, how fast and how much of an available upgrade image is transferred from the OTAP Server. The OTAP Client also decides which is the image transfer method based on its capabilities. Two image transfer methods are supported at this moment: the ATT Transfer Method and the L2CAP PSM CoC Transfer Method.

The ATT Transfer Method is supported by all devices which support Bluetooth LE but it has the disadvantage of a small data payload size and a larger Bluetooth LE stack protocols overhead leading to a lower throughput. This disadvantage has been somewhat reduced by the introduction of the Long Frames feature in the Bluetooth LE specification 4.2 Link Layer which allows for a larger ATT_MTU value. The L2CAP PSM CoC Transfer Method is an optional feature available for devices running a Bluetooth stack version 4.1 and later. The protocol overhead is smaller and the data payload is higher leading to a high throughput. The L2CAP PSM Transfer Method is the preferred transfer method and it is available on all Bluetooth LE Devices if the application requires it.

Based on application requirements and device resources and capabilities the OTAP Clients can request blocks of OTAP images divided into chunks. To minimize the protocol overhead and maximize throughput an OTAP Client makes a data block request specifying the block size and the chunk size and the OTAP Server sends the requested data chunks (which have a sequence number) without waiting for confirmation. The block size, chunk size and number of chunks per block are limited and suitable values must be used based on application needs.

The OTAP Client can stop or restart an image block transfer at any time if the application requires it or a transfer error occurs. The OTAP Server implementation can be almost completely stateless. The OTAP Server does not need to remember what parts of an image have been transferred, this is the job of the OTAP Client which can request any part of an image at any time. This allows it to download parts of the image whenever and how fast its resources allow it. The OTAP Server simply sends image information and image parts on request.

The Bluetooth LE OTAP Protocol is designed to be used not only on Bluetooth LE transport medium but on any transport medium, for example a serial communication interface or another type of wireless interface. This may be useful when transferring an upgrade image from a PC or a mobile device to the OTAP Server to be sent via Bluetooth LE to the OTAP Clients which require it. In the OTAP Demo Applications the embedded OTAP Server relays OTAP commands between an OTAP Client and a PC via a serial interface and using a FSCI type protocol. Effectively the OTAP Client downloads the upgrade image from the PC and not from the OTAP Server. Other transfer methods may be used based on application needs.

11.3.2 Bluetooth Low Energy OTAP commands

The Bluetooth LE OTAP Commands general format is shown below. A command consists of two parts, a Command ID, and a Command Payload as shown in the table below.

Table 12. Bluetooth LE OTAP General Command Format

Field Name	CmdId	CmdPayload
Size (Bytes)	1	variable

Commands are sent over the transport medium starting with the Command ID and continuing with the Command Payload.

All multibyte command parameters in the Command Payload are sent in a least significant octet first order (little endian).

A summary of the commands supported by the Bluetooth LE OTAP Protocol is shown in the table below. Each of the commands is then detailed in its own section.

Table 13. Bluetooth LE OTAP Commands Summary

CmdId	Command Name
0x01	New Image Notification
0x02	New Image Info Request
0x03	New Image Info Response
0x04	Image Block Request
0x05	Image Chunk
0x06	Image Transfer Complete
0x07	Error Notification
0x08	Stop Image Transfer

11.3.2.1 New image notification command

This command can be sent by an OTAP Server to an OTAP Client, usually immediately after the first connection, to notify the OTAP Client of the available images on the OTAP Server.

Table 14. New Image Notification Command Parameters

CmdId	Name	Dir	Parameters	Param Size (Bytes)	Description	Total Size (CmdId +Payload)
0x01	New Image Notification	S->C	ImageId	2	Short image identifier used for transactions between the OTAP Server and OTAP Client. Should be unique for all images on a server.	15
			ImageVersion	8	Image file version. Contains sufficient information to identify the target hardware, stack version and build version.	
			ImageFileSize	4	Image file size in bytes.	

The *ImageId* parameter should not be '0x0000', which is the reserved value for the current running image or 0xFFFF, which is the reserved value for "no image available".

11.3.2.2 New image info request command

This command can be sent by an OTAP Client to an OTAP Server to inquire about available upgrade images on the OTAP Server.

Table 15. New Image Info Request Command Parameters

CmdId	Name	Dir	Parameters	Param Size (Bytes)	Description	Total Size (CmdId +Payload)
0x02	New Image Info Request	C->S	CurrImageId	2	Id of the currently running image. Should be 0x0000.	11
			CurrImageVer	8	Version of the currently running image. A value of all zeroes signals that the client is looking for all images available on an OTAP Server. A value of all	

Table 15. New Image Info Request Command Parameters*...continued*

CmdId	Name	Dir	Parameters	Param Size (Bytes)	Description	Total Size (CmdId +Payload)
					zeroes requests information about all images on the server.	

The *CurrImageId* parameter should be set to 0x0000 to signify the current running image.

The *CurrImageVer* parameter should contain sufficient information about the target device for the OTAP Server to determine if it has an upgrade image available for the requesting OTAP Client.

A value of all zeroes for the *CurrImageVer* means that an OTAP Client is requesting information about all available images on an OTAP Server and the OTAP Server should send a New Image Info Response for each image.

11.3.2.3 New image info response command

This command is sent by the OTAP Server to the OTAP Client as a response to a New Image Information Request Command.

Table 16. New Image Info Response Command Parameters

CmdId	Name	Dir	Parameters	Param Size (Bytes)	Description	Total Size (CmdId +Payload)
0x03	New Image Info Response	S->C	ImageId	2	Image Id. Value 0xFFFF is reserved as "no image available"	15
			ImageVersion	8	Image file version.	
			ImageFileSize	4	Image file size.	

The *ImageId* parameter with a value of 0xFFFF is reserved for the situation where no upgrade image is available for the requesting device.

11.3.2.4 Image block request command

This command is sent by the OTAP Client to the OTAP Server to request a part of the upgrade image after it has determined the OTAP Server has an upgrade image available.

When an OTAP Server Receives this command it should stop any image file chunk transfer sequences in progress.

Table 17. Image Block Request Command Parameters

CmdId	Name	Dir	Parameters	Param Size (Bytes)	Description	Total Size (CmdId +Payload)
0x04	Image Block Request	C->S	ImageId	2	Image Id	16

Table 17. Image Block Request Command Parameters*...continued*

CmdId	Name	Dir	Parameters	Param Size (Bytes)	Description	Total Size (CmdId +Payload)
			StartPosition	4	Start position of the image block to be transferred.	
			BlockSize	4	Requested total block size in bytes.	
			ChunkSize	2	Should be optimized to the Transfer Channel type. The maximum number of chunks per block is 256. Value is in bytes.	
			TransferMethod	1	0x00 - ATT 0x01 – L2CAP PSM Credit based channel	
			L2capChannelOrPsm	2	0x0004 - ATT Other values – PSM for credit based channels	

The *ImageId* parameter contains the ID of the upgrade image.

The *StartPosition* parameter specifies the location in the image upgrade file at which the requested block starts.

The *BlockSize* and *ChunkSize* parameters specify the size in bytes of the block to be transferred and the size of the chunks into which a block is separated. The *ChunkSize* value must be chosen in such a way that the total number of chunks can be represented by the *SeqNumber* parameter of the Image Chunk Command. At the moment this parameter is 1 byte in size so there are a maximum of 256 chunks per block. The chunk sequence number goes from 0 to 255 (0x00 to 0xFF). If this condition is not met or the requested block is not entirely into the image file bounds an error is sent to the OTAP Client when the OTAP Server receives this misconfigured Image Block Request Command.

The maximum value of the *ChunkSize* parameter depends on the maximum ATT_MTU and L2CAP_MTU supported by the Bluetooth LE stack version and implementation.

The *TransferMethod* parameter is used to select the transfer method which can be ATT or L2CAP PSM CoC. The *L2capChannelOrPsm* parameter must contain the value 0x0004 for the ATT transfer method and another value representing the chosen PSM for the L2CAP PSM transfer method. The default PSM for the Bluetooth LE OTAP demo applications is 0x004F for both the OTAP Server and the OTAP Client although the specification allows different values at the 2 ends of the L2CAP PSM connection. The PSM must be in the range reserved by the Bluetooth specification which is 0x0040 to 0x007F.

The optimal value of the *ChunkSize* parameter depends on the chosen transfer method and the Link Layer payload size. Ideally it must be chosen in such a way that full packets are sent for every chunk in the block.

The default Link Layer payload is 27 bytes form which we subtract 4 for the L2CAP layer and 3 for the ATT layer (1 for the ATT Cmd Opcode and 2 for the Handle) leaving us with a 20 byte OTAP protocol payload. From these 20 bytes we subtract 1 for the OTAP CmdId and 1 for the chunk sequence number leaving us with an optimum chunk size of 18 for the ATT transfer method – which is the default in the demo applications. For the L2CAP PSM transfer method the chosen default chunk size is 111. This was chosen so as a chunk fits exactly 5 link layer packets. The default L2CAP payload of 23 (27 - 4) multiplied by 5 gives us 115 from which we subtract 2 bytes for the SDU Length (which is only sent in the first packet), 1 byte for the OTAP CmdId and 1 byte for the chunk sequence number which leaves exactly 111 bytes for the actual payload.

If the Link layer supports Long Frames feature then the chunk size should be set according to the negotiated ATT MTU for the ATT transfer method. From the negotiated ATT MTU (*att_mtu*) subtract 3 bytes for the ATT layer (1 for the ATT Cmd Opcode and 2 for the Handle) then subtract 2 bytes for the OTAP protocol (1 for the CmdId and 1 for the chunk sequence number) to determine the optimum chunk size (*optimum_att_chunk_size = att_mtu – 3 – 2)*. For the L2CAP PSM transfer method the chunk size can be set based on the maximum L2CAP SDU size (*max_l2cap_sdu_size*) from which 4 bytes should be subtracted, 2 for the SDU Length and 2 for the OTAP protocol (*optimum_l2cap_chunk_size = max_l2cap_sdu_size – 3 – 2*). In some particular cases reducing the L2CAP chunk size could lead to better performance. If the L2CAP chunk size needs to be reduced it should be reduced so it fits exactly a number of link layer packets. An example of how to compute an optimal reduced L2CAP chunk size is given in the previous paragraph.

11.3.2.5 Image chunk command

One or more Image Chunk Commands are sent from the OTAP Server to the OTAP Client after an Image Block Request is received by the former. The image chunks are sent via the ATT Write Without Response mechanism if the ATT transfer method is chosen and directly via L2CAP if the L2CAP PSM CoC transfer method is chosen.

Table 18. Image Chunk Command Parameters

CmdId	Name	Dir	Parameters	Param Size (Bytes)	Description	Total Size (CmdId +Payload)
0x05	Image Chunk	S->C	SeqNumber	1	In the range 0 -> BlockSize/ ChunkSize - calculated by Server, checked by Client. The command code is present even when ATT is used.	3 or more
			Data	variable	Actual data	

The *SeqNumber* parameter is the chunk sequence number and it has incremental values from 0 to 255 (0x00 to 0x FF) for a maximum of 256 chunks per block.

The *Data* parameter is an array containing the actual image part being transferred starting from the *BlockStartPosition* + *SeqNumber* * *ChunkSize* position in the image file and containing *ChunkSize* or less bytes depending on the position in the block. Only the last chunk in a block can have less than *ChunkSize* bytes in the Image Chunk Command data payload.

11.3.2.6 Image transfer complete command

This command is sent by the OTAP Client to the OTAP Server when an image file has been completely transferred and its integrity has been checked.

Table 19. Image Transfer Complete Command Parameters

CmdId	Name	Dir	Parameters	Param Size (Bytes)	Description	Total Size (CmdId +Payload)
0x06	Image Transfer Complete	C->S	ImageId	2	Image Id	4
			Status	1	Status of the image transfer. 0x00 - Success	

The *ImageId* parameter contains the ID of the image file that was transferred.

The *Status* parameter is 0x00 (Success) if image integrity and possibly other checks have been successfully made after the image is transferred and another value if integrity or other kind of errors have occurred.

If the status is 0x00 the OTAP Client can trigger the Bootloader to start flashing the new image. The image flashing should take about 15 seconds for a 160 KB flash memory.

11.3.2.7 Error notification command

This command can be sent by both the OTAP Server and the OTAP Client when an error of any kind occurs. When an OTAP Server Receives this command it should stop any image file chunk transfer sequences in progress.

Table 20. Error Notification Command Parameters

CmdId	Name	Dir	Parameters	Param Size (Bytes)	Description	Total Size (CmdId +Payload)
0x07	Error Notification	Bidir	CmdId	1	Id of the command which generated the error.	3
			ErrorStatus	1	Error Status: Examples: out of image bounds, chunk too small, chunk too large, image verification failure, bad command format, image not available, unknown command	

The *CmdId* parameter contains the ID of the command which caused the error (if applicable).

The *ErrorStatus* parameter contains the source of the error. All error statuses are defined in the *otapStatus_t* enumerated type in the *otap_interface.h* file.

11.3.2.8 Stop image transfer command

This command is sent from the OTAP Client to the OTAP Server whenever the former wants to stop the transfer of an image block which is currently in progress, or from OTAP Server to the OTAP Client when the image transfer is stopped from application (Test Tool).

Table 21. Stop Image Transfer Command Parameters

CmdId	Name	Dir	Parameters	Param Size (Bytes)	Description	Total Size (CmdId +Payload)
0x08	Stop Image Transfer	C->S	ImageId	2	Image Id	3

The *ImageId* parameter contains the ID of the image being transferred.

11.3.3 OTAP client–server interactions

The interactions between the OTAP Server and OTAP Client start immediately after the connection, discovery of the OTAP Service characteristics and writing of the OTAP Control Point CCC Descriptor by the OTAP Server.

The first command sent could be a New Image Notification sent by the OTAP Server to the OTAP Client or a New Image Info Request sent by the OTAP Client. The OTAP Server can respond with a New Image Info response if it has a new image for the device which sent the request (this can be determined from the *ImageVerison* parameter). The best strategy depends on application requirements.

After the OTAP Client has determined that the OTAP Sever has a newer image it can start downloading the image. This is done by Sending Image Block Request commands to retrieve parts of the image file. The OATP Server answers to these requests with one or more Image Chunk Commands via the requested transfer method or with an Error Notification if there are improper parameters in the Image Block Request. The OTAP Client makes as many Image Block Requests as it is necessary to transfer the entire image file.

The OTAP Client decides how often Image Block Request Commands are sent and can even stop a block transfer which is in progress via the Stop Image Transfer Command. The OTAP Client is in complete control of the image download process and can stop it and restart it at any time based on its resources and application requirements.

A typical **Bluetooth LE OTAP Image Transfer** scenario is shown in the message sequence chart Figure 18.

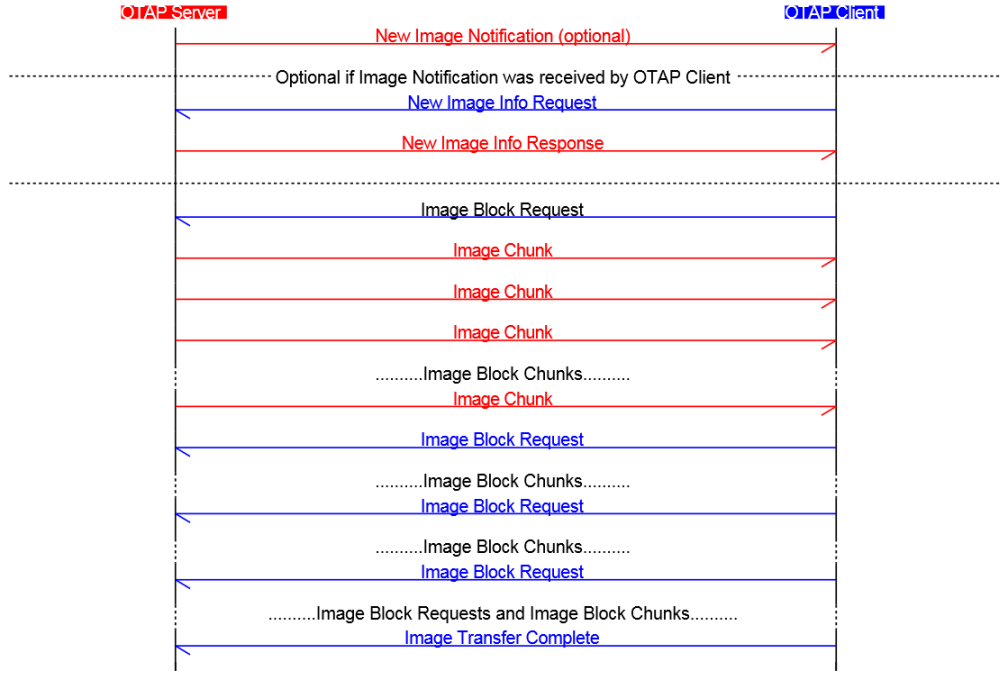

Figure 18. Typical Bluetooth LE OTAP Image Transfer Scenario Message Sequence Chart

11.4 Bluetooth Low Energy OTAP image file format

The Bluetooth LE OTAP Image file has a binary file format. It is composed of a header followed by a number of sub-elements. The header describes general information about the file. There are some predefined sub-elements of a file but an end manufacturer could add manufacturer-specific sub-elements. The header does not have details of the sub-elements. Each element is described by its type.

The general format of an image file is shown in the table below.

Table 22. Bluetooth LE OTAP Image File General Format

Image File Element	Value Field Length (bytes)	Description
Header	Variable	The header contains general information about the image file.
Upgrade Image Sub-element	Variable	This sub-element contains the actual binary executable image, which is copied into the flash memory of the target device. The maximum size of this sub-element depends on the target hardware.
Image File CRC Sub-element	2	This is a 16-bit CCITT type CRC which is calculated over all elements of the image file with the exception of the Image File CRC sub-element itself. This must be the last sub-element in an image file.

Each sub-element in a Bluetooth LE OTAP Image File has a Type-Length-Value (TLV) format. The type identifier provides forward and backward compatibility as new sub-elements are introduced. Existing devices that do not understand newer sub-elements may ignore the data.

The following table shows the general format of a Bluetooth LE Image File sub-element.

Table 23. Bluetooth LE OTAP Image File Sub-element Format

Subfield	Size (Bytes)	Format	Description
Type	2	uint16	Type Identifier – determines the format of the data contained in the value field
Length	4	uint32	Length of the *Value* field of the sub-element.
Value	variable	uint8[]	Data payload

Some sub-element type identifiers are reserved while others are left for manufacturer-specific use. The table below shows the reserved type identifiers and the manufacturer-specific ranges.

Table 24. Sub-element Type Identifiers Ranges

Type Identifiers	Description
0x0000	Upgrade Image
0x0001 – 0xefff	Reserved
0xf000 – 0xffff	Manufacturer-Specific Use

The OTAP Demo applications use two of the manufacturer-specific sub-element type identifiers while the rest remain free to use. The two are shown in the table below along with a short description.

Table 25. Manufacturer-specific Sub-element Type Identifiers Used by OTAP Demo Applications

Manufacturer-specific Type Identifiers	Sub-element Name	Notes
0xf000	Sector Bitmap	Bitmap that signals the bootloader the sectors of the internal flash, which should be overwritten and which should remain as is.
0xf100	Image File CRC	16-bit CRC that is computed over the image file with the exception of the CRC sub-element itself.

11.4.1 Bluetooth Low Energy OTAP header

The format and fields of the Bluetooth Low Energy OTAP Header are summarized in the table below.

Table 26. Bluetooth Low Energy OTAP Header Fields

Octets	Data Types	Field Name	Mandatory/Optional
4	Unsigned 32-bit integer	Upgrade File Identifier	M
2	Unsigned 16-bit integer	Header Version	M
2	Unsigned 16-bit integer	Header Length	M
2	Unsigned 16-bit integer	Header Field Control	M
2	Unsigned 16-bit integer	Company Identifier	M
2	Unsigned 16-bit integer	Image ID	M
8	8 byte array	Image Version	M
32	Character string	Header String	M
4	Unsigned 32-bit integer	Total Image File Size (including header)	M

The fields are shown in the order they are placed in memory from the first location to the last.

The total size of the header without the optional fields (if defined by the *Header Field Control*) is 58 bytes.

All the fields in the header have a little endian format with the exception of the *Header String* field which is an ASCII character string.

A packed structure type definition for the contents of the Bluetooth LE OTAP Header can be found in the *otap_interface.h* file.

11.4.1.1 Upgrade file identifier

Fixed value 4 byte field used to identify the file as being a Bluetooth LE OTAP Image File. The predefined value is "**0x0B1EF11E**".

11.4.1.2 Header version

This 2 byte field contains the major and minor version number. The high byte contains the major version and the low byte contains the minor version. The current value is "**0x0100**" with the major version "**01**" and the minor version **"00**". A change to the minor version means the OTA upgrade file format is still backward compatible, while a change to the major version suggests incompatibility.

11.4.1.3 Header length

Length of all the fields in the header including the ***Upgrade File Identifier*** field, ***Header Length*** field and all the optional fields. The value insulates existing software against new fields that may be added to the header. If new header fields added are not compatible with current running software, the implementations should process all fields they understand and then skip over any remaining bytes in the header to process the image or CRC sub-element. The value of the ***Header Length*** field depends on the value of the Header Field Control field, which dictates which optional header fields are included.

11.4.1.4 Header field control

This is a 2-byte bit mask that specifies the optional fields present in the OTAP Header.

In case no optional fields are defined, this whole field is reserved and should be set to "**0x0000**".

11.4.1.5 Company identifier

This is the company identifier assigned by the Bluetooth SIG. The Company Identifier used for the OTAP demo applications is "**0x01FF**".

11.4.1.6 Image ID

This is a unique short identifier for the image file. It is used to request parts of an image file. This number should be unique for all images available on a Bluetooth LE OTAP Server.

- The value 0x0000 is reserved for the current running image.
- The value 0xFFFF is reserved as a "no image available" code for New Image Info Response commands.

This field value must be used in the *ImageID* field in the *New Image Notification* and *New Image Info Response* commands.

11.4.1.7 Image version

This is the full identifier of the image file. It should allow a Bluetooth LE OTAP Client to identify the target hardware, stack version, image file build version, and other parameters if necessary. The recommended format of this field (which is used by the OTAP Demo applications) is shown below but an end device manufacturer could choose different format. The subfields are shown in the order they are placed in memory from the first location to the last. Each subfield has a little-endian format, if applicable. Refer Table 27

Table 27. Suggested Image Version Field Format

Subfield	Size (bytes)	Format	Description
Build Version	3	uint8[]	Image build version.
Stack Version	1	uint8	0x41 for example for Bluetooth Low Energy Stack version 4.1.
Hardware ID	3	uint8[]	Unique hardware identifier.
End Manufacturer Id	1	uint8	ID of the hardware–specific to the end manufacturer

This field value must be used in the *ImageVersion* field in the New Image Notification and New Image Info Response commands.

11.4.1.8 Header string

This is a manufacturer-specific string that may be used to store other necessary information as seen fit by each manufacturer. The idea is to have a human readable string that can prove helpful during the development cycle. The string is defined to occupy 32 bytes of space in the OTAP Header. The default string used for the Bluetooth LE OTAP demo application is "`BLE OTAP Demo Image File`".

11.4.1.9 Total image file size

The value represents the total image size in bytes. This is the total of data in bytes that is transferred over-the-air from the server to the client. In most cases, the total image size of an OTAP upgrade image file is the sum of the sizes of the OTAP Header and all the other sub-elements on the file. If the image contains any integrity and/or source identity verification fields then the **`Total Image File Size`** also includes the sizes of these fields.

11.5 Building Bluetooth Low Energy OTAP image file from SREC file

A SREC (Motorola S-record) file is an ASCII format file which contains binary information. Common file extensions are: `.srec`, `.s19`, `.s28`, `.s37` and others. Most modern compiler toolchains can generate an SREC format executable.

The steps described in this section enable the creation of a SREC file for your embedded application in IAR Embedded Workbench.

For this, open the target properties and go to the **Output Converter** tab. Activate the **Generate additional output** checkbox and choose the **Motorola** option from the **Output format** drop down menu. From the same pane you can also override the name of the output file. A screenshot of the described configuration is shown in Figure 19.

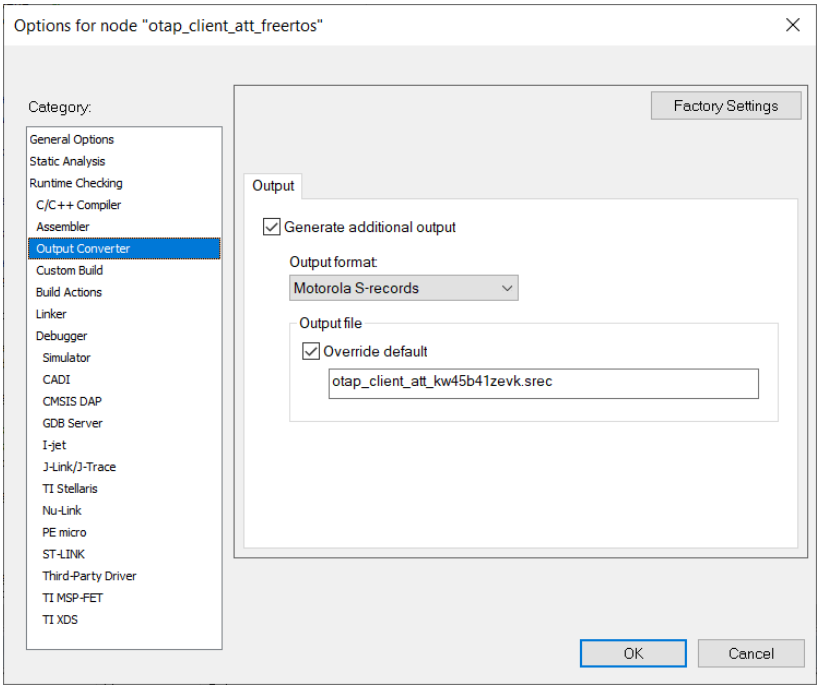

Figure 19. Enabling Options for Node "otap_client_att_freertos" in IAR Embedded Workbench

In MCUXpresso IDE, go to **Project properties** -> **Settings** -> **Build steps** window and press the "**Edit**" button for the Post-build steps. A Post-build steps window shows up in which the following command must be added:

```
arm-none-eabi-objcopy -v -O srec --only-section=.text --only-section=.data --only-
section=.ARM.exidx
"${BuildArtifactFileName}"
"${BuildArtifactFileBaseName}.srec"
```

A snapshot of this window is shown in the Figure 20.

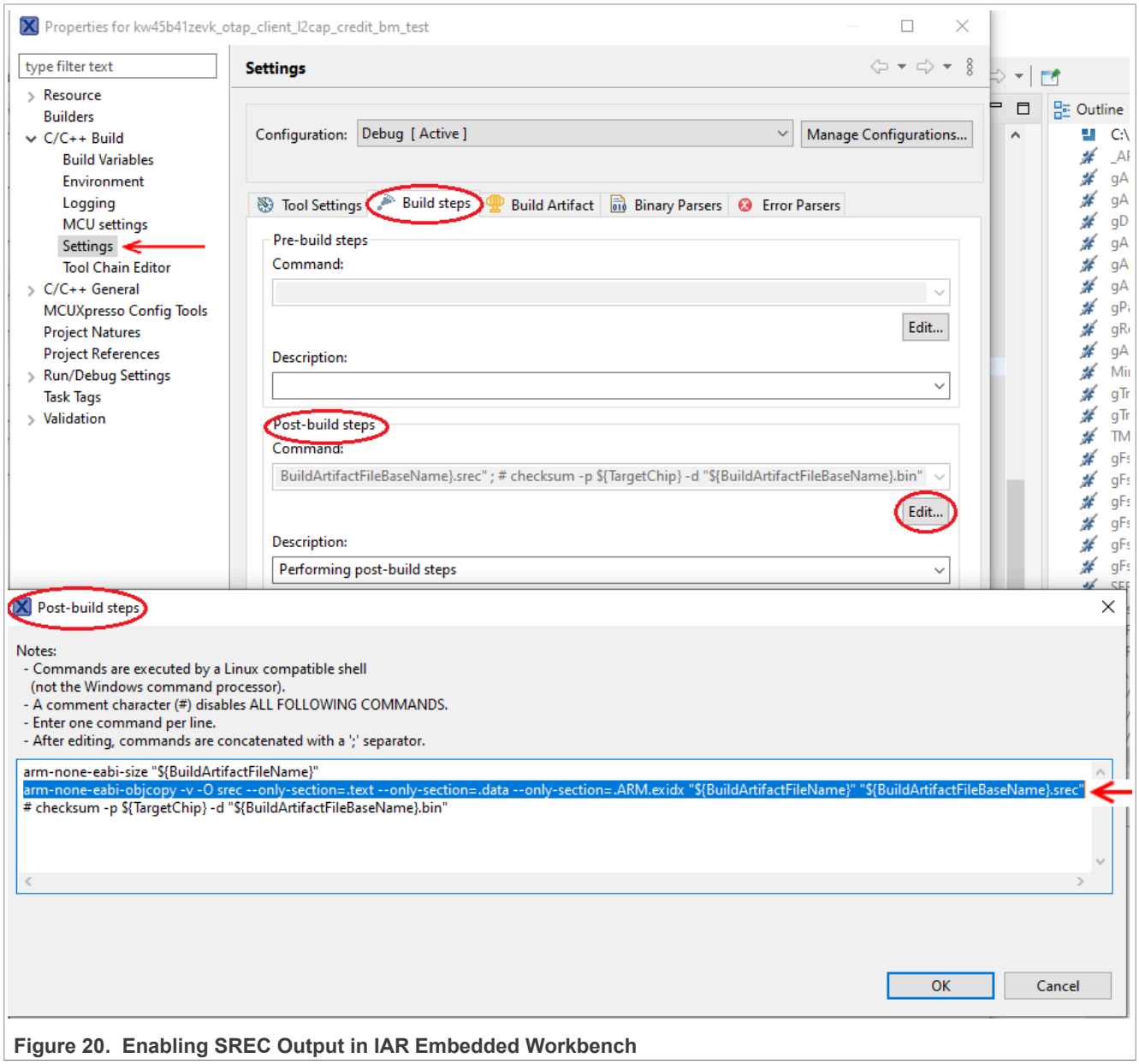

Figure 20. Enabling SREC Output in IAR Embedded Workbench

The format of the SREC file is shown in Table 28. It contains lines of text called records which have a specific format. An example of the contents of a SREC file is shown below.

```
S02000006F7461705F636C69656E745F6174745F4672656552544F532E73726563A1
S1130000F83F0020EB050000750600007506000075060000AF
S113001075060000750600007506000075060000F0
```

```
S1130020750600007506000075060000750600000E0
S1130030750600007506000075060000750600000D0
S113004000000000000000000000000000000000AC
S113005000000000000000000000000000000000009C
.............
S2140117900121380004F05FF8002866D12A003100E4
S2140117A06846008804F022F8A689002E16D0002884
S2140117B014D12569278801A868A11022F7F782FCB1
S2140117C06B4601AA0121380004F045F800284CD1E7
S2140117D02A0031006846008804F008F8A68A002E20
```

All records start with the ASCII letter 'S' followed by an ASCII digit from '0' to '9'. These two characters from the record type identify the format of the data field of the record.

The next 2 ASCII characters are 2 hex digits that indicate the number of bytes (hex digit pairs) which follow the rest of the record (address, data, and checksum).

The address that follows next can have 4, 6, or 8 ASCII hex digits, depending on the record type.

The data field is placed after the address and it contains 2 * n ASCII hex digits for 'n' bytes of actual data.

The last element of the S record is the checksum, which comprises 2 ASCII hex digits. The checksum is computed by adding all the bytes of the byte count, address, and data fields. Then the ones complement of the least significant octet of the sum is computed to determine the checksum.

Table 28. Format of an S Record

Field	Record Type	Count	Address	Data	Checksum	Line Terminator
Format	"Sn", n=0..9	ASCII hex digits	ASCII hex digits	ASCII hex digits	ASCII hex digits	"\r\n"
Length (characters)	2	2	4,6,8	*Count* –len*(Address)* – len(*Checksum)*	2	2

More details about the SREC file format can be found at this location: en.wikipedia.org/wiki/SREC_(file_format).

We are only interested in records that contain actual data. These are S1, S2, and S3 records. The other types of records can be ignored.

The S1, S2, and S3 records are used to build the Upgrade Image Sub-element of the image file simply by placing the record data at the location specified by the record address in the *Value* field of the Sub-element. It is recommended to fill all gaps in S record addresses with 0xFF.

To build an OTAP Image File from a SREC file, follow the procedure described below:

- Generate the SREC file by correctly configuring your toolchain to do so.
- Create the image file header.
  - Set the Image ID field of the header to be unique on the OTAP Server.
  - Leave the Total Image File Size Field blank for the moment.
- Create the Upgrade Image Sub-element
  - Read the S1, S2, and S3 records from the SREC file and place the binary record data to the record addresses in the *Value* filed of the sub-element. Fill all address gaps in the S records with 0xFF.
  - Fill in the *Length* field of the sub-element with the length of the written *Value* field.
- Create the Sector Bitmap Sub-element
  - A default working setting would be all byes 0xFF for the *Value* field of this sub-element.
- Create the Image File CRC Sub-element
  - Compute the total image file size as the length of the header + the length of all 3 sub-elements and fill in the appropriate filed in the header with this value.

– Compute and write the *Value* field of this sub-element using the header and all sub-elements except this one.
– The *OTA_CrcCompute()* function in the *OtaSupport.c* file can be used to incrementally compute the CRC.

If the Image ID is not available when the image file is created, then the CRC cannot be computed. It can be computed later after the Image ID is established and written in the appropriate field in the header.

11.6 Building Bluetooth Low Energy OTAP image file from BIN file

A BIN file is an binary file which contains an executable image. The most common extension for this type of file is `.bin`. Most modern compiler toolchains can output a BIN format executable.

To enable the creation of a BIN file for your embedded application in IAR Embedded Workbench open the target properties and go to the *Output Converter* tab. Activate the *"Generate additional output"* checkbox and choose the *binary* option from the *"Output format"* drop down menu. From the same pane you can also override the name of the output file. The Figure 21 shows a screenshot of the described configuration.

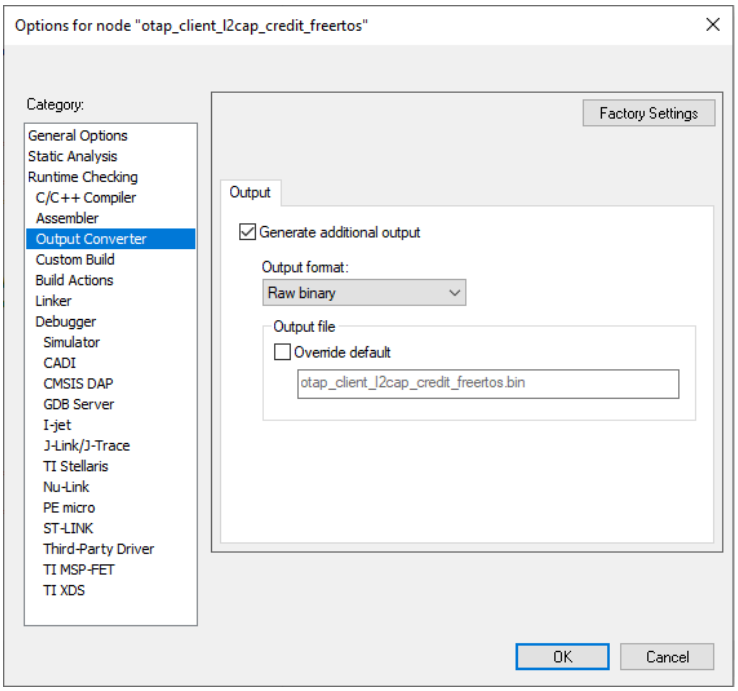

Figure 21. Enabling BIN Output in IAR Embedded Workbench

In MCUXpresso IDE, go to **Project properties** -> **Settings** -> **Build steps** window and press the "**Edit**" button for the Post-build steps. A Post-build steps window shows up in which the following command must be added:

```
arm-none-eabi-objcopy -v -O binary --only-section=.text --only-section=.data --
only-section=.ARM.exidx
   "${BuildArtifactFileName}"
   "${BuildArtifactFileBaseName}.bin"
```

The Figure 22 shows the Build steps and Post-build steps in **Settings** window.

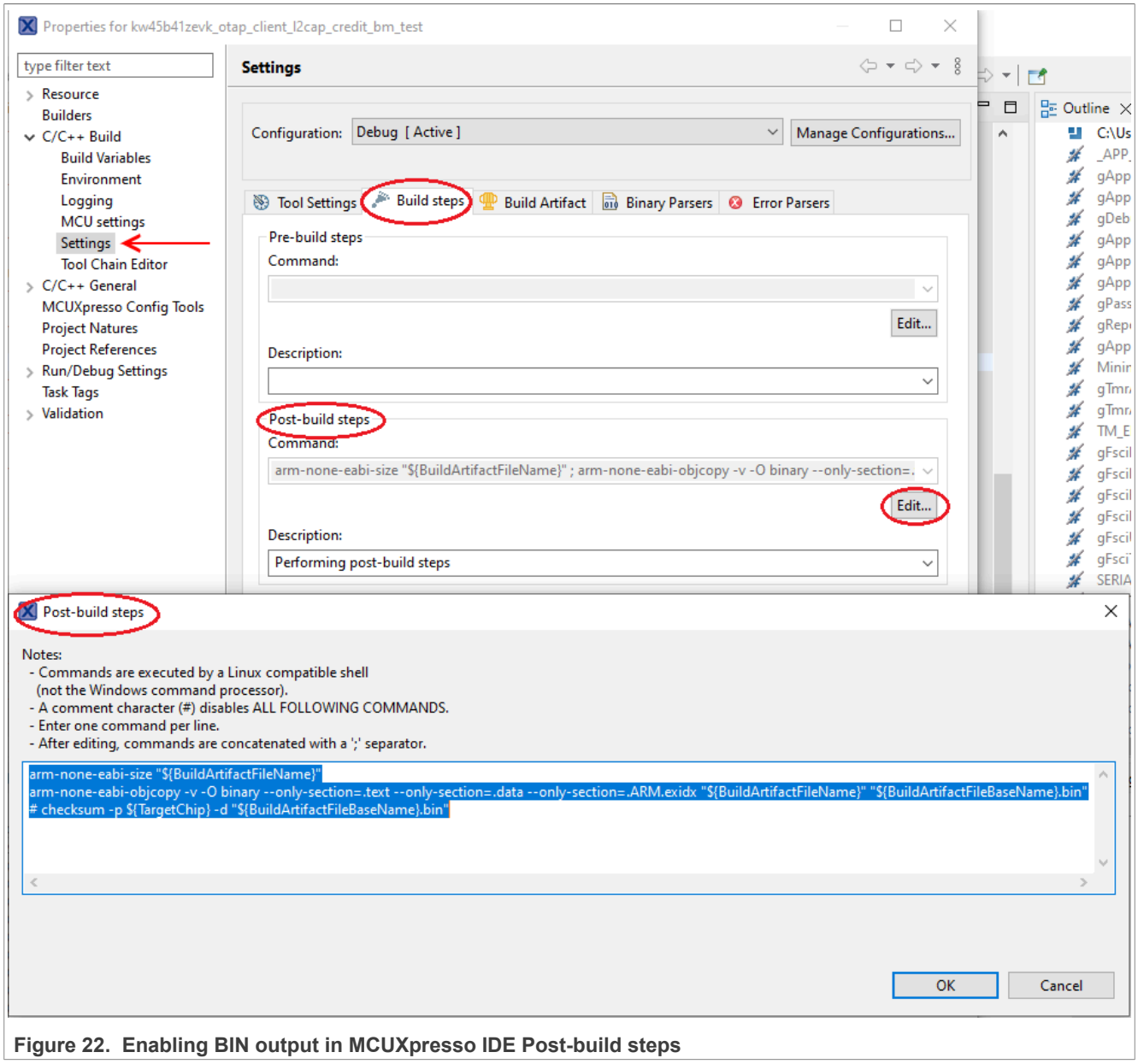

Figure 22. Enabling BIN output in MCUXpresso IDE Post-build steps

The format of the BIN file is very simple. It contains the executable image in binary format as is, starting from address 0 and up to the highest address. This type of file does not have any explicit address information.

To build an OTAP Image File from a BIN file, follow the procedure below:

- Generate the BIN file by correctly configuring your toolchain to do so.
- Create the image file header
  - Set the `Image ID` field of the header to be unique on the OTAP Server.
  - Leave the `Total Image File Size` field blank for the moment.
- Create the Upgrade Image Sub-element
  - Copy the entire contents of the BIN file as is into the *Value* filed of the sub-element.

  - Fill in the *Length* field of the sub-element with the length of the written *Value* filed.
- Create the Sector Bitmap Sub-element
  - A default working setting would be all bytes `0xFF` for the *Value* field of this sub-element.
- Create the Image File CRC Sub-element
  - Compute the total image file size as the length of the header + the length of all 3 sub-elements and fill in the appropriate filed in the header with this value.
  - Compute and write the *Value* field of this sub-element using the header and all sub-elements except this one.
  - The *OTA_CrcCompute()* function in the *OtaSupport.c* file can be used to incrementally compute the CRC.

If the Image ID is not available when the image file is created, then the CRC cannot be computed. It can be computed later after the Image ID is established and written in the appropriate field in the header.

11.7 Bluetooth Low Energy OTAP application integration

The Bluetooth Low Energy OTAP demo applications are standalone applications that only run the OTAP Server and the OTAP Client. In practice, however the OTAP Server and OTAP Client are used alongside with other functions. The OTAP functionality is used as a tool along with the main application on a device.

This section contains some guidelines on how to integrate OTAP functionality into other Bluetooth Low Energy applications.

11.7.1 OTAP server

Before any OTAP transactions can be done the application which acts as an OTAP Server must connect to a peer device and perform ATT service and characteristic discovery. Once the handles of the OTAP Service, OTAP Control Point and OTAP Data characteristics and their descriptors are found then OTAP communication can begin.

A good starting point for OTAP transactions for both the OTAP Server and The OTAP client is the moment the Server writes the OTAP Control Point CCCD to receive ATT Indications from the OTAP Client. At that point the Server can send a New Image Notification to the Client if it finds out what kind of device the client is through other means than the OTAP server. How this can be done is entirely application-specific. If the OTAP Server does not know exactly what kind of device is the OTAP Client it can wait for the Client to send a New Image Info Request. Again, the best behavior depends on application requirements.

Once OTAP communication begins then the OTAP Server just has to wait for commands from the OTAP Client and answer them. This behavior is almost completely stateless. An example state diagram for the OTAP Server application is shown in Figure 23.

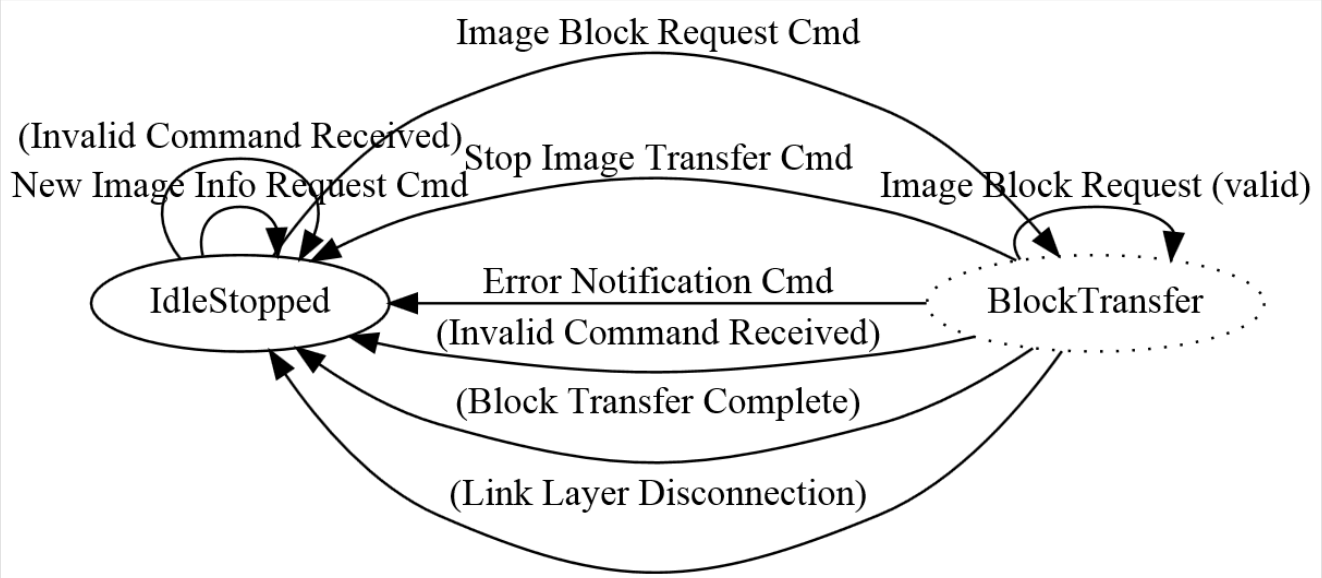

Figure 23. OTAP Server Example State Diagram

The OTAP Server waits in an idle state until a valid Image Block Request command is received and then moves to a pseudo-state and starts sending the requested block. The transfer can be interrupted by some commands (Error Notification, Stop Image Transfer, and so on) or other events (disconnection, user interruption, and so on).

The *otap_interface.h* file contains infrastructure for sending and receiving OTAP Commands and parsing OTAP image files. Packed structure types are defined for all OTAP commands and type enumerations are defined for command parameter values and some configuration values like the data payloads for the different transfer methods.

To receive ATT Indications and ATT Write Confirmations from the OTAP Client the OTAP Server application registers a set of callbacks in the stack. This is done in the *BluetoothLEHost_Initialized* function.

```
App_RegisterGattClientProcedureCallback (BleApp_GattClientCallback);
App_RegisterGattClientIndicationCallback (BleApp_GattIndicationCallback);
```

This *BleApp_GattIndicationCallback()* function is called when any attribute is indicated so the handle of the indicated attribute must be checked against a list of expected handles. In our case, we are looking for the OTAP Control Point handle that was obtained during the discovery procedure.

The *BleApp_GattIndicationCallback()* function from the demo calls an application-specific function called *BleApp_AttributeIndicated()* in which the OTAP Commands are handled.

```
static void BleApp_AttributeIndicated
(
    deviceId_t      deviceId,
    uint16_t        handle,
    uint8_t*        pValue,
    uint16_t        length
)
{
    if (handle == mPeerInformation.customInfo.otapServerConfig.hControlPoint)
    {
       otapCommandVars.pValueTemp = pValue;
        otapCommand_t*  pOtaCmd = otapCommandVars.otapCommandTemp;
        /* ... Missing code here ... */
```

```
        /* If the OTAP Server does not have internal storage then all commands
 must be forwarded
          *  via the serial interface. */
            FsciBleOtap_SendPkt (&(pOtaCmd->cmdId),
                        (uint8_t*)(&(pOtaCmd->cmd)),
                        length - gOtap_CmdIdFieldSize_c);
    }
    elseif (handle == otherHandle)
    {
        /* Handle other attribute indications here */
        /* ... Missing code here ... */
    }
    else
    {
        /*! A GATT Client is trying to GATT Indicate an unknown attribute value.
         * This should not happen. Disconnect the link. */
        Gap_Disconnect (deviceId);
    }
}
```

OTAP Server demo does not have internal storage, so all commands are forwarded via the serial interface.

To send OTAP Commands to the OTAP Client the application running the OTAP Server calls the *OtapServer_SendCommandToOtapClient()* function, which performs an ATT Write operation on the OTAP Control Point attribute.

```
static void OtapServer_SendCommandToOtapClient
        (deviceId_t  otapClientDevId,
         void*       pCommand,
         uint16_t    cmdLength)
{
    /* GATT Characteristic to be written - OTAP Client Control Point */
    gattCharacteristic_t    otapCtrlPointChar;
    bleResult_t             bleResult;

    /* Only the value handle element of this structure is relevant for this operation. */
    otapCtrlPointChar.value.handle =
 mPeerInformation.customInfo.otapServerConfig.hControlPoint;
    otapCtrlPointChar.value.valueLength = 0;
    otapCtrlPointChar.cNumDescriptors = 0;
    otapCtrlPointChar.aDescriptors = NULL;

    bleResult = GattClient_SimpleCharacteristicWrite (mPeerInformation.deviceId,
                                                      &otapCtrlPointChar,
                                                      cmdLength,
                                                      pCommand);

    if (gBleSuccess_c == bleResult)
    {
        otapServerData.lastCmdSentToOtapClient = (otapCmdIdt_t)
(((otapCommand_t*)pCommand)->cmdId);
    }
    else
    {
        /*! A Bluetooth Low Energy error has occurred - Disconnect */
        (void)Gap_Disconnect (otapClientDevId);
    }
}
```

The ATT Confirmation for the ATT Write is received in the *BleApp_GattClientCallback()* set up earlier which receives a GATT procedure success message for a *gGattProcWriteCharacteristicValue_c* procedure type.

```
static void BleApp_GattClientCallback(
    deviceId_t              serverDeviceId,
    gattProcedureType_t     procedureType,
    gattProcedureResult_t   procedureResult,
    bleResult_t             error
)
{
    union
    {
        uint8_t                     errorTemp;
        attErrorCode_t              attErrorCodeTemp;
    }attErrorCodeVars;

    if (procedureResult == gGattProcError_c)
    {
        attErrorCodeVars.errorTemp = (uint8_t)error & 0xFFU;
        attErrorCode_t attError = attErrorCodeVars.attErrorCodeTemp;
        if (attError == gAttErrCodeInsufficientEncryption_c     ||
            attError == gAttErrCodeInsufficientAuthorization_c  ||
            attError == gAttErrCodeInsufficientAuthentication_c)
        {
    #if gAppUsePairing_d
            /* Start Pairing Procedure */
            (void)Gap_Pair (serverDeviceId, &gPairingParameters);
    #endif
        }

        BleApp_StateMachineHandler (serverDeviceId, mAppEvt_GattProcError_c);
    }
    else if (procedureResult == gGattProcSuccess_c)
    {
        switch(procedureType)
        {
            /* ... Missing code here... */
            case gGattProcWriteCharacteristicValue_c:
            {
                BleApp_HandleValueWriteConfirmations (serverDeviceId);
            }
            break;

            default:
                ; /* For MISRA compliance */
            break;
        }

        BleApp_StateMachineHandler(serverDeviceId, mAppEvt_GattProcComplete_c);
    }
    else
    {
        ; /* For MISRA compliance */
    }
}
```

The *BleApp_HandleValueWriteConfirmations()* function deals with ATT Write Confirmations based on the requirements of the application.

There are two possible transfer methods for Image Chunks, the ATT transfer method and the L2CAP transfer method. The OTAP server is prepared to handle both, as requested by the OTAP Client.

To be able to use the L2CAP transfer method, the OTAP Server application must register a L2CAP LE PSM and 2 callbacks: a data callback and a control callback. This is done by using the *BluetoothLEHost_Initialized()* function.

```
/* Register OTAP L2CAP PSM */
  L2ca_RegisterLePsm (gOtap_L2capLePsm_c,
                      gOtapCmdImageChunkCocLength_c); /*!< The negotiated MTU
 must be higher than the biggest data chunk that is sent fragmented */
...
  App_RegisterLeCbCallbacks(BleApp_L2capPsmDataCallback,
 BleApp_L2capPsmControlCallback);
```

The data callback *BleApp_L2capPsmDataCallback()* is not used by the OTAP Server.

The control callback is used to handle L2CAP LE PSM connection requests from the OTAP Client and other events: PSM disconnections, No peer credits, and so on. The OTAP Client must initiate the L2CAP PSM connection if it wants to use the L2CAP transfer method.

```
static void BleApp_L2capPsmControlCallback(l2capControlMessageType_t messageType,
                                                                    void*
 pMessage)
{
    switch (messageType)
    {
        case gL2ca_LePsmConnectRequest_c:
        {
            l2caLeCbConnectionRequest_t *pConnReq = ( l2caLeCbConnectionRequest_t *)pMessage;
            /* Respond to the peer L2CAP CB Connection request - send a connection response. */
            L2ca_ConnectLePsm (gOtap_L2capLePsm_c,
                               pConnReq-> deviceId,
                               mAppLeCbInitialCredits_c);
            break;
        }
        case gL2ca_LePsmConnectionComplete_c:
        {
            l2caLeCbConnectionComplete_t *pConnComplete = ( l2caLeCbConnectionComplete_t *)pMessage;
            if (pConnComplete->result == gSuccessful_c)
            {
                /* Set the application L2CAP PSM Connection flag to TRUE because there is no
 gL2ca_LePsmConnectionComplete_c
                 * event on the responder of the PSM connection. */
                otapServerData. l2capPsmConnected = TRUE;
                otapServerData. l2capPsmChannelId = pConnComplete->cId;
            }
            break;
        }
        case gL2ca_LePsmDisconnectNotification_c:
        {
            l2caLeCbDisconnection_t *pCbDisconnect = ( l2caLeCbDisconnection_t *)pMessage;
            /* Call App State Machine */
            BleApp_StateMachineHandler (pCbDisconnect-> deviceId, mAppEvt_CbDisconnected_c);
            otapServerData. l2capPsmConnected = FALSE;
            break;
        }
        case gL2ca_NoPeerCredits_c:
        {
            l2caLeCbNoPeerCredits_t *pCbNoPeerCredits = ( l2caLeCbNoPeerCredits_t *)pMessage;
            L2ca_SendLeCredit (pCbNoPeerCredits-> deviceId,
                               otapServerData. l2capPsmChannelId,
                               mAppLeCbInitialCredits_c);
            break;
        }
        case gL2ca_LocalCreditsNotification_c:
        {
            l2caLeCbLocalCreditsNotification_t *pMsg = ( l2caLeCbLocalCreditsNotification_t
 *)pMessage;
            break;
        }
        default:
```

```
            break;
    }
}
```

The ATT transfer method is supported by default but the L2CAP transfer method only works if the OTAP Client opens an L2CAP PSM credit-oriented channel.

To send data chunks to the OTAP Client the OTAP Server application calls the *OtapServer_SendCImgChunk ToOtapClient()* function which delivers the chunk via the selected transfer method. For the ATT transfer method the chunk is sent via the *GattClient_CharacteristicWriteWithoutResponse()* function and for the L2CAP transfer method the chunk is sent via the *L2ca_SendLeCbData()* function.

```
static void OtapServer_SendCImgChunkToOtapClient (deviceId_t otapClientDevId,
                                                  void*      pChunk,
                                                  uint16_t   chunkCmdLength)
{
    bleResult_t bleResult = gBleSuccess_c;
    if (otapServerData.transferMethod == gOtapTransferMethodAtt_c)
    {
        /* GATT Characteristic to be written without response - OTAP Client Data */
        gattCharacteristic_t otapDataChar;
        /* Only the value handle element of this structure is relevant for this operation. */
        otapDataChar.value.handle = mPeerInformation.customInfo.otapServerConfig.hData;
        bleResult = GattClient_CharacteristicWriteWithoutResponse
                                                 (mPeerInformation.deviceId,
                                                  &otapDataChar,
                                                  chunkCmdLength,
                                                  pChunk);
    }
    else if (otapServerData.transferMethod == gOtapTransferMethodL2capCoC_c)
    {
        bleResult = L2ca_SendLeCbData (mPeerInformation.deviceId,
                                       otapServerData.l2capPsmChannelId,
                                       pChunk,
                                       chunkCmdLength);
    }
    if (gBleSuccess_c != bleResult)
    {
        /*! A Bluetooth Low Energy error has occurred - Disconnect */
        Gap_Disconnect (otapClientDevId);
    }
}
```

The OTAP Server demo application relays all commands received from the OTAP Client to a PC through the FSCI type protocol running over a serial interface. It also directly relays all responses from the PC back to the OTAP Client.

Other implementations can bring the image to an external memory through other means of communication and directly respond to the OTAP Client requests.

11.7.2 OTAP client

An application running an OTAP Client must wait for an OTAP Server to connect and perform service and characteristic discovery before performing any OTAP-related operations. OTAP transactions can begin only after the OTAP Server writes the OTAP Control point CCC Descriptor to receive ATT Notifications. After this is done, bidirectional communication is established between the OTAP Server and Client and OTAP transactions can begin.

The OTAP Client can advertise the OTAP Service via the demo application. Optionally, the OTAP Server may already know the advertising device has an OTAP Service based on application-specific means. In both situations, the OTAP Server must discover the handles of the OTAP Service and its characteristics.

In addition to the OTAP Service instantiated in the GATT Database, the OTAP Client needs to have some storage capabilities for the downloaded image file.

How to put the OTAP Service in the GATT Database is described in The OTAP Service and Characteristics.

The upgrade image storage capabilities in the demo OTAP Client applications are handled by the *OtaSupport* module from the Framework, which contains support modules and drivers. The *OtaSupport* module has support for both internal storage (a part of the internal flash memory is reserved for storing the upgrade image) and external storage (a SPI flash memory chip).

The demo applications use internal storage by default. The internal storage is viable only if there is enough space in the internal flash for the upgrade image – the flash in this case should be at least twice the size of the largest application. The *OtaSupport* module also needs the *Eeprom* module from the Framework to work correctly.

The OtaSupport module also includes functionality for configuring the OTACFG IFR sections after the image is received in order to enable the ROM bootloader to perform the actual image update.

To use the *OtaSupport* module several configuration options must be set up in both the source files and the linker options of the toolchain.

To use internal storage, set up the `gUseInternalStorageLink_d=1` symbol in the linker configuration window (**Linker->Config tab** in the IAR project properties) and set the `gAppOtaExternalStorage_c value` to (`0`) in the `app_preinclude.h` file:

```
/*! Define as 1 to place OTA storage in external flash */
  #define gAppOtaExternalStorage_c (0)
```

The OTAP demo applications for the IAR EW IDE have some settings in the Linker options tab which must be configured to use OtaSupport and the OTAP Bootloader. In the **Project Target Options->Linker->Config** tab, 3 symbols must be correctly defined. To use NVM storage, the `gUseNVMLink_d` symbol must be set to `1`. The `gUseInternalStorageLink_d` symbol must be set to `0` when OTAP external storage is used and to `1` when the internal storage is used. The `gEraseNVMLink_d` must be set to 0.

An example linker configuration window for IAR is shown in Figure 24.

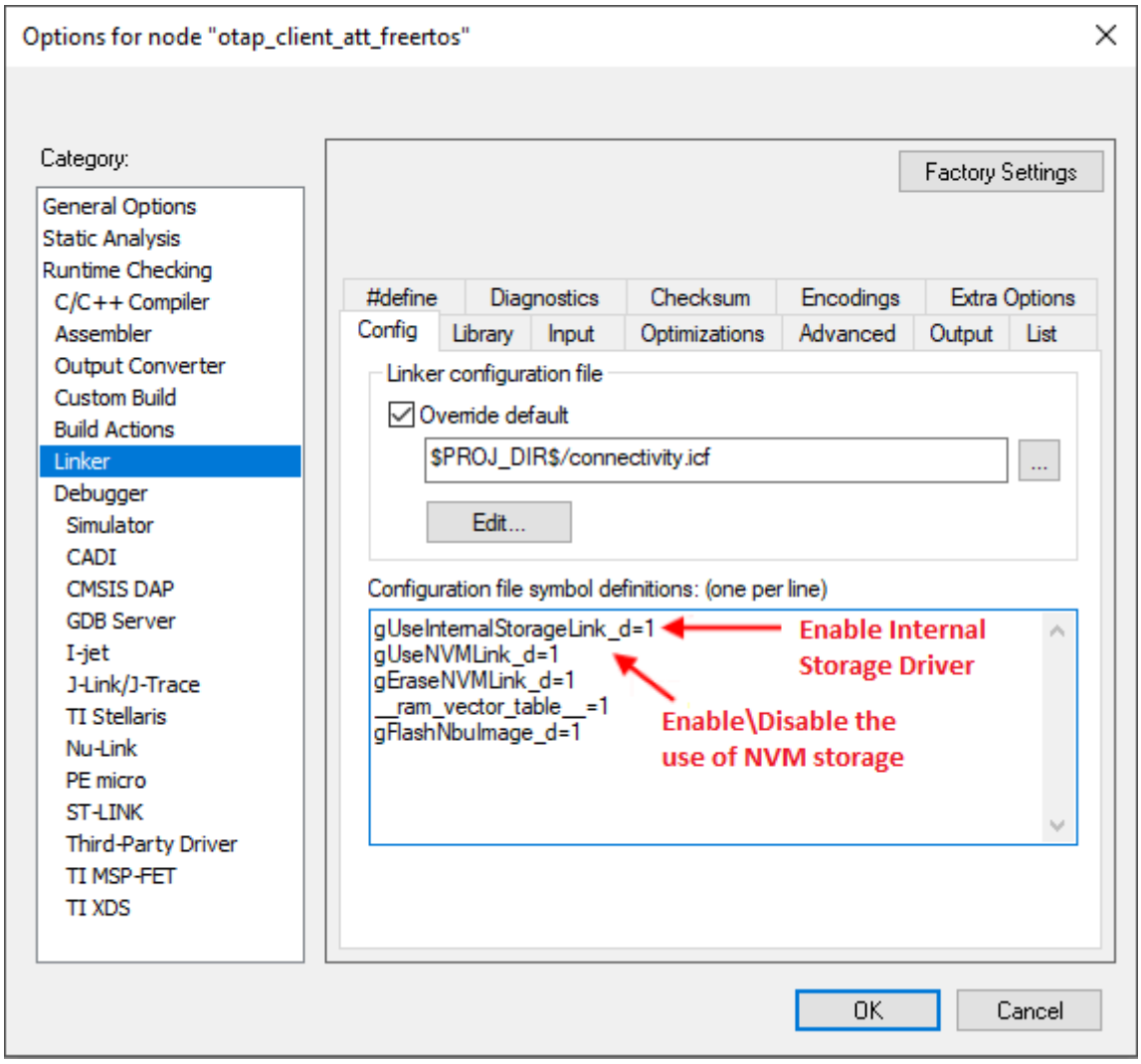

Figure 24. Linker Config IAR EW IDE - OTAP Client External Storage and NVM Configuration

Note: *The gEraseNVMLink_d=1 IAR linker flag places some dummy bytes into the NVM region to invalidate the data and force the application to erase the entire NVM region. When generating an image for the OTA upgrade, this flag must be set to 0. This results in a smaller image size being transferred and lower power consumption. If the NVM region must be erased after the upgrade process, the "Preserve NVM" checkbox (from the Over The Air programming tool) should be unchecked.*

For MCUXpresso IDE, the linker settings required for OTAP applications can be set up from the "**SDK Import Wizard**" or from the "**Project Properties** -> **MCU settings**". Refer to Figure 25.

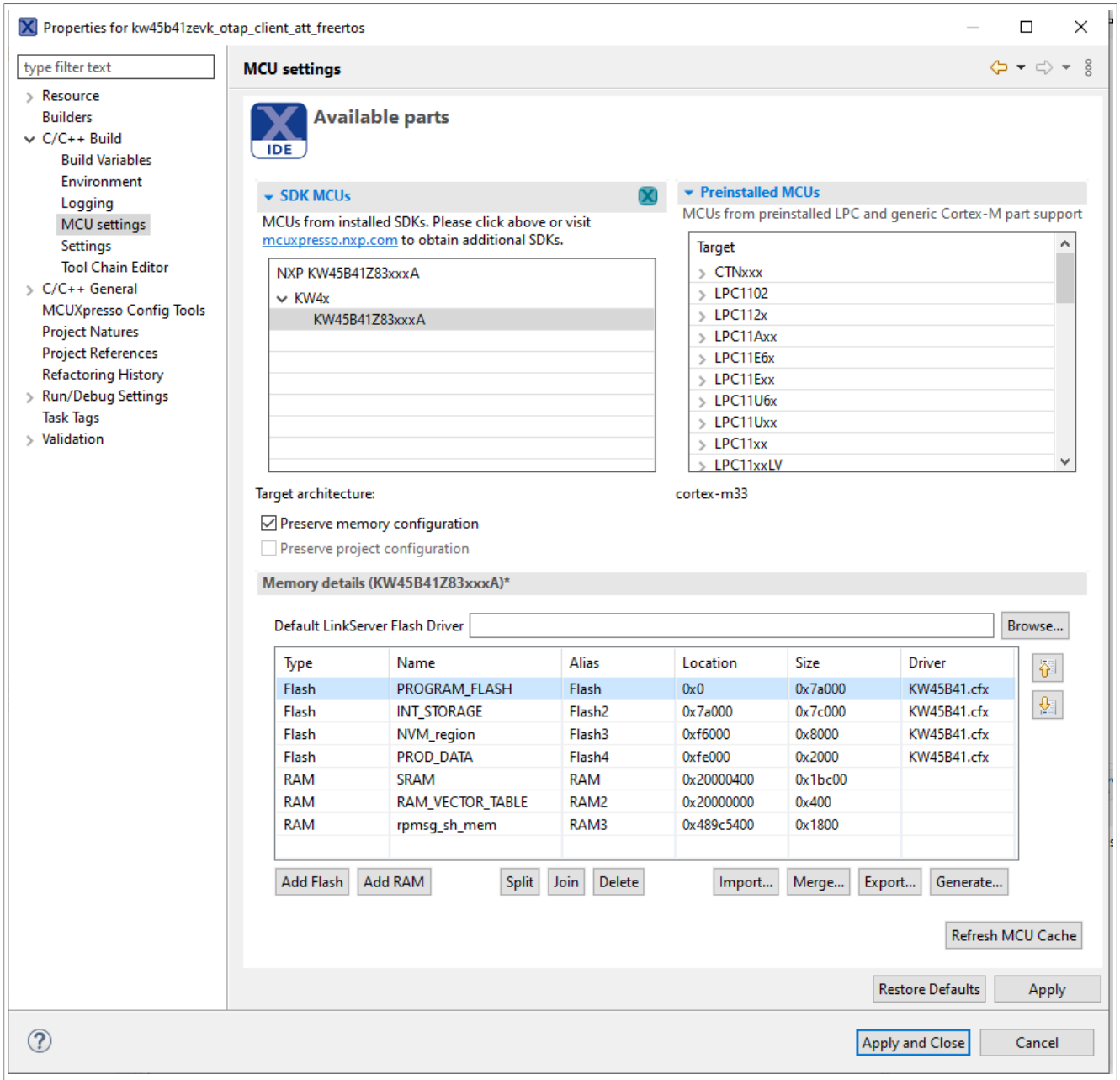

Figure 25. MCUX memory

The demo applications use internal storage by default. To enable external storage support for MCUX, set the `gAppOtaExternalStorage_c` value to (`1`) in the `app_preinclude.h` file. Also remove the `INT_STORAGE` section (from **Project Properties** -> **MCU settings**) and extend the `PROGRAM_FLASH` section as shown in the Figure 26.

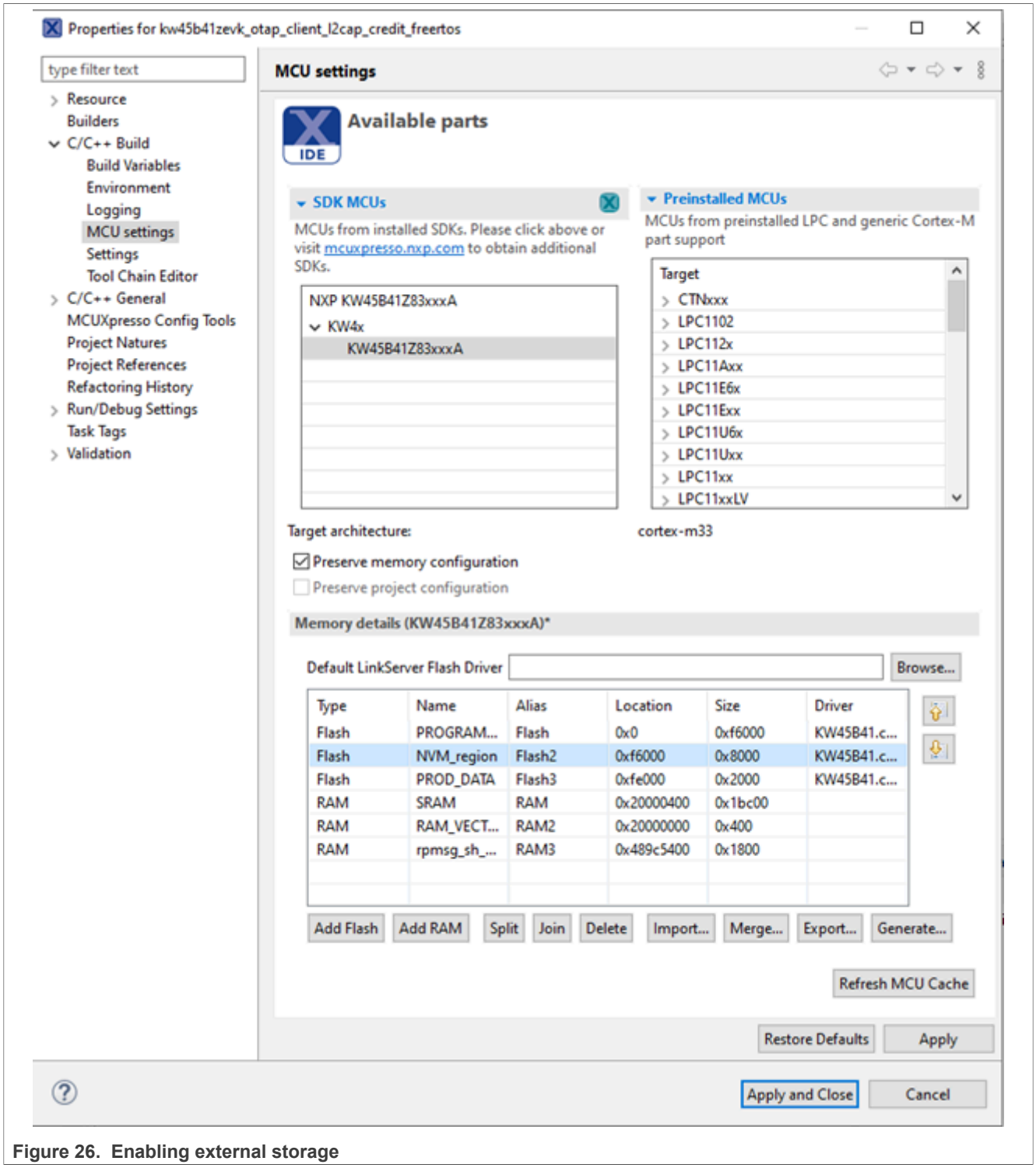

Figure 26. Enabling external storage

Once the application starts and bidirectional OTAP communication is established via the OTAP Service, then the OTAP Client must determine if the connected OTAP Server has a newer image than the one currently present on the device. This can be done in two ways:

- The OTAP Server knows by some application-specific means that it has a newer image and sends a New Image Notification to the OTAP Client or

- The OTAP Client sends a New Image Info Request to the OTAP Server and waits for a response. This example application uses the second method.

The New Image Info Request contains enough information about the currently running image to allow the OTAP Server to determine if it has a newer image for the requesting device. The New Image Info Response contains enough information for the OTAP Client to determine if the "deadvertised" image is newer and it wants to download it. The best method is entirely dependent on application requirements.

An example function that checks if an *ImageVerison* field from a New Image Notification or a New Image Info Response corresponds to a newer image (based on the suggested format of this field) is provided in the OTAP Client demo applications. The function is called *OtapClient_IsRemoteImageNewer()*.

The OTAP Client application is a little more complicated than the OTAP Server application because more state information needs to be handled (current image position, current chunk sequence number, image file parsing information, and so on). An example state diagram for the OTAP Client is shown below. The Figure 27 briefly lists the steps of the image download process. Note that some of the states may not be explicitly present in the demo applications.

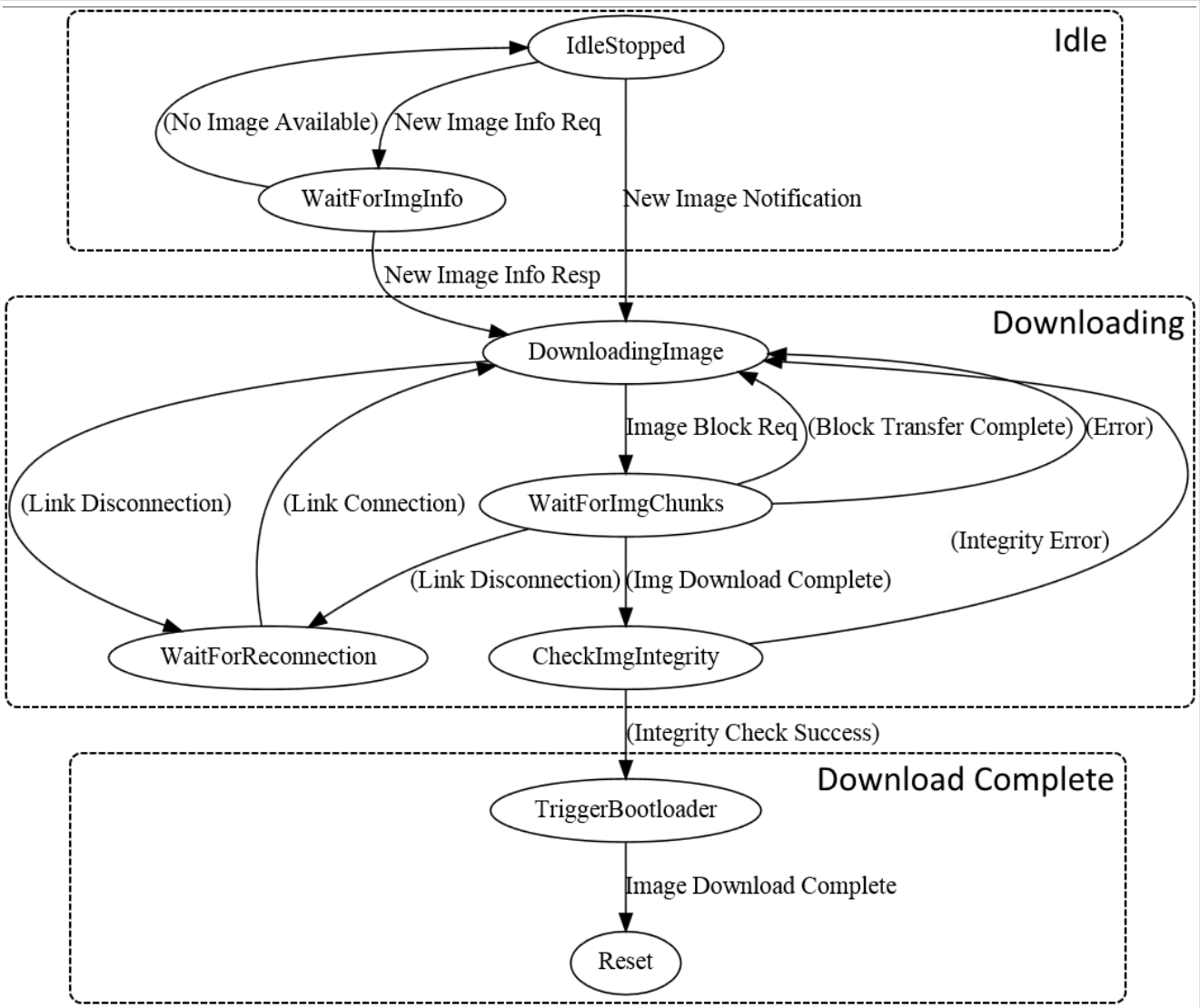

Figure 27. OTAP Client Example State Diagram

After the OTAP Client determines that the peer OTAP Server has a suitable upgrade image available, it can start the download process. This is done by sending multiple Image Block Request messages and waiting for the Image Chunks via the selected transfer method.

While receiving the image file blocks, the OTAP Client application parses the image file. In case any parameter of an image file sub-element is invalid or the image file format is invalid, it sends an Error Notification to the OTAP Server and tries to restart the download process from the beginning or a known good position.

When an Image Chunk is received, its sequence number is checked and its content is parsed in the context of the image file format. If the sequence number is not as expected, then the block transfer is restarted from the last known good position. When all chunks of an Image Block are received, the next block is requested, if there are more blocks to download. When the last Image Block in an image file is received, then the image integrity is checked (the received CRC from the Image File CRC sub-element is compared to the computed CRC).

The computed image integrity initialization and intermediary value must be reset to '0' before starting or restarting an image download. If the image integrity check fails then the image download process is restarted from the beginning. If the image integrity check is successful, then the `Image Download Complete` message is sent to the OTAP Server, the OTACFG IFR is updated and the MCU is restarted. After the restart, the ROM bootloader kicks in and writes the new image to the flash memory, afterwards giving CPU control to the newly installed application.

If at any time during the download process, a Link Layer disconnection occurs, then the image download process is restarted from the last known good position when the link is re-established.

As noted earlier, the OTAP Client application needs to handle a lot of state information. In the demo application, all this information is held in the *otapClientData* structure of the *otapClientAppData_t* type. The type is defined and the structure is initialized in the *otap_client.c* file of the application. This structure is defined and initialized differently for the OTAP Client ATT and L2CAP example applications. Mainly, the *transferMethod* member of the structure is constant and has different values for the two example applications and the L2CAP application structure has an extra member.

To receive write notifications when the OTAP Server writes the OTAP Control Point attribute and ATT Confirmations when it indicates the OTAP Control Point attribute, the OTAP Client application must register a GATT Server callback and enable write notifications for the OTAP Control Point attribute. This is done in the *BluetoothLEHost_Initialized()* function in the *otap_client_att.c/otap_client_l2cap_credit.c* file.

```
static void BluetoothLEHost_Initialized(void)
{
    /* ... Missing code here ... */

    /* Register stack callbacks */
    (void)App_RegisterGattServerCallback (BleApp_GattServerCallback);

    /* ... Missing code here ... */
}
```

The *BleApp_GattServerCallback()* function handles all incoming communication from the OTAP Server.

```
static void BleApp_GattServerCallback (deviceId_t deviceId, gattServerEvent_t*
 pServerEvent)
{
    switch (pServerEvent->eventType)
    {
        /* ... Missing code here ... */

        case gEvtCharacteristicCccdWritten_c:
        {
            OtapClient_CccdWritten (deviceId,
```

```
                                    pServerEvent-
>eventData.charCccdWrittenEvent.handle,
                                    pServerEvent-
>eventData.charCccdWrittenEvent.newCccd);
        }
        break;

        case gEvtAttributeWritten_c:
        {
            OtapClient_AttributeWritten (deviceId,
                                         pServerEvent-
>eventData.attributeWrittenEvent.handle,
                                         pServerEvent-
>eventData.attributeWrittenEvent.cValueLength,
                                         pServerEvent-
>eventData.attributeWrittenEvent.aValue);
        }
        break;

        case gEvtAttributeWrittenWithoutResponse_c:
        {
            OtapClient_AttributeWrittenWithoutResponse (deviceId,
                                                        pServerEvent-
>eventData.attributeWrittenEvent.handle,
                                                        pServerEvent-
>eventData.attributeWrittenEvent.cValueLength,
                                                        pServerEvent-
>eventData.attributeWrittenEvent.aValue);
        }
        break;

        case gEvtHandleValueConfirmation_c:
        {
            OtapClient_HandleValueConfirmation (deviceId);
        }
        break;

        /* ... Missing code here ... */

        default:
            ; /* For MISRA compliance */
        break;
    }
}
```

When the OTAP Server Writes a CCCD the *BleApp_GattServerCallback()* function calls the *OtapClient_CccdWritten()* function which sends a New Image Info Request when the OTAP Control Point CCCD is written it – this is the starting point of OTAP transactions in the demo applications.

When an ATT Write Request is made by the OTAP Server the the *BleApp_GattServerCallback()* function calls the *OtapClient_AttributeWritten()* function which handles the data as an OTAP command. Only writes to the OTAP Control Point are handled as OTAP commands. For each command received from the OTAP Server there is a separate handler function which performs required OTAP operations. These are:

- *OtapClient_HandleNewImageNotification()*
- *OtapClient_HandleNewImageInfoResponse()*
- *OtapClient_HandleErrorNotification()*

When an ATT Write Command (GATT Write Without Response) is sent by the OTAP Server the *BleApp_GattServerCallback()* function calls the *OtapClient_AttributeWrittenWithoutResponse()* function which handles Data Chunks if the selected transfer method is ATT and returns an error if any problems are encountered. Data chunks are handled by the *OtapClient_HandleDataChunk()* function.

```
static void BleApp_AttributeWrittenWithoutResponse (deviceId_t deviceId,
                                                    uint16_t
 handle,
                                                    uint16_t
 length,
                                                    uint8_t*
 pValue)
{
    /* ... Missing code here ... */
    if (handle == value_otap_data)
    {
        /* ... Missing code here ... */
        if (otapClientData.transferMethod == gOtapTransferMethodAtt_c)
        {
            if (((otapCommand_t*)pValue)->cmdId == gOtapCmdIdImageChunk_c)
            {
                OtapClient_HandleDataChunk (deviceId,
                                            length,
                                            pValue);
            }
        }
        /* ... Missing code here ... */
    }
    /* ... Missing code here ... */
}
```

Finally, when an ATT Confirmation is received for a previously sent ATT Indication the *BleApp_GattServerCallback()* function calls the *OtapClient_HandleValueConfirmation()* function, which performs the necessary OTAP operations based on the last sent command to the OTAP Server. This is done using separate confirmation handling functions for each command that is sent to the OTAP Server. These functions are:

- *OtapClient_HandleNewImageInfoRequestConfirmation()*
- *OtapClient_HandleImageBlockRequestConfirmation()*
- *OtapClient_HandleImageTransferCompleteConfirmation()*
- *OtapClient_HandleErrorNotificationConfirmation()*
- *OtapClient_HandleStopImageTransferConfirmation()*

Outgoing communication from the OTAP Client to the OTAP Server is done using the *OtapCS_SendCommandToOtapServer()* function. This function writes the value to be indicated to the OTAP Control Point attribute in the GATT database and then calls the *OtapCS_SendControlPointIndication()* which checks if indications are enabled for the target device and sends the actual ATT Indication. Both functions are implemented in the *otap_service.c* file.

```
bleResult_t OtapCS_SendCommandToOtapServer (uint16_t serviceHandle,
                                            void* pCommand,
                                            uint16_t cmdLength)
{
    union
    {
        uint8_t*               uuid_char_otap_control_pointTemp;
        bleUuid_t*             bleUuidTemp;
```

```
    }bleUuidVars;

    uint16_t  handle;
    bleResult_t result;
    bleUuidVars.uuid_char_otap_control_pointTemp = uuid_char_otap_control_point;
    bleUuid_t* pUuid = bleUuidVars.bleUuidTemp;

    /* Get handle of OTAP Control Point characteristic */
    result = GattDb_FindCharValueHandleInService(serviceHandle,
                                                 gBleUuidType128_c, pUuid,
 &handle);

    if (result == gBleSuccess_c)
    {
        /* Write characteristic value */
        result = GattDb_WriteAttribute(handle,
                                       cmdLength,
                                       (uint8_t*)pCommand);

        if (result == gBleSuccess_c)
        {
            /* Send Command to the OTAP Server via ATT Indication */
            result = OtapCS_SendControlPointIndication (handle);
        }
    }

    return result;
}

static bleResult_t OtapCS_SendControlPointIndication (uint16_t handle)
{
    uint16_t     hCccd;
    bool_t       isIndicationActive;
    /* Get handle of CCCD */
    GattDb_FindCccdHandleForCharValueHandle (handle, &hCccd);
    Gap_CheckIndicationStatus (...);
    return GattServer_SendIndication (...);
}
```

The *otap_interface.h* file contains all the necessary information for parsing and building OTAP commands (packed command structures type definitions, command parameters enumerations, and so on).

For the two possible image transfer methods (ATT and L2CAP) there are two separate demo applications. To be able to use the L2CAP transfer method the OATP Client application must register a L2CAP LE PSM and 2 callbacks: a data callback and a control callback. This is done in the *OtapClient_Config()* function.

```
/* Register OTAP L2CAP PSM */
L2ca_RegisterLePsm (gOtap_L2capLePsm_c,
gOtapCmdImageChunkCocLength_c); /*!< The negotiated MTU must be higher than the
 biggest data chunk that is sent fragmented */
...
App_RegisterLeCbCallbacks(BleApp_L2capPsmDataCallback,
 BleApp_L2capPsmControlCallback);
```

The control callback is used to handle L2CAP LE PSM-related events: PSM disconnections, PSM Connection Complete, No peer credits, and so on.

```
static void BleApp_L2capPsmControlCallback
            (l2capControlMessageType_t   messageType,
```

```
                void*   pMessage)
{
    switch (messageType)
    {
        case gL2ca_LePsmConnectRequest_c:
        {
            l2caLeCbConnectionRequest_t *pConnReq =
                            (l2caLeCbConnectionRequest_t *)pMessage;
            /* This message is unexpected on the OTAP Client, the OTAP Client
 sends L2CAP
             * PSM connection requests and expects L2CAP PSM connection
 responses.
             * Disconnect the peer. */
            Gap_Disconnect (pConnReq->deviceId);
            break;
        }
        case gL2ca_LePsmConnectionComplete_c:
        {
            l2caLeCbConnectionComplete_t *pConnComplete =
                             (l2caLeCbConnectionComplete_t *)pMessage;
            /* Call the application PSM connection complete handler. */
            OtapClient_HandlePsmConnectionComplete (pConnComplete);
        break;
    }
    case gL2ca_LePsmDisconnectNotification_c:
    {
        l2caLeCbDisconnection_t *pCbDisconnect = (l2caLeCbDisconnection_t
 *)pMessage;
        /* Call the application PSM disconnection handler. */
        OtapClient_HandlePsmDisconnection (pCbDisconnect);
        break;
    }
    case gL2ca_NoPeerCredits_c:
    {
        l2caLeCbNoPeerCredits_t *pCbNoPeerCredits =
                        (l2caLeCbNoPeerCredits_t *)pMessage;
        L2ca_SendLeCredit (pCbNoPeerCredits->deviceId,
                           otapClientData.l2capPsmChannelId,
                           mAppLeCbInitialCredits_c);
        break;
    }
   /* ... Missing code here ... */
    case gL2ca_Error_c:
        {
            /* Handle error */
            break;
        }
        default:
            ; /* For MISRA compliance */
        break;
}
```

The OTAP Client must initiate the L2CAP PSM connection if it wants to use the L2CAP transfer method; this can be done using the *L2ca_ConnectLePsm()* function. The *L2ca_ConnectLePsm()* function is called by the *OtapClient_ContinueImageDownload()* if the transfer method is L2CAP and the PSM is found to be disconnected.

```
static void OtapClient_ContinueImageDownload (deviceId_t deviceId)
{
```

```
    /* ... Missing code here ... */
    /* Check if the L2CAP OTAP PSM is connected and if not try to connect and
 exit immediately. */
    if ((otapClientData.l2capPsmConnected == FALSE) &&
                (otapClientData.state !=
 mOtapClientStateImageDownloadComplete_c))
    {
        L2ca_ConnectLePsm (gOtap_L2capLePsm_c,
                           deviceId,
                           mAppLeCbInitialCredits_c);
        bValidState = FALSE;;
    }
    /* ... Missing code here ... */
}
```

The PSM data callback *BleApp_L2capPsmDataCallback()* is used by the OTAP Client to handle incoming image file parts from the OTAP Server.

```
static void BleApp_L2capPsmDataCallback (deviceId_t  deviceId,
   uint8_t*    pPacket,
   uint16_t    uint16_t lePsm,
   uint16_t    packetLengt
{
   OtapClient_HandleDataChunk (deviceId,
                               packetLength,
                               pPacket);
}
```

All data chunks regardless of their source (ATT or L2CAP) are handled by the *OtapClient_HandleDataChunk()* function. This function checks the validity of Image Chunk messages, parses the image file, requests the continuation or restart of the image download and triggers the bootloader when the image download is complete.

```
static void OtapClient_HandleDataChunk (deviceId_t deviceId, uint16_t length,
 uint8_t* pData);
```

The Image File CRC Value is computed on the fly as the image chunks are received using the *OTA_CrcCompute()* function from the *OtaSupport* module which is called by the *OtapClient_HandleDataChunk()* function. The *OTA_CrcCompute()* function has a parameter for the intermediary CRC value which must be initialized to 0 every time a new image download is started.

The actual write of the received image parts to the storage medium is also done in the *OtapClient_HandleDataChunk()* function using the *OtaSupport* module. This is achieved using the following functions:

- *OTA_StartImage()* – called before the start of writing a new image to the storage medium.
- *OTA_CancelImage()* – called whenever an error occurs and the image download process needs to be stopped/restarted from the beginning.
- *OTA_PushImageChunk()* – called to write a received image chunk to the storage medium. Note that only the Upgrade Image Sub-element of the image file is actually written to the storage medium.
- *OTA_CommitImage()* - called to set up what parts of the downloaded image are written to flash and other information for the bootloader. The Value field of the Sector Bitmap Sub-element of the Image File is given as a parameter to this function.
- *OTA_SetNewImageFlag()* - called to configure the OTACFG IFR when a new image has been successfully received. When the MCU is reset, the ROM bootloader transfers the new image from the storage medium to the program flash.

To continue the image download process after a block is transferred or to restart it after an error has occurred the *OtapClient_ContinueImageDownload()* function is called. This function is used in multiple situations during the image download process.

To summarize, an outline of the steps required to perform the image download process is shown below:

- Wait for a connection from an OTAP Server
- Wait for the OTAP Server to write the OTAP Control Point CCCD
- Ask or wait for image information from the server
- If a new image is available on the server, start the download process using the *OtapClient_ContinueImageDownload()* function.
  - If the transfer method is L2CAP CoC, then initiate a PSM connection to the OTAP Server
- Repeat while image download is not complete.
  - Wait for image chunks.
  - *Call the OtapClient_HandleDataChunk()* function for all received image chunks regardless of the selected transfer method.
    - Check image file header integrity using the *OtapClient_IsImageFileHeaderValid()* function.
    - Write the Upgrade Image Sub-element to the storage medium using *OtaSupport* module functions.
    - When the download is complete, check image integrity.
      - If the integrity check is successful, commit the image using the Sector Bitmap Sub-element and trigger the bootloader
      - If integrity check fails, restart the image download from the beginning
    - If the download is not complete, ask for a new image chunk.
  - If any error occurs during the processing of the image chunk, restart the download from the last known good position.
- If an image was successfully downloaded and transferred to the storage medium and the bootloader triggered, then reset the MCU to start the flashing process of the new image.

11.8 Secured OTAP

The security features of the KW45/K32W1 devices enable them to use secured OTAP, meaning the new images can be authenticated and encrypted. The decryption/authentication keys are programmed into hardware fuses. For information on how to prepare the board, refer to the accompanying document related to board provisioning. For information on how to obtain the secured image, refer to the *Bluetooth Low Energy Demo Applications User's Guide (BLEDAUG)*.

12 Creating a Bluetooth LE application when the Host Stack runs on another processor

This section describes how to create a Bluetooth Low Energy application (host), when the Bluetooth Low Energy Host Stack is running on another processor (blackbox). The section alsp provides sample code to explain how to achieve this.

The supported serial interfaces between the two chips (application and the Bluetooth Low Energy Host Stack) are UART, SPI, or USB.

Typical applications employing Bluetooth LE Host Stack blackboxes are host systems such as a PC tool or an embedded system that has an application implementation. This chapter describes an embedded application.

For more information, refer to *Bluetooth Low Energy Host Stack FSCI Reference Manual*. This document provides explicit information on exercising the Bluetooth Low Energy Host Stack functionality through a serial communication interface to a host system.

12.1 Serial manager and FSCI configuration

For creating an embedded application that communicates with the Bluetooth Low Energy Host Stack using the serial interface, the following steps must be done:

12.1.1 Serial manager initialization

The function that must be called for Serial Manager initialization is located in *SerialManager.h*:

```
/* Init serial manager */
SerialManager_Init();
```

12.1.2 FSCI configuration and initialization

By default, the FSCI module is disabled. It must be enabled by setting `gFsciIncluded_c` to `1`. Also, `gFsciLenHas2Bytes_c` must be set to `1` because Bluetooth Low Energy Host Stack interface commands and events need serial packets bigger than 255 octets.

For more information on the following configuration parameters, refer to the FSCI chapter of the *Connectivity Framework Reference Manual*.

To configure the FSCI module, the following parameters can be set on both the Bluetooth Low Energy Application project and the Bluetooth Low Energy FSCI blackbox:

```
/* Enable/Disable FSCI */
#define gFsciIncluded_c                 1

/* Enable/Disable FSCI Low Power Commands*/
#define gFSCI_IncludeLpmCommands_c      0

/* Defines FSCI length - set this to FALSE is FSCI length has 1 byte */
#define gFsciLenHas2Bytes_c             1

/* Defines FSCI maximum payload length */
#define gFsciMaxPayloadLen_c            1660

/* Enable/Disable Ack transmission */
#define gFsciTxAck_c                    0
```

```
/* Enable/Disable Ack reception */
#define gFsciRxAck_c                    0

/* Enable FSCI Rx restart with timeout */
#define gFsciRxTimeout_c                1
#define mFsciRxTimeoutUsePolling_c      1

/* Use Misra Compliant version of FSCI module */
#define gFsciUseDedicatedTask_c         1

/* FSCI task size */
#if defined(DEBUG)
#define gFsciTaskStackSize_c            4600
#else
#define gFsciTaskStackSize_c            2600
#endif
```

To perform the FSCI module initialization, the following code can be used:

```
/*Define fsci serial manager handle*/
#if defined(gFsciIncluded_c) && (gFsciIncluded_c > 0)
extern serial_handle_t g_fsciHandleList[gFsciIncluded_c];
#endif /*gFsciIncluded_c > 0*/

void BluetoothLEHost_AppInit(void)
{
    /* Init FSCI */
    FSCI_commInit( g_fsciHandleList );

    /* Register BLE handlers in FSCI */
    fsciBleRegister(0);
    ...
}
```

12.1.3 FSCI handlers (GAP, GATT, and GATTDB) registration

For receiving messages from all the Bluetooth Low Energy Host Stack serial interfacing layers (GAP, GATT, and GATTDB), a function handler must be registered in FSCI for each layer:

```
fsciBleRegister(0);
```

12.2 Bluetooth Low Energy Host Stack initialization

The Bluetooth Low Energy Host Stack must be initialized when platform setup is complete and all RTOS tasks have been started. This initialization is done by restarting the blackbox using a FSCI CPU Reset Request command. This is performed automatically by the ***BIe_Initialize****(App_GenericCallback)* function.

```
/* Send FSCI CPU reset command to BlackBox */
FSCI_transmitPayload(gFSCI_ReqOpcodeGroup_c, mFsciMsgResetCPUReq_c, NULL, 0,
 fsciInterface);
```

The completion of the Bluetooth Low Energy Host Stack initialization is signaled by the reception of the *GAP-GenericEventInitializationComplete.Indication* event (over the serial communication interface, in FSCI). The *Bluetooth Low Energy-HostInitialize.Request* command is not required to be sent to the blackbox (the entire initialization is performed by the blackbox, when it resets).

12.3 GATT database configuration

The GATT database always resides on the same processor as the entire Bluetooth Low Energy Host Stack, so the attributes must be added by the host application using the serial communication interface.

To create a GATT database remotely, *GATTDBDynamic* commands must be used. The GATTDBDynamic API is provided to the user that performs all the required memory allocations and sends the FSCI commands to the blackbox. The result of the operation is returned, including optionally the service, characteristic, and '*cccd*' handles returned by the blackbox.

Current supported API for adding services is the following:

```
bleResult_t GattDbDynamic_AddGattService (gattServiceHandles_t*
 pOutServiceHandles);
bleResult_t GattDbDynamic_AddGapService (gapServiceHandles_t*
 pOutServiceHandles);
bleResult_t GattDbDynamic_AddIpssService (ipssServiceHandles_t*
 pOutServiceHandles);
bleResult_t GattDbDynamic_AddHeartRateService (heartRateServiceHandles_t*
 pOutServiceHandles);
bleResult_t GattDbDynamic_AddBatteryService (batteryServiceHandles_t*
 pOutServiceHandles);
bleResult_t GattDbDynamic_AddDeviceInformationService
 (deviceInfoServiceHandles_t* pOutServiceHandles);
```

The service handles are optional.

Also, a generic function is provided, so that the user can add any generic service to the database:

```
bleResult_t GattDbDynamic_AddServiceInDatabase (serviceInfo_t* pServiceInfo);
```

Usually, a Bluetooth Low Energy Application is ported from a single chip solution, where the Bluetooth Low Energy Application and the Bluetooth Low Energy stack reside on the same processor and the GATT database is populated statically. The user should remove all the attribute handles from any structure and replace them with *gGattDbInvalidHandle_d* . The attribute handles should be populated after the services are added dynamically to the database with the handles returned by the previous API.

12.4 FSCI host layer

The Bluetooth Low Energy GAP, GATT, GATTDB, and L2CAP APIs are included in the Bluetooth Low Energy interface. When these APIs reside on a separate processor than the Bluetooth Low Energy stack, they are implemented as an FSCI Host Layer that should be added to the Bluetooth Low Energy Application project.

This layer is responsible for serializing API to the corresponding FSCI commands. The layer also sends these APIs to the blackbox, receives and deserializes FSCI statuses and events, presents them to the Bluetooth Low Energy Application, and arbitrates access from multiple tasks to the serial interface.

All the GAP, GATT, GATTDB, and L2CAP APIs are executed asynchronously, so the user context blocks waiting for the response from the blackbox. The response can be the status of the request or optionally an FSCI event, which includes the output parameters of a synchronous function.

There are also functions without parameters that are not executed synchronously and they are provided asynchronously through a later FSCI event. It is the responsibility of the FSCI Host layer to keep the application-allocated memory between the time of the request and the completion of the event with the actual values of the output parameters and populate them accordingly.

The Bluetooth Low Energy API execution inside the FSCI Host layer first waits for gaining access to the serial interface through a mutex. Once the access is gained, the FSCI request is sent to the serial interface to the

blackbox. Then, by default, the serial interface response is received by polling until the whole FSCI packet is received. The other option available is to block the user task to wait for an OS event that is set by the FSCI module when the status is received. For more information on the FSCI module, see the Connectivity Framework Reference Manual. See Section 13 "References".

The API can have output parameters that are to be received immediately after the status of the request. In such as case, if the status of the request is 'success', the polling mechanism continues to receive the whole FSCI packet of the Bluetooth Low Energy event. The output parameters are obtained and the values are filled in the memory space provided by the application. After obtaining the status and optionally the event, the execution of the request is considered completed, the mutex to the serial interface is unlocked, and the execution flow is returned to the user calling context.

13 References

For more information, refer to the following documents:

- Bluetooth Low Energy Demo Applications User's Guide (*KW45_K32W1_BLEDAUG*)
- Bluetooth Low Energy Host Stack API Reference Manual (*KW45_K32W1_BLEHSAPIRM*)
- Bluetooth Low Energy Host Stack FSCI (Framework Serial Connectivity Interface) API Reference Manual (*KW45_K32W1_BLEHSFSCIAPIRM*)
- Connectivity Framework Reference Manual (*KW45_K32W1_CONNFWRM*)
- Bluetooth Low Energy CCC Digital Key R3 Application Note (*AN12791*)

14 Acronyms and abbreviations

The following acronyms are used in this document.

Table 29. Acronyms and abbreviations

Acronym	Description
Bluetooth LE	Bluetooth Low Energy
CCCD	Client Characteristic Configuration Descriptor
CSRK	Connection Signature Resolving Key
ELKE	EdgeLock Secure Enclave
FSCI	Framework Serial Connectivity Interface
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GATTDB	Generic Attribute Profile Database
HCI	Host Controller Interface
IRK	Identity Resolving Key
LTK	Long Term Key
LL	Link Layer
L2CAP	Logical Link Control and Adaptation Protocol
MTU	Maximum Transmission Unit
PDU	Protocol Data Unit
PAwR	Periodic Advertising with Responses
RPA	Resolvable Private Address
RSSI	Received Signal Strength Indicator
RTOS	Real Time Operating System
RX	Receiver
SDK	Software Development Kit
TX	Transmitter
WFI	Wait For Interrupt

15 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2022-2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

16 Revision history

This table summarizes revisions to this document.

Table 30. Revision history

Document ID	Release date	Description
UG10184 v.1.0	26 November 2024	Document is aligned to KW47 EAR 2.1 24.12.00-pvw2 release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Amazon Web Services, AWS, the Powered by AWS logo, and FreeRTOS — are trademarks of Amazon.com, Inc. or its affiliates.

Bluetooth — the Bluetooth wordmark and logos are registered trademarks owned by Bluetooth SIG, Inc. and any use of such marks by NXP Semiconductors is under license.

EdgeLock — is a trademark of NXP B.V.

IAR — is a trademark of IAR Systems AB.

Contents

1 Introduction 2
2 Prerequisites 3
2.1 Task and event queues 3
2.2 GATT database 3
2.3 Non-Volatile Memory (NVM) access 4
3 Bluetooth LE Host Stack Initialization and APIs 7
3.1 Initialization 7
3.2 Main function to initialize the Bluetooth LE Host Stack 7
3.3 HCI entry and exit points 8
3.4 Bluetooth LE Host Stack libraries and API availability 9
3.5 Synchronous and asynchronous functions 9
3.6 Radio TX Power level 10
4 Generic Access Profile (GAP) Layer 11
4.1 Peripheral setup 12
4.1.1 Advertising 12
4.1.2 Pairing and bonding (peripheral) 13
4.2 Central setup 17
4.2.1 Scanning 17
4.2.2 Initiating and closing a connection 19
4.2.3 Pairing and bonding (Central) 20
4.3 LE data packet length extension 24
4.4 Privacy feature 25
4.4.1 Introduction 25
4.4.2 Host privacy 27
4.4.3 Controller privacy 27
4.5 Setting PHY mode in a connection 29
4.6 Data management of bonded devices 29
4.6.1 Application removal of bonded devices data 32
4.7 Controller enhanced notifications 32
4.8 Extended advertising 35
4.8.1 Peripheral setup 36
4.8.2 Central setup 38
4.9 Periodic Advertising 40
4.9.1 Peripheral Setup 41
4.9.2 Central Setup 42
4.10 Periodic Advertising with Responses (PAwR) 43
4.10.1 Central Setup 43
4.10.2 Peripheral Setup 43
4.11 Encrypted Advertising Data 45
4.11.1 Central Setup 45
4.11.2 Peripheral Setup 45
4.12 L2CAP credit-based channels 45
4.13 Enhanced ATT 48
4.13.1 EATT Credits management 48
4.13.2 EATT Connection establishment 49
4.13.3 EATT Bearer reconfiguration 50
4.13.4 EATT Bearer disconnection 51
5 Generic Attribute Profile (GATT) Layer 52
5.1 Client APIs 52
5.1.1 Installing client callbacks 52
5.1.2 MTU exchange 55
5.1.3 Service and characteristic discovery 56
5.1.4 Reading and writing characteristics 65
5.1.5 Reading and writing characteristic descriptors 72
5.1.6 Resetting procedures 74
5.2 Server APIs 75
5.2.1 Server callback 75
5.2.2 Sending notifications and indications 76
5.2.3 Attribute write notifications 77
6 GATT database application interface 79
6.1 Writing and reading attributes 79
6.2 Finding attribute handles 79
7 Creating GATT database 81
7.1 Creating static GATT database 81
7.1.1 Declaring custom 128-bit UUIDs 81
7.1.2 Declaring a service 82
7.1.3 Declaring a characteristic 83
7.1.4 Static GATT database definition examples 84
7.2 Creating a GATT database dynamically 86
7.2.1 Memory considerations 86
7.2.2 Initialization and release 86
7.2.3 Adding services 86
7.2.4 Adding characteristics and descriptors 87
7.2.5 Removing services and characteristics 87
7.3 Gatt caching 87
7.3.1 Service change feature 87
7.3.2 Robust caching 88
8 Creating a Custom Profile 92
8.1 Defining custom UUIDs 92
8.2 Creating service functionality 92
8.3 GATT client interactions 93
9 Application Structure 95
9.1 Folder structure 95
9.2 Application main framework 96
9.2.1 Start task 96
9.2.2 Application messaging 97
9.3 Bluetooth LE Connection Manager 97
9.3.1 GAP generic event 97
9.3.2 GAP configuration 98
9.3.3 GAP connection event 98
9.3.4 Privacy 98
9.4 GATT database 99
9.5 RTOS specifics 99
9.5.1 Operating system selection 99
9.5.2 Bluetooth LE Host task configuration 99
9.6 Board configuration 100
9.7 Bluetooth Low Energy initialization 100
9.8 Bluetooth Low Energy Host Stack configuration 101
9.9 Profile configuration 102
9.9.1 Application code 103
9.10 Multiple connections 105
9.11 Bluetooth address generation 106
9.12 Repeated attempts 106
9.13 Advanced Secure Mode (kw45_k32w) 106
10 Low-Power Management 108
10.1 System considerations 108

10.2 When/how to enter low power 108
11 Over the Air Programming (OTAP) 110
11.1 General functionality 110
11.2 Bluetooth Low Energy OTAP service-profile .. 111
11.2.1 OTAP service and characteristics 111
11.2.2 OTAP server and OTAP client interactions 112
11.3 Bluetooth LE OTAP protocol 113
11.3.1 Protocol design considerations 113
11.3.2 Bluetooth Low Energy OTAP commands 114
11.3.3 OTAP client–server interactions 120
11.4 Bluetooth Low Energy OTAP image file format 121
11.4.1 Bluetooth Low Energy OTAP header 122
11.5 Building Bluetooth Low Energy OTAP image file from SREC file 124
11.6 Building Bluetooth Low Energy OTAP image file from BIN file 127
11.7 Bluetooth Low Energy OTAP application integration 129
11.7.1 OTAP server 129
11.7.2 OTAP client 134
11.8 Secured OTAP 146
12 Creating a Bluetooth LE application when the Host Stack runs on another processor 147
12.1 Serial manager and FSCI configuration 147
12.1.1 Serial manager initialization 147
12.1.2 FSCI configuration and initialization 147
12.1.3 FSCI handlers (GAP, GATT, and GATTDB) registration 148
12.2 Bluetooth Low Energy Host Stack initialization 148
12.3 GATT database configuration 149
12.4 FSCI host layer 149
13 References 151
14 Acronyms and abbreviations 152
15 Note about the source code in the document 153
16 Revision history 154
Legal information 155