Push every boundary.™

CSR µEnergy®

GATT Client

Application Note

Issue 3

Document History

Revision	Date	History
1	05 FEB 14	Original publication of this document
2	06 FEB 14	Update current measurement values
3	02 JUL 14	Updated for SDK 2.4.0

Contacts

General information	www.csr.com
Information on this product	sales@csr.com
Customer support for this product	www.csrsupport.com
More detail on compliance and standards	product.compliance@csr.com
Help with this document	comments@csr.com

Trademarks, Patents and Licences

Unless otherwise stated, words and logos marked with ™ or ® are trademarks registered or owned by CSR plc and/or its affiliates.

Bluetooth® and the Bluetooth logos are trademarks owned by Bluetooth SIG, Inc. and licensed to CSR.

Other products, services and names used in this document may have been trademarked by their respective owners.

The publication of this information does not imply that any licence is granted under any patent or other rights owned by CSR plc or its affiliates.

CSR reserves the right to make technical changes to its products as part of its development programme.

While every care has been taken to ensure the accuracy of the contents of this document, CSR cannot accept responsibility for any errors.

Use of this document is permissible only in accordance with the applicable CSR licence agreement.

Safety-critical Applications

CSR's products are not designed for use in safety critical devices or systems such as those relating to: (i) life support; (ii) nuclear power; and/or (iii) civil aviation applications, or other applications where injury or loss of life could be reasonably foreseeable as a result of the failure of a product. The customer agrees not to use CSR's products (or supply CSR's products for use) in such devices or systems.

Performance and Conformance

Refer to www.csrsupport.com for compliance and conformance to standards information.

Contents

Tables, Figures and Equations

1. Introduction

This document demonstrates how to create a Generic Attribute Profile (GATT) Client application. An example GATT Client application is provided with the CSR µEnergy Software Development Kit (SDK) and should be used together with this document.

GATT Client applications access data and resources provided by GATT Servers over the air using Bluetooth technology. The GATT Server may enforce access permissions on the data. The data may be protected using various security levels, so only authorised GATT Clients may access selected values. The GATT Client may need to negotiate an encrypted link with the GATT Server to access some or all data to prevent unauthorised third parties from intercepting the data.

As a minimum, the GATT Client application must support the Generic Access Profile (GAP) to discover which services are provided by each GATT Server, and to understand security levels. The GATT Client must support GATT, built upon the Attribute Protocol (ATT), which defines how GATT service characteristics are accessed.

Optionally, the application may support other profiles. The application may require access to hardware, e.g. to perform various actions dependent on the values reported by GATT services.

This document demonstrates how a typical GATT Client application may be implemented using the example application as a template. It then demonstrates how the example application may be extended to access additional GATT services.

1.1. Application Overview

1.1.1. Profiles Supported

This example application supports GATT to access characteristics provided by a GATT Server. It supports GAP to discover services and establish connections with GATT Servers.

The example application uses ATT to transport data between Client and Server.

See the *GATT Server Application Note* for an overview of the supported profiles. See the *ATT Specification*, *GATT Specification* and *GAP Specification* for more detailed information.

1.1.2. Application Topology

The example application implements GATT in the Client role, and GAP in the Central role, see Table 1.1:

Role	GAP Service	GATT Service	Device Information Service	Battery Service
GATT Role	Client	Client	Client	Client
GAP Role	Central	Central	Central	Central

Table 1.1: Application Topology

Role	Responsibility
GATT Client	It initiates commands and requests towards a server and receives responses, notifications and indications sent by the GATT Server.
GAP Central	It initiates a connection towards the peer device and acts as a master in the connection.

Table 1.2: Responsibilities

For more information about GATT Client and GAP Central, see *Bluetooth Core Specification version 4.1*.

1.1.3. Services

This application acts as a GATT Client for the following services:

- Device Information (version 1.1)
- Battery (version 1.0)
- GAP
- GATT

GAP and GATT services are mandated by *Bluetooth Core Specification version 4.1*. The Device Information Service and Battery Service are optional as shown in Figure 1.1.

For more information on the Device Information and Battery services, see *Device Information Service Specification version 1.1* and *Battery Service Specification version 1.0* respectively. For more information on GATT and GAP services, see *Bluetooth Core Specification version 4.1*.

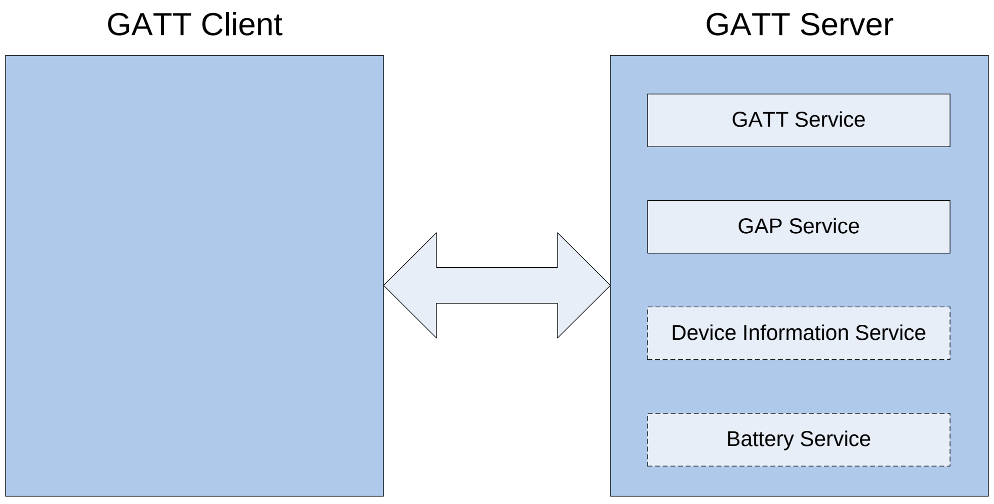


Figure 1.1: Primary Services

2. Using the Application

This section describes how the GATT Client application can be used with GATT Servers.

2.1. Demonstration Kit

The application can be demonstrated using the following components:

Component	Hardware	Application
GATT Client	▪ CSR1011 Development Board, e.g. CNS12016	GATT Client Application
GATT Server	▪ CSR101x Development Board, e.g. CNS12016 or CNS10004	GATT Server, e.g. GATT Server Application
Host	▪ PC ▪ CSR µEnergy USB to SPI Adapter Board, CNS10020	Terminal emulator, e.g. HyperTerminal or PuTTY

Table 2.1: Demonstration Components

2.1.1. GATT Client

The SDK is used to build and download the GATT Client application to the development board. See the *CSR µEnergy xIDE User Guide* for further information.

Figure 2.1 shows a CSR1011 board with the switch set to receive power from the battery. Set the switch to **USB** to receive power from the mini-USB connector when debugging and downloading the firmware image. During normal use, set the switch to the **Batt** position to use the coin cell battery.

2.1.1.1. To power on the device:

- Ensure the power source is provided i.e. either the mini-USB cable is attached to the USB to SPI adapter (connected to the PC), or the coin cell battery is fitted.
- Ensure that the corresponding power source is selected using the power slider switch.

2.1.1.2. To power off the device, either:

- Remove the power source (currently selected by the power slider switch) i.e. by disconnecting the mini-USB cable, or removing the battery, or
- Ensure the power slider switch is set to a power source that is not provided.

Note:

When the mini-USB cable has been disconnected, wait at least 1 minute before switching the board to receive power from the coin cell. This allows any residual charge received from the SPI connector to dissipate.

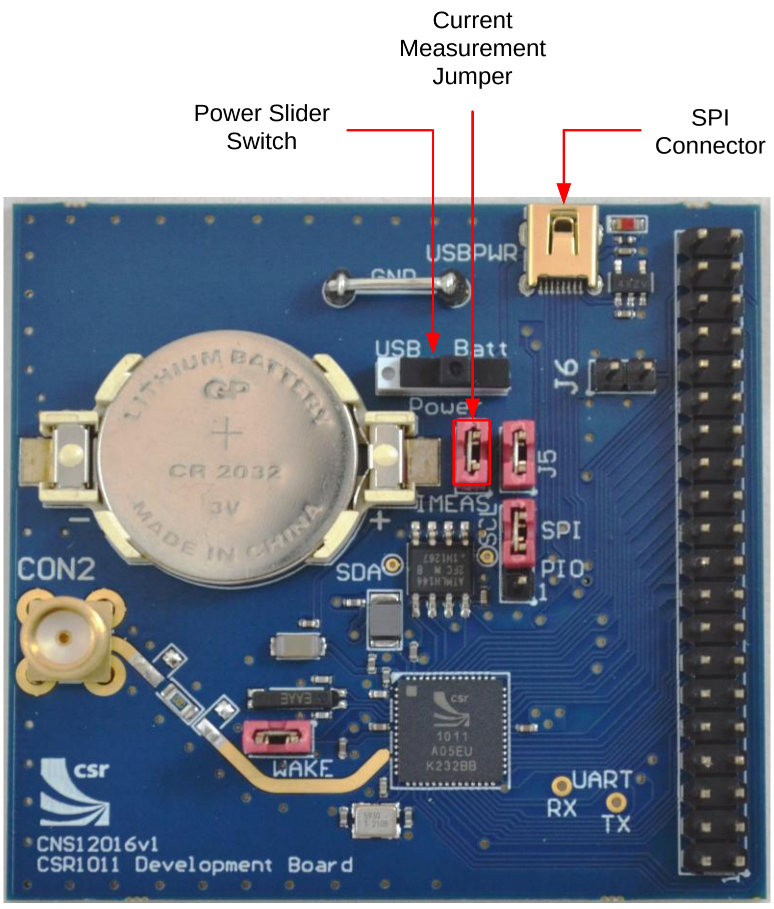


Figure 2.1: CSR1011 Development Board

2.1.2. GATT Server

The GATT Client will connect to any GATT Server that is broadcasting advertisements, however for demonstration purposes it may be desirable to use the GATT Server example application provided in the SDK.

For more information regarding the use of the GATT Server example application see the *GATT Server Application Note*.

2.1.3. Host

The GATT Client optionally provides debug output over a UART interface, typically connected to a PC through a CSR µEnergy USB to SPI adapter board, see Figure 2.2. To receive the debug output the host must run a terminal emulator such as HyperTerminal or PuTTY connected to the appropriate COM port. By default the UART is configured for 115200 baud, 8 data bits, 1 stop bit, and no parity bit.

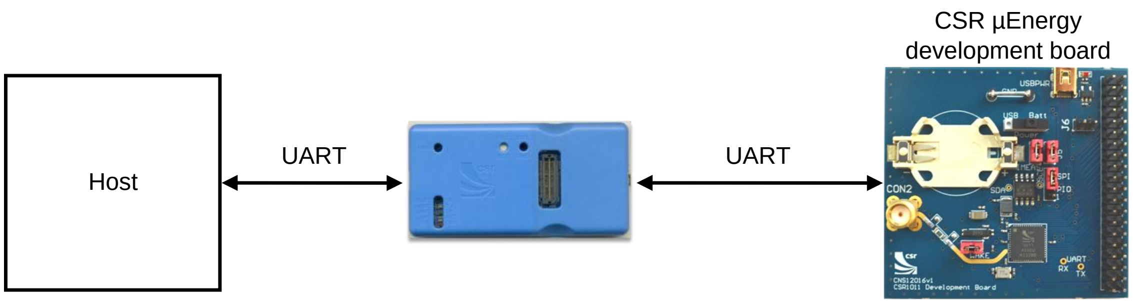
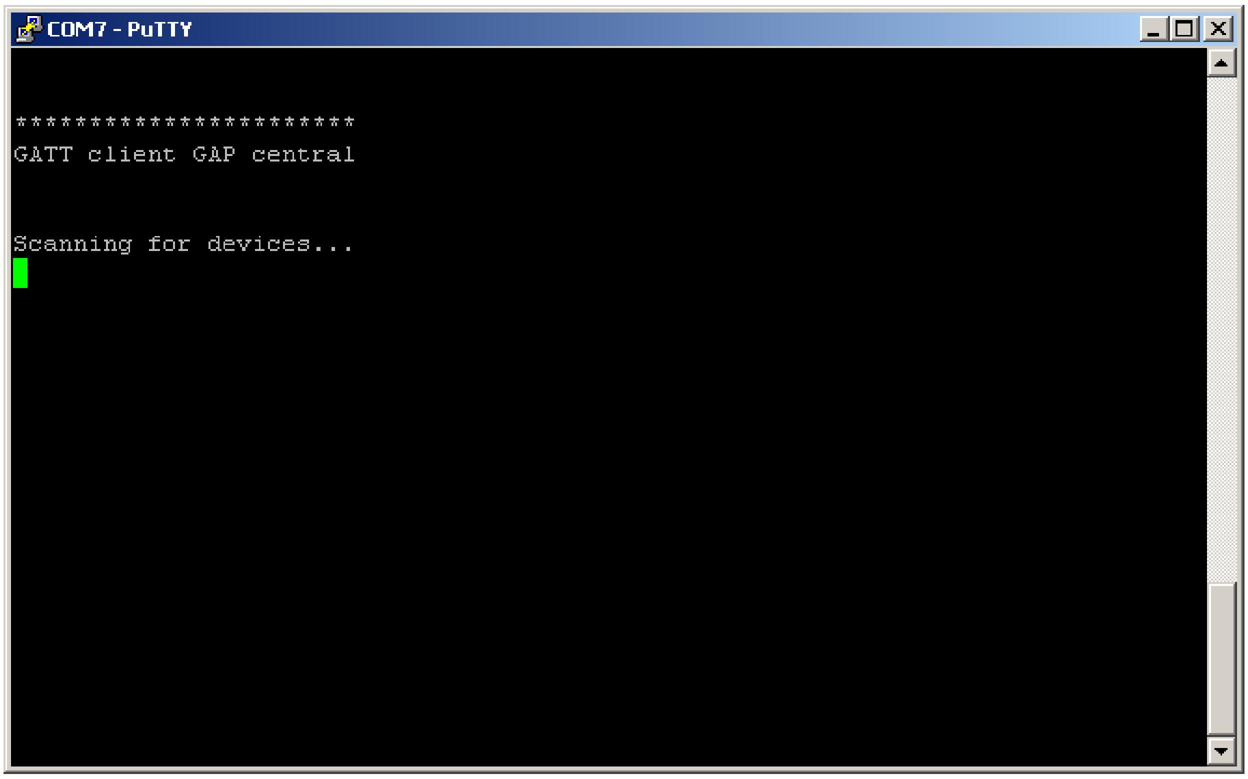


Figure 2.2: Connecting GATT Client Device to a Host

2.2. Demonstration Procedure

1. Connect the development board to a PC using the CSR µEnergy USB to SPI adapter board.
2. Run a terminal emulator and select the appropriate COM port and communications configuration, see section 2.1.3.
3. Build and download the GATT Client application to the development board. See the *CSR µEnergy xIDE User Guide* for further information. When the application is running a message is displayed over UART, see Figure 2.3:

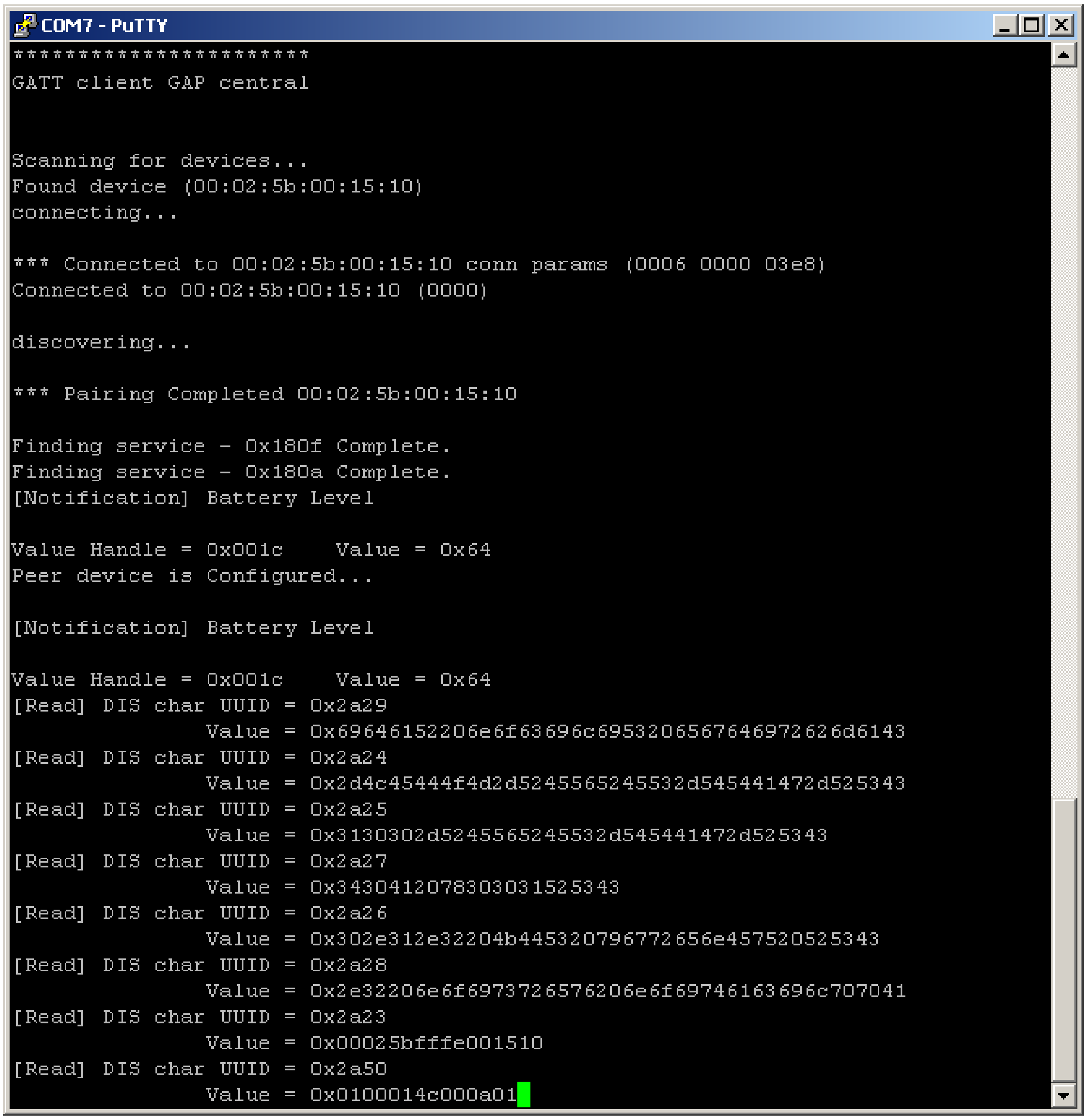


```
***********************
GATT client GAP central


Scanning for devices...
```

Figure 2.3: Terminal Output Following Initialisation

4. After the example application has initialised it will scan for advertisements broadcast by a GATT Server. It will then connect to the GATT Server and initiate a Discovery Procedure to find which services are available. If the Device Information Service is found, the details are displayed over the UART. Similarly, if the Battery Service is found, then the battery level is displayed and a request for notifications made, see Figure 2.4:



```
***********************
GATT client GAP central


Scanning for devices...
Found device (00:02:5b:00:15:10)
connecting...

*** Connected to 00:02:5b:00:15:10 conn params (0006 0000 03e8)
Connected to 00:02:5b:00:15:10 (0000)

discovering...

*** Pairing Completed 00:02:5b:00:15:10

Finding service - 0x180f Complete.
Finding service - 0x180a Complete.
[Notification] Battery Level

Value Handle = 0x001c    Value = 0x64
Peer device is Configured...

[Notification] Battery Level

Value Handle = 0x001c    Value = 0x64
[Read] DIS char UUID = 0x2a29
               Value = 0x69646152206e6f63696c69532065676469726 26d6143
[Read] DIS char UUID = 0x2a24
               Value = 0x2d4c45444f4d2d52455652455 32d545441472d525343
[Read] DIS char UUID = 0x2a25
               Value = 0x3130302d52455652455 32d545441472d525343
[Read] DIS char UUID = 0x2a27
               Value = 0x343041207830303152 5343
[Read] DIS char UUID = 0x2a26
               Value = 0x302e312e32204b4453207967726 56e457520525343
[Read] DIS char UUID = 0x2a28
               Value = 0x2e32206e6f697372657620 6e6f69746163696c707041
[Read] DIS char UUID = 0x2a23
               Value = 0x00025bfffe001510
[Read] DIS char UUID = 0x2a50
               Value = 0x0100014c000a01
```

Figure 2.4: Terminal Output Following Service Discovery

5. Once the Discovery procedure is complete, the GATT Client will scan for the next advertising GATT Server.

6. The GATT Client will continue running until power is removed from the device or execution is interrupted using xIDE.

3. Application Structure

The example GATT Client application project contains service files and application files:

- Service files provide routines for accessing and manipulating the service characteristics available on a GATT Server. Typically each service will have its own dedicated file, together with corresponding header files. This improves readability, portability and maintenance, and makes it easier to support additional services.
- Application files implement application specific routines in a clearly identifiable way. Hardware- and GATT-specific routines are isolated in separate files.

3.1. Source Files

Table 3.1 lists the source files used in the example GATT Client application and their purpose:

File Name	Purpose
`debug_interface.c`	Implements routines for sending debug messages to the UART, if enabled.
`gap_access.c`	Implements routines that configure the GAP Service.
`gatt_access.c`	Implements routines for triggering scanning and GATT service Discovery Procedures
`gatt_client.c`	**Application entry point.** Implements all entry functions: `AppPowerOnReset()`, `AppInit()`, `AppProcessSystemEvent()` and `AppProcessLmEvent()`. It also contains handling functions for all the LM and system events, and implements the application's state machine.
`battery_service_data.c`	Implements a structure for storing the discovered Battery Service handles. It also implements functions for reading and configuring Battery Service characteristics on a GATT Server.
`dev_info_service_data.c`	Implements structure for storing the discovered Device Information Service handles. It also implements functions for reading Device Information Service characteristics on a GATT Server.
`nvm_access.c`	Implements routines for reading and writing NVM.

Table 3.1: Source Files

3.2. Header Files

Table 3.2 lists the header files included in the example GATT Client application and their purpose:

File Name	Purpose
`debug_interface.h`	Contains macro definitions and function prototypes for optionally sending debug messages to the UART
`gap_access.h`	Contains function prototypes for configuring the GAP Service.
`gatt_access.h`	Contains macro definitions, user defined data type definitions and function prototypes that are used across the application.
`gatt_client.h`	Contains user defined data type definitions and prototypes of externally referred functions defined in the file `gatt_client.c`.
`battery_service_data.h`	Contains macro definitions and prototypes of externally referred functions defined in the file `battery_service_data.c`.
`battery_uuids.h`	Contains macros for the UUIDs of the Battery Service.
`dev_info_service_data.h`	Contains macro definitions and prototypes of externally referred functions defined in the file `dev_info_service_data.c`.
`dev_info_uuids.h`	Contains macros for the UUIDs of the Device Information Service.
`gap_conn_params.h`	Contains macros for the connection and scanning parameters.
`nvm_access.h`	Contains prototypes of the externally referred functions defined in the file `nvm_access.c`.
`user_config.h`	Contains macros for user configuration data.

Table 3.2: Header Files

4. Code Overview

4.1. Application Entry Points

4.1.1. AppPowerOnReset

This is the first application entry point to be invoked by the firmware after the device has been powered on. It is also called after a wakeup from hibernation or dormant sleep states.

4.1.2. AppInit

This function is called once following a call to `AppPowerOnReset()`. The most recent sleep state is provided to the application via a parameter to this function. This function performs the following:

- Initialises the application timers
- Enables the UART interface (if compiled for debug messages)
- Configures the GATT service for Client role, and GAP service for Central role
- Configures NVM

4.1.3. AppProcessSystemEvent

This function is called whenever a system event (e.g. a battery low notification) is generated. Appropriate event information is delivered to the application via its parameters. The example application does not handle any system events, so at present this function does nothing.

4.1.4. AppProcessLmEvent

This function is invoked whenever a LM-specific event is received by the system. Appropriate event information is delivered to the application via its parameters. The following events are handled:

4.1.4.1. LS Events

- `LS_CONNECTION_UPDATE_SIGNALLING_IND`: This event is received when the application receives a connection parameter update request from the peer device. It is normally handled by a call to the Firmware Application Program Interface (API) function `LsConnectionUpdateSignalingRsp()` accepting or rejecting the proposed connection parameters. At present the example application ignores this event.

4.1.4.2. SM Events

- `SM_KEY_REQUEST_IND`: This event indicates that the Security Manager could not find security keys for the host in persistent store. The application responds with either the security keys or a null pointer.
- `SM_KEYS_IND`: This event contains the keys and associated security information used on a connection that has completed Short Term Key Generation or Transport Specific Key Distribution. The application extracts the keys from this event and stores them in NVM for future use.
- `SM_SIMPLE_PAIRING_COMPLETE_IND`: This event is received upon completion of the Pairing Procedure.

4.1.4.3. LM Events

- `LM_EV_CONNECTION_COMPLETE`: This event is received upon connection completion. The application retrieves the connection parameter values from this event.
- `LM_EV_ENCRYPTION_CHANGE`: This event is received upon encryption completion. The application ignores this event.
- `LM_EV_DISCONNECT_COMPLETE`: This event is received upon disconnection. The application restarts scanning upon receiving this event.
- `LM_EV_ADVERTISING_REPORT`: This event is received on discovering an advertising device. The advertisement is checked for supported services, and if at least one supported service is found, scanning is paused and an attempt is made to connect to the device.

4.1.4.4. GATT Events

- GATT_CONNECT_CFM: This event is received on connection establishment. On receiving this event the application starts the Discovery Procedure. If the peer device has a Resolvable Random Address the application also starts the Pairing Procedure.
- GATT_DISCONNECT_IND: Indicates that the peer device wishes to disconnect, or that the link has been lost. The underlying Bluetooth connection is not released until the firmware has sent LM_EV_DISCONNECT_COMPLETE. Therefore the example application does nothing until LM_EV_DISCONNECT_COMPLETE is received.
- GATT_DISCONNECT_CFM: Confirms completion of the GattDisconnectReq() function. The underlying Bluetooth connection is not released until the firmware has sent LM_EV_DISCONNECT_COMPLETE. Therefore the example application does nothing until LM_EV_DISCONNECT_COMPLETE is received.
- GATT_CANCEL_CONNECT_CFM: This event confirms completion of the GattCancelConnectReq() function. It means that a connection attempt was made, but that it was not completed in time. The application will start scanning for the next device.
- GATT_WRITE_CHAR_VAL_CFM: This event is received on completion of a write characteristic value procedure. Depending upon the write result and the attribute handle, the application takes appropriate action.
- GATT_READ_CHAR_VAL_CFM: This event is received on completion of a read characteristic value procedure. Depending upon the read result and the attribute value read the application takes appropriate action.
- GATT_IND_CHAR_VAL_IND: This event is received when the application receives an indication for a characteristic on the peer device. Depending on the attribute handle indicated the application takes the appropriate action.
- GATT_NOT_CHAR_VAL_IND: This event is received when the application receives a notification for a characteristic on the peer device. Depending upon the attribute handle notified the application takes appropriate action.

The following events are received during the Discovery Procedure. See section 6.3 for more information:

- GATT_DISC_PRIM_SERV_BY_UUID_IND: This event is received each time a primary service with a specific UUID is found on the peer device. The event may be received multiple times depending on how many instances of the service are found.

  The application retrieves the service handles from the received event.
- GATT_DISC_PRIM_SERV_BY_UUID_CFM: This event is received on completion of the GattDiscoverPrimaryServiceByUuid() function, indicating that all instances of the primary service with the specified UUID have been discovered.

  On receiving this event the application triggers discovery of the next primary service. When all the supported services have been discovered the procedure for discovering all characteristics of each discovered service is started.
- GATT_CHAR_DECL_INFO_IND: This event is received for each service characteristic discovered by on the peer device. A service may contain multiple characteristics, and this event will be received for each one.

  The application retrieves the characteristic handles from this event.
- GATT_DISC_SERVICE_CHAR_CFM: This event is received on completion of the GattDiscoverServiceChar() function.

  On receiving this event the application will start the characteristic descriptors discovery procedure.
- GATT_CHAR_DESC_INFO_IND: This event is received for every characteristic descriptor discovered on the peer device. As each characteristic may have multiple descriptors, this event may be received multiple times.

  If the discovered descriptor is a Client Characteristic Configuration Descriptor then the application stores the discovered handle, otherwise the event is ignored.

- `GATT_DISC_ALL_CHAR_DESC_CFM`: This event is received on completion of the `GattDiscoverAllCharDescriptors()` function, indicating all the descriptors for a given characteristics have been discovered.

  On receiving this event, the application repeats the descriptor discovery procedure for all the characteristics one by one. After the procedure is complete for all the characteristics in the current service, the characteristics of the next service are discovered. When all the descriptors of all the characteristics of all the supported services have been discovered, the application stops the Discovery Procedure.

4.2. Internal State Machine

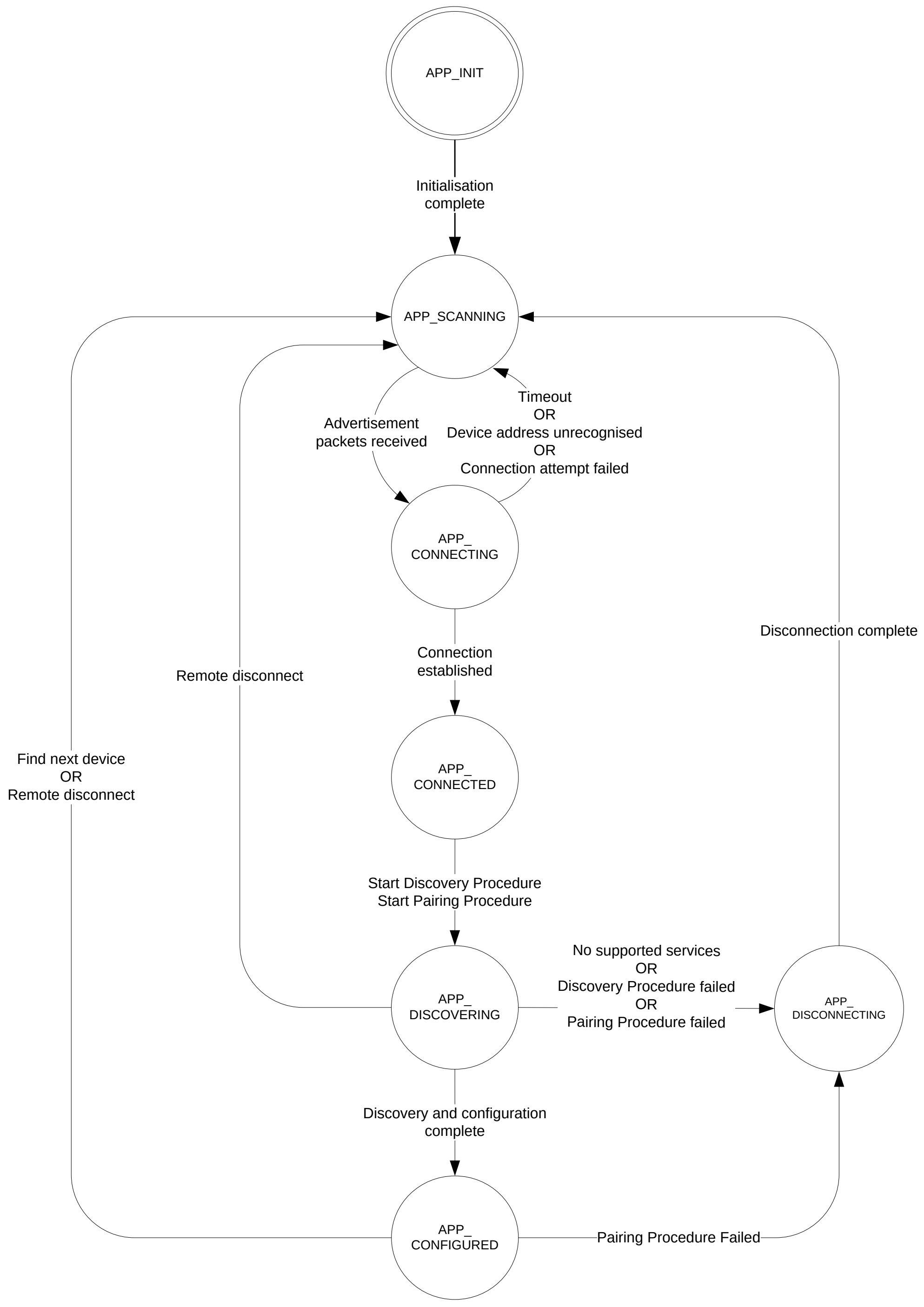

APP_INIT
Initialisation complete
APP_SCANNING
Advertisement packets received
Timeout OR Device address unrecognised OR Connection attempt failed
APP_ CONNECTING
Connection established
APP_ CONNECTED
Start Discovery Procedure Start Pairing Procedure
APP_ DISCOVERING
No supported services OR Discovery Procedure failed OR Pairing Procedure failed
APP_ DISCONNECTING
Discovery and configuration complete
APP_ CONFIGURED
Pairing Procedure Failed
Disconnection complete
Remote disconnect
Find next device OR Remote disconnect

Figure 4.1: Internal State Machine

4.2.1. APP_INIT

When the device is powered on or reset, the application enters this state. In this state, the application initialises timers and local data, installs the GATT client functionality, configures the NVM manager and initialises the Security Manager. When initialisation is complete, the application moves to the **APP_SCANNING** state.

4.2.2. APP_SCANNING

In this state the application configures the GAP Service for the GAP Central role (`GapSetMode()`) and instructs the firmware to scan for advertisements (`LsStartStopScan()`). Any advertisement report received (`LM_EV_ADVERTISING_REPORT`) is optionally filtered for supported services (see section 7.5) and if passed causes the application to stop scanning (`LsStartStopScan()`) and send a connection request (`GattConnectReq()`). The application then moves to the **APP_CONNECTING** state.

4.2.3. APP_CONNECTING

In this state the application waits for the result of the connection request. If the connection process takes longer than is allowed by the Connection Request Timer (see section 7.6) or is rejected by the peer device, then the application returns to the **APP_SCANNING** state.

After a connection is established (`GATT_CONNECT_CFM`) the application reads NVM to check whether the Client device was previously bonded with the peer device, and if so, link keys are retrieved from NVM. The application then moves to the **APP_CONNECTED** state.

4.2.4. APP_CONNECTED

If pairing is enabled (see section 7.3) and the peer device has provided a Resolvable Random Address, the application will wait for the peer device to initiate pairing. If pairing has not been initiated within the time specified by the Pairing Timer (see section 7.6) then the application will initiate the Pairing Procedure (see section 6.2).

After a duration specified by the Discovery Start Timer (see section 7.6) the application commences the Discovery Procedure (see section 6.3) and moves to the **APP_DISCOVERING** state.

4.2.5. APP_DISCOVERING

In this state the application interrogates the GATT Server to find all supported services. The characteristic handles for the supported services are stored locally for subsequent access.

When all the supported services have been discovered, the application configures those service characteristics for which it requires notification or indication reports. Following configuration, the application enters the **APP_CONFIGURED** state.

4.2.6. APP_CONFIGURED

The application is now ready to start running its main function: for example, monitoring the services and reporting updates on a display, or perhaps sending control signals to the peer device in response to commands received from a user interface. The example application just reports the GATT Server's Device Information Service characteristic values to UART if debugging is enabled (see section 7.4) before returning to the **APP_SCANNING** state and scanning for another device. If the peer device disconnects during this state the application resets its data structures for that connection and returns to the **APP_SCANNING** state.

4.2.7. APP_DISCONNECTING

This state is entered when an error occurs following connection. The application disconnects from the peer device and when the disconnection is complete resets its data structures for that connection and returns to the **APP_SCANNING** state.

The connection to the peer device may be broken at any time following successful connection. In this case the data structures for the peer device are reset and if the application is in the **APP_DISCOVERING** or **APP_CONFIGURED** state (that is, it is not in the process of connecting to another device) it will return to the **APP_SCANNING** state, otherwise the application remains in its current state.

5. NVM Memory Map

The application stores keys for each device it is paired with in NVM so that this information is not lost on power cycle or reset. The memory map is implemented in macros defined in `gatt_client.c`, and listed for the first three paired devices in Table 5.1:

Entity Name	Type	Size of Entity (Words)	NVM Offset (Words)
Sanity Word	uint16	1	0
Bonded Flag [0]	boolean	1	1
Link Keys [0]	structure	38	2
Bonded Flag [1]	boolean	1	40
Link Keys [1]	structure	38	41
Bonded Flag [2]	boolean	1	79
Link Keys [2]	structure	38	80

Table 5.1: NVM Memory Map

Notes:

1. The application does not pack data while storing it into NVM so storing a boolean value takes one word of NVM memory.
2. The Sanity Word is a 16-bit value unique to each application that allows the application to check whether the NVM has been initialised and whether it holds relevant information.

6. Application Procedures

6.1. Scanning Procedure

The GATT Client needs to scan for relevant advertisements in order to find a GATT Server to connect to. Advertisements and scan responses may include the following information:

- Device name (full or shortened)
- Appearance
- Preferred connection parameters
- List of the main application services (through their UUIDs)

6.1.1. Connection Establishment

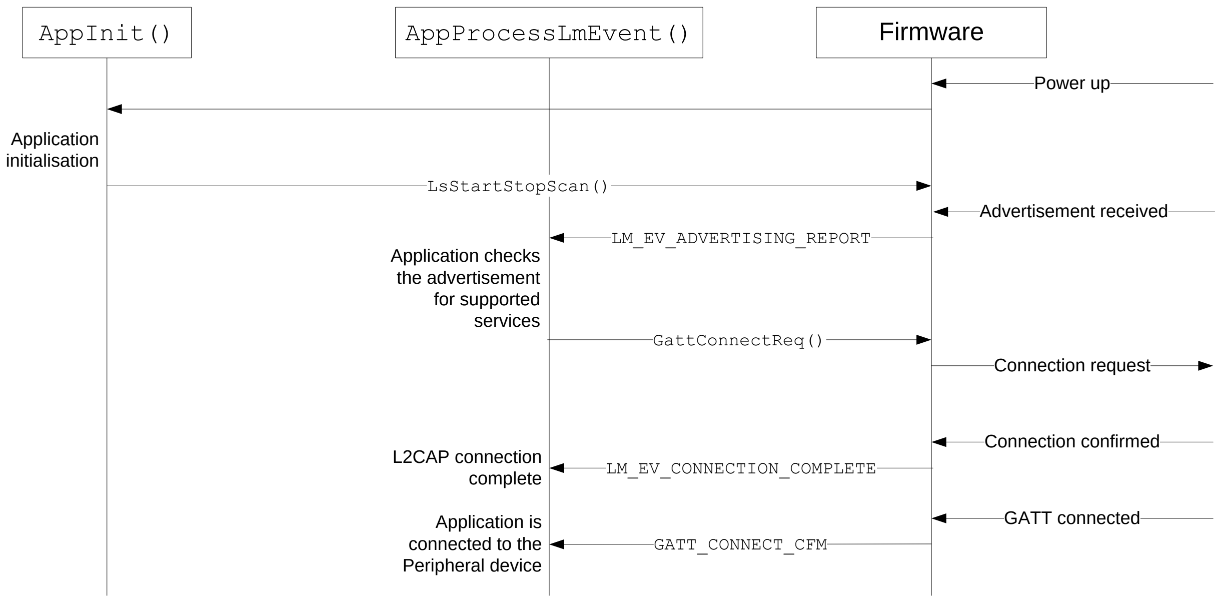


Figure 6.1: Sequence of Events in a Typical Connection Establishment Scenario

Figure 6.1 shows the various steps and events associated with a Peripheral device when a Central device initiates a connection. The Central device is not limited to one connection; scanning may continue if the application supports connection to multiple Peripheral devices.

Note:

While the GAP Central role permits a device to connect to multiple Peripheral devices, the GATT Client example application supplied with the CSR µEnergy SDK may only connect to one device at a time.

6.1.2. Disconnection

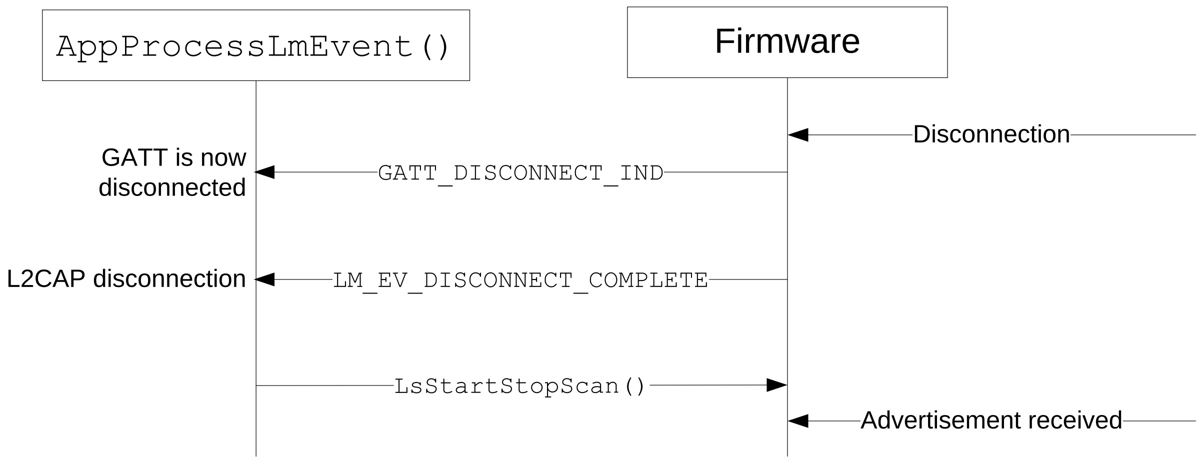


Figure 6.2: Sequence of Events Following a Disconnection

Figure 6.2 describes the sequence of operations for a typical Central device following a disconnection. Scanning is restarted (if not already running) following the L2CAP disconnection. The `GATT_DISCONNECT_IND` event will not be issued to the application unless a successful GATT connection has been established.

6.2. Pairing Procedure

The example GATT Client supports pairing with a GATT Server but will only initiate pairing if the `PAIRING_SUPPORT` macro is defined in `user_config.h`.

When the `PAIRING_SUPPORT` macro is defined the application starts the Pairing Procedure after a connection has been established (i.e. upon entering the **APP_CONNECTED** state), or whenever an attempt is made to access a service characteristic that requires authorisation and/or authentication:

1. If the GATT Server has supplied a Resolvable Random Address then the Pairing Procedure exits to allow the GATT Server to initiate the pairing process.
2. The Firmware API function `SMRequestSecurityLevel()` is called to start security procedures on the link. This will result in the event `SM_KEY_REQUEST_IND` if the Security Manager cannot find the security keys for the peer device in its persistent store.
3. If the `SM_KEY_REQUEST_IND` event is received the application checks to see whether it has previously bonded with the peer device by looking for its address in the NVM. If so the keys are extracted from NVM and passed to the Security Manager using the Firmware API function `SMKeyRequestResponse()`. If not, `NULL` is passed instead and new keys will be generated.
4. If pairing is successful and `SMKeyRequestResponse()` was called with `NULL`, then the keys generated during the pairing process will be returned by the Security Manager in the `SM_KEYS_IND` event. The application checks whether it has previously bonded with the peer device, and if not stores the keys in NVM so that they are accessible next connection, and are not lost should the device be reset or power cycled.
5. Once the pairing process is complete the application receives event `SM_SIMPLE_PAIRING_COMPLETE_IND` from the Security Manager. This signals the end of the Pairing Procedure. If the event indicates that the PIN or key was missing, and the peer device was already bonded, then the Pairing Procedure is repeated, but this time `NULL` is passed to the Security Manager instead of the previously stored keys, to force new keys to be generated. Otherwise if the event indicates that pairing failed the Client disconnects from the peer device and starts scanning for the next device.

Figure 6.3 illustrates the Pairing Procedure when the GATT Client initiates pairing with the GATT Server:

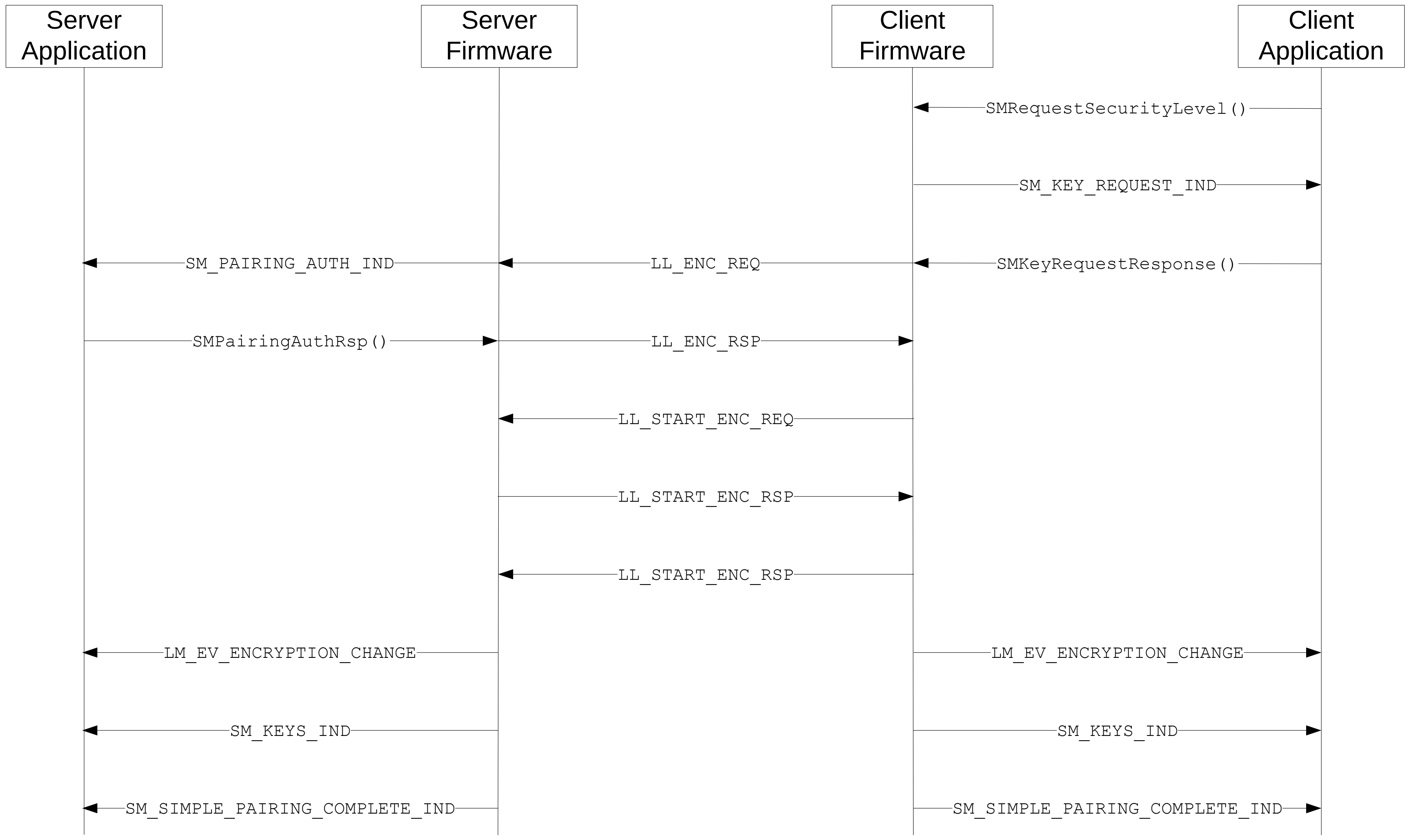


Figure 6.3: Pairing Procedure

6.3. Discovery Procedure

6.3.1. Overview

The application explores the Server's GATT database by performing a three step Discovery Procedure for each service present in the application's supported services list, `g_supported_services[]`:

Primary Service Discovery:

- The application uses the Firmware API function `GattDiscoverPrimaryServiceByUuid()` to perform the service discovery. The service's UUID is passed to the function.
- The application receives a `GATT_DISC_PRIM_SERV_BY_UUID_IND` event for each instance of the service found in the peer device's GATT database. The application extracts the service start and end handles from this event.
- If the service is not present in the GATT database, the application does not receive this event, and the Discovery Procedure moves onto the next service in the application's supported services list.
- The application receives the event `GATT_DISC_PRIM_SERV_BY_UUID_CFM` on service discovery completion.

Characteristic Discovery:

- The application uses the Firmware API function `GattDiscoverServiceChar()` to perform the service characteristic discovery. The service's start and end handles from the Primary Service Discovery are passed to the function.
- The application receives a `GATT_CHAR_DECL_INFO_IND` event for each characteristic discovered. The application extracts the characteristic start handles from these events. The characteristic end handle has to be calculated by the application.
- On completion of the characteristic discovery procedure, the application receives the event `GATT_DISC_SERVICE_CHAR_CFM`.

Characteristic Descriptor Discovery:

- The application only performs characteristic descriptor discovery on the characteristics that it is interested in for each service. The Firmware API function `GattDiscoverAllCharDescriptors()` is used to discover the descriptors of a particular characteristic.
- The application receives the `GATT_CHAR_DESC_INFO_IND` event for each descriptor discovered. The example application is only interested in client characteristic configuration descriptors. If the received event is for the client characteristic configuration descriptor, it extracts the descriptor handle from the event.
- On completion of characteristic descriptor discovery, the application receives the event `GATT_DISC_ALL_CHAR_DESC_CFM`.
- The characteristic descriptor discovery is performed on all the relevant characteristics one by one.

If the Discovery Procedure fails at any of the above mentioned steps, the control moves to the next service in the service list. After performing discovery for all the services the application starts configuring services with the function `appGattConfigureServices()` which marks the completion of the GATT database discovery.

Figure 6.4 illustrates the Discovery Procedure. Application function calls are in italics, and Firmware API function calls are in normal text:

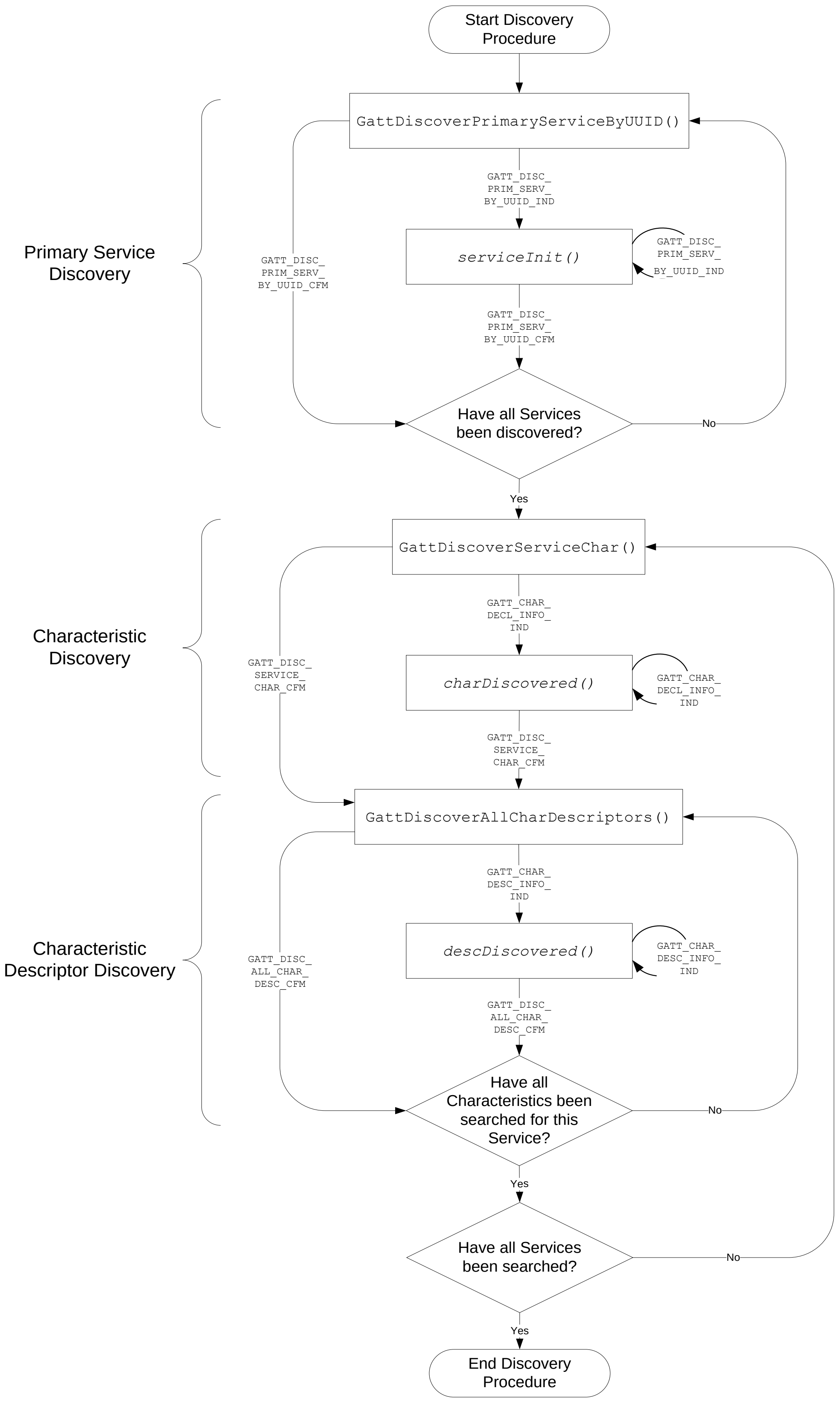

Start Discovery Procedure
GattDiscoverPrimaryServiceByUUID()
GATT_DISC_PRIM_SERV_BY_UUID_IND
serviceInit()
GATT_DISC_PRIM_SERV_BY_UUID_IND
GATT_DISC_PRIM_SERV_BY_UUID_CFM
GATT_DISC_PRIM_SERV_BY_UUID_CFM
Have all Services been discovered?
No
Yes
Primary Service Discovery
GattDiscoverServiceChar()
GATT_CHAR_DECL_INFO_IND
charDiscovered()
GATT_CHAR_DECL_INFO_IND
GATT_DISC_SERVICE_CHAR_CFM
GATT_DISC_SERVICE_CHAR_CFM
Characteristic Discovery
GattDiscoverAllCharDescriptors()
GATT_CHAR_DESC_INFO_IND
descDiscovered()
GATT_CHAR_DESC_INFO_IND
GATT_DISC_ALL_CHAR_DESC_CFM
GATT_DISC_ALL_CHAR_DESC_CFM
Characteristic Descriptor Discovery
Have all Characteristics been searched for this Service?
No
Yes
Have all Services been searched?
No
Yes
End Discovery Procedure

Figure 6.4: Discovery Procedure

6.3.2. Implementation

The GATT database Discovery Procedure is implemented in `gatt_client.c`, `gatt_access.c` and `gatt_access.h` and calls various functions provided by each service to store GATT data:

- `gatt_client.c` contains a list of supported services implemented as a variable called `g_supported_services[]`. `gatt_client.c` is responsible for triggering the Discovery Procedure after successful connection to a Server.
- `gatt_access.c` implements the Discovery Procedure. Control is handed back to `gatt_client.c` when the Discovery Procedure is complete.
- `gatt_access.h` defines the various structures (`CHARACTERISTIC_T`, `ATTRIBUTE_T` and `SERVICE_FUNC_POINTERS_T`) used to organise the GATT data used by the service and identified during the Discovery Procedure.
- Each supported service provides a source file and a header file named after the service (e.g. `battery_service_data.c`, `battery_service_data.h`). These files provide a variable to store GATT data relevant to that service, a collection of functions that provide access to the GATT data, and a variable of type `SERVICE_FUNC_POINTERS_T` that provides service-independent access to the functions. This allows `gatt_access.c` to build a local representation of the Server's GATT database in a service independent fashion. Table 6.1 lists each function supported:

Function	Description
`serviceUuid()`	Returns the Service UUID and the UUID type (16- or 128-bit)
`isMandatory()`	Returns TRUE if the Server must support this service
`serviceInit()`	This is called when the service is discovered on the Server. It allows the service to initialise its data structures.
`checkHandle()`	This is called when the Server indicates a change in a characteristic value through a `GATT_IND_CHAR_VAL_IND` or `GATT_NOT_CHAR_VAL_IND` event. It returns TRUE if the handle supplied is supported by the service, or FALSE if not.
`getHandles()`	This returns the range of handles that the service supports. It is used during the Discovery Procedure to find all the characteristics for this service that are supported by the Server.
`charDiscovered()`	This is called during the Discovery Procedure for each service characteristic discovered on the Server. The service should store the characteristic data for future reference.
`descDiscovered()`	This is called during the Discovery Procedure for each characteristic descriptor discovered on the Server. The service should store the descriptor data for future reference.
`discoveryComplete()`	This is called when the Discovery Procedure has finished for this service. The service should store the supplied connection handle for future reference, and take any action required after the service has been discovered. Note that the Discovery Procedure may still be in operation for other supported services when this function is called, and it may be preferable to wait until all the supported services have been discovered before configuring individual services or accessing service characteristics on the Server.
`configureServiceNotif()`	This function may be used to configure service notifications. It is not called directly in the example application.
`serviceIndNotifHandler()`	This is called when the Server indicates a change in a characteristic value through a `GATT_IND_CHAR_VAL_IND` or `GATT_NOT_CHAR_VAL_IND` event. It

Function	Description
	allows the service to take action on a change in characteristic value.
`writeRequest()`	This callback is provided to request a write to a service characteristic on the Server. It is not used in the example application.
`writeConfirm()`	This callback is called when a `GATT_WRITE_CHAR_VAL_CFM` event is received in response to a request to update a characteristic value on the Server.
`readRequest()`	This callback may be used when the application wishes to read the value of a service characteristic.
`readConfirm()`	This function is called when a `GATT_READ_CHAR_VAL_CFM` event is received in response to a read request.
`configureService()`	This function is called after the Discovery Procedure completes to configure the service. This may involve setting up characteristics for notification or indication, or initialising characteristic values on the Server.
`isServiceFound()`	This function should return TRUE if the Server supports this service, or FALSE if not.
`resetServiceData()`	This function causes the service data structure to be reset. It is called upon Server disconnection and allows the service to free resources for future connections.

Table 6.1: Service Callback Functions

7. Customising the Application

7.1. Scanning Parameters

The macros for these values are defined in `gatt_access.h`. These values will affect the overall current consumption and response time of the application. See the *Bluetooth Core Specification version 4.1* for scanning parameter range.

Parameter Name	Description	Parameter Value
Scan Window	How long to scan for advertisements	400 ms
Scan Interval	How often to scan for advertisements	400 ms

Table 7.1: Scanning Parameters

7.2. Connection Parameters

The macros for these values are defined in `gap_conn_params.h`. These values will affect the overall current consumption and response time of the application. See the *Bluetooth Core Specification version 4.1* for connection parameter range.

Parameter Name	Description	Parameter Value
Minimum Connection Interval	Minimum time between the start of two consecutive connection events	20 ms
Maximum Connection Interval	Maximum time between the start of two consecutive connection events	20 ms
Slave Latency during Discovery Procedure	Number of connection events for which the slave is not required to listen to the master	0 intervals
Slave Latency following Discovery Procedure	Number of connection events for which the slave is not required to listen to the master	60 intervals
Supervision Timeout	Maximum time between receiving packets before considering the connection to be broken	10 s
Scan Interval	Scan interval during connection establishment	400 slots
Scan Window	Scan window during connection establishment	400 slots

Table 7.2: Connection Parameters

7.3. Pairing Support

The GATT Client is configured to initialise pairing where necessary. This may be disabled by commenting out or deleting the `PAIRING_SUPPORT` macro definition in `user_config.h`.

7.4. Debug Output

Debug output over UART may be enabled by un-commenting the `DEBUG_OUTPUT_ENABLED` macro definition in `user_config.h`.

7.5. Filtering

The GATT Client can be configured to filter out advertisements from GATT Servers. This allows the GATT Client to ignore GATT Servers that don't meet specific conditions.

7.5.1. Filtering by Advertised Services

The GATT Client may be configured to ignore GATT Servers that do not advertise any services that are supported by the Client. Supported services are recognised by their UUID.

Whether to apply filtering based on service UUID is set at runtime by calling `GattStartScan()` in `gatt_access.c` with the appropriate arguments. In the example application, this filtering may be disabled at compile time by commenting out or deleting the `FILTER_DEVICE_BY_SERVICE` macro in `user_config.h`.

7.5.2. Application Specific Filtering

After filtering by service UUID, the function `appGattCheckFilter()` from `gatt_access.c` is called. This function returns `TRUE` to accept an advertisement and connect to the GATT Server, and `FALSE` to ignore it.

In the example application this function does nothing, but it could be modified to reject advertisements based on device name or Bluetooth Address, or any other condition.

7.6. Application Timers

The GATT Client makes use of several timers to control how long to wait for certain events to occur. They are all defined in `gatt_client.c`:

Timer Name	Timer Macro	Timer Description	Timer Value
Connection Request Timer	`CONNECTING_STATE_EXPIRY_TIMER`	How long to wait for a connection to be established with a GATT Server	15 s
Pairing Timer	`PAIRING_TIMER_VALUE`	How long to wait after a connection has been established before starting the Pairing Procedure	150 ms
Discovery Start Timer	`DISCOVERY_START_TIMER`	How long to wait after a connection has been established before starting the Discovery Procedure	300 ms

Table 7.3: Application Timers

7.7. Non-Volatile Memory

The GATT Client uses one of the following macros to store and retrieve persistent data in either the EEPROM or Flash-based memory.

- `NVM_TYPE_EEPROM` for I^{2}C EEPROM.
- `NVM_TYPE_FLASH` for SPI Flash.

Note:

The macros are enabled by selecting the NVM type using the Project Properties in xIDE. This macro is defined during compilation to let the application know which NVM type it is being built for. If EEPROM is selected `NVM_TYPE_EEPROM` will be defined and for SPI Flash the macro `NVM_TYPE_FLASH` will be defined.

7.8. Service Configuration

All the service configuration macros are defined in `user_config.h` and are listed in Table 7.4:

Macro Name	Description	Default Value
`MAX_SUPPORTED_SERVICES`	Maximum number of services supported by the Client. This may be different from the number of services provided by a Server, as the Server may provide a number of un-supported services, or only a subset of the supported services.	10
`MAX_SUPPORTED_SERV_PER_DEVICE`	Maximum number of supported services for each Server. Although the Client may support more services overall, it can only handle up to this number for each connected device. (There is no reason why this value should not be the same as `MAX_SUPPORTED_SERVICES`, provided there is sufficient memory).	5
`MAX_CONNECTED_DEVICES`	Maximum number of devices that may be connected at any given time. Maximum value is 1.	1
`MAX_BONDED_DEVICES`	Maximum number of devices that the Client may be paired with. Must not exceed 3.	1

Table 7.4: Service Configuration Macros

7.9. Accessing Additional Services

The example application may be extended to allow access to additional services. The following steps describe how to add support for an example service called New to the GATT Client example application.

7.9.1. Create Service Specific Source Files

Characteristic handling for each service accessible by the Client is implemented in its own set of source files: `<service>_service_data.c`, `<service>_service_data.h` and `<service>_uuids.h`.

For the New service, the existing files for the Battery Service are copied and renamed accordingly, to produce `new_service_data.c`, `new_service_data.h` and `new_uuids.h`. The files are then customised for the New service:

- For the `new_service_data.c` source file:
  - Include `new_service_data.h` instead of `battery_service_data.h`
  - Update the `MAXIMUM_NUMBER_OF_CHARACTERISTIC` macro to indicate how many characteristics the New service provides
  - The name of the service data type needs updating, from `BATTERY_SERVICE_DATA_T` to `NEW_SERVICE_DATA_T`
  - The service data variable is renamed appropriately, from `g_bs_serv_data` to `g_new_serv_data`.
- For the `new_service_data.h` header file:
  - Update the guard macro definition protecting the body of the file, so instead of `__BATTERY_SERVICE_DATA_H__` it is called `__NEW_SERVICE_DATA_H__`:

```
#ifndef __NEW_SERVICE_DATA_H__
#define __NEW_SERVICE_DATA_H__
...
#endif /* __NEW_SERVICE_DATA_H__
```

  - Include `new_uuids.h` instead of `battery_uuids.h`.

    - Enumerate the service characteristics that will be accessed. For example, the New service may support two characteristics called `new_level` and `new_threshold`, and the `battery_char_t` enumeration is renamed and modified accordingly:

      ```
      typedef enum {
          new_level     = 0,
          new_threshold = 1,
          new_invalid_type
      } new_char_t;
      ```

    - Export the callback function table by renaming `BatteryServiceFuncStore` to `NewServiceFuncStore`:

      ```
      extern SERVICE_FUNC_POINTERS_T NewServiceFuncStore;
      ```

    - Define function prototypes for all the supported callback functions. For the New service, replace `Battery` in the callback function names with `New`.
- For the `new_uuids.h` file:
    - Update the guard macro definition protecting the body of the file, by renaming `__BATTERY_UUIDS_H__` to `__NEW_UUIDS_H__`:

      ```
      #ifndef __NEW_UUIDS_H__
      #define __NEW_UUIDS_H__
      ...
      #endif /* __NEW_UUIDS_H__
      ```

    - Replace `UUID_BATTERY_SERVICE` with the UUID of the New service. This is how the service is identified in the Server GATT database. For a Bluetooth SIG standardised profile such as the Battery Service, a 16-bit UUID is defined:

      ```
      #define UUID_BATTERY_SERVICE  0x180f
      ```

      For a custom service, a full 128-bit UUID must be defined. 128-bit values are not supported in C and may not be used directly in application code. To work around this, 128-bit UUIDs may be split into 16 octets, and defined alongside the full 128-bit version. For example:

      ```
      #define UUID_NEW_SERVICE      0x00001c00d10211e19b2300025b00a5a5

      #define UUID_NEW_SERVICE_1    0x00
      #define UUID_NEW_SERVICE_2    0x00
      #define UUID_NEW_SERVICE_3    0x1c
      #define UUID_NEW_SERVICE_4    0x00
      #define UUID_NEW_SERVICE_5    0xd1
      #define UUID_NEW_SERVICE_6    0x02
      #define UUID_NEW_SERVICE_7    0x11
      #define UUID_NEW_SERVICE_8    0xe1
      #define UUID_NEW_SERVICE_9    0x9b
      #define UUID_NEW_SERVICE_10   0x23
      #define UUID_NEW_SERVICE_11   0x00
      #define UUID_NEW_SERVICE_12   0x02
      #define UUID_NEW_SERVICE_13   0x5b
      #define UUID_NEW_SERVICE_14   0x00
      #define UUID_NEW_SERVICE_15   0xa5
      #define UUID_NEW_SERVICE_16   0xa5
      ```

    - Delete the characteristics defined for the Battery Service, and replace with the UUIDs of each of the relevant characteristics the New service provides.

7.9.2. Implement Callback Functions

Each service accessed by the GATT Client requires that a variable of type `SERVICE_FUNC_POINTERS_T` be defined, that contains pointers to the functions called by the application during initialisation, service discovery, and normal running. This variable is defined in `<service>_service_data.c`, and the `SERVICE_FUNC_POINTERS_T` structure is defined in `gatt_access.h`.

To provide access to the New service, edit `new_service_data.c` and rename `BatteryServiceFuncStore` to `NewServiceFuncStore`, and rename the callback functions replacing `Battery` with `New`:

```
SERVICE_FUNC_POINTERS_T NewServiceFuncStore = {
    .serviceUuid            = &NewServiceUuid,
    .isMandatory            = &NewServiceIsMandatory,
    .serviceInit            = &NewServiceDataInit,
    .checkHandle            = &NewServiceCheckHandle,
    .getHandles             = &NewServiceGetHandles,
    .charDiscovered         = &NewServiceCharDiscovered,
    .descDiscovered         = &NewServiceCharDescDisc,
    .discoveryComplete      = &NewServiceDiscoveryComplete,
    .serviceIndNotifHandler = &NewServiceHandlerNotifInd,
    .configureServiceNotif  = NULL,
    .writeRequest           = &NewServiceWriteRequest,
    .writeConfirm           = &NewServiceWriteConfirm,
    .readRequest            = &NewServiceReadRequest,
    .readConfirm            = &NewServiceReadConfirm,
    .configureService       = &NewServiceConfigure,
    .isServiceFound         = &NewServiceFound,
    .resetServiceData       = &NewServiceResetData
};
```

Callbacks that are not required for the service may be replaced by NULL, for example:

```
    ...
    .configureServiceNotif  = NULL,
    ...
```

Each callback function needs to be implemented in `new_service_data.c` as described below. See Table 6.1 for an overview of the callback functions. Detailed information is provided in the comments for each function implemented for the existing Battery and Device Information services.

7.9.2.1. NewServiceUuid

This function is called from the main application to find the service's UUID and its type (16- or 128-bit). For 16-bit UUIDs the value may be returned directly:

```
void BatteryServiceUuid(GATT_UUID_T *type, uint16 *uuid)
{
    *type = GATT_UUID16;
    uuid[0] = UUID_BATTERY_SERVICE;
}
```

128-bit values may not be assigned directly in C, so for 128-bit UUIDs the value is broken down into 16 octets, with each octet then loaded into an array to re-form the 128-bit value with little-endian order (i.e. the least significant octet is loaded first):

```
void NewServiceUuid(GATT_UUID_T *type, uint16 *uuid)
{
    *type = GATT_UUID128;
    uuid[0] = UUID_NEW_SERVICE_16;
    uuid[1] = UUID_NEW_SERVICE_15;
    uuid[2] = UUID_NEW_SERVICE_14;
    uuid[3] = UUID_NEW_SERVICE_13;
    uuid[4] = UUID_NEW_SERVICE_12;
    uuid[5] = UUID_NEW_SERVICE_11;
    uuid[6] = UUID_NEW_SERVICE_10;
    uuid[7] = UUID_NEW_SERVICE_9;
    uuid[8] = UUID_NEW_SERVICE_8;
    uuid[9] = UUID_NEW_SERVICE_7;
    uuid[10] = UUID_NEW_SERVICE_6;
    uuid[11] = UUID_NEW_SERVICE_5;
    uuid[12] = UUID_NEW_SERVICE_4;
    uuid[13] = UUID_NEW_SERVICE_3;
    uuid[14] = UUID_NEW_SERVICE_2;
    uuid[15] = UUID_NEW_SERVICE_1;
}
```

7.9.2.2. NewServiceDataInit

This function is called to initialise service information during the Discovery Procedure when the New service has been discovered in the Server's GATT database. The whole of `g_new_serv_data` is reset to initial values before storing the discovered service's start and end handles.

7.9.2.3. NewServiceCheckHandle

This function returns `TRUE` if the supplied handle falls within the range of characteristic handles supported by the New service. It is called whenever the GATT Client application receives a `GATT_IND_CHAR_VAL_IND` or `GATT_NOT_CHAR_VAL_IND` event to check whether it needs to take any action - if the handle is recognised, then `NewServiceHandlerNotifInd` will be called to handle the event.

7.9.2.4. NewServiceGetHandles

This function is used during the Discovery Procedure.

For each service it is called with `type` set to `service_type`, to return the range of characteristic handles supported by the service.

It is then called for each characteristic with `type` set to `characteristic_type`, to return the range of characteristic descriptor handles supported by the characteristic. In this mode it returns `TRUE` if there are more descriptors to be discovered or `FALSE` if the range has been exhausted.

In both cases the range of available handles is returned.

7.9.2.5. NewServiceCharDiscovered

This function is called during the Discovery Procedure for each New service characteristic discovered in the Server's GATT database and handles the `GATT_CHAR_DECL_INFO_IND` event.

A check is made that the characteristic is of interest to the GATT Client (as the Client may not require access to all the available characteristics) and if so the characteristic value is stored along with the discovered properties.

7.9.2.6. NewServiceCharDescDisc

This function is called during the Discovery Procedure for each New service characteristic descriptor discovered in the Server's GATT database and handles the `GATT_CHAR_DESC_INFO_IND` event.

A check is made that the characteristic descriptor is of interest to the GATT Client and if so the descriptor's UUID and handle are stored.

The GATT Client example application only needs access to the client characteristic configuration descriptor but support may be extended to other descriptors if required.

7.9.2.7. NewServiceDiscoveryComplete

This function is called during the Discovery Procedure when there are no more characteristics or characteristic descriptors to be discovered in the Server's GATT database for the New service. This callback is provided to allow the GATT Client to perform any actions that can only be performed after the full service discovery is complete, but before the Discovery Procedure is complete.

The GATT Client currently uses this callback to reset some control variables and store the connection handle used to communicate with the Server for later use.

7.9.2.8. NewServiceHandlerNotifInd

This function is called to handle `GATT_IND_CHAR_VAL_IND` and `GATT_NOT_CHAR_VAL_IND` events for characteristics supported by the New service.

A check is made that the characteristic handle is of interest to the GATT Client before taking action dependent on the affected characteristic and the new value.

For example, in the case of the Battery Service, the GATT Client example application only requires access to the battery level characteristic, and this event causes the new battery level to be displayed on the UART.

7.9.2.9. NewServiceConfigNotif

This callback is no longer used and should be set to `NULL` in the function callback structure `NewServiceFuncStore`.

7.9.2.10. NewServiceWriteRequest

This function modifies the value of a characteristic in the Server's GATT database. The resulting `GATT_WRITE_CHAR_VAL_CFM` event will be handled by `NewServiceWriteConfirm`. If the result of the write request indicates that the Client has insufficient authentication/authorisation before the characteristic value can be modified then the application will start the Pairing Procedure, if `PAIRING_SUPPORT` is defined (see section 7.3), or disconnect from the peer device if not.

7.9.2.11. NewServiceWriteConfirm

This function is called to handle the `GATT_WRITE_CHAR_VAL_CFM` event after successfully writing a characteristic value into the Server's GATT database.

7.9.2.12. NewServiceReadRequest

This function reads the current value of a characteristic from the Server's GATT database. The resulting `GATT_READ_CHAR_VAL_CFM` event will be handled by `NewServiceReadConfirm`. If the result of the read request indicates that the Client has insufficient authentication/authorisation before the characteristic value can be read then the application will start the Pairing Procedure, if `PAIRING_SUPPORT` is defined (see section 7.3), or disconnect from the peer device if not.

7.9.2.13. NewServiceReadConfirm

This function is called to handle a `GATT_READ_CHAR_VAL_CFM` event after a characteristic value has been read successfully from the Server's GATT database.

7.9.2.14. NewServiceConfigure

This function configures the Server's GATT database by enabling notifications for selected characteristics. If successful, the Client will receive `GATT_NOT_CHAR_VAL_IND` events when the configured characteristics' values change, which are handled by `NewServiceHandlerNotifInd`.

7.9.2.15. NewServiceFound

This function is called from the main application to check whether the New service is present and supported on a particular Server. It returns `TRUE` if it is, and `FALSE` if it is not. It is usually sufficient to check whether the service handles have been modified since initialisation.

7.9.2.16. NewServiceResetData

Reset the service information stored for the specified device. This is called from the main application when a device disconnects, so that resources can be freed for a new connection. The whole of `g_new_serv_data` is reset to initial values.

7.9.3. Integrate Access to New Service

The GATT Client application needs to know which services will be accessed. By example, this may be accomplished for the New service as follows:

- Edit `gatt_client.c`
- Include the New service header file `new_service_data.h`.
- Add the address of the New service callback structure to the `g_supported_services` list:

```
static SERVICE_FUNC_POINTERS_T *g_supported_services[] = {
    &BatteryServiceFuncStore,     /* Battery Service */
    &DeviceInfoServiceFuncStore, /* Device Information Service */
    &NewServiceFuncStore          /* New Service */
};
```

8. Current Consumption

The current consumed by the application can be measured by removing the Current Measuring Jumper (see Figure 2.1) and installing an ammeter in its place. The ammeter should be set to DC, measuring current from µA to mA.

The setup used while measuring current consumption is described in section 2.1.

Table 8.1 shows the typical current consumption values measured during testing under noisy RF conditions with typical connection parameter values using the CSR1011 development board. See the Release Notes for the actual current consumption values measured for the application.

Test Scenario	Description	Average Current Consumption
Active Scanning	▪ Interval: 400 ms ▪ Window: 400 ms	19 mA
Active Scanning	▪ Interval: 400 ms ▪ Window: 200 ms	10 mA
Connected	▪ Connection interval: 20 ms ▪ Slave latency: 60 intervals	374 µA
Notes: ▪ Average current consumption is measured at 3.0 V ▪ Ammeter used: Agilent 34411A		

Table 8.1: Current Consumption Values

Current consumption may be reduced by reducing the scan window value to between 40ms and 400ms, while keeping the scan interval fixed at 400ms. The lower the ratio of the scan window value to the scan interval value the lower the current consumption, and the greater the risk of missing advertisement reports. The scan interval and scan window values should be optimised to achieve the required level of connectivity while minimising current consumption. This optimal value will vary from application to application.

Document References

Document	Reference
Bluetooth Core Specification version 4.1	http://www.bluetooth.org/
GAP Specification	Volume 3, Part C of the *Bluetooth Core Specification version 4.1*
ATT Specification	Volume 3, Part F of the *Bluetooth Core Specification version 4.1*
GATT Specification	Volume 3, Part G of the *Bluetooth Core Specification version 4.1*
Battery Service Specification version 1.0	http://www.bluetooth.org/
Device Information Service Specification version 1.1	http://www.bluetooth.org/
SDK Reference Documentation	Supplied with the CSR µEnergy SDK as the Firmware Library Documentation
CSR µEnergy xIDE User Guide	CS-212742-UG
GATT Server Application Note	CS-305206-AN

Terms and Definitions

AFH	Adaptive Frequency Hopping
API	Application Program Interface
ATT	Attribute Protocol: a Bluetooth protocol for discovering, reading and writing attributes on a peer device
BLE	Bluetooth Low Energy: a Bluetooth technology designed for ultra-low power consumption
Bluetooth®	Set of technologies providing audio and data transfer over short-range radio connections
Bluetooth Smart	A term used to indicate devices supporting BLE
COM	Communication, normally used to refer to the RS-232 serial ports on a PC.
CSR	Cambridge Silicon Radio
CSR µEnergy®	Group term for CSR's range of Bluetooth Smart wireless technology chips
EEPROM	Electrically Erasable Programmable Read Only Memory
e.g.	*exempli gratia* (for example)
i.e.	*id est* (that is)
GAP	Generic Access Profile: a Bluetooth profile that defines generic procedures related to the discovery of Bluetooth devices and link management. It also defines procedures related to the use of different security levels, and common format requirements for parameters accessible on the user interface level
GATT	Generic Attribute Profile: a Bluetooth profile that describes a service framework using the Attribute Protocol to discover services, and for reading and writing characteristic values on a peer device
I^2C	Inter-Integrated Circuit
L2CAP	Logical Link Control and Adaptation Layer: A Bluetooth protocol that provides connection-oriented and connectionless data services to upper layer protocols
LM	Link Manager
LS	Link Supervisor
NVM	Non-Volatile Memory
PC	Personal Computer
PIN	Personal Identification Number
PS	Persistent Store
RS-232	Serial communications standard
SDK	Software Development Kit
SIG	Special Interest Group
SM	Security Manager
SPI	Serial Peripheral Interface

UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
UUID	Universally Unique Identifier