Push every boundary.™

CSR µEnergy®

GATT Server

Application Note

Issue 2

© Cambridge Silicon Radio Limited 2014

CSR

Document History

Revision	Date	History
1	05 FEB 14	Original publication of this document
2	06 FEB 14	Update current consumption values

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.

CSR

Contents

Tables, Figures and Equations

© Cambridge Silicon Radio Limited 2014

© Cambridge Silicon Radio Limited 2014

1. Introduction

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

GATT Server applications provide GATT Clients with access to data and resources 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 that only authorised GATT Clients may access selected values. The GATT Server may request an encrypted link for some or all data access to prevent unauthorised third parties from intercepting the data.

As a minimum, the GATT Server application must support the Generic Access Profile (GAP) to allow its services to be discovered by potential GATT Clients, and to provide access to security levels. The GATT Server 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, to generate indications or to respond to GATT Client requests, for example.

This document describes the mandatory components that a GATT Server must include, and then demonstrates how a typical GATT Server application may be implemented using the example application as a template.

1.1. Application Overview

1.1.1. Profiles Supported

This example application supports GATT to provide access to service characteristics to a GATT Client. It supports GAP to allow services to be discovered and establish connections with GATT Clients.

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

1.1.1.1. Attribute Protocol Overview

This section provides a quick overview of the Attribute Protocol (ATT), see the *ATT Specification* for detailed information.

All of the current Bluetooth Low Energy (BLE) profiles use the Attribute Protocol for transport. The Attribute Protocol defines two roles: a Client and a Server. A Server exposes a set of attributes which are made accessible to a Client via the Attribute Protocol.

Client-Server Architecture

Client and Server roles do not depend on the link establishment roles. The connection initiator can be an ATT Server or an ATT Client or both at the same time.

An attribute is a discrete value consisting of a handle, an attribute type (UUID), permissions and a value, see Figure 1.1:

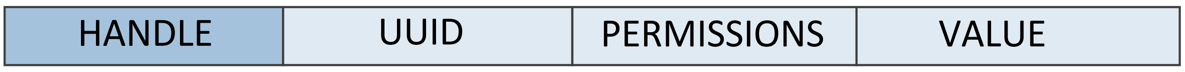
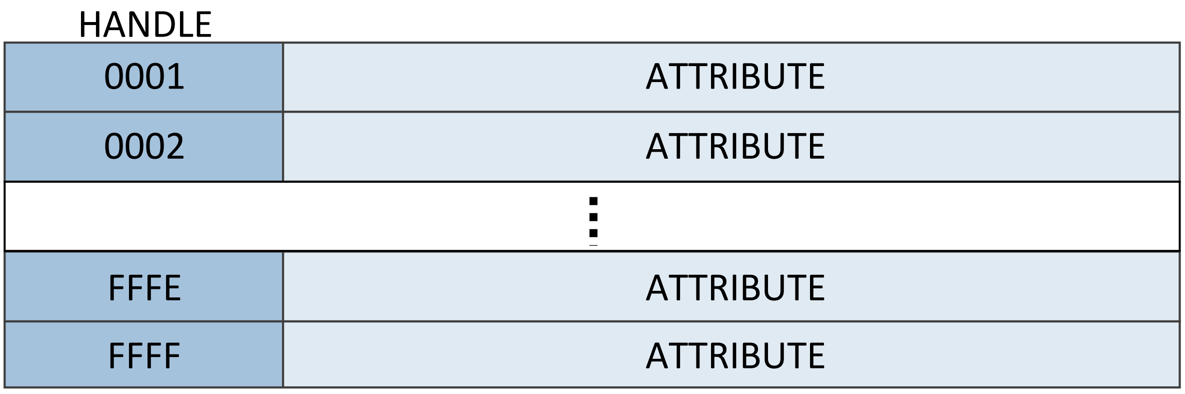


Figure 1.1: Contents of an Attribute as Defined by the Attribute Protocol

Data is arranged on the Server in a table of attributes, with each row indexed by an attribute handle, see Figure 1.2:



HANDLE	
0001	ATTRIBUTE
0002	ATTRIBUTE
⋮	
FFFE	ATTRIBUTE
FFFF	ATTRIBUTE

Figure 1.2: ATT Server Attributes Table

The Attribute Protocol allows a Client to:

- Find information about the attributes
- Read attributes
- Write attributes
- Queue attribute writes

See Figure 1.3 for an example showing the transfer of attribute values between a Client and a Server:

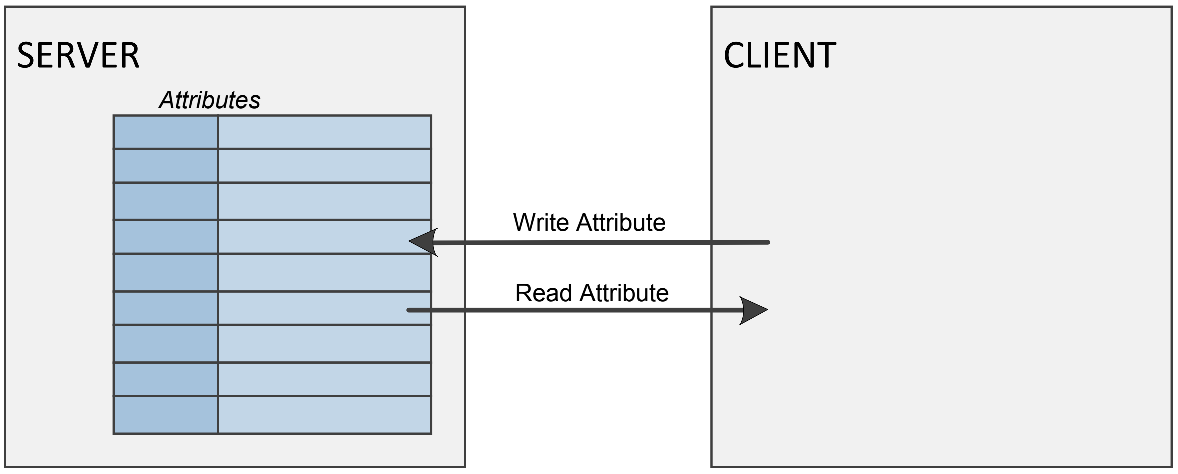


Figure 1.3: Client Access to a Table of Attributes, Initiated by the Client

The Attribute Protocol allows a Server to:

- Send notifications of changes in the attributes to a Client
- Send indications of changes in the attributes to a Client, which are in turn acknowledged by that Client.

See Figure 1.4 for an example showing the transfer of attributes values to a Client triggered by a Server:

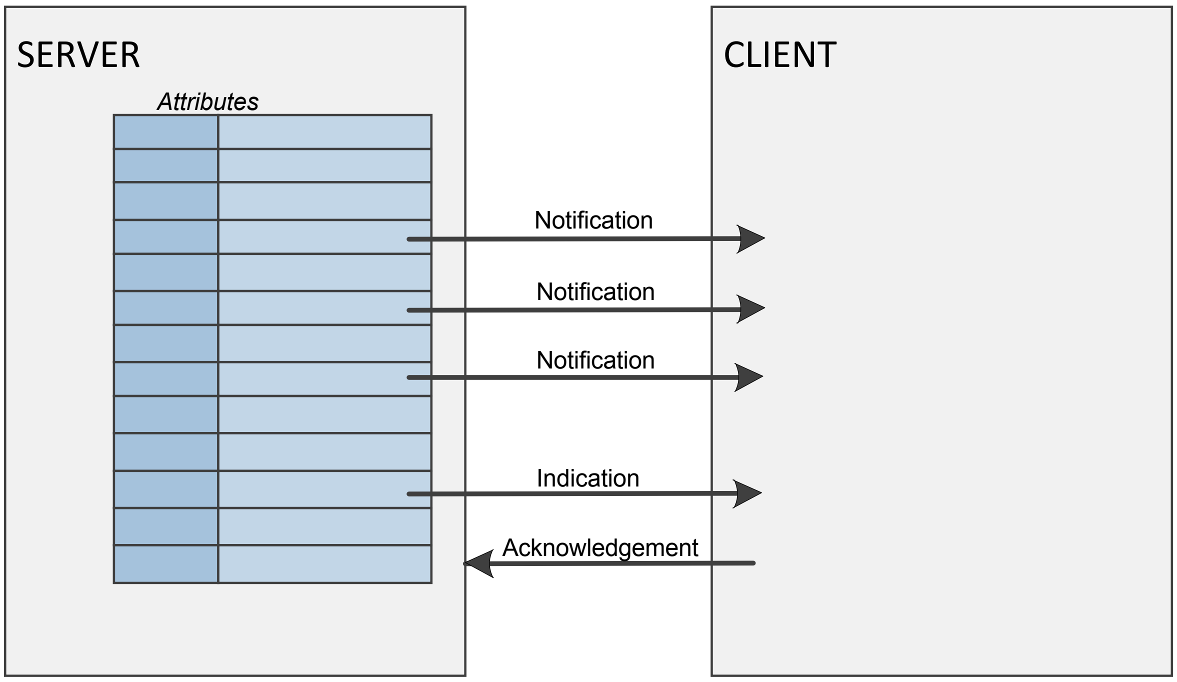


Figure 1.4: Client Receives Updates to a Table of Attributes, Initiated by the Server

1.1.1.2. Generic Attribute Profile Overview

This section provides a quick overview of the Generic Attribute Profile (GATT). See the *GATT Specification* for detailed information.

The GATT Profile is a service framework that defines procedures and formats of services and their characteristics. It is based on the Attribute Protocol. Attribute Protocol data structures and the client-server architecture are used by GATT. As with the Attribute Protocol, the GATT Server and Client roles do not depend on the link establishment role.

The GATT Server data is arranged in services with characteristics. The GATT protocol defines various procedures for discovering, reading and writing, notifying and indicating characteristics. A GATT Service runs on the Server to implement the GATT protocol. All BLE profiles are based on GATT.

GATT Database

The GATT Server data is conceptually arranged into services, e.g. the Battery Service. The actual data, such as the battery level or the battery state, is represented by characteristics of that service. The characteristics may be further described using characteristic descriptors, e.g. characteristic extended properties, client characteristic configuration etc., see Figure 1.5.

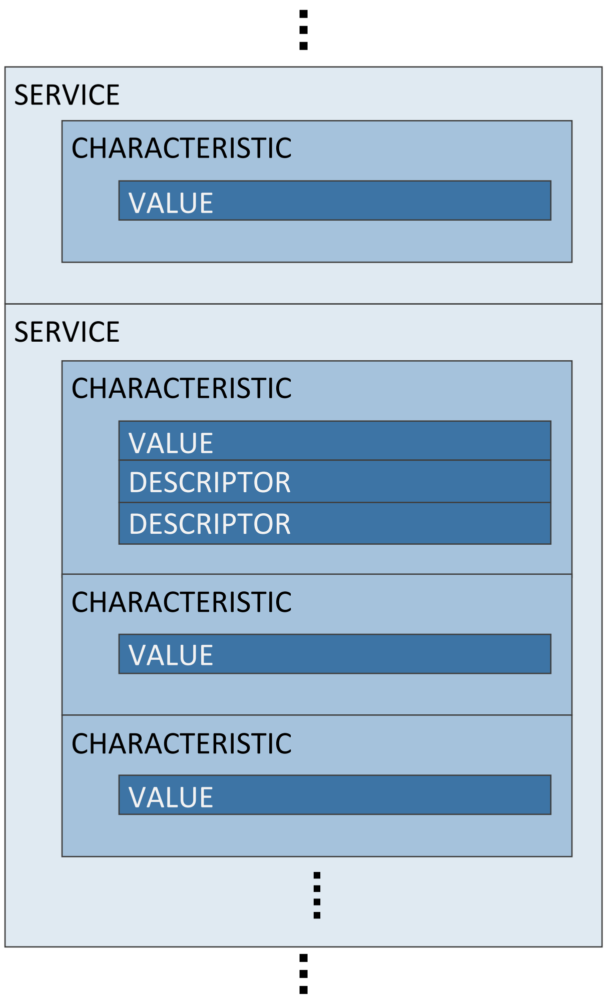
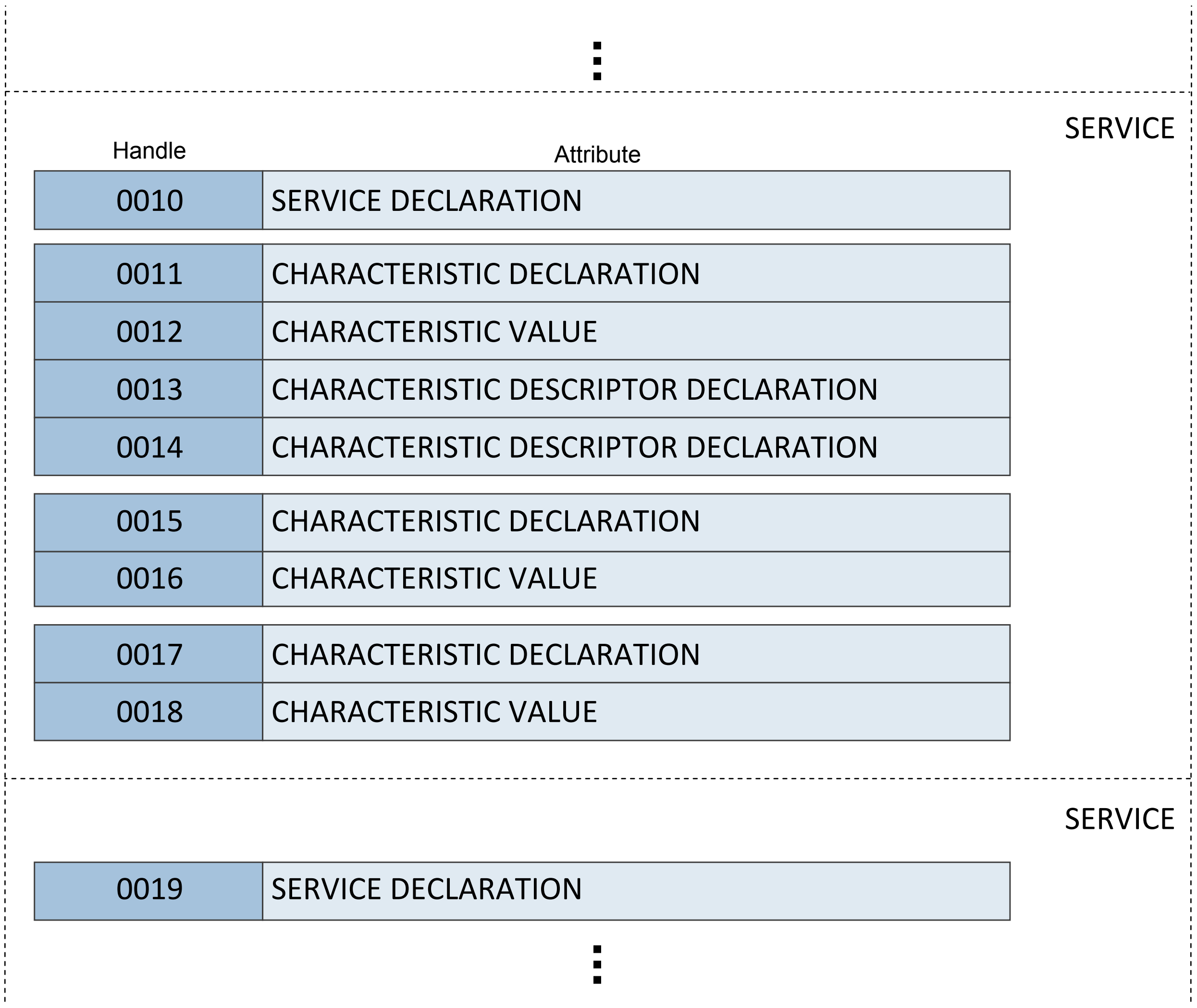


Figure 1.5: GATT Server Data Conceptual Organisation

The services, characteristics and descriptors are mapped on top of the Attribute Protocol table structure. Individual services and characteristics are identified by UUIDs.

© Cambridge Silicon Radio Limited 2014



Handle	Attribute
0010	SERVICE DECLARATION
0011	CHARACTERISTIC DECLARATION
0012	CHARACTERISTIC VALUE
0013	CHARACTERISTIC DESCRIPTOR DECLARATION
0014	CHARACTERISTIC DESCRIPTOR DECLARATION
0015	CHARACTERISTIC DECLARATION
0016	CHARACTERISTIC VALUE
0017	CHARACTERISTIC DECLARATION
0018	CHARACTERISTIC VALUE
0019	SERVICE DECLARATION

Figure 1.6: GATT Server Data Mapped to the Table of Attributes

The GATT Server data mapped to the table of attributes shown in Figure 1.6 is referred to as the GATT Database.

GATT defines procedures for a Client to:

- Discover services, characteristics and descriptors
- Read values of characteristics and descriptors
- Write values of characteristics and descriptors

See Figure 1.7 for an example of a GATT Client writing a value to a characteristic and reading a value from a characteristic in a GATT Server database:

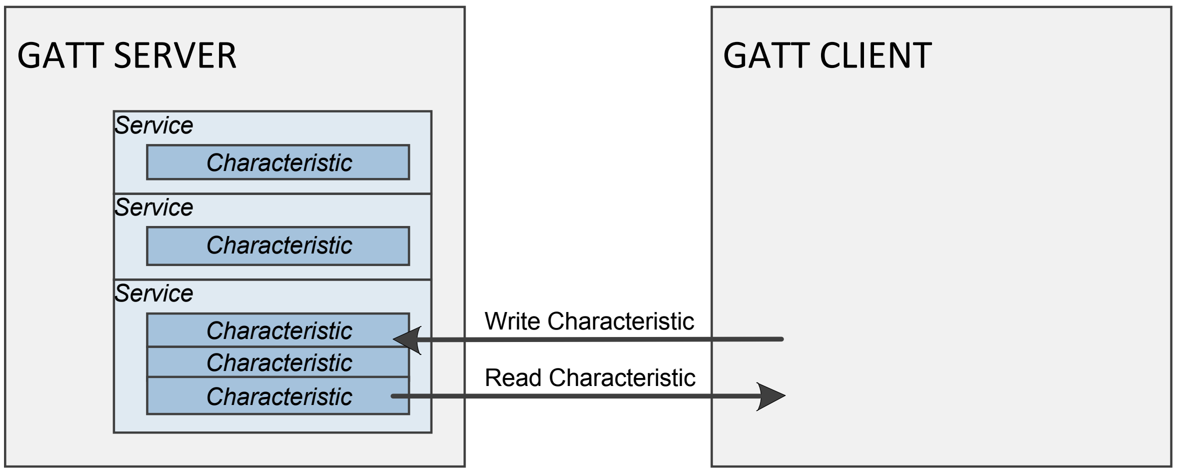


Figure 1.7: GATT Client Access to a GATT Server Database, Initiated by the Client

GATT defines procedures for a Server to:

- Notify a Client regarding changes to a characteristic value
- Send indications to a Client regarding changes to a characteristic value, which are acknowledged by the Client

See Figure 1.8 for an example showing the transfer of characteristic values to a Client triggered by a Server:

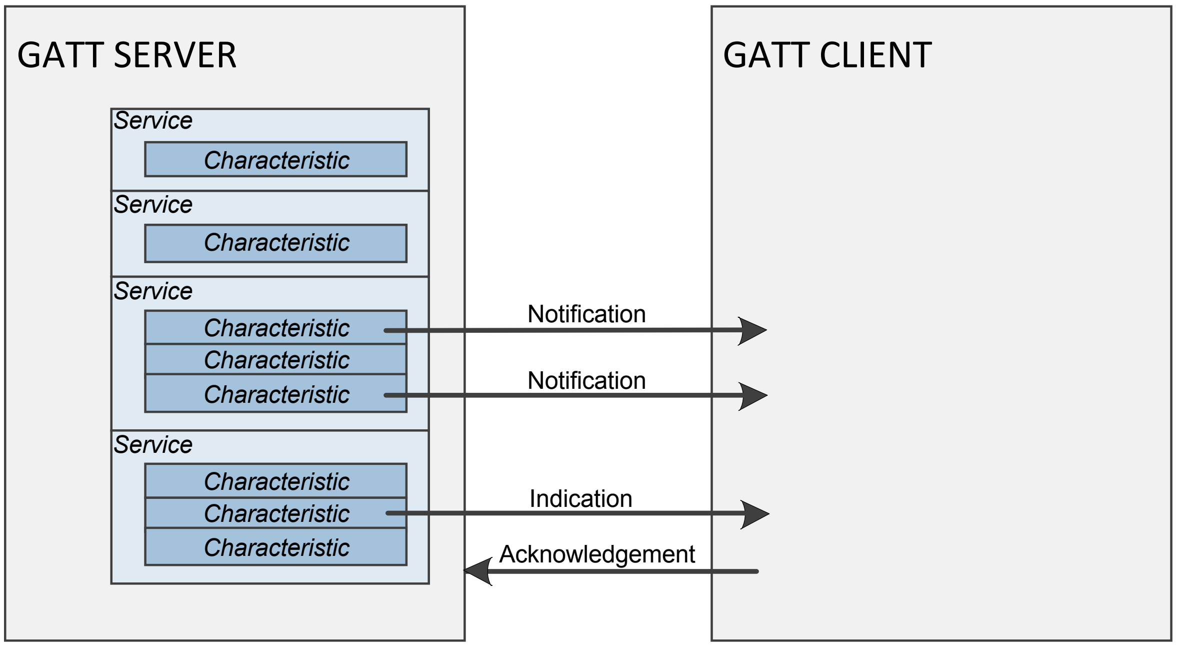


Figure 1.8: GATT Client Receives Updates to a GATT Server Database, Initiated by the Server

1.1.1.3. Generic Access Profile Overview

This section provides a quick overview of the Generic Access Profile (GAP). See the *GAP Specification* for detailed information.

The GAP defines procedures for:

- Discovering identities, names and basic capabilities
- Exchanging security information
- Establishing connections between devices
- Creating bonds between devices
- Resolving private addresses

GAP also defines advertising and scan response data formats.

There are four GAP roles, see Table 1.1:

Role	Description
Broadcaster	Broadcasts advertising events only.
Observer	Scans for advertising events only.
Peripheral	Advertises the services it provides and is connected to one device at a time. It takes on a slave role in link establishment
Central	Scans for advertisements and initiates multiple connections to other devices. It takes the master role in link establishment

Table 1.1: GAP Roles

1.1.2. Application Topology

The example application implements GATT in the Server role, and GAP in the Peripheral role, see Table 1.2:

Role	GAP Service	GATT Service	Device Information Service	Battery Service
GATT Role	Server	Server	Server	Server
GAP Role	Peripheral	Peripheral	Peripheral	Peripheral

Table 1.2: Application Topology

Role	Responsibility
GATT Server	It accepts incoming commands and requests from the client and sends responses, indications and notifications to the client.
GAP Peripheral	It accepts connection requests from the remote device and acts as a slave in the connection.

Table 1.3: Responsibilities

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

© Cambridge Silicon Radio Limited 2014

1.1.3. Services

This application acts as a GATT Server 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 1.9.

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*.

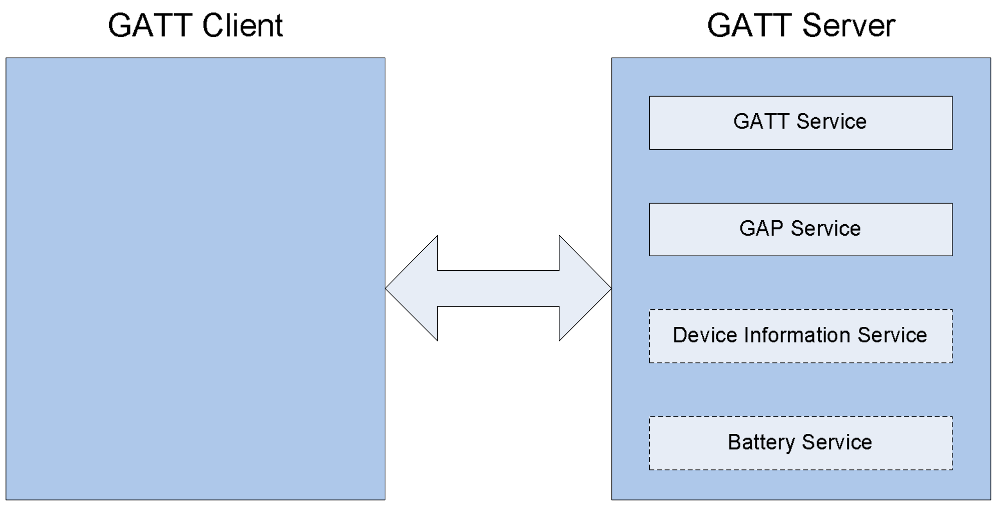


Figure 1.9: Primary Services

© Cambridge Silicon Radio Limited 2014

2. Using the Application

This section describes how the GATT Server application can be used with GATT Clients.

2.1. Demonstration Kit

The application can be demonstrated using the following components:

Component	Hardware	Application
GATT Server	▪ CSR10x0 Development Board, e.g. CNS10004, or ▪ CSR10x1 Development Board, e.g. CNS10017 or CNS12016	GATT Server Application
GATT Client	▪ CSR1011 Development Board, e.g. CNS12016	GATT Client Application
Host	▪ PC ▪ 2 x CSR µEnergy USB to SPI Adapters, CNS10020	Terminal emulator, e.g. HyperTerminal or PuTTY

Table 2.1: Demonstration Components

2.1.1. GATT Server

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

Figure 2.1 shows a CSR1001 development board with the switch set 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. Figure 2.2 shows a CSR1011 board with the switch set to receive power from the battery. Figure 2.3 shows a CSR10x0 development board, with the power slider switch in the **Off** position.

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.
- On CSR1001 and CSR1011 development boards ensure that the corresponding power source is selected using the power slider switch.
- On CSR10x0 development boards ensure that the power slider switch is in the **On** position.

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
- On CSR1001 and CSR1011 development boards ensure the power slider switch is set to a power source that is not provided. Figure 2.1 shows the switch in the **USB** position.
- On CSR10x0 development boards ensure that the power slider switch is in the **Off** position.

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.

© Cambridge Silicon Radio Limited 2014

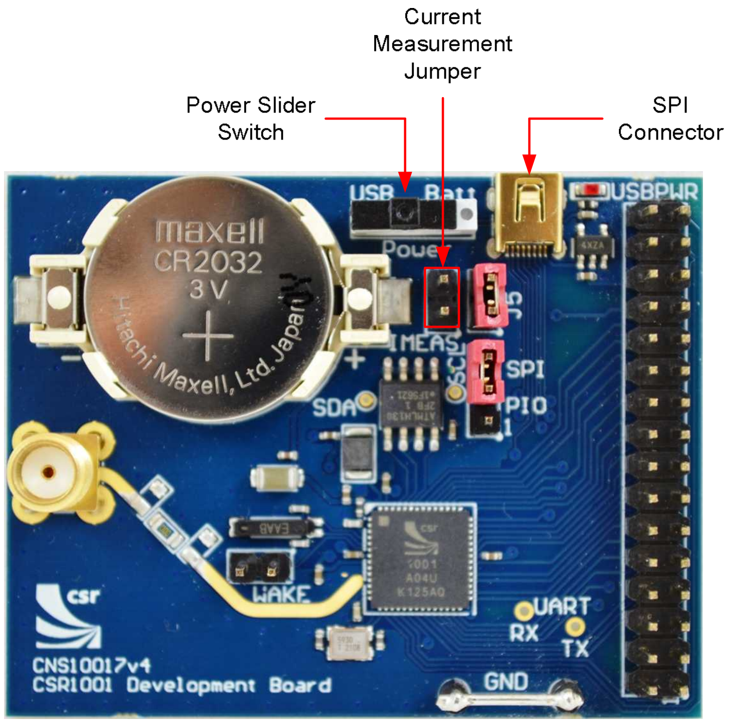


Figure 2.1: CSR1001 Development Board

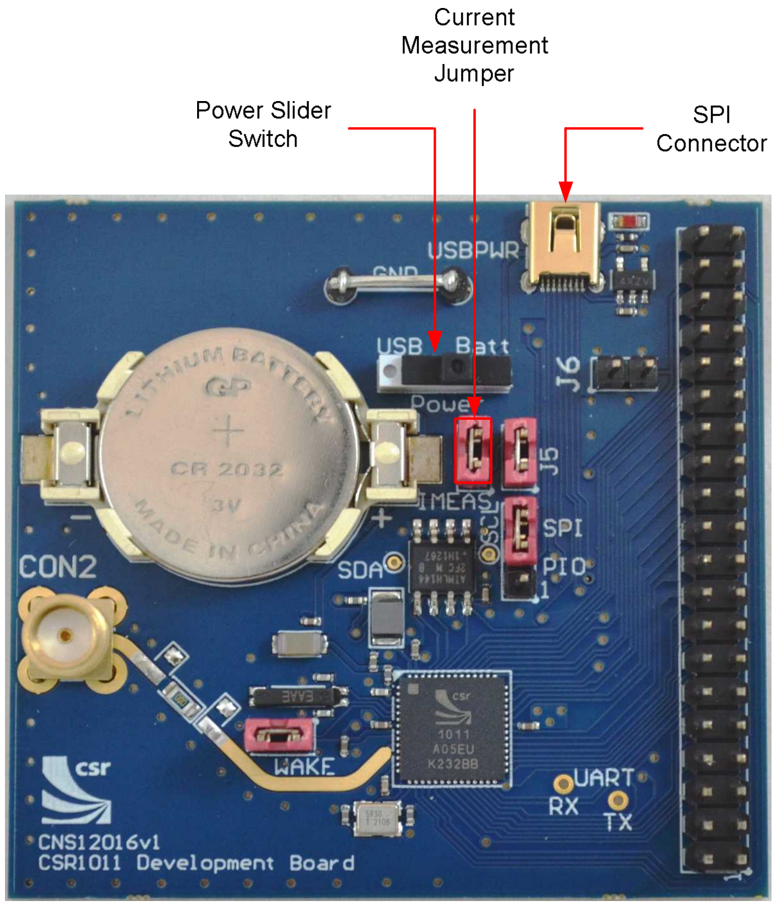


Figure 2.2: CSR1011 Development Board

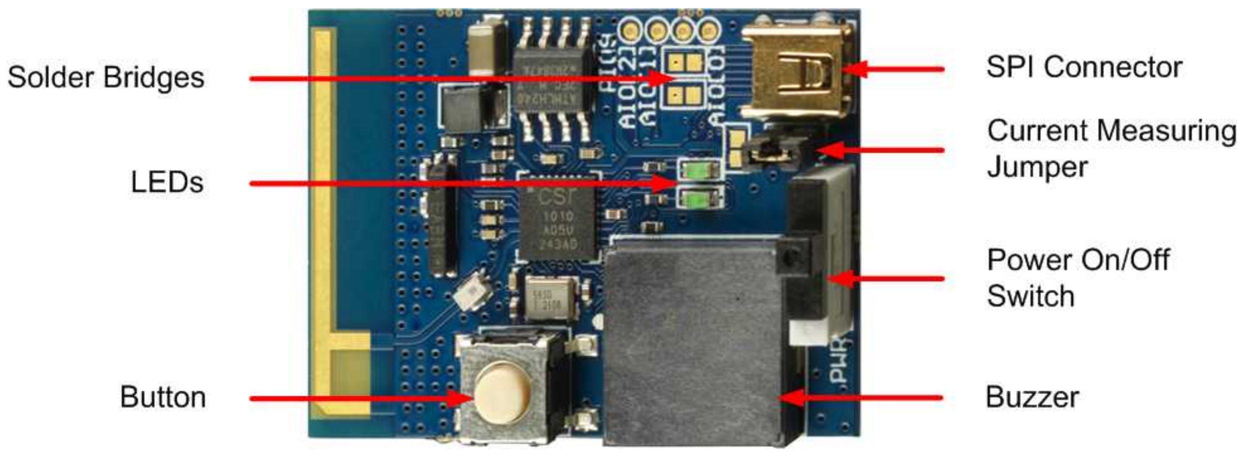


Figure 2.3: CSR10x0 Development Board

2.1.2. GATT Client

The GATT Server will connect to any GATT Client that issues a connection request, however for demonstration purposes the GATT Client example application provided in the CSR µEnergy SDK shall be used.

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

2.1.3. Host

The GATT Server optionally provides debug output over a UART interface, typically connected to a PC through a CSR µEnergy USB to SPI adapter, see Figure 2.4. 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.

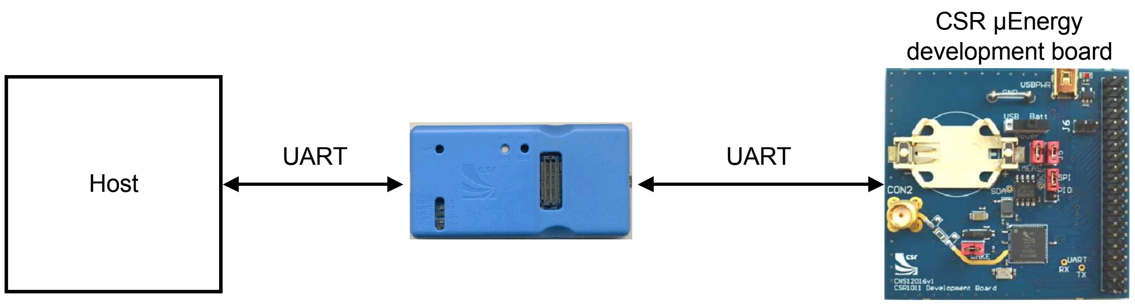


Figure 2.4: Connecting GATT Server Device to a Host

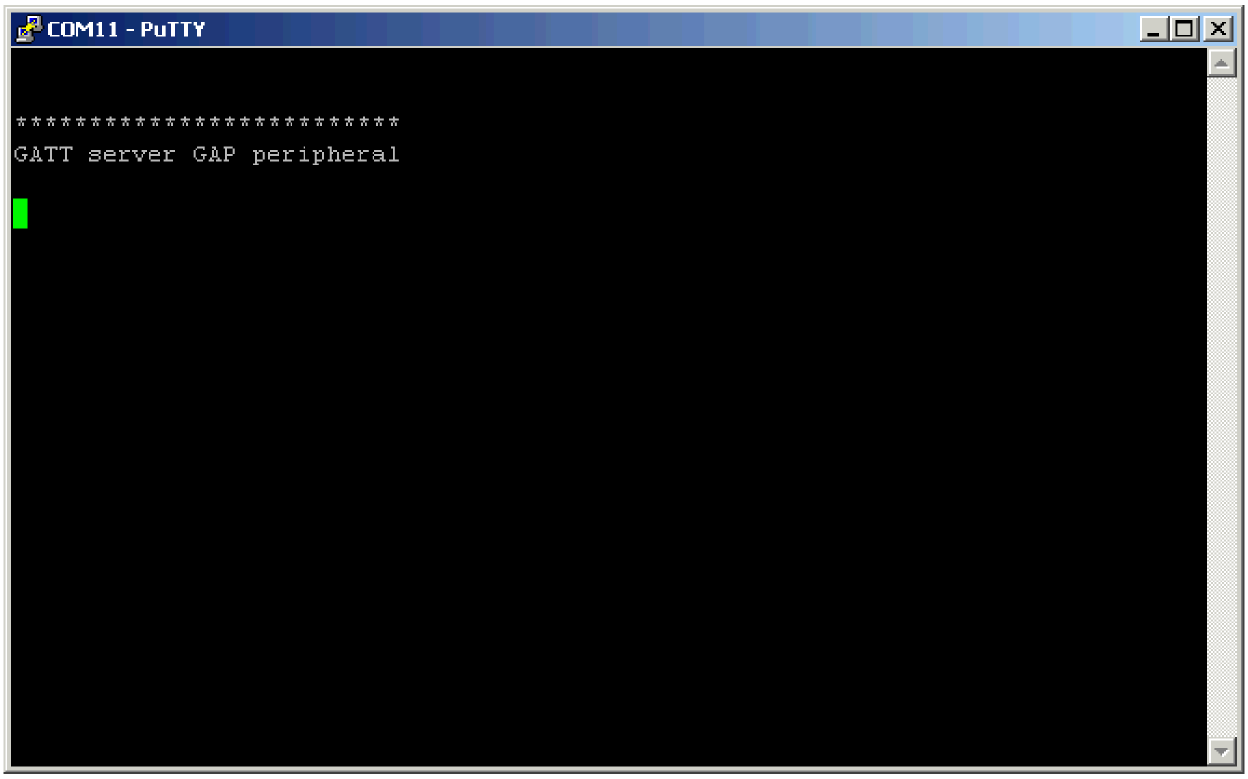
For this demonstration the GATT Client output is used to demonstrate the GATT Server functionality.

Note:

On CSR10x0 development boards, access to the UART is enabled by soldering the bridges indicated in Figure 2.3. See the *CSR µEnergy CSR1000 Development Kit Quick Start Guide* or the *CSR µEnergy CSR1010 Development Kit Quick Start Guide* for more information.

2.2. Demonstration Procedure

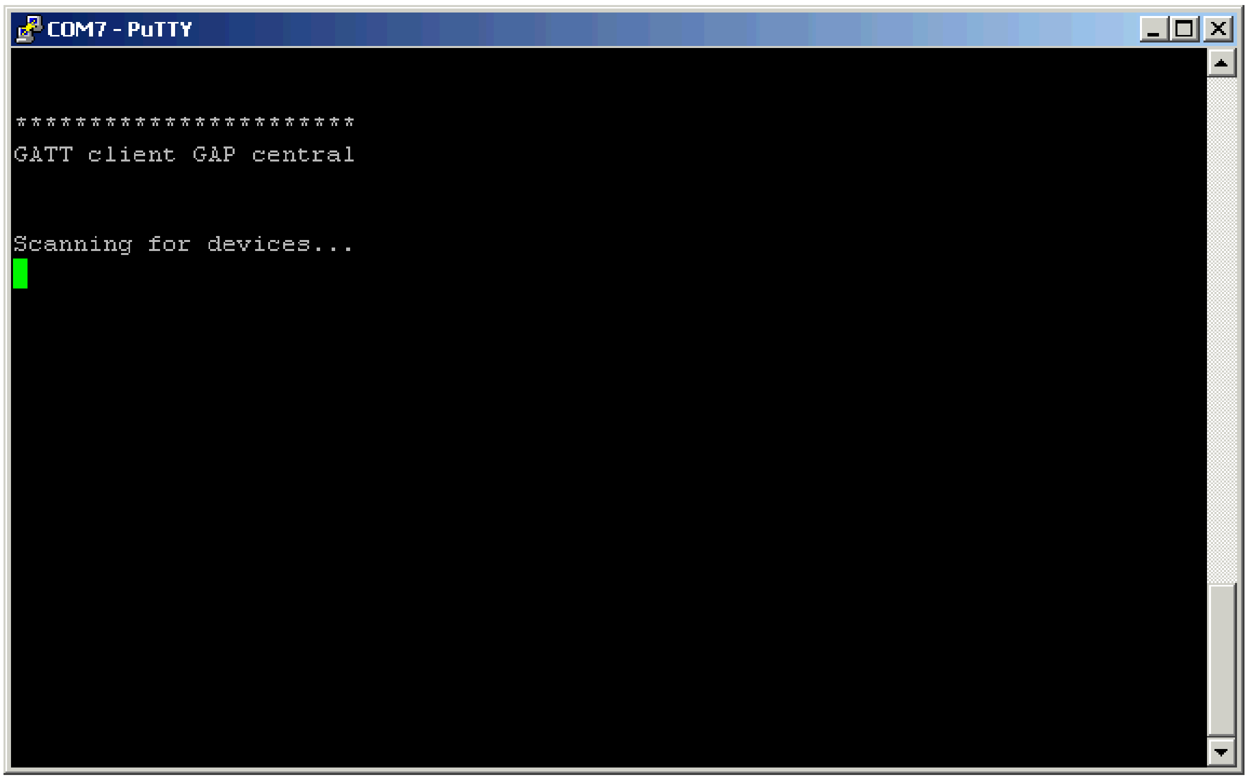
1. Connect two development boards to a PC using two CSR µEnergy USB to SPI adapters.
2. Run two terminal emulators and select the appropriate COM ports and communications configuration (see section 2.1.3).
3. Build and download the GATT Server application to one development board. See the *CSR µEnergy xIDE User Guide* for further information. When the application is running a message is displayed over UART, see Figure 2.5:



```
**************************
GATT server GAP peripheral
```

Figure 2.5: GATT Server Terminal Output Following Initialisation

4. Build and download the GATT Client application to the other development board. When the application is running a message is displayed over UART, see Figure 2.6:

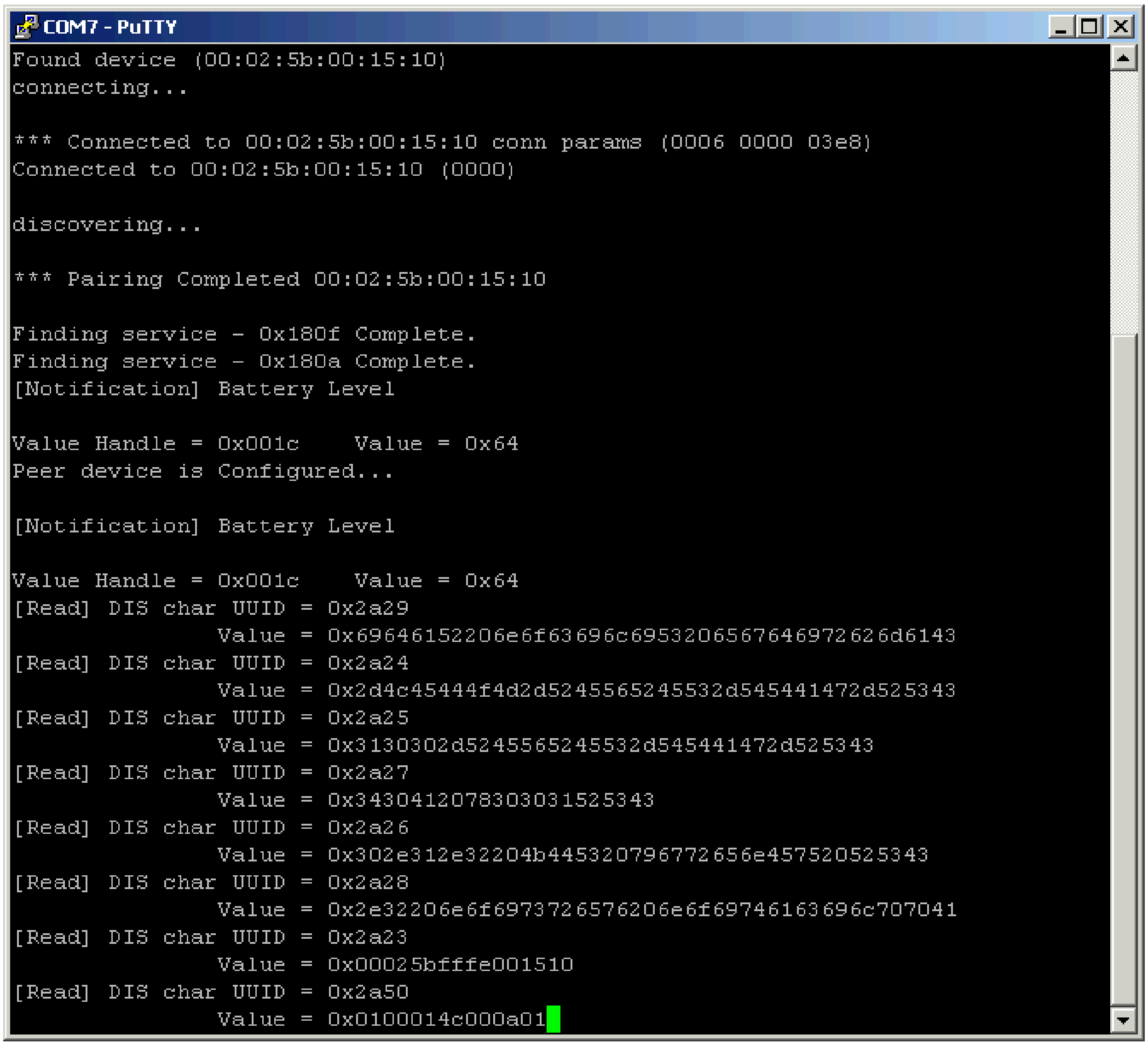


```
***********************
GATT client GAP central


Scanning for devices...
```

Figure 2.6: GATT Client Terminal Output Following Initialisation

5. After the example GATT Client has initialised it will scan for advertisements broadcast by the GATT Server. It will then connect to the GATT Server and initiate a Discovery Procedure to find which services are available. The GATT Server's Device Information Service (DIS) attribute values are displayed over the UART, and the GATT Server device's battery level is displayed and a request for notifications made, see Figure 2.7:



```
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 = 0x69646152206e6f63696c69532065676469726d6143
[Read] DIS char UUID = 0x2a24
               Value = 0x2d4c45444f4d2d52455652455532d545441472d525343
[Read] DIS char UUID = 0x2a25
               Value = 0x3130302d52455652455532d545441472d525343
[Read] DIS char UUID = 0x2a27
               Value = 0x34304120783030315 25343
[Read] DIS char UUID = 0x2a26
               Value = 0x302e312e32204b445320796772656e457520525343
[Read] DIS char UUID = 0x2a28
               Value = 0x2e32206e6f697372657620 6e6f69746163696c707041
[Read] DIS char UUID = 0x2a23
               Value = 0x00025bfffe001510
[Read] DIS char UUID = 0x2a50
               Value = 0x0100014c000a01
```

Figure 2.7: GATT Client Terminal Output Following GATT Server Discovery

6. The GATT Server will not accept connections from any other GATT Client until the link is disconnected.

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

3. Application Structure

A typical GATT Server application project contains database files, profile service files, and application files:

- Database files have file extension `.db`. Typically there is one `.db` file for each service, and a master `.db` file that is empty apart from `#include` directives to bring all the service `.db` files together. The master `.db` file is processed by the GATT Database Generator tool to produce a header file and source file (see Section 7.10). In the example applications provided with the CSR µEnergy SDK the master database file is always called `app_gatt_db.db` but that filename is not mandatory.
- Profile service files are used to implement each of the profile services. Typically each profile will have its own file implementing all the services it provides, together with a corresponding header file. The profile service files will use the header files generated by the GATT Database Generator to refer to the attributes provided by each service. This improves readability, portability and maintenance.
- 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 Server application and their purpose:

File Name	Purpose
`battery_service.c`	Implements routines required for the Battery Service e.g. reading battery level, notifying it to the remote device and handling access indications on the Battery Service specific ATT attributes.
`buzzer.c`	Provides access to the buzzer hardware.
`debug_interface.c`	Implements routines for sending debug messages to the UART, if enabled.
`dev_info_service.c`	Implements routines required for the Device Information Service e.g. handling read/write access indications on the Device Information Service specific ATT attributes.
`gap_service.c`	Implements routines for the GAP Service e.g. handling read/write access indication on the GAP Service characteristics, reading/writing device name on NVM etc.
`gatt_access.c`	Implements routines for triggering advertising and handling access to characteristic values implemented by the application.
`gatt_server.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.
`hw_access.c`	Provides routines to access the hardware (mainly used in button press detection).
`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 Server application and their purpose:

File Name	Purpose
`appearance.h`	Contains the appearance value macro of the GATT Server application.
`battery_service.h`	Contains prototypes of externally referenced functions defined in `battery_service.c`
`battery_uuids.h`	Contains macro definitions for UUIDs of the Battery Service and related characteristics
`buzzer.h`	Provides prototypes and macro definitions for accessing functions defined in `buzzer.c`
`debug_interface.h`	Contains macro definitions and function prototypes for optionally sending debug messages to the UART
`dev_info_service.h`	Contains prototypes of externally referenced functions defined in `dev_info_service.c`
`dev_info_uuids.h`	Contains macro definitions for UUIDs of the Device Information Service and related characteristics
`gap_conn_params.h`	Contains macro definitions for fast/slow advertising, preferred connection parameters, maximum number of connection parameter update requests etc.
`gap_service.h`	Contains prototypes of the externally referred functions defined in the `gap_service.c` file.
`gap_uuids.h`	Contains macro definitions for UUIDs of the GAP service and related characteristics.
`gatt_access.h`	Contains macro definitions, user defined data type definitions and function prototypes that are used across the application.
`gatt_server.h`	Contains user defined data type definitions and prototypes of externally referred functions defined in the file `gatt_server.c`.
`gatt_service_uuids.h`	Contains macros for UUID values of the GATT Service.
`hw_access.h`	Contains prototypes of externally referred hardware access functions defined in `hw_access.c`.
`nvm_access.h`	Contains prototypes of the externally referred functions defined in the `nvm_access.c`.
`user_config.h`	Contains macros for user configuration data.

Table 3.2: Header Files

© Cambridge Silicon Radio Limited 2014

3.3. Database Files

The SDK uses database files to generate the attribute database for the application. For more information on how to write database files, see the *GATT Database Generator User Guide*.

Table 3.3 lists the database files and their purpose.

File name	Purpose
`app_gatt_db.db`	Master database file which includes all service specific database files. This file is imported by the GATT Database Generator.
`battery_service_db.db`	Contains information related to Battery Service characteristics, their descriptors and values. See section B.1 for more information on Battery Service characteristics.
`dev_info_service_db.db`	Contains information related to Device Information Service characteristics, their descriptors and values. See section B.2 for Device Information Service characteristics.
`gap_service_db.db`	Contains information related to GAP Service characteristics, their descriptors and values. See section B.3 for GAP Service characteristics.
`gatt_service_db.db`	Contains information related to GATT Service characteristics, their descriptors and values.

Table 3.3: Database Files and Their Purpose

© Cambridge Silicon Radio Limited 2014

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 initialisation:

- Initialises the application timers, application data structures and hardware
- Configures GATT entity for server role
- Configures the NVM manager
- Initialises all the services
- Reads the persistent store
- Registers the GATT database with the firmware

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. It currently handles two system events:

- `sys_event_battery_low`: This event is received whenever the battery voltage crosses the low battery voltage threshold. If connected and notifications are configured, the GATT Server application notifies the battery level to the GATT Client.
- `sys_event_pio_changed`: This event indicates a change in PIO value. Whenever the user presses or releases the button, the corresponding PIO value changes. The application receives a PIO changed event and takes the appropriate action.

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. Database Access

- `GATT_ADD_DB_CFM`: This confirmation event marks the completion of database registration with the firmware. On receiving this event, the GATT Server application starts advertising.
- `GATT_ACCESS_IND`: This indication event is received when a GATT Client tries to access an ATT characteristic managed by the application.

4.1.4.2. LS Events

- `LS_CONNECTION_PARAM_UPDATE_CFM`: This confirmation event is received in response to the connection parameter update request by the slave. The connection parameter update request from the slave triggers the L2CAP connection parameter update signalling procedure. See Volume 3, Part A, Section 4.20 of *Bluetooth Core Specification version 4.1*.
- `LS_CONNECTION_PARAM_UPDATE_IND`: This indication event is received when the remote central device updates the connection parameters. On receiving this event, the application validates the new connection parameters against the preferred connection parameters and triggers a connection parameter update request if the new connection parameters do not comply with the preferred connection parameters.

4.1.4.3. SM Events

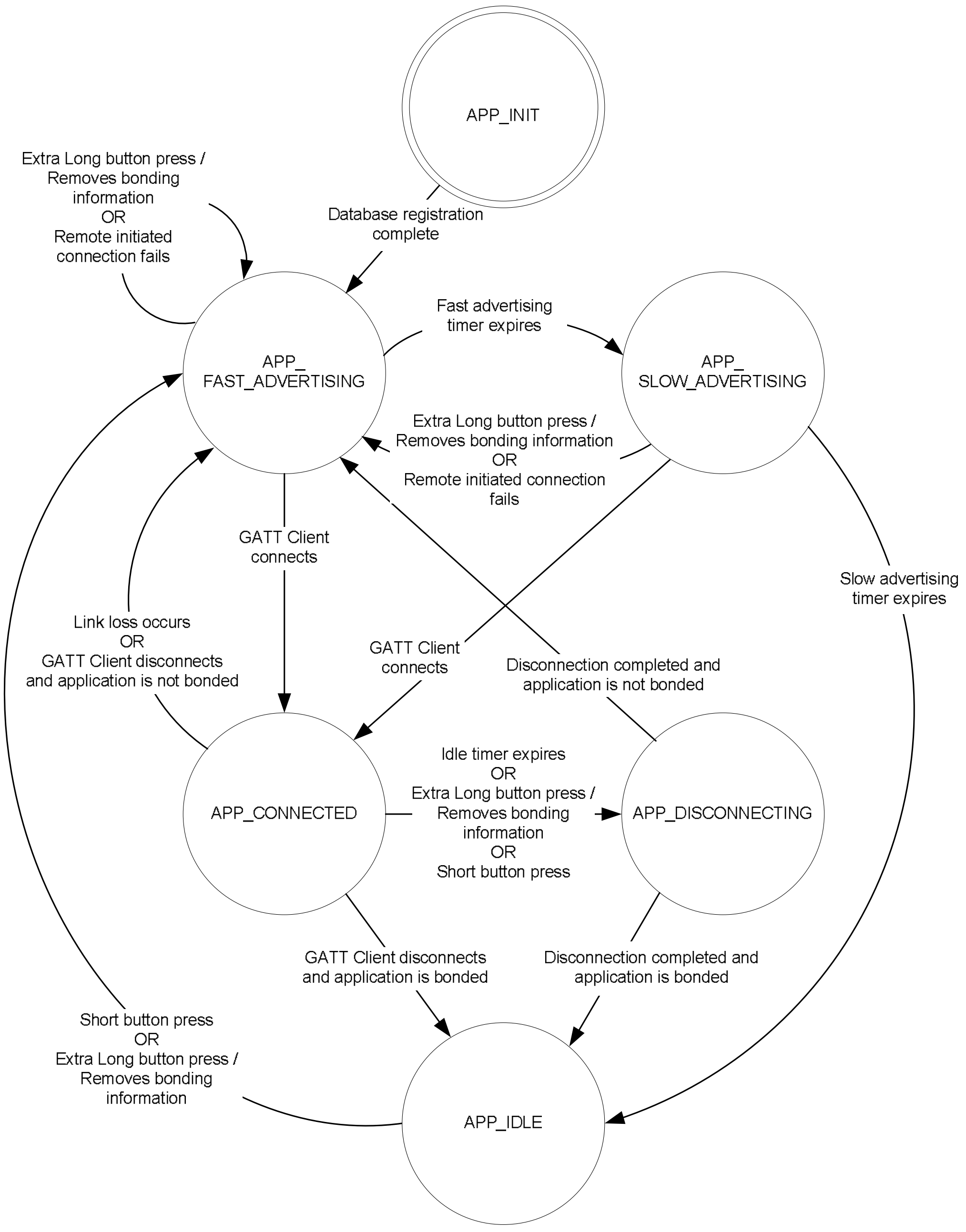
- `SM_KEYS_IND`: This indication event is received on completion of the bonding procedure. It contains keys and security information used on a connection that has completed short term key generation. The application stores the received Diversifier (DIV) and Identity Resolving Key (IRK) (if the GATT Client is using a resolvable random address) to the NVM. See Volume 3, Part H, section 2.1 of the *Bluetooth Core Specification version 4.1*.
- `SM_SIMPLE_PAIRING_COMPLETE_IND`: This indication event indicates that the pairing has completed. See Volume 3, Part H, Section 2.3 of the *Bluetooth Core Specification version 4.1*. In the case of a successful completion of the pairing procedure, the GATT Server is bonded with the GATT Client and bonding information is stored in the NVM. The bonded device address will be added to the white list if it is not a resolvable random address.
- `SM_DIV_APPROVE_IND`: This indication event is received when the remote connected device re-encrypts the link or triggers encryption at the time of reconnection. The firmware sends the diversifier in this event and waits for the application to approve or disapprove the encryption. The application approves the diversifier if the received diversifier is the same as the one stored by the application.
- `SM_PAIRING_AUTH_IND`: This indication is received when the remote connected device initiates pairing. The application can either accept or reject the pairing request from the peer device. The application shall reject the pairing request if it is already bonded to the GATT Client to prevent any new device disguising itself as one previously bonded to the GATT Server.

4.1.4.4. Connection Events

- `GATT_CONNECT_CFM`: This confirmation event indicates that the connection procedure has completed. If it has not successfully completed, the application restarts advertising. If the application is bonded to a device with a resolvable random address and the connection is established, the application tries to resolve the connected device address using the IRK stored in the non-volatile memory (NVM). If the application fails to resolve the address, it disconnects the link and restarts advertising.
- `GATT_CANCEL_CONNECT_CFM`: This confirmation event confirms the cancellation of the connection procedure. This signal confirms the successful stopping of advertisements by the GATT Server application when the application stops advertisements to change advertising parameters or to save power.
- `LM_EV_CONNECTION_COMPLETE`: This event is received with the connection with the master is considered to be complete and includes the new connection parameters.
- `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.
- `LM_EV_DISCONNECT_COMPLETE`: This event is received on link disconnection. Disconnection could be due to link loss, locally triggered or triggered by the remote connected device.
- `LM_EV_ENCRYPTION_CHANGE`: This event indicates a change in the link encryption.

4.2. Internal State Machine

The GATT Server application has six internal states, see Figure 4.1:



APP_INIT
Database registration complete
Extra Long button press / Removes bonding information OR Remote initiated connection fails
APP_ FAST_ADVERTISING
Fast advertising timer expires
APP_ SLOW_ADVERTISING
Extra Long button press / Removes bonding information OR Remote initiated connection fails
GATT Client connects
Link loss occurs OR GATT Client disconnects and application is not bonded
GATT Client connects
Disconnection completed and application is not bonded
Slow advertising timer expires
APP_CONNECTED
Idle timer expires OR Extra Long button press / Removes bonding information OR Short button press
APP_DISCONNECTING
GATT Client disconnects and application is bonded
Disconnection completed and application is bonded
Short button press OR Extra Long button press / Removes bonding information
APP_IDLE

Figure 4.1: Internal State Machine Diagram

4.2.1. APP_INIT

When the application is powered on or the chip resets, the GATT Server application initialises in this state and registers the database with the firmware. When the application receives a successful confirmation for the database registration it enters the **APP_FAST_ADVERTISING** state.

4.2.2. APP_IDLE

The GATT Server application is not connected to any GATT Client. The application sounds a **long beep** while entering this state.

- On a **short button press**, the application enters the **APP_FAST_ADVERTISING** state.
- On an **extra long button press**, the application removes the bonding information, clears the white list and enters the **APP_FAST_ADVERTISING** state. See *Bluetooth Core specification version 4.1* for more information on white lists.

4.2.3. APP_FAST_ADVERTISING

The GATT Server application sends advertisements and sounds two **short beeps** to indicate the start of advertisements. Fast advertising parameters are used in this state. If a GATT Client connects, the GATT Server stops advertising and enters the **APP_CONNECTED** state. If the fast advertising timer expires before a connection is made, the application enters the **APP_SLOW_ADVERTISING** state. See section 7.2 for more information on advertisement timers.

On an **extra long button press**, the application stops advertising, removes bonding and restarts advertising without using the white list.

4.2.4. APP_SLOW_ADVERTISING

The application uses slow advertising parameters in this state. If a GATT Client connects, the GATT Server stops advertising and enters the **APP_CONNECTED** state. If the slow advertising timer expires before a connection is made, the application enters the **APP_IDLE** state.

On an **extra long button press**, the application stops advertisements, removes bonding and restarts advertising in the **APP_FAST_ADVERTISING** state without using the white list.

4.2.5. APP_CONNECTED

The GATT Server application is connected to a GATT Client.

- On a **short button press**, the application disconnects the link and enters the **APP_DISCONNECTING** state.
- On an **extra long button press**, the application removes the bonding information, clears the white list, disconnects the link and enters the **APP_DISCONNECTING** state.
- The application disconnects the link if kept idle for some time and enters the **APP_DISCONNECTING** state. See section 7.4 for more information on the idle timer.
- If a link loss occurs, the application enters the **APP_FAST_ADVERTISING** state.
- In the case of a remote triggered disconnection and if the application is not bonded to any collector, the application enters the **APP_FAST_ADVERTISING** state, otherwise it enters the **APP_IDLE** state.

4.2.6. APP_DISCONNECTING

The GATT Server application waits for a disconnect confirmation for a disconnection initiated by itself.

- On receiving a disconnect confirmation, bonding to any GATT Client is checked:
    - If the application is bonded, it enters the **APP_IDLE** state and waits for user activity.
    - If the application is not bonded, it enters the **APP_FAST_ADVERTISING** state.
- On an **extra long button press**, the application removes the bonding information and clears the white list.

© Cambridge Silicon Radio Limited 2014

5. NVM Memory Map

The applications can store data in the NVM to prevent data loss in the event of a power off or chip panic. The GATT Server application uses the following memory map for NVM:

Entity Name	Type	Size of Entity (Words)	NVM Offset (Words)
Sanity Word	uint16	1	0
Bonded Flag	boolean	1	1
Bonded Device Address	structure	5	2
Diversifier	uint16	1	7
IRK	uint16 array	8	8

Table 5.1: NVM Memory Map for Application

Entity Name	Type	Size of Entity (Words)	NVM Offset (Words)
GAP Device Name Length	uint16	1	16
GAP Device Name	uint8 array	20	17

Table 5.2: NVM Memory Map for GAP Service

Entity Name	Type	Size of Entity (Words)	NVM Offset (Words)
Battery Level Characteristic Client Configuration Descriptor	uint16	1	37

Table 5.3: NVM Memory Map for Battery Service

Note:

The application does not pack data while storing it into NVM so storing a boolean value takes one word of NVM memory.

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. Advertising Procedure

The GATT Server application goes through two different modes of advertising based on the rate at which advertisements are sent:

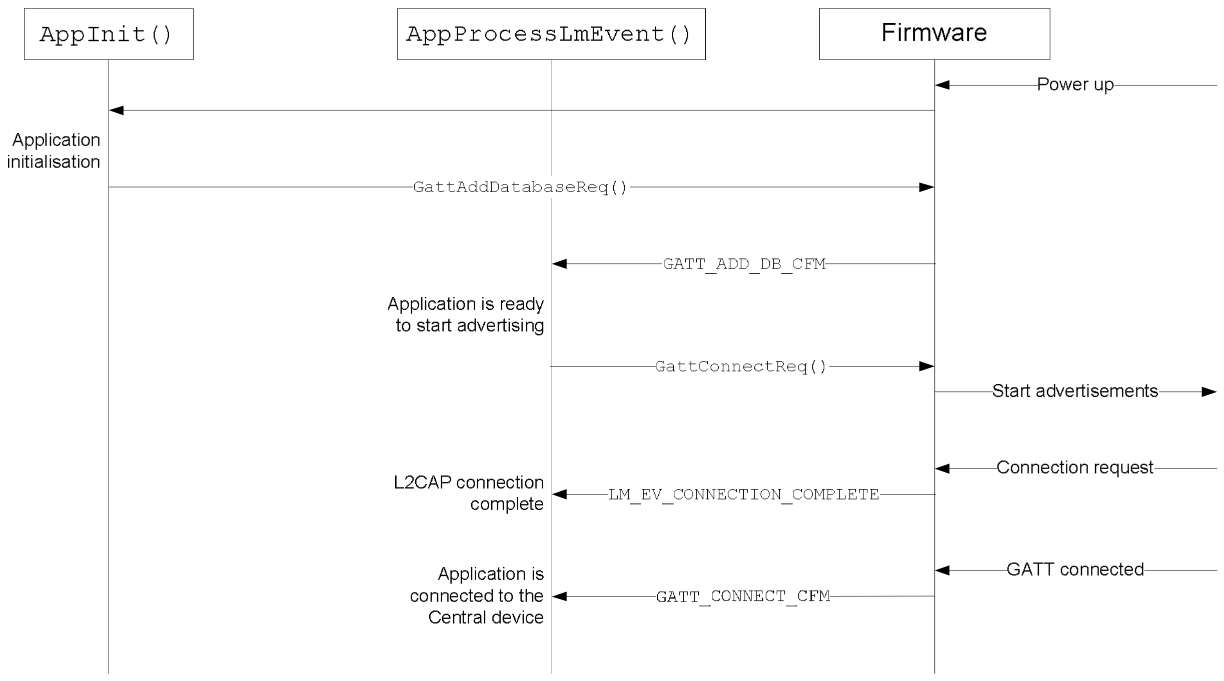
- Faster advertisements, by default, are carried out at 60 ms intervals for a duration of 30 s.
- Slow advertisements, by default, are carried out at 1.28 s intervals for a duration of 1 minute.

Advertisements and scan responses include the following information:

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

Figure 6.1 and Figure 6.2 show various steps in starting advertisements.

6.1.1. Connection Establishment



AppInit()
AppProcessLmEvent()
Firmware
Power up
Application initialisation
GattAddDatabaseReq()
GATT_ADD_DB_CFM
Application is ready to start advertising
GattConnectReq()
Start advertisements
Connection request
L2CAP connection complete
LM_EV_CONNECTION_COMPLETE
GATT connected
Application is connected to the Central device
GATT_CONNECT_CFM

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.

© Cambridge Silicon Radio Limited 2014

6.1.2. Disconnection

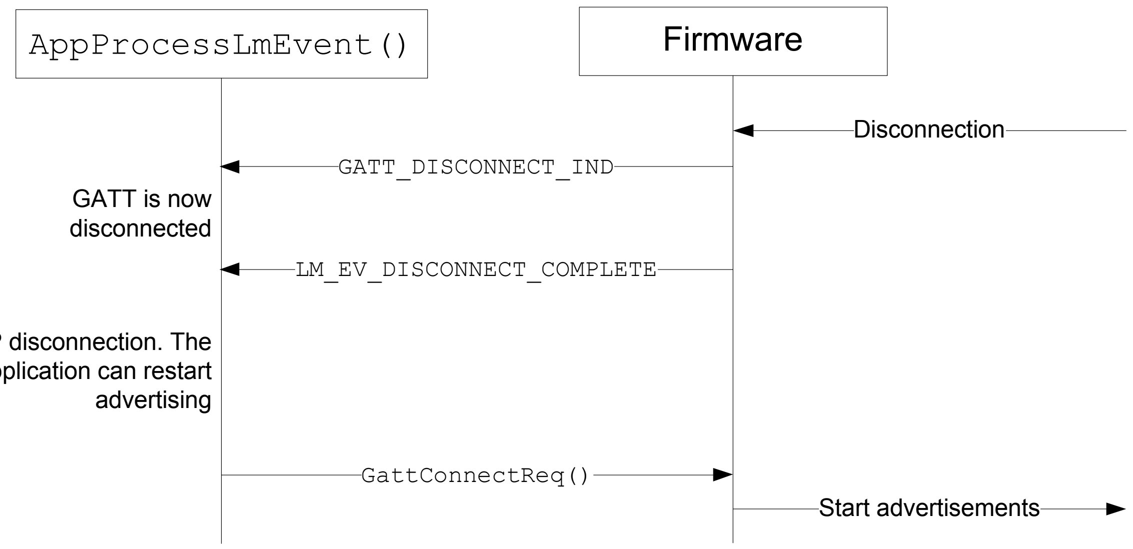


Figure 6.2: Sequence of Events Following a Disconnection

Figure 6.2 describes the sequence of operations for a typical Peripheral device following a disconnection. The advertisements are restarted following the L2CAP disconnection. The `GATT_DISCONNECT_IND` event will not be issued to the application if a GATT connection was not established.

6.2. Pairing Procedure

The example GATT Server supports pairing with a GATT Client 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):

1. If the GATT Client has supplied a Resolvable Random Address then the Pairing Procedure exits to allow the GATT Client to initiate the pairing process.
2. The Firmware Application Program Interface (API) function `SMRequestSecurityLevel()` is called to start security procedures on the link.
3. The `SM_PAIRING_AUTH_IND` event is received to give the GATT Server a chance to reject the pairing request in case it has been initiated by the GATT Client. The example GATT Server application rejects the request if it finds that it is already bonded with the GATT Client.
4. Assuming pairing is successful, the keys generated during the pairing process will be returned by the Security Manager in event `SM_KEYS_IND`. The application 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 pairing has failed after repeated attempts the link is disconnected immediately.
   - If the GATT Client was bonded but pairing has failed then the application starts a timer and allows the GATT Client to re-try pairing. If the Pairing Procedure is not completed successfully before the timer expires (configured by the `BONDING_CHANCE_TIMER` macro in `gatt_access.h`) then the link is disconnected.

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

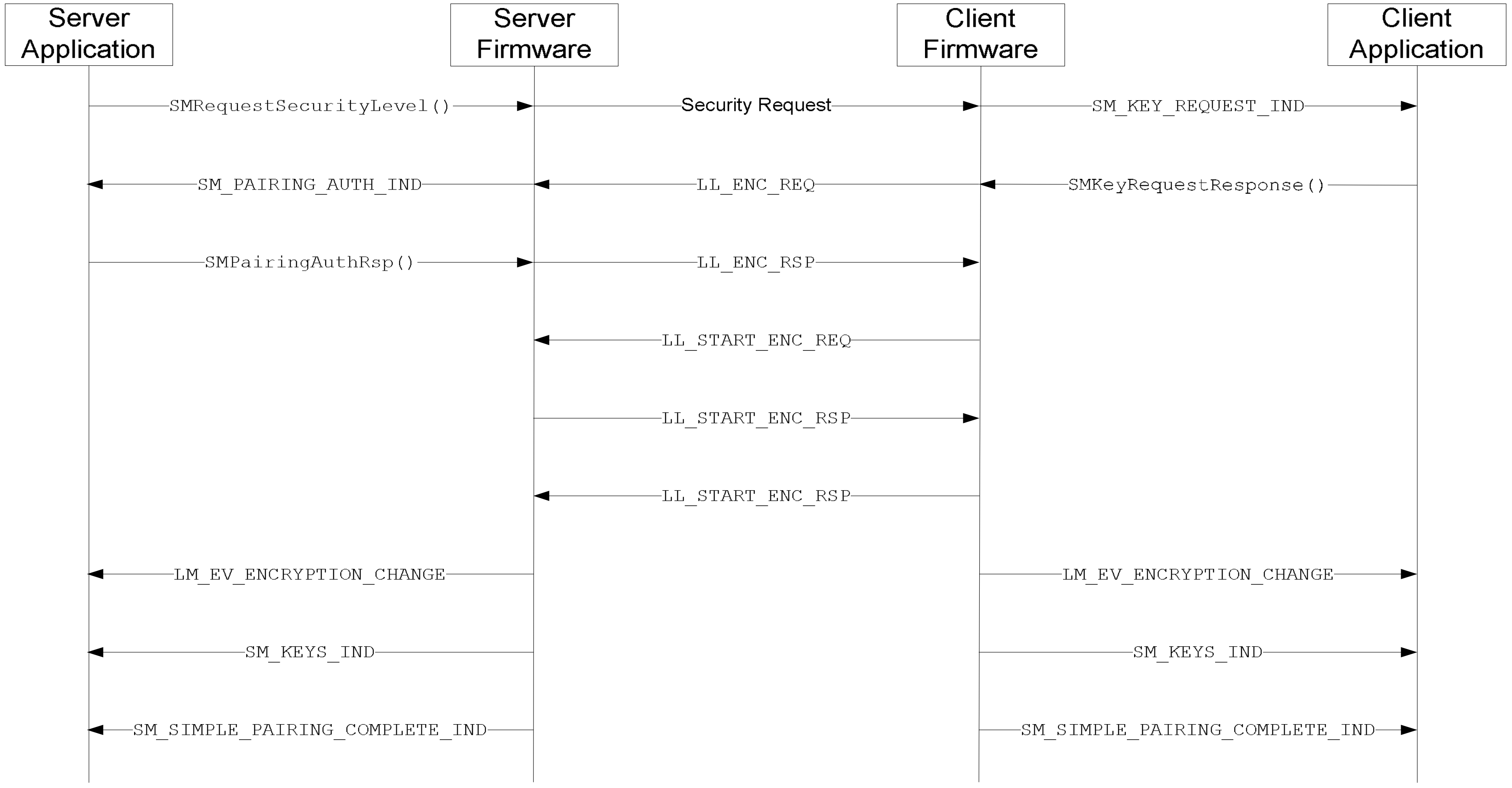


Figure 6.3: Pairing Procedure

7. Customising the Application

7.1. Advertising Parameters

The GATT Server application uses the parameters in Table 7.1 for fast and slow advertisements. The macros for these values are defined in file `gap_conn_params.h`. These values have been chosen by considering the overall current consumption of the device. See *Bluetooth Core Specification version 4.1* for advertising parameter range.

Parameter Name	Slow Advertisements	Fast Advertisements
Minimum Advertising Interval	1280 ms	60 ms
Maximum Advertising Interval	1280 ms	60 ms

Table 7.1: Advertising Parameters

7.2. Advertisement Timers

The GATT Server application enters the appropriate state on expiry of the advertisement timers. See section 4.2 for more information. The macros for these timer values are defined in file `gatt_access.h`.

Timer Name	Timer Value
Fast Advertisement Timer Value	30 s
Slow Advertisement Timer Value	60 s

Table 7.2: Advertisement Timers

7.3. 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	500 ms
Maximum Connection Interval	Maximum time between the start of two consecutive connection events	500 ms
Slave Latency	Number of connection events for which the slave is not required to listen to the master	4 intervals
Supervision Timeout	Maximum time between receiving packets before considering the connection to be broken	10 s

Table 7.3: Connection Parameters

© Cambridge Silicon Radio Limited 2014

7.4. Idle Connected Timeout

The GATT Server application provides a facility to disconnect a GATT Client after a defined period of time. Typically this would occur if the GATT Client did not perform any data accesses within the specified duration. The macro for this idle connected timeout is defined in file `user_config.h` and is commented out by default.

Timer Name	Timer Macro	Timer Value
Idle Connected Timeout	`CONNECTED_IDLE_TIMEOUT_VALUE`	5 minutes

Table 7.4: Idle Connected Timeout

7.5. Device Name

The user can change the device name for the application. By default, it is set to `CSR GATT Server` in file `gap_service.c`. The maximum length of the device name is 20 octets.

7.6. Buzzer

The GATT Server application can use the buzzer available on the CNS10004 development boards to indicate different states and events to the user. The user can enable or disable the buzzer as required by the application. The buzzer should be disabled while taking current consumption readings, see section 8. The macro `ENABLE_BUZZER`, defined in file `user_config.h`, can be commented out to disable the buzzer.

7.7. Non-Volatile Memory

The GATT Server 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. Pairing Support

The GATT Server is configured to support bonding with GATT Client devices. This may be disabled by commenting out or deleting the `PAIRING_SUPPORT` macro definition in `user_config.h`.

If `PAIRING_SUPPORT` is defined, then following an unsuccessful pairing attempt the application starts a timer. On expiry of the timer the link to the GATT Client is disconnected. The duration of this timer is configured by the `BONDING_CHANCE_TIMER` macro defined in `gatt_access.h`.

Timer Name	Timer Macro	Timer Value
Bonding Chance Timer	`BONDING_CHANCE_TIMER`	30 s

Table 7.5: Bonding Chance Timer

7.9. Debug Output

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

7.10. GATT Server Database

7.10.1. Overview

The GATT Server Database describes how data is associated with various services. Typically the Database will include:

- Mandatory GAP service data
  - Device name
  - Appearance
  - Preferred connection parameters
- Mandatory GATT service data
  - (Usually empty)
- Battery service data
  - Battery level
- Application service data
  - Data specific to the application profile(s) implemented

In the CSR µEnergy SDK, the Database is specified using database files with extension `.db`, and is included in the application's project. The GATT Server Database is generated from the `.db` files using the GATT Database Generator tool included in the SDK. The generated Database contains an array of attributes in a format specifically designed for CSR µEnergy firmware.

The application registers the generated Database with the firmware via a call to `GattAddDatabaseReq()`. The firmware will then take ownership of the memory associated with the Database which the application must not subsequently access.

7.10.2. GATT Database Generator Overview

A brief description of the GATT Database Generator follows. For more information see the *GATT Database Generator User Guide*.

The GATT Database Generator is a tool supplied with the CSR µEnergy SDK that converts a GATT database specified in a human readable format into a form optimised for use by the CSR µEnergy firmware. The service information provided in the database is usually static and can be specified at build time, although it is also possible to specify attributes whose values change at run time.

7.10.2.1. `.db` File Format

The `.db` file format is based on JavaScript Object Notation (*JSON*) which provides a lightweight text-based human-readable format. The format supports comments, whitespace, indentation and separators along with attribute definitions. The syntax supports a subset of C pre-processor directives such as `#define` and `#include`. This allows the developer to write a database file in an easily readable and maintainable manner without the need for complex binary representations.

The GATT Database Generator only accepts a single `.db` file. To improve readability and for easier maintenance, each service may be defined with its own `.db` file, and a "master" `.db` file created that includes all the required service `.db` files using the `#include` pre-processor directive.

7.10.2.2. Generator Output

The GATT Database Generator converts the input `.db` file into a header file and a source file to be included in the application.

The header file consists of:

- Macros defining the start and end handles for all the services
- Macros defining the handles for various characteristics and descriptors associated with the services
- Macros defining the length of the data associated with the characteristics

© Cambridge Silicon Radio Limited 2014

The source file consists of:

- An array containing the generated GATT Server Database in a form suitable for the firmware
- A helper function that returns a pointer to the Database, along with its size in words:
  `uint16 *GattGetDatabase(uint16 *length)`

7.10.2.3. Example

Input

```
primary_service {
            uuid:         0x1234,
            name:         "PRIMARY_SERVICE_1",

            characteristic {
                        uuid:         0x5678,
                        name:         "SERVICE_CHARACTERISTIC_1",
                        properties:   read,

                        value:        0x00 0x01 0x02 0x03
            },

            characteristic {
                        uuid:         0x0123456789ABCDEF0123456789ABCDEF,
                        name:         "SERVICE_CHARACTERISTIC_2",
                        properties:   [read | notify],
                        flags:        [FLAG_IRQ | FLAG_ENCR_R],
                        value:        0,

                        client_config {
                                    flags:        [FLAG_IRQ | FLAG_ENCR_W],
                                    name:         "SERVICE_CHAR_2_CLIENT_CONFIG"
                        }

            }
},
```

Figure 7.1: Excerpt from a GATT Database File

Figure 7.1 shows an excerpt from a typical GATT database file. It defines a service called `PRIMARY_SERVICE_1` with 16-bit UUID set to 0x1234. The service contains two characteristics:

- `SERVICE_CHARACTERISTIC_1` with 16-bit UUID 0x5678:
  - This characteristic is set with `read` permission only so Clients can only read the value. The characteristic value is 4 octets in size and set to { 0, 1, 2, 3 }
- `SERVICE_CHARACTERISTIC_2` with 128-bit UUID 0x01234567-89AB-CDEF-0123-456789ABCDEF:
  - This characteristic has `read` and `notify` permissions so its value may be read or sent in notifications only.
  - `FLAG_ENCR_R` indicates that an encrypted link is required when reading or requesting notifications.
  - `FLAG_IRQ` indicates that the application will supply the value rather than the firmware. Hence the value specified in the characteristic declaration (`value: 0`) is ignored and the firmware will ask the application to supply the value when a read request is received.
  - `SERVICE_CHAR_2_CLIENT_CONFIG` is a client configuration descriptor which can be set by each Client independently:
    - `FLAG_ENCR_W` indicates that this descriptor cannot be modified except through an encrypted link.

© Cambridge Silicon Radio Limited 2014

- `FLAG_IRQ` indicates that the value is handled by the application and not by the firmware.

Output

Given the input provided in Figure 7.1 the GATT Database Generator will produce source code, an excerpt of which is illustrated in Figure 7.2:

```
#define HANDLE_PRIMARY_SERVICE_1                                  (0x0001)
#define HANDLE_PRIMARY_SERVICE_1_END                              (0x0007)

#define HANDLE_SERVICE_CHARACTERISTIC_1                           (0x0003)
#define ATTR_LEN_SERVICE_CHARACTERISTIC_1                         (4)

#define HANDLE_SERVICE_CHARACTERISTIC_2                           (0x0005)
#define ATTR_LEN_SERVICE_CHARACTERISTIC_2                         (1)

#define HANDLE_SERVICE_CHAR_2_CLIENT_CONFIG                       (0x0007)
```

Figure 7.2: Generated Handles

A header file will be generated that contains:

- For each service, macros for the first and last handle in the service
- For each characteristic, a handle to the attribute with the characteristic value, along with the attribute size
- For each characteristic descriptor, a handle to the attribute with the characteristic descriptor value
- A function prototype used to return a pointer to the GATT Server Database and its length:

  `extern uint16 *GattGetDatabase(uint16 *len);`

A source file will be generated that contains:

- An array of data representing the GATT Server Database in a form optimised for use by the firmware:

  `uint16 gattDatabase[] = { ... };`
- A function that returns a pointer to the GATT Server Database and its length

7.10.2.4. Dynamic Values

Values supplied to the GATT Server Database are static. They may be modified by a Client but the firmware will not change the values unilaterally. They are stored in the Database and retrieved or modified by the firmware upon request.

It is possible to specify dynamic values by declaring a characteristic with the `FLAG_IRQ` property. In this case the attribute value is handled by the application rather than the firmware. Whenever the value is accessed, the firmware will forward the request onto the application using the `GATT_ACCESS_IND` event. The application should then immediately handle the event and respond to the firmware by calling `GattAccessRsp()`, see Figure 7.3:

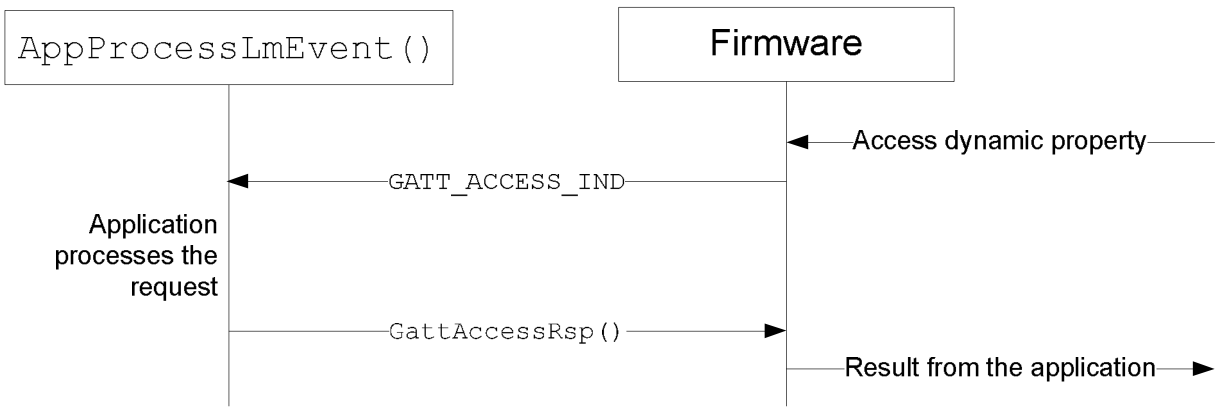


Figure 7.3: Accessing Dynamic Attribute Values

7.11. Custom GATT Profiles

Besides Bluetooth profiles adopted by the Bluetooth SIG, such as the Heart Rate Profile (HRP) or Glucose Profile (GLP), it is also possible to specify custom GATT profiles. This section will demonstrate the implementation of a custom GATT profile.

7.11.1. Use Case

The first step in implementing a custom profile is to consider how it is going to be used. The properties (characteristics) which will be made available by particular services in the GATT Server must be determined.

7.11.2. Specification

A UUID must be defined for each service and characteristic desired, as specified in the Attribute Protocol.

Custom services and characteristics should all use 128-bit UUIDs, and not 16-bit UUIDs. 16-bit UUIDs are aliases for 128-bit UUIDs using the Bluetooth Base UUID which may be written 0000xxxx-0000-1000-8000-00805F9B34FB, with the 16-bit alias substituted in the location indicated by xxxx. As such, 16-bit UUIDs are reserved by the Bluetooth SIG and are assigned to often-used, registered purposes. See the *Bluetooth Assigned Numbers* document for more information.

The base part may be generated from a MAC address and a timestamp, see *A Universally Unique IDentifier (UUID) URN Namespace* for details.

Alternatively a fixed base part may be chosen. For example, the CSR µEnergy SDK examples use a base UUID of xxxxxxxx-D102-11E1-9B23-00025B00A5A5.

7.11.3. Adding a Service to the GATT Database

A header file should be created that contains macros for all the UUIDs. The UUIDs will be defined in full for the GATT Database Generator tool. For example, the Battery Service included in the example GATT Server defines one service and one characteristic in `battery_uuids.h`:

```
#define UUID_BATTERY_SERVICE      0x180f
#define UUID_BATTERY_LEVEL        0x2a19
```

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_CUSTOM_SERVICE       0x00001c00d10211e19b2300025b00a5a5

#define UUID_CUSTOM_SERVICE_1     0x00
#define UUID_CUSTOM_SERVICE_2     0x00
#define UUID_CUSTOM_SERVICE_3     0x1c
#define UUID_CUSTOM_SERVICE_4     0x00
#define UUID_CUSTOM_SERVICE_5     0xd1
```

```
#define UUID_CUSTOM_SERVICE_6    0x02
#define UUID_CUSTOM_SERVICE_7    0x11
#define UUID_CUSTOM_SERVICE_8    0xe1
#define UUID_CUSTOM_SERVICE_9    0x9b
#define UUID_CUSTOM_SERVICE_10   0x23
#define UUID_CUSTOM_SERVICE_11   0x00
#define UUID_CUSTOM_SERVICE_12   0x02
#define UUID_CUSTOM_SERVICE_13   0x5b
#define UUID_CUSTOM_SERVICE_14   0x00
#define UUID_CUSTOM_SERVICE_15   0xa5
#define UUID_CUSTOM_SERVICE_16   0xa5
```

The GATT Database is created based on the profile specification. The example GATT Server defines one GATT Database for each service. For example, `battery_service_db.db`:

```
#include "battery_uuids.h"

/* Primary service declaration of Battery service */
primary_service {
    uuid : UUID_BATTERY_SERVICE,
    name : "BATTERY_SERVICE", /* Name will be used in handle name macro */

    /* Battery level characteristic */
    characteristic {
        uuid : UUID_BATTERY_LEVEL,
        name : "BATT_LEVEL",

        flags : [FLAG_IRQ],
        properties : [read, notify],
        value : 0x00,
        client_config {
            flags : [FLAG_IRQ],
            name : "BATT_LEVEL_C_CFG"
        }
    }
}
```

The GATT Database Generator tool only accepts a single database file for input. So a "master" database file is created that includes all of the required service database files. In the example GATT Server the master database is called `app_gatt_db.db`:

```
#include "gap_service_db.db"
#include "gatt_service_db.db"
#include "battery_service_db.db"
#include "dev_info_service_db.db"
```

The GATT Database Generator will combine each of the included databases and produce a header file (`app_gatt_db.h`) that defines the GATT Database in a form optimised for the CSR µEnergy firmware, together with macros for the attribute handles and characteristic sizes:

```
#define HANDLE_BATTERY_SERVICE          (0x001a)
#define HANDLE_BATTERY_SERVICE_END      (0xffff)

#define ATTR_LEN_BATTERY_SERVICE        (2)
#define HANDLE_BATT_LEVEL               (0x001c)
#define ATTR_LEN_BATT_LEVEL             (1)

#define HANDLE_BATT_LEVEL_C_CFG         (0x001d)
#define ATTR_LEN_BATT_LEVEL_C_CFG       (0)
```

The handles indicate the position of the attributes in the GATT Database.

7.11.4. Adding a Service Advertisement and GAP Service Data

The default device name is added to `gap_service.c`:

```
/* Default device name - Added two for storing AD Type and Null ('\0') */
uint8 g_device_name[DEVICE_NAME_MAX_LENGTH + 2] = {
    AD_TYPE_LOCAL_NAME_COMPLETE,
    'C', 'S', 'R', ' ', 'G', 'A', 'T', 'T', ' ', 'S', 'e', 'r', 'v', 'e', 'r',
    '\0'
};
```

The device appearance is defined in `appearance.h` using a custom defined value:

```
#define APPEARANCE_APPLICATION_VALUE APPEARANCE_GATT_SERVER_VALUE
```

The preferred connection parameters are maintained by the GAP Service. Default values are provided for use by the application, but they may be changed in `gap_conn_params.h` if required.

The application services are specified with a call to `GetSupported128BitUUIDServiceList()` in `app_gatt.c`. 128-bit values may not be assigned in C, so instead the 128-bit UUID 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):

```
uint16 GetSupported128BitUUIDServiceList(uint8 *p_service_uuid_ad)
{
        uint16 i = 0;

        /* Add 128-bit UUID for supported main service */
        p_service_uuid_ad[i++] = AD_TYPE_SERVICE_UUID_128BIT_LIST;

        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_16;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_15;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_14;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_13;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_12;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_11;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_10;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_9;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_8;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_7;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_6;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_5;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_4;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_3;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_2;
        p_service_uuid_ad[i++] = UUID_CUSTOM_SERVICE_1;

        return i;
}
```

Finally, the device Bluetooth address should be stored in a `.keyr` file:

```
// (0001) - Bluetooth device address
&BDADDR = 1510 5b00 0002
```

© Cambridge Silicon Radio Limited 2014

7.11.5. Implementing a Service

The example application services provided with the CSR µEnergy SDK are implemented using a consistent interface. It is recommended that custom profile services follow the same conventions for ease of maintenance:

Function Name	Description
`<Service>InitChipReset()`	Performs service specific initialisation when the chip is started. Called from `AppInit()`.
`<Service>DataInit()`	Initialise service specific internal data
`<Service>ReadDataFromNVM()`	Called during initialisation if the service needs to read data from the Non-Volatile Memory (NVM)
`<Service>CheckHandleRange()`	Returns TRUE when the supplied handle belongs to the service
`<Service>HandleAccessRead()` `<Service>HandleAccessWrite()`	Called in response to `GATT_ACCESS_IND` events to handle characteristics with dynamic values

Table 7.6: Common Service Interface Functions

The recommended functions may be present but left empty when there is no immediate need for them. For example, `<Service>InitChipReset()` could be left empty if there is no requirement to reset the service data when the device is reset. Similarly if there is no service data to be stored in NVM `<Service>ReadDataFromNVM()` may also be left empty. In a future version of the code the NVM could be used to store data that needs to be recalled after a power cycle.

The GATT Database may specify that some characteristics will be handled by the application. This means the application will be responsible for that data, and will need to allocate space for the characteristic value, initialise the characteristic to a reasonable value, and handle access events received from the control device.

The value could be stored in the service source file, and initialised in `<Service>DataInit()`. Taking the Battery Service in the example GATT Server:

```
/* Battery service data type */
typedef struct
{
    /* Battery level in percent */
    uint8     level;
} BATT_DATA_T;

/* Battery service data instance */
static BATT_DATA_T g_batt_data;
```

The battery level will be initialised in `<Service>DataInitChipReset()`:

```
void BatteryInitChipReset(void)
{
    g_batt_data.level = 0;
}
```

Handling access to the value requires the supplied handle be validated first, which is performed by `<Service>CheckHandleRange()`:

```
bool BatteryCheckHandleRange(uint16 handle)
{
        bool rc = FALSE;

        if ((handle >= HANDLE_BATTERY_SERVICE) &&
             (handle <= HANDLE_BATTERY_SERVICE_END))
                rc = TRUE;

        return rc;
}
```

When the application receives a `GATT_ACCESS_IND` event, it uses `<Service>CheckHandleRange()` to work out which service is responsible for the supplied handle. It then calls `<Service>HandleAccessRead()` or `<Service>HandleAccessWrite()` as appropriate.

```
...
else if (BatteryCheckHandleRange(event->handle))
{
        /* Attribute handle belongs to Battery service */
        BatteryHandleAccessRead(event);
}
...
```

The service may implement read access as follows:

```
void BatteryHandleAccessRead(GATT_ACCESS_IND_T *event)
{
        uint16      length = 0;                /* Attribute size */
        uint8      *data;                      /* Pointer to attribute value */
        sys_status  rc = sys_status_success;   /* Function status */

        switch (event->handle)
        {
                case HANDLE_BATT_LEVEL:
                        /* Reading battery level */
                        length = 1;         /* one octet */
                        g_batt_data.level = readBatteryLevel();
                        data = &g_batt_data.level;
                        break;

                default:
                        /* No more IRQ characteristics */
                        rc = gatt_status_read_not_permitted;
                        break;
        }

        /* Send access response */
        GattAccessRsp(event->cid, event->handle, rc, length, data);
}
```

The service may need to communicate with the hardware. For the example applications provided with the CSR µEnergy SDK, the functions used to communicate with hardware are all placed in a file called <`app>_hw.c`.

8. Current Consumption

The current consumed by the application can be measured by removing the Current Measuring Jumper (see Figure 2.3) and installing an ammeter in its place. The ammeter should be set to DC, measuring current from µA to mA. Any code that sounds the buzzer should be disabled before measuring the actual current consumed. See section 7.6 for more information on how to disable the buzzer.

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 CSR1010 development board. See the Release Notes for the actual current consumption values measured for the application.

Test Scenario	Description	Average Current Consumption	Remarks
Fast Advertisements	1. Switch on the GATT Server device 2. Wait for 5 s 3. Take the measurement	393 µA	▪ Advertisement interval: 60 ms ▪ Advertisement data length: 31 octets ▪ Measurement time duration: 20 s
Slow Advertisements	1. Switch on the GATT Server device 2. Wait for 40 s 3. Take the measurement	27 µA	▪ Advertising interval: 1280 ms ▪ Advertisement data length: 31 octets ▪ Measurement time duration: 40 s
Connected Idle	1. Connect to the GATT Client device 2. Wait for 30 s 3. Take the measurement	11 µA	▪ Minimum connection interval: 500 ms ▪ Maximum connection interval: 500 ms ▪ Slave latency: 4 intervals ▪ Measurement time duration: 60 s
Connected active	1. Connect to the GATT Client device 2. Send a battery notification every 5 s 3. Take the measurement	18 µA	▪ Minimum connection interval: 500 ms ▪ Maximum connection interval: 500 ms ▪ Slave latency: 4 intervals ▪ Measurement time duration: 60 s
Disconnected Idle	1. Do not connect to any GATT Client device 2. Wait for 100 s 3. Take the measurement	4 µA	▪ Minimum connection interval: 500 ms ▪ Maximum connection interval: 500 ms ▪ Slave latency: 4 intervals ▪ Measurement time duration: 60 s
Notes: ▪ Average current consumption is measured at 3.0 V ▪ Ammeter used: Agilent 34411A ▪ Buzzer disabled			

Table 8.1: Current Consumption Values

© Cambridge Silicon Radio Limited 2014

Appendix A Definitions

Term	Meaning
short button press	Button press for less than 4 seconds
extra long button press	Button press for greater than or equal to 4 seconds
short beep	Beep for 100 ms
long beep	Beep for 500 ms

Table A.1: Definitions

© Cambridge Silicon Radio Limited 2014

Appendix B Service Characteristics

Characteristics are managed by either the firmware or the application. The characteristics managed by the application have flags set to `FLAG_IRQ` in the corresponding database file. When the remote connected device accesses that characteristic, the application receives an `GATT_ACCESS_IND` LM event which is handled in the `AppProcessLmEvent()` function defined in the `gatt_server.c` file. See section 4.1.4.1 for more information on handling of the `GATT_ACCESS_IND` LM event. For more information on flags, see the *GATT Database Generator User Guide*.

B.1 Battery Service Database

Characteristic Name	Database Handle	Access Permissions	Managed By	Security Permissions	Value
Battery Level	`0x001c`	Read, Notify	Application	Security mode 1 and Security level 2	Current battery level
Battery Level-Client Configuration Descriptor	`0x001d`	Read, Write	Application	Security mode 1 and Security level 2	Current client configuration for “Battery level” characteristic

Table B.1: Battery Service Database

For more information on Battery Service, see *Battery Service Specification version 1.0*.

B.2 Device Information Service Database

Characteristic Name	Database Handle	Access Permissions	Managed By	Security Permissions	Value
Serial Number String	`0x000b`	Read	Firmware	Security Mode 1 and Security Level 2	"CSR-GATT-SERVER-001"
Model Number String	`0x000d`	Read	Firmware	Security Mode 1 and Security Level 2	"CSR-GATT-SERVER-MODEL-001"
System ID	`0x000f`	Read	Application	Security Mode 1 and Security Level 2	Organizationally Unique identifier is `0x00025b` Manufacturer Identifier depends on the device address
Hardware Revision String	`0x0011`	Read	Firmware	Security Mode 1 and Security Level 2	<Chip Identifier>
Firmware Revision String	`0x0013`	Read	Firmware	Security Mode 1 and Security Level 2	<SDK version>
Software Revision String	`0x0015`	Read	Firmware	Security Mode 1 and Security Level 2	<Application version>
Manufacturer Name String	`0x0017`	Read	Firmware	Security Mode 1 and Security Level 2	"Cambridge Silicon Radio"
PnP ID	`0x0019`	Read	Firmware	Security Mode 1 and Security Level 2	Vendor Id source is BT Vendor Id is `0x000a` Product Id is `0x014c` Product Version is 1.0.0

Table B.2: Device Information Service Database

For more information on Device Information Service, see *Device Information Service Specification version 1.1*.

For more information on Security permissions, see *Bluetooth Core Specification version 4.1*.

© Cambridge Silicon Radio Limited 2014

B.3 GAP Service Database

Characteristic Name	Database Handle	Access Permissions	Managed By	Security Permissions	Value
Device Name	`0x0003`	Read, Write	Application	Security Mode 1 and Security Level 2	Device name. Default name: "CSR GATT Server"
Appearance	`0x0005`	Read	Firmware	Security Mode 1 and Security Level 1	Unknown Value - `0x0000`
Peripheral preferred connection parameters	`0x0007`	Read	Firmware	Security Mode 1 and Security Level 1	Min connection interval - 500 ms Max connection interval - 500 ms Slave latency - 4 Connection timeout -10 s

Table B.3: GAP Service Database

For more information on GAP service and security permissions, see *Bluetooth Core Specification version 4.1*.

Document References

Document	Reference
A Universally Unique IDentifier (UUID) URN Namespace	http://www.ietf.org/rfc/rfc4122.txt
Adopted Bluetooth Profiles and Services	https://www.bluetooth.org/en-us/specification/adopted-specifications
ATT Specification	Volume 3, Part F of the Bluetooth Core Specification version 4.1
Battery Service Specification version 1.0	http://www.bluetooth.org/
Bluetooth Assigned Numbers	https://www.bluetooth.org/en-us/specification/assigned-numbers
Bluetooth Core Specification version 4.1	http://www.bluetooth.org/
CSR µEnergy xIDE User Guide	CS-212742-UG
Device Information Service Specification version 1.1	http://www.bluetooth.org/
GAP Specification	Volume 3, Part C of the Bluetooth Core Specification version 4.1
GATT Client Application Note	CS-308754-AN
GATT Database Generator User Guide	CS-219225-UG
GATT Specification	Volume 3, Part G of the Bluetooth Core Specification version 4.1
JSON	RFC 4627 http://www.json.org/
SDK Reference Documentation	Supplied with the CSR µEnergy SDK as the Firmware Library Documentation
CSR µEnergy CSR1000 Development Kit Quick Start Guide	CS-213120-UG
CSR µEnergy CSR1010 Development Kit Quick Start Guide	CS-232523-UG

© Cambridge Silicon Radio Limited 2014

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
CSR	Cambridge Silicon Radio
CSR µEnergy®	Group term for CSR's range of Bluetooth Smart wireless technology chips
DC	Direct Current
DIS	Device Information Service
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
GLP	Glucose Profile
HRP	Heart Rate Profile
I^2C	Inter-Integrated Circuit
JSON	JavaScript Object Notation
L2CAP	Logical Link Control and Adaptation Layer: A Bluetooth protocol that provides connection-oriented and connectionless data services to upper layer protocols
MAC	Media Access Control
NVM	Non-Volatile Memory
PC	Personal Computer
PIO	Programmable Input/Output
PS	Persistent Store
SDK	Software Development Kit
SIG	Special Interest Group

© Cambridge Silicon Radio Limited 2014

SM	Security Manager
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver-Transmitter
URN	Uniform Resource Name
USB	Universal Serial Bus
UUID	Universally Unique Identifier

© Cambridge Silicon Radio Limited 2014