Sample Application User's Guide

Document 2012-0022
Revision 1.1
September 30, 2015

Table of Contents

1 Introduction

Wicentric's Bluetooth low energy sample applications provide example source code for products such as a proximity keyfob, health sensor, and watch.

1.1 Overview

Wicentric's sample applications are designed with a product-oriented focus, with each application supporting one or more Bluetooth LE profile. The table below summarizes the different sample applications with their supported profiles and device roles.

Application Name	Description	Supported Profiles	Device Role
"datc"	Proprietary data client	Proprietary Profile	Master
"dats"	Proprietary data server	Proprietary Profile	Slave
"fit"	Fitness sensor	Heart Rate Profile Battery Service	Slave
"gluc"	Glucose sensor	Glucose Profile	Slave
"medc"	Health data collector	Blood Pressure Profile Glucose Profile Heart Rate Profile Weight Scale Profile Health Thermometer Profile	Master
"meds"	Health sensor	Blood Pressure Profile Weight Scale Profile Health Thermometer Profile	Slave
"tag"	Proximity tag	Find Me Profile Proximity Profile	Slave
"watch"	Watch accessory with message alerts	Alert Notification Profile Phone Alert Status Profile Time Profile	Slave
"keyboard"	HID Computer keyboards	HID Service Battery Service	Slave
"mouse"	HID Computer mice	HID Service Battery Service	Slave
"remote"	HID Remote controls	HID Service Battery Service	Slave

(Note that the applications and profiles listed above are not necessarily included in all customer releases; the release you receive will only contain the applications for the profiles you have licensed.)

2 Sample Application Operation

The sample applications are designed to interact with the user via buttons and to provide user feedback via LEDs, sounds, or other mechanisms depending on the capabilities of the target hardware platform.

The applications use the following button press durations:

Press Duration	Description
Short Press	Button pressed for less than 1.6 seconds.
Medium Press	Button pressed for greater than 1.6 seconds and less than 3.2 seconds.
Long Press	Button pressed for greater than 3.2 seconds and less than 4.8 seconds.
Extra Long Press	Button pressed for greater than 4.8 seconds.

2.1 datc

The datc application implements the master role of a proprietary data transfer application. It has an "auto connect" feature, where it scans for and then automatically connects to a matching proprietary data transfer slave application. Once connected, the application can send and receive simple data messages.

2.1.1 Button Operation

Button Press	Description
When disconnected	
Button 1 Short	Initiate auto connect, or cancel auto connect if already initiated.
Button 1 Long	Clear bonded device information.
When connected	
Button 1 Short	Send data packet.
Button 1 Long	Disconnect.

2.2 dats

The dats application implements the slave role of a proprietary data transfer application. When the application starts it will begin advertising. The application advertises continuously when not connected.

The dats application does not use any button presses. When it receives a data message from the peer device it will automatically send a fixed data message back.

2.3 fit

The fit application implements a heart rate profile sensor. When the application starts it will advertise for 60 seconds. A button press is used to restart advertising if it has stopped.

2.3.1 Button Operation

Button Press	Description
When disconnected	
Button 1 Short	Restart advertising.
Button 1 Medium	Enter discoverable and bondable mode and start advertising.
Button 1 Long	Clear bonded device information and then start advertising.
When connected	

Button 1 Short	Increment simulated heart rate.
Button 1 Long	Disconnect.
Button 2 Short	Decrement simulated heart rate.

2.4 gluc

The gluc application implements a glucose profile sensor. When the application starts it will advertise for 60 seconds. A button press is used to restart advertising if it has stopped.

2.4.1 Button Operation

Button Press	Description
When disconnected	
Button 1 Short	Restart advertising.
Button 1 Medium	Enter discoverable and bondable mode and start advertising.
Button 1 Long	Clear bonded device information and then start advertising.
When connected	
Button 1 Long	Disconnect.

2.5 medc

The medc application implements the collector role of several different health profiles. The selected profile is configured at run time or compile time. Note that although the application supports multiple profiles it does not support simultaneous operation of multiple profiles.

The medc application has an "auto connect" feature, where it scans for and then automatically connects to a matching profile.

2.5.1 Compile Options

The following compile options can be configured in **medc_main.c**.

Name	Description
MEDC_HRP_INCLUDED	TRUE if heart rate profile included.
MEDC_BLP_INCLUDED	TRUE if blood pressure profile included.
MEDC_GLP_INCLUDED	TRUE if glucose profile included.
MEDC_WSP_INCLUDED	TRUE if weight scale profile included.
MEDC_HTP_INCLUDED	TRUE if health thermometer profile included.
MEDC_PROFILE	Default profile to use.

The values for macro MEDC_PROFILE are as follows:

Name	Description
MEDC_ID_HRP	Heart rate profile.
MEDC_ID_BLP	Blood pressure profile.
MEDC_ID_GLP	Glucose profile.

MEDC_ID_WSP	Weight scale profile.
MEDC_ID_HTP	Health thermometer profile.

2.5.2 Default Button Operation, All Profiles

Button Press	Description
When disconnected	
Button 1 Short	Initiate auto connect, or cancel auto connect if already initiated.
Button 1 Long	Clear bonded device information.
When connected	
Button 1 Long	Disconnect.

2.5.3 Button Operation, Glucose Profile

Button Press	Description
When connected	
Button 1 Short	Report all records.
Button 1 Medium	Report records greater than sequence number.
Button 2 Short	Report number of records.
Button 2 Medium	Report number of records greater than sequence number.
Button 2 Long	Abort.
Button 2 Extra Long	Delete all records.

2.6 meds

The meds application implements the sensor role of several different health profiles. The selected profile is configured at run time or compile time. Note that although the application supports multiple profiles it does not support simultaneous operation of multiple profiles.

When the application starts it will advertise for 60 seconds. A button press is used to restart advertising if it has stopped.

The application uses simulated sensor values for its sensor data.

2.6.1 Compile Options

The following compile options can be configured in **meds_main.c**.

Name	Description
MEDS_BLP_INCLUDED	TRUE if blood pressure profile included.
MEDS_WSP_INCLUDED	TRUE if weight scale profile included.
MEDS_HTP_INCLUDED	TRUE if health thermometer profile included.
MEDS_PROFILE	Default profile to use.

The values for macro MEDS_PROFILE are as follows:

Name	Description

MEDS_ID_BLP	Blood pressure profile.
MEDS_ID_WSP	Weight scale profile.
MEDS_ID_HTP	Health thermometer profile.

2.6.2 Default Button Operation, All Profiles

Button Press	Description
When disconnected	
Button 1 Short	Restart advertising.
Button 1 Medium	Enter discoverable and bondable mode and start advertising.
Button 1 Long	Clear bonded device information and then start advertising.
When connected	
Button 1 Long	Disconnect.

2.6.3 Button Operation, Blood Pressure Profile

Button Press	Description
When connected	
Button 1 Short	Press to start a measurement. If already started, press again to complete measurement and send final measurement value.

2.6.4 Button Operation, Weight Scale Profile

Button Press	Description
When connected	
Button 1 Short	Send final measurement value.

2.6.5 Button Operation, Health Thermometer Profile

Button Press	Description
When connected	
Button 1 Short	Press to start a measurement. If already started, press again to complete measurement and send final measurement value.
Button 2 Short	Set units to Fahrenheit.
Button 2 Medium	Set units to Celsius.

2.7 tag

The tag application implements the proximity and find me profiles. When the application starts it will begin advertising. The application advertises continuously when not connected.

2.7.1 Button Operation

Button Press	Description
When disconnected	
Button 1 Short	Restart advertising.
Button 1 Medium	Enter discoverable and bondable mode and start advertising.
Button 1 Long	Clear bonded device information and then start advertising.

When connected	
Button 1 Short	Send immediate alert.
Button 1 Medium	Stop immediate alert.
Button 1 Long	Disconnect.

2.8 watch

The watch application implements several profiles applicable to a watch. When the application starts it will begin advertising. The application advertises continuously when not connected.

2.8.1 Button Operation

Button Press	Description
When disconnected	
Button 1 Short	Restart advertising.
Button 1 Medium	Enter discoverable and bondable mode and start advertising.
Button 1 Long	Clear bonded device information then start advertising.
When connected	
Button 1 Short	Mute ringer once.
Button 1 Medium	Toggle between silencing ringer and enabling ringer.
Button 1 Long	Disconnect.

2.9 keyboard

The keyboard application implements several profiles applicable to HID Keyboard. When the application starts it will begin advertising. The application advertises continuously when not connected. A full keyboard cannot be implemented with two buttons. Therefore, the keyboard application demonstrates the implementation of a HID keyboard that only supports the Up Arrow and Down Arrow keys.

2.9.1 Button Operation

Button Press	Description
When connected	
Button 1 Short	Transmit Up Arrow keypress.
Button 2 Short	Transmit Down Arrow keypress.
All other button events	Transmit None keypress.

2.10 mouse

The mouse application implements several profiles applicable to HID Computer Mice. When the application starts it will begin advertising. The application advertises continuously when not connected. A full mouse cannot be implemented with two buttons. Therefore, the mouse application demonstrates the implementation of left button and right button.

2.10.1 Button Operation

Button Press	Description
When connected	

Button 1 Short	Transmit Left Mouse Button.
Button 2 Short	Transmit Right Mouse Button.
All other button events	Transmit No Button event.

2.11 remote

The remote application implements several profiles applicable to a HID Consumer Remote Control. When the application starts it will begin advertising. The application advertises continuously when not connected. A full remote control cannot be implemented with two buttons. Therefore, the remote application demonstrates the implementation of a play button and a stop button.

2.11.1 Button Operation

Button Press	Description
When connected	
Button 1 Short	Transmit Play.
Button 2 Short	Transmit Stop.
All other button events	Transmit No Button event.

3 Software Design

3.1 Software System

The sample applications are part of Wicentric's *exactLE* Profiles software system, as shown in Figure 1. The sample applications interface to the Profiles and Services, which provide interoperable components designed to Bluetooth specification requirements. The sample applications also interface to the App Framework, which provides connection and device management services, user interface services, a device database, and a hardware sensor interface.

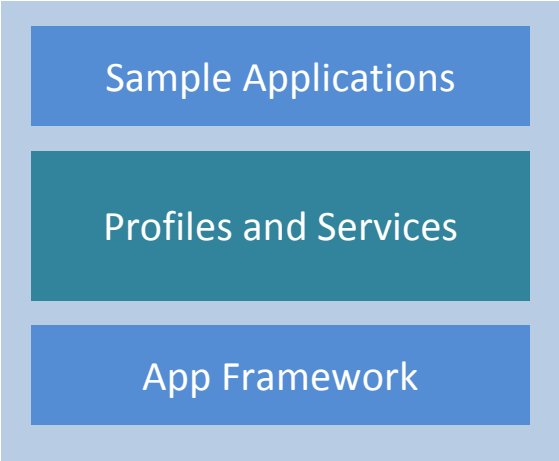

Figure 1. The exactLE Profiles software system.

For a complete description of the App Framework API and Profiles and Services API see their respective API documents [1] and [2].

3.2 Sample Application Design

All sample applications follow the same basic design model and consist of the following:

- **Configurable parameters**: Data structures that control the behavior of advertising, security, and connections.
- **Attribute protocol (ATT) data**: Data structures and constants that configure service discovery and manage client characteristic configuration descriptor (CCCD) data for the ATT client and server.
- **Protocol stack and App Framework callbacks**: These functions interface the exactLE protocol stack and App Framework to the sample application event handler.
- **Button press handler**: This function controls the application behavior on button press events. For example, start advertising on a short button press.
- **Event handler and event processing functions**: These functions handle events from the protocol stack and perform actions specific to the sample application. For example, generate a UI alert when the connection is closed.

4 Sample Application Code Walkthrough

This code walkthrough provides a detailed description of how a sample application works. The example code in this walkthrough is taken from the "tag" sample application in file tag_main.c.

4.1 Configurable Parameters

This section of the code contains configurable parameters for slave, security, and connection update.

4.1.1 Slave Parameters

The slave parameters configuration structure `appAdvCfg_t` configures the interval and duration of advertising. The structure contains three interval-duration pairs.

A code example is shown below:

```
/*! configurable parameters for slave */
static const appAdvCfg_t tagSlaveCfg =
{
  {15000, 45000,     0},    /*! Advertising durations in ms */
  {   56,   640,  1824}     /*! Advertising intervals in 0.625 ms units */
};
```

Note that the advertising interval is in 0.625ms units. For example: 56 * 0.625ms = 35ms. Also, if the advertising duration is zero then advertising will not time out and will continue until a connection is established or advertising is stopped by the application.

This example creates the following advertising behavior: Advertise with a 35ms interval for 15 seconds, then a 400ms interval for 45 seconds, then continuously with a 1140ms interval.

4.1.2 Security Parameters

The security parameters structure `appSecCfg_t` configures the security options for the application. A code example is shown below:

```
/*! configurable parameters for security */
static const appSecCfg_t tagSecCfg =
{
  DM_AUTH_BOND_FLAG,  /*! Authentication and bonding flags */
  0,                  /*! Initiator key distribution flags */
  DM_KEY_DIST_LTK,    /*! Responder key distribution flags */
  FALSE,              /*! TRUE if Out-of-band pairing data is present */
  TRUE                /*! TRUE to initiate security upon connection */
};
```

This example creates the following security behavior:

- Request bonding and use "just works" pairing without a PIN.
- Only distribute the minimum required keys.
- Out-of-band data (used instead of a PIN for pairing) is not present.
- Initiate a request for security upon connection.

4.1.3 Connection Update Parameters

The structure `appUpdateCfg_t` configures the connection update parameters. These parameters are used after a connection is established to reconfigure a connection for low power and/or low latency.

The `appUpdateCfg_t` structure is currently only used by slave devices.

A code example is shown below:

```
/*! configurable parameters for connection parameter update */
static const appUpdateCfg_t tagUpdateCfg =
{
  6000,  /*! Connection idle period in ms before attempting
             connection parameter update; set to zero to disable */
  640,   /*! Minimum connection interval in 1.25ms units */
  800,   /*! Maximum connection interval in 1.25ms units */
  0,     /*! Connection latency */
  600,   /*! Supervision timeout in 10ms units */
  5      /*! Number of update attempts before giving up */
};
```

Note that the connection interval is in 1.25ms units. For example, 640 * 1.25 = 800ms.

This example creates the following behavior:

- Request a connection parameter update after the connection has been idle for at least 6 seconds. The connection is considered idle when there is no pending security procedure or ATT discovery procedure.
- Request a connection interval between 800 and 1000ms.
- Request a connection latency of zero, meaning that the slave and master have equal connection intervals.
- Set the supervision timeout to 6 seconds. If the connection is lost for 6 seconds the devices will disconnect.
- Attempt a connection parameter update 5 times. The master device may reject a connection parameter update if it is busy; if this occurs the connection parameter update will be attempted again.

4.1.4 HID Parameters

Applications using the HID service (e.g. keyboard, mouse, and remote) must register a hidConfig_t with the HID profile.

A code example is shown below:

```
/*! HID Profile Configuration */
static const hidConfig_t mouseHidConfig =
{
  HID_DEVICE_TYPE_MOUSE,                              /* Type of HID device */
  (uint8_t*) mouseReportMap,                          /* Report Map */
  sizeof(mouseReportMap),                             /* Size of report map in bytes */
  (hidReportIdMap_t*) mouseReportIdSet,               /* Report ID to Attribute Handle map */
  sizeof(mouseReportIdSet)/sizeof(hidReportIdMap_t), /* ID to Handle map size (bytes) */
  NULL,                                               /* Output Report Callback */
  NULL,                                               /* Feature Report Callback */
  mouseInfoCback                                      /* Info Callback */
};

HidInit(&mouseHidConfig);
```

This example creates the following behavior:

- A HID Mouse device, defined by the HID_DEVICE_TYPE_MOUSE. HID mice support Boot Mouse HID Reports. Alternative device types are HID_DEVICE_TYPE_KEYBOARD which support the Boot Keyboard Reports, and HID_DEVICE_TYPE_GENERIC which do not support the HID Boot Protocol Mode or HID Boot Reports.
- Registers a HID Report Map defined by the mouseReportMap structure shown below:

```
static const uint8_t mouseReportMap[] =
{
  0x05, 0x01,                    /* USAGE_PAGE (Generic Desktop) */
  0x09, 0x02,                    /* USAGE (Mouse) */
  0xa1, 0x01,                    /* COLLECTION (Application) */
  0x09, 0x01,                    /*   USAGE (Pointer) */
  0xa1, 0x00,                    /*   COLLECTION (Physical) */
  0x05, 0x09,                    /*     USAGE_PAGE (Button) */
  0x19, 0x01,                    /*     USAGE_MINIMUM (Button 1) */
  0x29, 0x03,                    /*     USAGE_MAXIMUM (Button 3) */
  0x15, 0x00,                    /*     LOGICAL_MINIMUM (0) */
  0x25, 0x01,                    /*     LOGICAL_MAXIMUM (1) */
  0x95, 0x03,                    /*     REPORT_COUNT (3) */
  0x75, 0x01,                    /*     REPORT_SIZE (1) */
  0x81, 0x02,                    /*     INPUT (Data,Var,Abs) */
  0x95, 0x01,                    /*     REPORT_COUNT (1) */
  0x75, 0x05,                    /*     REPORT_SIZE (5) */
  0x81, 0x03,                    /*     INPUT (Cnst,Var,Abs) */
  0x05, 0x01,                    /*     USAGE_PAGE (Generic Desktop) */
  0x09, 0x30,                    /*     USAGE (X) */
  0x09, 0x31,                    /*     USAGE (Y) */
  0x15, 0x81,                    /*     LOGICAL_MINIMUM (-127) */
  0x25, 0x7f,                    /*     LOGICAL_MAXIMUM (127) */
  0x75, 0x08,                    /*     REPORT_SIZE (8) */
  0x95, 0x02,                    /*     REPORT_COUNT (2) */
  0x81, 0x06,                    /*     INPUT (Data,Var,Rel) */
  0xc0,
  0xc0
};
```

The HID Report Map is a HID Report Descriptor for the HID device. A detailed description of HID Report Descriptors can be found in the USB HID spec.

- A map between the HID Report ID and the ATT attribute handle as specified by the code below:

```
static const hidReportIdMap_t mouseReportIdSet[] =
{
  /* type                  ID           handle */
  {HID_REPORT_TYPE_INPUT, 0,           HIDM_INPUT_REPORT_HDL},  /* Input Report */
  {HID_REPORT_TYPE_INPUT, HID_BOOT_ID, HIDM_MOUSE_BOOT_IN_HDL}, /* Boot Input Report */
};
```

- An information callback that receives notification of HID Control Point and HID Protocol Mode messages via the mouseInfoCback function.
- An application that does not receive HID feature or output reports. Applications wishing to receive HID output or feature reports must provide callback functions to the outputCback or featureCback parameters of the hidConfig_t structure.

4.2 Advertising Data

The advertising data and scan response data is configured via simple byte arrays. There can be separate sets of advertising and scan response data for connectable and discoverable mode (as we'll see later on in Section 4.7.2).

The contents of advertising and scan response data follow a simple "length-type-value" format as defined by the Bluetooth specification. The length byte contains the length of the type byte and value bytes that follow. The type byte contains the advertising data type, or AD type, specifying a particular type of data. The value bytes, if present, are set according to the AD type.

Example advertising and scan response data is shown below:

```
/*! advertising data, discoverable mode */
static const uint8_t tagAdvDataDisc[] =
{
  /*! flags */
  2,                                    /*! length */
  DM_ADV_TYPE_FLAGS,                    /*! AD type */
  DM_FLAG_LE_LIMITED_DISC |             /*! flags */
  DM_FLAG_LE_BREDR_NOT_SUP,

  /*! tx power */
  2,                                    /*! length */
  DM_ADV_TYPE_TX_POWER,                 /*! AD type */
  0,                                    /*! tx power */

  /*! device name */
  14,                                   /*! length */
  DM_ADV_TYPE_LOCAL_NAME,               /*! AD type */
  'w',
  'i',
  'c',
  'e',
  'n',
  't',
  'r',
  'i',
  'c',
  ' ',
  'a',
  'p',
  'p'
};

/*! scan data, discoverable mode */
static const uint8_t tagScanDataDisc[] =
{
  /*! service UUID list */
  7,                                    /*! length */
  DM_ADV_TYPE_16_UUID,                  /*! AD type */
  UINT16_TO_BYTES(ATT_UUID_LINK_LOSS_SERVICE),
  UINT16_TO_BYTES(ATT_UUID_IMMEDIATE_ALERT_SERVICE),
  UINT16_TO_BYTES(ATT_UUID_TX_POWER_SERVICE)
};
```

The advertising data consists of three AD type fields: flags, TX power, and device name. The flags are set to "limited discoverable" mode. The TX power is set to 0dBm. The device name is set to "wicentric app".

The scan response data is set to the service UUID list. This contains a list of services supported by the device. The list in this example contains the Link Loss Service, Immediate Alert Service, and TX Power Service.

More information on AD types is in the Bluetooth 4.0 specification Volume 3, Part C, Chapter 11.

4.3 ATT Client Discovery Data

The ATT client discovery data is used for service discovery and to manage the client handle list containing the handles of discovered characteristics and attributes.

The handle list is an integer array defined by the sample application. Handles are set in the list by App Framework discovery functions used to find the characteristics and attributes of desired services on a peer device. For bonded peer devices, the handle list is stored in the device database so it can be restored on subsequent connections without performing discovery again.

In the following example, the ATT client discovery data is set up to discover the GATT Service and Immediate Alert Service (IAS).

```
/*! Discovery states:  enumeration of services to be discovered */
enum
{
  TAG_DISC_GATT_SVC,      /* GATT service */
  TAG_DISC_IAS_SVC,       /* Immediate Alert service */
  TAG_DISC_SVC_MAX        /* Discovery complete */
};

/*! the Client handle list, tagCb.hdlList[], is set as follows:
 *
 *  ------------------------------ <- TAG_DISC_GATT_START
 *  | GATT svc changed handle     |
 *  ------------------------------
 *  | GATT svc changed ccc handle |
 *  ------------------------------ <- TAG_DISC_IAS_START
 *  | IAS alert level handle      |
 *  ------------------------------
 */

/*! Start of each service's handles in the the handle list */
#define TAG_DISC_GATT_START       0
#define TAG_DISC_IAS_START        (TAG_DISC_GATT_START + GATT_HDL_LIST_LEN)
#define TAG_DISC_HDL_LIST_LEN     (TAG_DISC_IAS_START + FMPL_IAS_HDL_LIST_LEN)

/*! Pointers into handle list for each service's handles */
static uint16_t *pTagGattHdlList = &tagCb.hdlList[TAG_DISC_GATT_START];
static uint16_t *pTagIasHdlList = &tagCb.hdlList[TAG_DISC_IAS_START];
```

The discovery state enumeration is a list of the services to be discovered. These values are used in the App Framework discovery callback (see Section 4.9).

Then some constants and pointers are defined for accessing the handle list, as illustrated in the figure below.

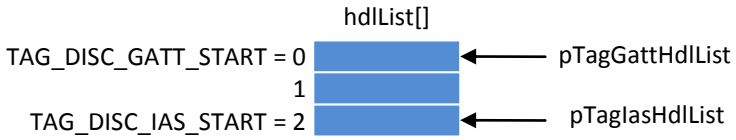

Figure 2. Example ATT client handle list and associated data.

In this example the handle list stores three handles: Two GATT handles and one IAS handle. Constants `TAG_DISC_GATT_START` and `TAG_DISC_IAS_START` are set to the start index of the handles for their respective services in the handle list. The pointers `pTagGattHdlList` and `pTagIasHdlList` point to the start of the handles for their respective services in the handle list. These pointers are used by the profile service discovery functions (e.g. `GattDiscover()` and `FmplIasDiscover()`) to access the handle list.

4.4 ATT Client Data

When service and characteristic discovery is complete, a profile typically requires that certain characteristics are read or written to configure the profile and the services it uses. For example, client characteristic configuration descriptors (CCCDs) are typically written to enable indications or notifications for their respective characteristics.

The ATT client data consists of constants and data structures used to configure a list of discovered characteristics. The data is used with the `AppDiscConfigure()` function of the App Framework API.

The data structure of type `attcDiscCfg_t` contains of a list of characteristics to read or write. Each entry in the list contains a value (if it is to be written), the value length, and the handle index of the discovered attribute or characteristic. An example is shown below:

```
/* Default value for GATT ccc descriptor */
static const uint8_t tagGattScCccVal[] =
  {UINT16_TO_BYTES(ATT_CLIENT_CFG_INDICATE)};

/* List of characteristics to configure */
static const attcDiscCfg_t tagDiscCfgList[] =
{
  /* Write:  GATT service changed ccc descriptor */
  {tagGattScCccVal, sizeof(tagGattScCccVal),
    (GATT_SC_CCC_HDL_IDX + TAG_DISC_GATT_START)}
};

/* Characteristic configuration list length */
#define TAG_DISC_CFG_LIST_LEN   (sizeof(tagDiscCfgList) / sizeof(attcDiscCfg_t))
```

In this example the characteristic list has a single entry that contains data used to write the CCCD of the GATT "service changed" characteristic. The value to be written will enable indications. (Note that the value is formatted as a little-endian byte array.) The handle index is set to `(GATT_SC_CCC_HDL_IDX + TAG_DISC_GATT_START)`. The value `GATT_SC_CCC_HDL_IDX` is the handle index of the CCCD

discovered by the GATT profile (see gatt_api.h). The value `TAG_DISC_GATT_START` is the start index of the GATT portion of the applications handle list, as described in Section 4.3.

4.5 ATT Server Data

The ATT server data contains constants and data structures defining the client characteristic configuration descriptors (CCCDs) used in the services supported by the device in its own server. The data is used by the ATT server CCCD management service. The data consists of an enumeration of each CCCD in the ATT server and a table of settings for each CCCD. An example is shown below:

```
/*! enumeration of client characteristic configuration descriptors used in local
ATT server */
enum
{
  TAG_GATT_SC_CCC_IDX,        /*! GATT service, service changed characteristic */
  TAG_NUM_CCC_IDX             /*! Number of ccc's */
};

/*! client characteristic configuration descriptors settings, indexed by ccc
enumeration */
static const attsCccSet_t tagCccSet[TAG_NUM_CCC_IDX] =
{
  /* cccd handle          value range                 security level */
  {GATT_SC_CH_CCC_HDL,   ATT_CLIENT_CFG_INDICATE,    DM_SEC_LEVEL_ENC}
};
```

In this example the ATT server database contains a single CCCD for the GATT "service changed" characteristic. The table of CCCD settings has a single entry, containing the handle of the CCCD, the value range, and the security level required for an indication or notification to be sent for the characteristic value associated with the CCCD. In this example the CCCD supports indications, and encryption is required before an indication can be sent.

4.6 Protocol Stack Callbacks

The protocol stack callbacks interface the sample application to the exactLE protocol stack.

4.6.1 DM Callback

The DM callback function is executed when the stack has a device management event to send to the application. The function simply copies the callback event parameters to a message and sends the message to the sample application event handler.

4.6.2 ATT Callback

The ATT callback function is executed when the ATT protocol client or server has an event to send to the application. The function simply copies the callback event parameters to a message and sends the message to the sample application event handler.

4.6.3 ATT CCC Callback

The ATT CCC callback function is executed when a peer device writes a new value to a client characteristic configuration descriptor in the ATT server. It is also executed on connection establishment if the CCCD is initialized with a stored value from a previous connection.

The function first checks if this new CCCD value should be stored in the device database. If so, the value is stored. Then it sends a message to the sample application event handler with the CCCD value.

4.7 Event Handler Action Functions

A sample application defines event handler actions functions when a particular event, such as connection open or close, requires specific actions in the application. The following functions are examples from the "tag" sample application.

4.7.1 tagClose

This function performs an alert when the connection is closed.

4.7.2 tagSetup

This function is executed when the application is started after the stack is reset. It sets up the advertising and scan response data, and then starts advertising:

```
/* set advertising and scan response data for discoverable mode */
AppAdvSetData(APP_ADV_DATA_DISCOVERABLE, sizeof(tagAdvDataDisc),
              (uint8_t *) tagAdvDataDisc);
AppAdvSetData(APP_SCAN_DATA_DISCOVERABLE, sizeof(tagScanDataDisc),
              (uint8_t *) tagScanDataDisc);

/* set advertising and scan response data for connectable mode */
AppAdvSetData(APP_ADV_DATA_CONNECTABLE, 0, NULL);
AppAdvSetData(APP_SCAN_DATA_CONNECTABLE, 0, NULL);

/* start advertising; automatically set connectable/discoverable mode and
   bondable mode */
AppAdvStart(APP_MODE_AUTO_INIT);
```

Note that the advertising data is set to the constants described earlier in Section 4.1.4. Also note that the advertising data is set differently for discoverable mode and connectable mode, and that the advertising data is set to empty in connectable mode.

The device starts advertising by calling function `AppAdvStart()`. By using "auto init" mode, the connectable/discoverable and bondable mode of the device is set automatically based on whether the device has already bonded. If it has not bonded the device is set to discoverable and bondable mode. If it has bonded the device is set to connectable and non-bondable mode.

4.8 Button Handler Callback

The button handler callback function is part of the App Framework's user interface service. It is executed by the App Framework when a button press occurs. The button press value identifies the pressed button and the duration of the button press (short, medium, or long).

This function performs an action on a button press event specific to the sample application. The application will typically perform different actions when connected vs. not connected. For example in the "tag" application, an immediate alert is sent when a short button press occurs while connected.

4.9 Discovery Callback

This is the callback function for the App Framework discovery API. The App Framework provides a set of discovery APIs that simplify service and characteristic discovery as well as service configuration. The App Framework executes the callback at appropriate times to trigger the application to perform a discovery-related action. The status parameter to the function indicates the action to perform, or the status result of a completed action.

The status values and the associated action typically performed by the callback function are as follows:

- **APP_DISC_INIT:** This status value is used when the connection is opened. The function must call `AppDiscSetHdlList()` and pass in a memory buffer for the App Framework to store the handle list.
- **APP_DISC_START:** This status value is used when discovery is started. The function should initiate service discovery for the first service to be discovered, for example call `GattDiscover()`.
- **APP_DISC_CMPL and APP_DISC_FAILED:** These status values are used when the previously-initiated discovery procedure is complete. If there are more services to discover initiate discovery for the next service. Otherwise, call `AppDiscComplete(APP_DISC_CMPL)` to notify the App Framework that all discovery procedures are complete. If there is a configuration procedure to perform initiate the configuration procedure by calling `AppDiscConfigure()` using the ATT client data as described in Section 4.4.
- **APP_DISC_CFG_START:** This status value is used to start a configuration procedure. This status value is used when all discovery procedures are complete but configuration is not complete. If there is a configuration procedure to perform initiate the procedure. Otherwise, call `AppDiscComplete(APP_DISC_CFG_CMPL)` to notify the App Framework that all discovery procedures are complete.
- **APP_DISC_CFG_CONN_START:** This status value is used to start a connection setup configuration procedure. This can be used when an application needs to read or write certain characteristics of the peer device every time a connection is established. If applicable, call `AppDiscConfigure()` to perform the configuration procedure.
- **APP_DISC_CFG_CMPL:** This function is called when a configuration procedure is complete. Call `AppDiscComplete(APP_DISC_CFG_CMPL)` to notify the App Framework that all discovery procedures are complete.

4.10 Event Handler Processing Function

This function decodes received DM or ATT events and then executes an action function to perform an application-specific procedure. The sample application code also demonstrates how DM events can be mapped to UI events that are then passed to `AppUiAction()` to perform a platform-specific UI action, for example blink an LED when a connection is established.

4.11 Application Initialization Function

The application initialization function is executed on system startup when the WSF event handlers for the system are initialized. This function initializes App Framework configuration pointers and initializes any used App Framework components that require initialization.

4.12 Application Event Handler Function

This is the application's WSF event handler. It is executed by the WSF OS. Received messages are passed to the appropriate App Framework components and then are passed to the application's event handler processing function.

4.13 Application Start Function

Finally, this is the function that ties everything together for the application. This function is executed on system startup after WSF event handlers have been initialized. The function registers the application's protocol stack callback functions and App Framework callback functions. It then initializes the services used in the local ATT server database. Finally, function `DmDevReset()` is called to reset the stack and Bluetooth LE controller and then trigger the start of the application.

5 References

1. Wicentric, "App Framework API", 2011-0020.
2. Wicentric, "Profile and Service API", 2012-0021.

6 Definitions

AD	Advertising Data
ATT	Attribute protocol software subsystem
CCC or CCCD	Client Characteristic Configuration Descriptor
DM	Device Manager software subsystem
GATT	Generic Attribute Profile
HCI	Host Controller Interface
LE	(Bluetooth) Low Energy
LTK	Long Term Key
WSF	Wicentric Software Foundation software service and porting layer