Getting started with BLE in WICED Studio

1.0 Purpose

The purpose of this document is to help users who are already familiar with the PSoC Creator BLE design flow to quickly get started with their BLE designs in WICED Studio. This post tries to explain the structure of a WICED BLE application and in the end explains how a user can quickly get started with an application. This blog will be using the *hello_sensor* application *20706-A2 Bluetooth* SDK as an example. However, the same holds good for other WICED Bluetooth chips as well.

Note that this blog post does not aim to explain Bluetooth functionalities, rather assumes that the reader is already aware of BLE basics.

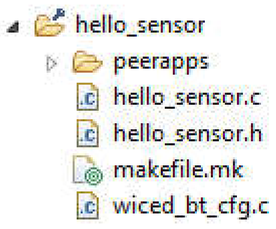
2.0 Content

- The application files
- Common sections in a BLE application
- WICED Bluetooth Designer
- Illustrating the design flow

2.1 The application files

All example applications are located in the *Apps* folder in the SDK. The *hello_sensor* example app can be found here as well. All Bluetooth applications generally contain the following files,

- Source and header files (hello_sensor.c and hello_sensor.h)
- The Bluetooth configuration file (wiced_bt_cfg.c)
- The make file. (makefile.mk)



```
hello_sensor
    peerapps
    hello_sensor.c
    hello_sensor.h
    makefile.mk
    wiced_bt_cfg.c
```

2.1.1 Hello_sensor.c

This file contains the main user application code. More details about this file in section 2.2.

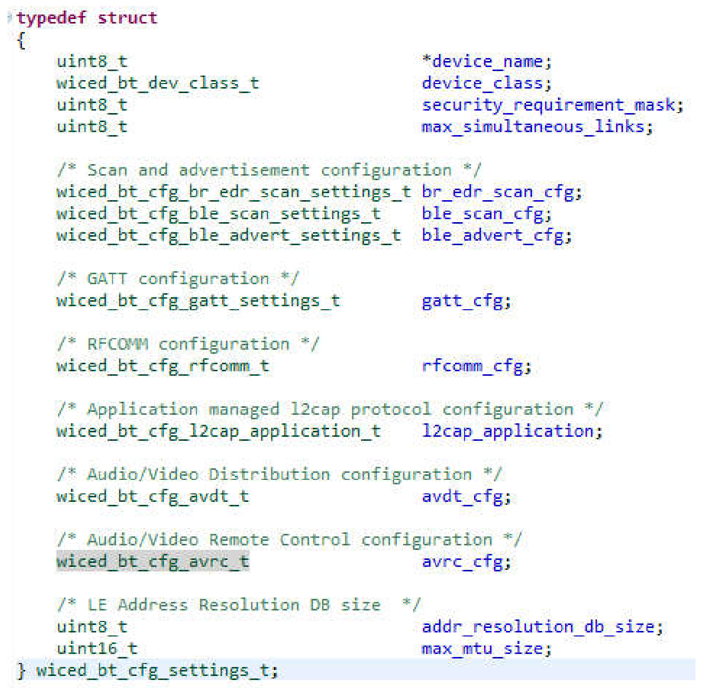
2.1.2 MAKEFILE.MK

All applications in WICED will have an associated makefile that can be modified by the user. This file is generally used to enable/disable certain features of the code during the build process. For example, you can enable or disable BT HCI or Debug trace capabilities by adding or removing the following flags,

```
C_FLAGS += -DWICED_BT_TRACE_ENABLE
#C_FLAGS += -DENABLE_HCI_TRACE
```

2.1.3 WICED_BT_CFG.C

This file is used for configuring all of the Bluetooth parameters. This file is analogous to the BLE component configuration window in PSoC Creator. The configuration parameters are initialized inside the structure *wiced_bt_cfg_settings_t* which is later passed as a parameter during Bluetooth stack initialization (Section 2.2.1). A brief description of all parameters are available as comments in the SDK.



```
typedef struct
{
    uint8_t                             *device_name;
    wiced_bt_dev_class_t                device_class;
    uint8_t                             security_requirement_mask;
    uint8_t                             max_simultaneous_links;

    /* Scan and advertisement configuration */
    wiced_bt_cfg_br_edr_scan_settings_t br_edr_scan_cfg;
    wiced_bt_cfg_ble_scan_settings_t    ble_scan_cfg;
    wiced_bt_cfg_ble_advert_settings_t  ble_advert_cfg;

    /* GATT configuration */
    wiced_bt_cfg_gatt_settings_t        gatt_cfg;

    /* RFCOMM configuration */
    wiced_bt_cfg_rfcomm_t               rfcomm_cfg;

    /* Application managed l2cap protocol configuration */
    wiced_bt_cfg_l2cap_application_t    l2cap_application;

    /* Audio/Video Distribution configuration */
    wiced_bt_cfg_avdt_t                 avdt_cfg;

    /* Audio/Video Remote Control configuration */
    wiced_bt_cfg_avrc_t                 avrc_cfg;

    /* LE Address Resolution DB size  */
    uint8_t                             addr_resolution_db_size;
    uint16_t                            max_mtu_size;
} wiced_bt_cfg_settings_t;
```

2.2 COMMON SECTIONS IN A BLE APPLICATION

All BLE applications comprise of some mandatory common sections. These sections are explained below with the help of the *hello_sensor* app.

- APPLICATION_START( )
- Bluetooth Management Callback
- GATT callback
- GATT database

2.2.1 APPLICATION_START( )

This function marks the beginning of the code execution for all BLE applications. The section is most commonly used to initialize the BT stack and to set up the WICED transport.

wiced_bt_stack_init is used to initialize the BT stack and also register the Bluetooth management callback (Section 2.2.2). To this function we also pass the Bluetooth configuration parameters (Section 2.1.3) and buffer pool configuration.

For more information on buffer pools, refer AN216403 – WICED- Application Buffer Pools in the DOC folder of WICED Studio.

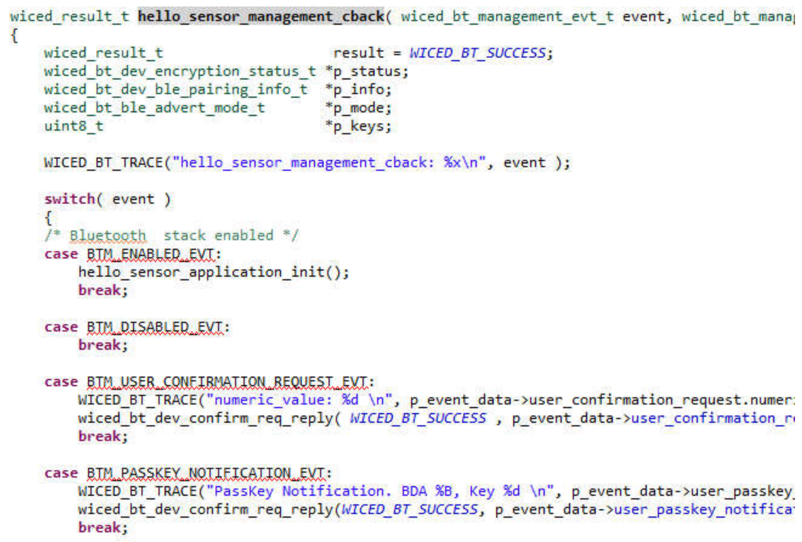
```
// Register call back and configuration with stack
wiced_bt_stack_init( hello_sensor_management_cback ,
                     &wiced_bt_cfg_settings, wiced_bt_cfg_buf_pools );
```

wiced_transport_init is used to configure the WICED transport, which is HCI_UART interface. In the hello_sensor application, the WICED transport is used to send out HCI traces when *C_FLAGS += -DENABLE_HCI_TRACE* is enabled in the makefile. The WICED transport can be configured as UART or SPI. Once configured, the WICED transport can be used to receive WICED HCI commands.

```
wiced_transport_init( &transport_cfg );
```

2.2.2 BLUETOOTH MANAGEMENT CALLBACK

This is the main callback function that return all major events for managing the Bluetooth link from the application. This callback function is registered in the *wiced_bt_stack_init* function. In the *hello_sensor* application we have registered *hello_sensor_management_cback* for management callbacks.



```
wiced_result_t hello_sensor_management_cback( wiced_bt_management_evt_t event, wiced_bt_mana
{
    wiced_result_t                    result = WICED_BT_SUCCESS;
    wiced_bt_dev_encryption_status_t *p_status;
    wiced_bt_dev_ble_pairing_info_t  *p_info;
    wiced_bt_ble_advert_mode_t       *p_mode;
    uint8_t                          *p_keys;

    WICED_BT_TRACE("hello_sensor_management_cback: %x\n", event );

    switch( event )
    {
    /* Bluetooth  stack enabled */
    case BTM_ENABLED_EVT:
        hello_sensor_application_init();
        break;

    case BTM_DISABLED_EVT:
        break;

    case BTM_USER_CONFIRMATION_REQUEST_EVT:
        WICED_BT_TRACE("numeric_value: %d \n", p_event_data->user_confirmation_request.numer
        wiced_bt_dev_confirm_req_reply( WICED_BT_SUCCESS , p_event_data->user_confirmation_r
        break;

    case BTM_PASSKEY_NOTIFICATION_EVT:
        WICED_BT_TRACE("PassKey Notification. BDA %B, Key %d \n", p_event_data->user_passkey_
        wiced_bt_dev_confirm_req_reply(WICED_BT_SUCCESS, p_event_data->user_passkey_notifica
        break;
```

Similar to the BLE callback function in PSoC Creator, this callback return events and event parameters associated with Bluetooth events such as Security, Advertisement, Scanning etc. All BT events except the

GATT events are handled here. GATT events have a separate callback (Section 2.2.3). All the available management events can be found in *wiced_bt_management_evt_t.*

It is worth mentioning that *BTM_ENABLED_EVT* is the first management event that is triggered after stack initialization. In *hello_sensor*, the rest of the application is initialized after this event. *BTM_ENABLED_EVT* is equivalent to CYBLE_EVT_STACK_ON event in PSoC Creator.

2.2.3 GATT Callback

This callback returns all GATT related events and event parameters that need to be handled by the application after connection has been established. The GATT callback is registered using *wiced_bt_gatt_register()* function. In the *hello_sensor* app, the GATT callback is registered after the BT stack initialization.

```
/* Register with stack to receive GATT callback */
gatt_status = wiced_bt_gatt_register( hello_sensor_gatts_callback );
```

All GATT events are available in *wiced_bt_gatt_evt_t.* These events inform the state of the connection and GATT database operation.

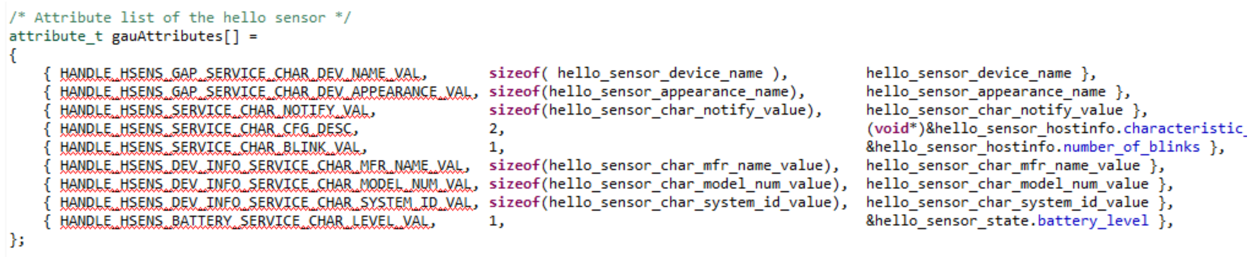
2.2.4 GATT Database

All BLE server applications must include a GATT database that is registered with the stack. The GATT database is custom made for each application and need to follow the correct format as defined by the spec. In the *hello_sensor* application, a custom service and a SIG defined device information service is defined in the GATT database. After defining the required GATT database, it is registered with the stack using the wiced_bt_gatt_db_init() API.

```
const uint8_t hello_sensor_gatt_database[]=
{
    // Declare mandatory GATT service
    PRIMARY_SERVICE_UUID16( HANDLE_HSENS_GATT_SERVICE, UUID_SERVICE_GATT ),

    // Declare mandatory GAP service. Device Name and Appearance are mandatory
    // characteristics of GAP service
    PRIMARY_SERVICE_UUID16( HANDLE_HSENS_GAP_SERVICE, UUID_SERVICE_GAP ),

        // Declare mandatory GAP service characteristic: Dev Name
        CHARACTERISTIC_UUID16( HANDLE_HSENS_GAP_SERVICE_CHAR_DEV_NAME, HANDLE_HSENS_GAP_SERVICE_CHAR_DEV_NAME_VAL,
            UUID_CHARACTERISTIC_DEVICE_NAME, LEGATTDB_CHAR_PROP_READ, LEGATTDB_PERM_READABLE ),

        // Declare mandatory GAP service characteristic: Appearance
        CHARACTERISTIC_UUID16( HANDLE_HSENS_GAP_SERVICE_CHAR_DEV_APPEARANCE, HANDLE_HSENS_GAP_SERVICE_CHAR_DEV_APPEARANCE_VAL,
            UUID_CHARACTERISTIC_APPEARANCE, LEGATTDB_CHAR_PROP_READ, LEGATTDB_PERM_READABLE ),

    // Declare proprietary Hello Service with 128 byte UUID
    PRIMARY_SERVICE_UUID128( HANDLE_HSENS_SERVICE, UUID_HELLO_SERVICE ),

        // Declare characteristic used to notify/indicate change
        CHARACTERISTIC_UUID128( HANDLE_HSENS_SERVICE_CHAR_NOTIFY, HANDLE_HSENS_SERVICE_CHAR_NOTIFY_VAL,
            UUID_HELLO_CHARACTERISTIC_NOTIFY, LEGATTDB_CHAR_PROP_READ | LEGATTDB_CHAR_PROP_NOTIFY | LEGATTDB_CHAR_PROP_INDICATE, LEGATTDB_PERM_READABLE ),

            // Declare client characteristic configuration descriptor
            // Value of the descriptor can be modified by the client
            // Value modified shall be retained during connection and across connection
            // for bonded devices.  Setting value to 1 tells this application to send notification
            // when value of the characteristic changes.  Value 2 is to allow indications.
            CHAR_DESCRIPTOR_UUID16_WRITABLE( HANDLE_HSENS_SERVICE_CHAR_CFG_DESC, UUID_DESCRIPTOR_CLIENT_CHARACTERISTIC_CONFIGURATION,
            LEGATTDB_PERM_READABLE | LEGATTDB_PERM_WRITE_REQ | LEGATTDB_PERM_AUTH_READABLE | LEGATTDB_PERM_AUTH_WRITABLE),
```

Manually writing a GATT database from scratch can be a tedious and error prone process. WICED Studio provides '*WICED Bluetooth Designer'* to simplify this process.

The actual values of the GATT database are stored in a separate attribute array along with their associated characteristic handles as shown in the figure below.



```
/* Attribute list of the hello sensor */
attribute_t gauAttributes[] =
{
    { HANDLE_HSENS_GAP_SERVICE_CHAR_DEV_NAME_VAL,       sizeof( hello_sensor_device_name ),         hello_sensor_device_name },
    { HANDLE_HSENS_GAP_SERVICE_CHAR_DEV_APPEARANCE_VAL, sizeof(hello_sensor_appearance_name),       hello_sensor_appearance_name },
    { HANDLE_HSENS_SERVICE_CHAR_NOTIFY_VAL,             sizeof(hello_sensor_char_notify_value),     hello_sensor_char_notify_value },
    { HANDLE_HSENS_SERVICE_CHAR_CFG_DESC,               2,                                          (void*)&hello_sensor_hostinfo.characteristic_
    { HANDLE_HSENS_SERVICE_CHAR_BLINK_VAL,              1,                                          &hello_sensor_hostinfo.number_of_blinks },
    { HANDLE_HSENS_DEV_INFO_SERVICE_CHAR_MFR_NAME_VAL,  sizeof(hello_sensor_char_mfr_name_value),   hello_sensor_char_mfr_name_value },
    { HANDLE_HSENS_DEV_INFO_SERVICE_CHAR_MODEL_NUM_VAL, sizeof(hello_sensor_char_model_num_value),  hello_sensor_char_model_num_value },
    { HANDLE_HSENS_DEV_INFO_SERVICE_CHAR_SYSTEM_ID_VAL, sizeof(hello_sensor_char_system_id_value),  hello_sensor_char_system_id_value },
    { HANDLE_HSENS_BATTERY_SERVICE_CHAR_LEVEL_VAL,      1,                                          &hello_sensor_state.battery_level },
};
```

2.3 WICED Bluetooth Designer

The BT Designer feature of the WICED BT SDK IDE helps software developers generate the Bluetooth Generic Attribute Profile (GATT) databases and initial code for Bluetooth WICED applications.

The BT Designer is initiated by clicking *File > New > WICED Bluetooth Designer* in the WICED Studio IDE.

Complete information about this tool can be found in the file *'Developing Custom Applications with BT Designer'* under the doc folder of WICED Studio.

2.4 Creating your own peripheral in WICED

The following section uses the above discussed information to create a heart rate sensor peripheral application from the existing *hello_sensor* code. The following section also draws parallels between how the heart rate sensor is implemented in Creator and how the same is done in WICED. The final example code *hr_sensor* is attached along with this blog post so that users can go through it and see the differences between hello_sensor and hr_sensor.

Note:

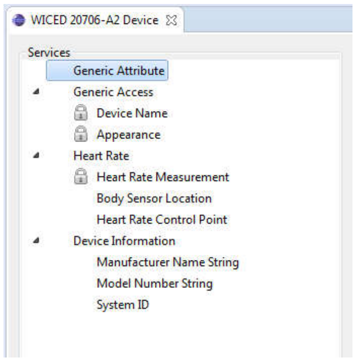
- This section just illustrates the simplest way to quickly get started with WICED Studio. Advanced users can simply opt to write the application code from scratch.
- Basic understanding of the hello_sensor application is required. The following section assumes that the user has gone through and understood the *hello_sensor* code using the information above.

The heart rate peripheral code can be classified into 3 major categories,

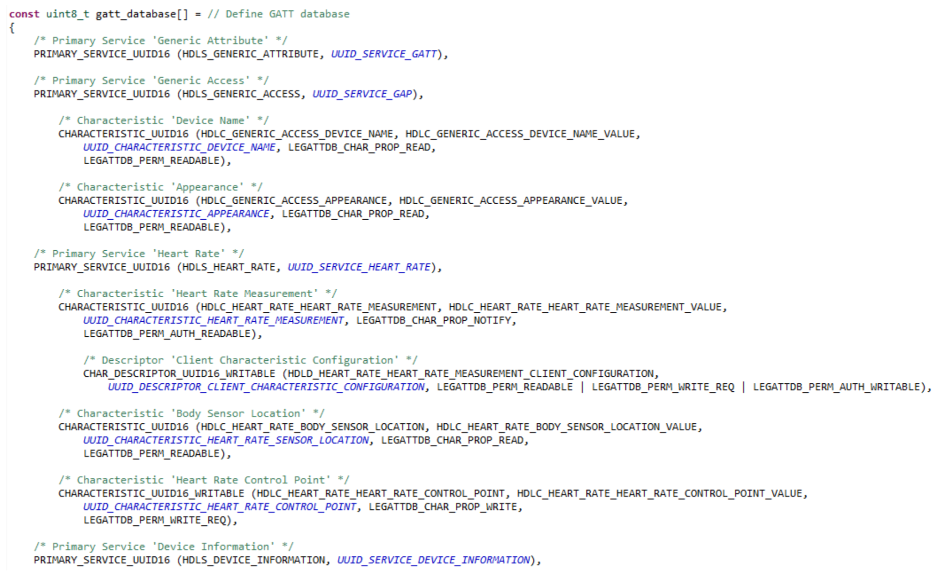
- GATT Database
- BLE Configuration
    - Connection and Advertisement Parameters
    - Security Parameters
- Application Activity

2.4.1 GATT DATABASE

A GATT database for the heart rate profile needs to be generated first. In PSoC Creator, this was done using the BLE Component window under the Profiles tab. In WICED Studio, we will use *WICED Bluetooth Designer* to generate our required GATT database. In the *hr_sensor* example project, we use two SIG defined services, *heart rate* and *device information*.

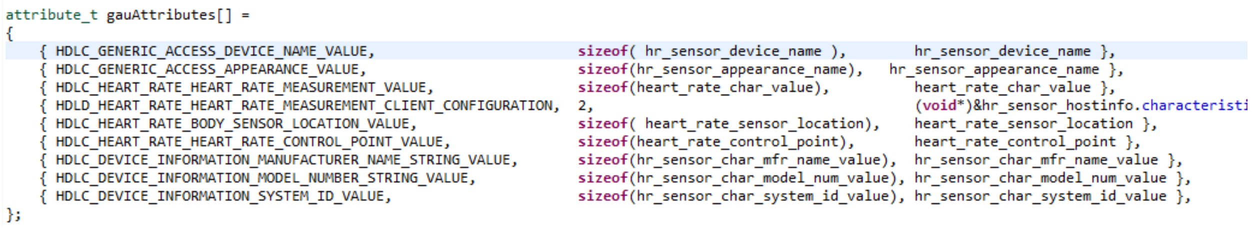


Once the required code is generated, copy the GATT database and its associated header definitions from the generated code into the hello_sensor application, replacing the existing hello_sensor GATT database.



```
const uint8_t gatt_database[] = // Define GATT database
{
    /* Primary Service 'Generic Attribute' */
    PRIMARY_SERVICE_UUID16 (HDLS_GENERIC_ATTRIBUTE, UUID_SERVICE_GATT),

    /* Primary Service 'Generic Access' */
    PRIMARY_SERVICE_UUID16 (HDLS_GENERIC_ACCESS, UUID_SERVICE_GAP),

        /* Characteristic 'Device Name' */
        CHARACTERISTIC_UUID16 (HDLC_GENERIC_ACCESS_DEVICE_NAME, HDLC_GENERIC_ACCESS_DEVICE_NAME_VALUE,
            UUID_CHARACTERISTIC_DEVICE_NAME, LEGATTDB_CHAR_PROP_READ,
            LEGATTDB_PERM_READABLE),

        /* Characteristic 'Appearance' */
        CHARACTERISTIC_UUID16 (HDLC_GENERIC_ACCESS_APPEARANCE, HDLC_GENERIC_ACCESS_APPEARANCE_VALUE,
            UUID_CHARACTERISTIC_APPEARANCE, LEGATTDB_CHAR_PROP_READ,
            LEGATTDB_PERM_READABLE),

    /* Primary Service 'Heart Rate' */
    PRIMARY_SERVICE_UUID16 (HDLS_HEART_RATE, UUID_SERVICE_HEART_RATE),

        /* Characteristic 'Heart Rate Measurement' */
        CHARACTERISTIC_UUID16 (HDLC_HEART_RATE_HEART_RATE_MEASUREMENT, HDLC_HEART_RATE_HEART_RATE_MEASUREMENT_VALUE,
            UUID_CHARACTERISTIC_HEART_RATE_MEASUREMENT, LEGATTDB_CHAR_PROP_NOTIFY,
            LEGATTDB_PERM_AUTH_READABLE),

            /* Descriptor 'Client Characteristic Configuration' */
            CHAR_DESCRIPTOR_UUID16_WRITABLE (HDLD_HEART_RATE_HEART_RATE_MEASUREMENT_CLIENT_CONFIGURATION,
                UUID_DESCRIPTOR_CLIENT_CHARACTERISTIC_CONFIGURATION, LEGATTDB_PERM_READABLE | LEGATTDB_PERM_WRITE_REQ | LEGATTDB_PERM_AUTH_WRITABLE),

        /* Characteristic 'Body Sensor Location' */
        CHARACTERISTIC_UUID16 (HDLC_HEART_RATE_BODY_SENSOR_LOCATION, HDLC_HEART_RATE_BODY_SENSOR_LOCATION_VALUE,
            UUID_CHARACTERISTIC_HEART_RATE_SENSOR_LOCATION, LEGATTDB_CHAR_PROP_READ,
            LEGATTDB_PERM_READABLE),

        /* Characteristic 'Heart Rate Control Point' */
        CHARACTERISTIC_UUID16_WRITABLE (HDLC_HEART_RATE_HEART_RATE_CONTROL_POINT, HDLC_HEART_RATE_HEART_RATE_CONTROL_POINT_VALUE,
            UUID_CHARACTERISTIC_HEART_RATE_CONTROL_POINT, LEGATTDB_CHAR_PROP_WRITE,
            LEGATTDB_PERM_WRITE_REQ),

    /* Primary Service 'Device Information' */
    PRIMARY_SERVICE_UUID16 (HDLS_DEVICE_INFORMATION, UUID_SERVICE_DEVICE_INFORMATION),
```

Modify the GATT attribute array in the application to match the heart rate sensor's GATT database as shown in the figure below,



```
attribute_t gauAttributes[] =
{
    { HDLC_GENERIC_ACCESS_DEVICE_NAME_VALUE,                        sizeof( hr_sensor_device_name ),       hr_sensor_device_name },
    { HDLC_GENERIC_ACCESS_APPEARANCE_VALUE,                         sizeof(hr_sensor_appearance_name),   hr_sensor_appearance_name },
    { HDLC_HEART_RATE_HEART_RATE_MEASUREMENT_VALUE,                 sizeof(heart_rate_char_value),         heart_rate_char_value },
    { HDLD_HEART_RATE_HEART_RATE_MEASUREMENT_CLIENT_CONFIGURATION,  2,                                     (void*)&hr_sensor_hostinfo.characteristi
    { HDLC_HEART_RATE_BODY_SENSOR_LOCATION_VALUE,                   sizeof( heart_rate_sensor_location),   heart_rate_sensor_location },
    { HDLC_HEART_RATE_HEART_RATE_CONTROL_POINT_VALUE,               sizeof(heart_rate_control_point),      heart_rate_control_point },
    { HDLC_DEVICE_INFORMATION_MANUFACTURER_NAME_STRING_VALUE,       sizeof(hr_sensor_char_mfr_name_value), hr_sensor_char_mfr_name_value },
    { HDLC_DEVICE_INFORMATION_MODEL_NUMBER_STRING_VALUE,            sizeof(hr_sensor_char_model_num_value), hr_sensor_char_model_num_value },
    { HDLC_DEVICE_INFORMATION_SYSTEM_ID_VALUE,                      sizeof(hr_sensor_char_system_id_value), hr_sensor_char_system_id_value },
};
```

2.4.2 BLE Configuration

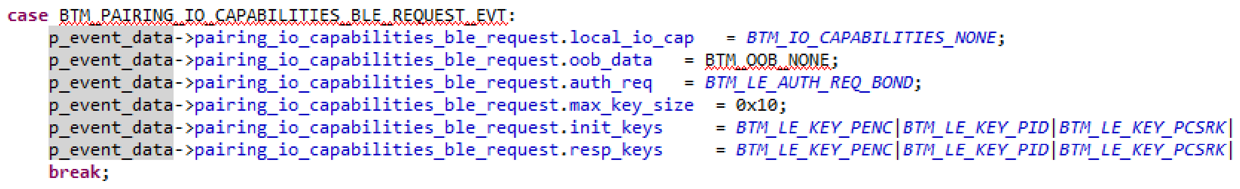
Advertisement Parameters:

As discussed in section 2.1.3, you can set the advertisement parameters in the *wiced_bt_cfg.c* file. The advertisement interval and duration can be set in *ble_advert_cfg.* Advertisement can then be started in the application by calling the *wiced_bt_start_advertisements* API.

Note: The list of APIs available can found inside the doc folder in the Project Explorer or WICED Studio.

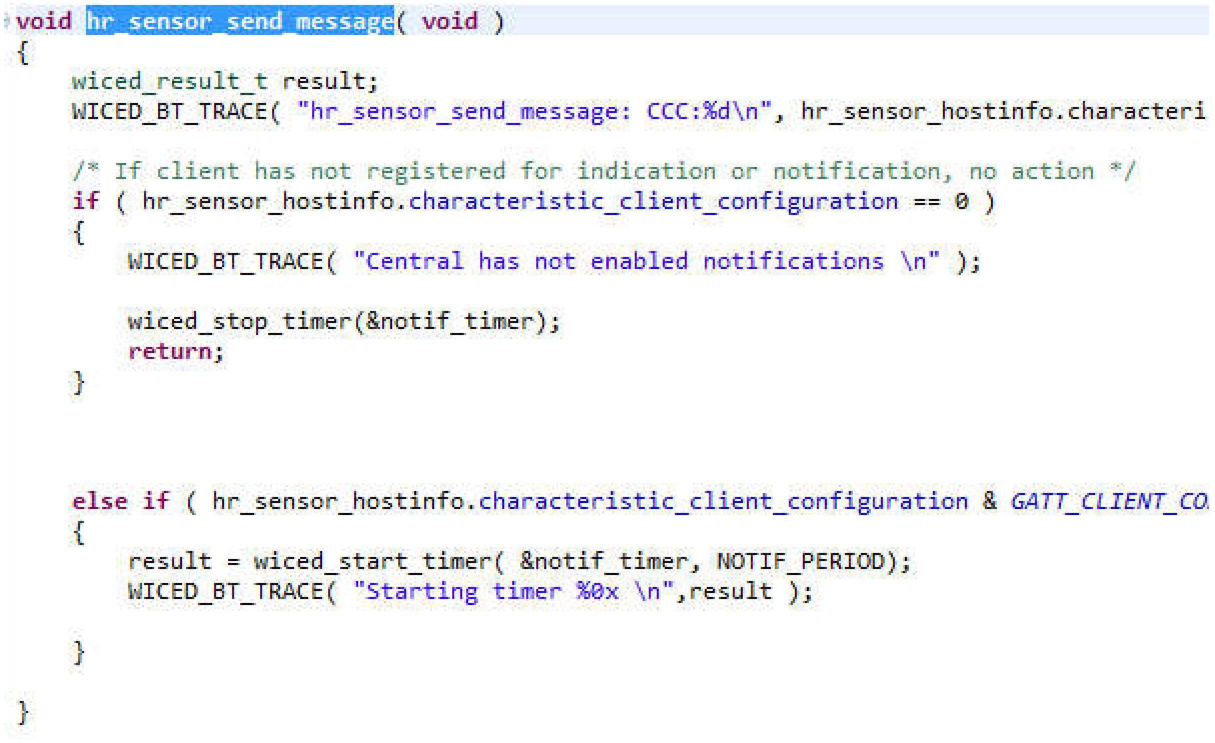
Security Parameters:

In PSoC Creator, security parameters are located in the BLE configuration dialog box, under the Security tab. In WICED, the security parameters are set from the application at runtime during the pairing procedure. Whenever a pairing process in initiated, the application receives the *BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT* management callback event along with the event parameter *wiced_bt_dev_ble_io_caps_req_t*. Here the application is expected the fill the *wiced_bt_dev_ble_io_caps_req_t* to set the security parameters such as Authentication mode, IO Capabilities, Max key size etc.

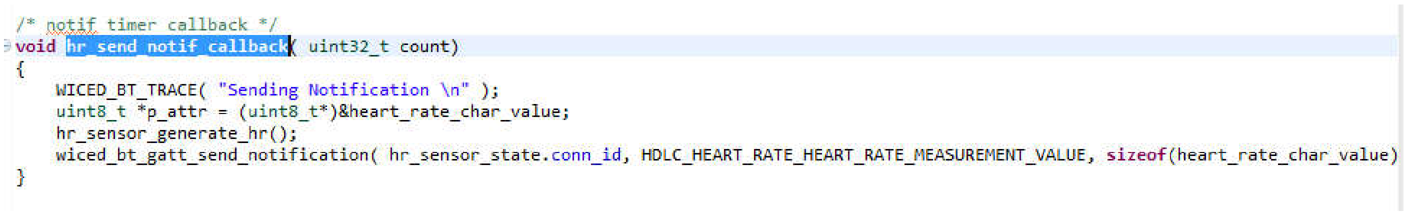

```
case BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT:
    p_event_data->pairing_io_capabilities_ble_request.local_io_cap    = BTM_IO_CAPABILITIES_NONE;
    p_event_data->pairing_io_capabilities_ble_request.oob_data     = BTM_OOB_NONE;
    p_event_data->pairing_io_capabilities_ble_request.auth_req     = BTM_LE_AUTH_REQ_BOND;
    p_event_data->pairing_io_capabilities_ble_request.max_key_size  = 0x10;
    p_event_data->pairing_io_capabilities_ble_request.init_keys     = BTM_LE_KEY_PENC|BTM_LE_KEY_PID|BTM_LE_KEY_PCSRK|
    p_event_data->pairing_io_capabilities_ble_request.resp_keys     = BTM_LE_KEY_PENC|BTM_LE_KEY_PID|BTM_LE_KEY_PCSRK|
    break;
```

2.4.3 Application activity

The application activity is nothing but sending of notifications once the client connects and enables notifications. PSoC Creator does not support an RTOS by default. So the entire application code runs in an infinite while loop where the send notifications API is called periodically. However, WICED supports an RTOS and the main application runs in a thread. Here we send out notifications based on a timer expiry. In hr_sensor, the function *hr_sensor_send_message()* takes care of starting the timer and sending notifications on notifications are enabled by the client.


```
void hr_sensor_send_message( void )
{
    wiced_result_t result;
    WICED_BT_TRACE( "hr_sensor_send_message: CCC:%d\n", hr_sensor_hostinfo.characteri

    /* If client has not registered for indication or notification, no action */
    if ( hr_sensor_hostinfo.characteristic_client_configuration == 0 )
    {
        WICED_BT_TRACE( "Central has not enabled notifications \n" );

        wiced_stop_timer(&notif_timer);
        return;
    }


    else if ( hr_sensor_hostinfo.characteristic_client_configuration & GATT_CLIENT_CO
    {
        result = wiced_start_timer( &notif_timer, NOTIF_PERIOD);
        WICED_BT_TRACE( "Starting timer %0x \n",result );

    }

}
```

Following is the image of the timer callback where the notifications are actually sent. Refer the HRS manual for more



```
/* notif timer callback */
void hr_send_notif_callback( uint32_t count)
{
    WICED_BT_TRACE( "Sending Notification \n" );
    uint8_t *p_attr = (uint8_t*)&heart_rate_char_value;
    hr_sensor_generate_hr();
    wiced_bt_gatt_send_notification( hr_sensor_state.conn_id, HDLC_HEART_RATE_HEART_RATE_MEASUREMENT_VALUE, sizeof(heart_rate_char_value)
}
```