

Chapter 8: Bluetooth SDK & Reference Flow

After completing this chapter, you will understand a top-level view of the Bluetooth SDK and how to use it with ModusToolbox.

| | | |
|-------------|--|-----------|
| 8.1 | INTRODUCTION | 3 |
| 8.2 | BLUETOOTH SDK ARCHITECTURE | 4 |
| 8.3 | STARTER APPLICATION SETS | 6 |
| 8.4 | APPLICATION ARCHITECTURE | 7 |
| 8.4.1 | PROJECT EXPLORER | 7 |
| 8.4.2 | README | 7 |
| 8.4.3 | APPLICATION CONFIGURATOR FILES | 7 |
| 8.4.4 | GENERATEDSOURCE DIRECTORY | 7 |
| 8.4.5 | LIBS | 8 |
| 8.4.6 | MAKEFILE | 8 |
| 8.4.7 | STACK CONFIGURATION FILES | 8 |
| 8.4.8 | APPLICATION C FILE | 8 |
| 8.5 | DEVICE CONFIGURATION | 11 |
| 8.6 | MANAGING LIBRARIES | 12 |
| 8.7 | RENAMING / COPYING / SHARING APPLICATIONS | 13 |
| 8.7.1 | IMPORT USING THE NEW APPLICATION WIZARD | 13 |
| 8.7.2 | RENAME | 13 |
| 8.7.3 | COPY/PASTE | 13 |
| 8.7.4 | ARCHIVE FILES | 14 |
| 8.8 | DOCUMENTATION | 15 |
| 8.9 | EXERCISES (PART 1) | 16 |
| 8.9.1 | TURN ON AN LED | 16 |
| 8.9.2 | ADD A TIMER TO GET THE LED TO BLINK AT 1Hz | 18 |
| 8.9.3 | USE AN RTOS THREAD TO BLINK THE LED | 18 |
| 8.10 | BLUETOOTH STACK EVENTS | 19 |
| 8.10.1 | ESSENTIAL BLUETOOTH MANAGEMENT EVENTS | 19 |
| 8.10.2 | ESSENTIAL GATT EVENTS | 20 |
| 8.10.3 | ESSENTIAL GATT SUB-EVENTS | 20 |
| 8.11 | FIRMWARE ARCHITECTURE | 21 |
| 8.11.1 | TURNING ON THE STACK | 21 |
| 8.11.2 | START ADVERTISING | 22 |
| 8.11.3 | PROCESSING CONNECTION EVENTS FROM THE STACK | 22 |
| 8.11.4 | PROCESSING CLIENT READ EVENTS FROM THE STACK | 23 |
| 8.11.5 | PROCESSING CLIENT WRITE EVENTS FROM THE STACK | 24 |
| 8.12 | SIMPLE PERIPHERAL DEMO WALKTHROUGH | 25 |
| 8.12.1 | RUNNING THE BLUETOOTH CONFIGURATOR | 26 |
| 8.12.2 | EDITING THE FIRMWARE | 29 |
| 8.12.3 | TESTING THE APPLICATION | 31 |
| 8.13 | WICED GATT DATABASE IMPLEMENTATION | 34 |
| 8.13.1 | GATT_DATABASE[] | 34 |
| 8.13.2 | GATT_DB_EXT_ATTR_TBL | 36 |
| 8.13.3 | UINT8_T ARRAYS FOR THE VALUES | 37 |
| 8.13.4 | APPLICATION PROGRAMMING INTERFACE | 37 |
| 8.14 | NOTIFICATIONS | 38 |
| 8.15 | NOTIFICATION DEMO WALKTHROUGH | 39 |

| | | |
|-------------|--|-----------|
| 8.15.1 | RUNNING THE BLUETOOTH CONFIGURATOR | 40 |
| 8.15.2 | EDITING THE FIRMWARE | 44 |
| 8.15.3 | TESTING THE APPLICATION | 46 |
| 8.16 | SECURITY | 48 |
| 8.16.1 | PAIRING..... | 48 |
| 8.16.2 | BONDING | 51 |
| 8.16.3 | PAIRING & BONDING PROCESS SUMMARY | 51 |
| 8.16.4 | AUTHENTICATION, AUTHORIZATION AND THE GATT DB..... | 51 |
| 8.16.5 | SECURITY IN THE BLUETOOTH CONFIGURATOR..... | 52 |
| 8.16.6 | LINK LAYER PRIVACY | 53 |
| 8.17 | FIRMWARE ARCHITECTURE FOR SECURITY | 54 |
| 8.18 | CYSMART | 59 |
| 8.18.1 | CYSMART PC APPLICATION | 59 |
| 8.18.2 | CYSMART MOBILE APPLICATION | 63 |
| 8.19 | EXERCISES (PART 2) | 64 |
| 8.19.1 | BASIC BLE PERIPHERAL..... | 64 |
| 8.19.2 | NOTIFICATIONS | 64 |
| 8.19.3 | PARING | 64 |
| 8.19.4 | BONDING | 67 |

8.1 Introduction

This chapter will introduce you to the WICED Bluetooth SDK and its use in ModusToolbox. It is meant as a getting started guide for those already familiar with the basics of Bluetooth and BLE.

Much of the material in this chapter is taken from the WICED Bluetooth 101 course. That is a 3-day course that covers much more ground (both in terms of topics and depth of coverage) than could fit in the time available for this single chapter. If you want more information about WICED Bluetooth, it is highly recommended that you attend that class or review it on your own. The class material can be found online at:

https://github.com/cypresssemiconductorco/CypressAcademy_WBT101_Files

Some of the topics from the full course that are not included here are:

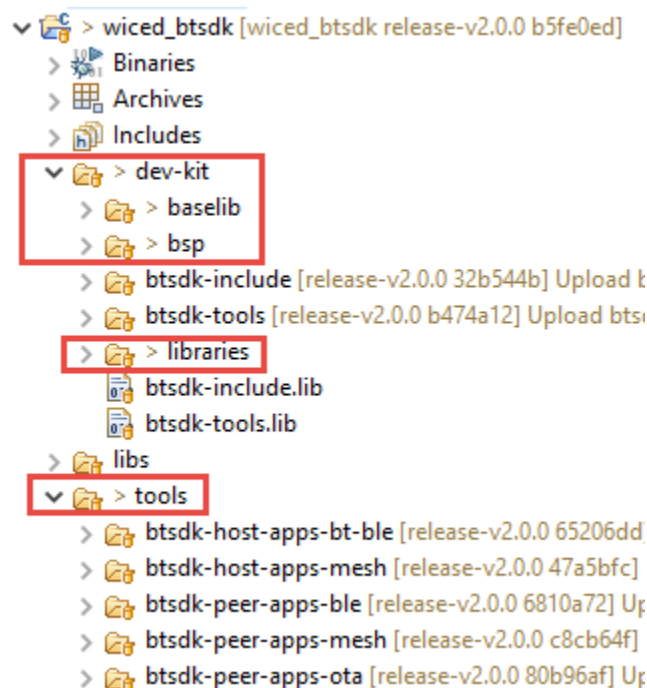
- BLE Protocol Details
- WICED MCU Peripherals
- WICED RTOS
- Advanced Security Topics such as Passkey and Numeric Comparison
- Low Power
- Beacons
- Over-The-Air Firmware Update
- BLE Centrals
- Debugging
- Classic Bluetooth
- Mesh

8.2 Bluetooth SDK Architecture

The Bluetooth SDK is downloaded from GitHub just like the other project dependencies. In the case of the SDK, it is contained in a separate Eclipse project called `wiced_btstack`. This project is referenced by all the applications in your workspace. That is, the `wiced_btstack` is shared among all applications in a single workspace.

The best practice is to create the `wiced_btstack` project first, but in future versions of the IDE and CLI, the project will be created automatically for you if it isn't found. It doesn't matter which hardware you select for the `wiced_sdk` project – one copy will work for all boards – but the name must NOT be changed.

The SDK contains the base libraries for all the supported devices in `wiced_btstack/dev-kit/baselib`, additional libraries in `wiced_btstack/dev-kit/libraries`, board support packages (BSPs) for all the supported boards in `wiced_btstack/dev-kit/bsp`, and Bluetooth tools such as ClientControl in `wiced_btstack/tools`.



Since the SDK is shared among all applications in a workspace, any changes to files contained in the SDK will affect all applications in that workspace. This is especially important for the BSP since it contains the Device Configurator files.

Therefore, if you want to use the Device Configurator to change settings for an application without affecting other applications, you can do one of three things: (1) override the device configurator files for a single application; (2) create a custom BSP inside the `wiced_btstack`; or (3) create a new workspace with a new copy of the `wiced_btstack` that you can modify. The first method is best when a single application requires a different configuration while 2 is better if you want a single custom configuration for multiple applications. Method 3 isn't really recommended since it results in a change to your copy of the

wiced_btstack repository or one of its dependent repositories. This prevents you from easily updating versions in the future.

The steps to override the Device Configurator files for a single application are:

1. Copy the *wiced_btstack/dev-kit/bsp/TARGET_<bsp_name>/COMPONENT_bsp_design_modus* directory and its contents to the application's project directory (the directory with the top-level source code). Change the name to something unique that still starts with COMPONENT_. For example:

COMPONENT_my_design_modus

2. Disable the configurator files from the BSP and include your custom configurator files by adding the following to the makefile (the makefile already has a blank line that says COMPONENTS += so you can add your component name to that line). Replace "my_design_modus" with whatever name you used for the directory (excluding COMPONENTS_).

```
COMPONENTS += my_design_modus
DISABLE_COMPONENTS += bsp_design_modus
```

3. Select your project in the Project Explorer and then open the Device Configurator from the Quick Panel. Make the required changes and save/exit the configurator.

The steps to create a custom BSP are:

1. Copy an existing BSP and paste it with a new name in the same directory, which is *wiced_btstack/dev-kit/bsp*.
2. Edit the makefile for the application to include the new BSP in the TARGET, SUPPORTED_TARGETS, and TARGET_DEVICE_MAP variables.
3. Select your project in the Project Explorer and open the Device Configurator from the Quick Panel. Make the required changes and save/exit the configurator.

8.3 Starter Application Sets

Most Bluetooth starter applications are really a "set" of applications instead of individual applications. For example, the BLE_20819EVB02 starter application contains 12 different BLE applications that are all created when you use that set as the starter application. For that reason, the name you provide is used as the prefix of the applications for that set. For example, BLE_20819EVB02 will give you projects called:

| | |
|-----------------------|-------------------------------------|
| BLE_20819EVB02.anc | (alert notification service client) |
| BLE_20819EVB02.ans | (alert notification service server) |
| BLE_20819EVB02.bac | (battery service client) |
| BLE_20819EVB02.bas | (battery service server) |
| BLE_20819EVB02.beacon | (beacon) |
| ... | |

On disk, this is done using hierarchy. For the example described above, the directory structure is:

```
BLE_20819EVB02/ble/anc/<application_files>
BLE_20819EVB02/ble/ans/<application_files>
BLE_20819EVB02/ble/bac/<application_files>
BLE_20819EVB02/ble/bas/<application_files>
BLE_20819EVB02/ble/beacon/<application_files>
...
```

The lowest level of the hierarchy is where the individual applications are stored. Go to that directory if you want to use the command line interface. Note that the middle level of the directory hierarchy (ble in this example) is not used in the Eclipse application name.

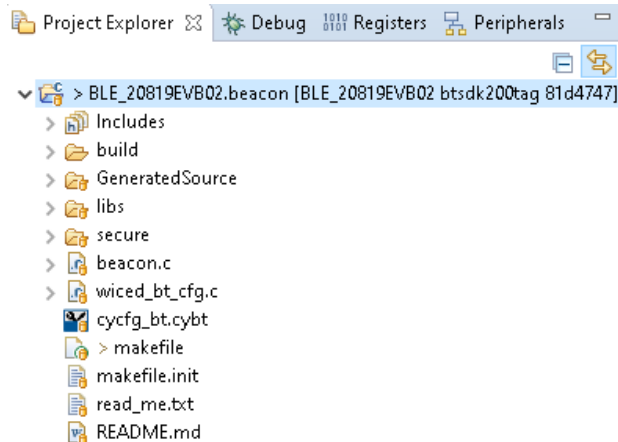
For some exercises in this class, we will use single application templates, so we will not include the extra hierarchy. In that case, the makefile is modified slightly from the starter applications – more on that later.

8.4 Application Architecture

At this point, we are ready to start developing the application. As mentioned before, a WICED Bluetooth application is the application project and the wiced_btsdk project. We looked briefly at wiced_btsdk earlier, so let's look at the application project now.

8.4.1 Project Explorer

In the Project Explorer window, you will see your application project and all its associated files.



The key parts of a project are:

- A directory with the name of the project
- Readme files (read_me.txt and/or README.md)
- Application configurator files (e.g. cycfg_bt.cybt for Bluetooth)
- A GeneratedSource directory with the output files from the application level configurators
- A libs directory
- makefile
- Stack configuration files (e.g. wiced_bt_cfg.c or app_bt_cfg.c)
- Application C source files

8.4.2 Readme

The first file to open in the file editor window will be the readme file included with the application, if there is one. This gives general information about the platform or the application that you started with.

8.4.3 Application Configurator Files

Files for configurators are distinguished by their file extension. For example, any file that ends with .cybt is a Bluetooth Configurator file.

8.4.4 GeneratedSource Directory

When you save from a configurator, the tool generates firmware in the GeneratedSource directory. The following are the most interesting/useful files (e.g. cycfg_gatt_db.c and cycfg_gatt_db.h for BLE).

Remember that the Device Configuration files (generated from design.modus) go in the BSP so they will NOT be in the GeneratedSource directory in the application.

8.4.5 libs

For Bluetooth starter applications, this directory just contains a file named index.html. It is used to populate the links in the Documents tab in the quick panel.

8.4.6 makefile

The makefile is used in the application creation process – it defines everything that ModusToolbox needs to know to create/build/program the application. This file is interchangeable between the ModusToolbox IDE and the Command Line Interface. So, once you create an application, you can go back and forth between the IDE and CLI at will.

Various build settings can be set in the makefile to change the build system behavior. These can be "make" settings or they can be settings that are passed to the compiler. Some examples are:

- Target Device (`TARGET=CYW920819EVB-02`)
- Relative path to the wiced_bt sdk (`CY_SHARED_PATH=$(CY_APP_PATH)/../../../wiced_bt sdk`)
- Method to generate the Bluetooth Device Address (`BT_DEVICE_ADDRESS?=default`)
- Enable OTA Firmware Upgrade (`OTA_FW_UPGRADE?=1`)
- Compiler Setting to Enabling Debug Trace Printing (`CY_APP_DEFINES+= -DWICED_BT_TRACE_ENABLE`)

You will get to experiment with some of these settings later.

In particular, note the `../../../` in the `CY_SHARED_PATH` that accounts for the hierarchy in the Bluetooth starter applications. In the single application templates we use, the hierarchy will be removed, so that will be changed to just `../`

8.4.7 Stack Configuration Files

Many of the templates you will use in this class include `app_bt_cfg.c` and `app_bt_cfg.h`, which create static definitions of the stack configuration and buffer pools. You will edit the stack configuration, for example, to optimize the scanning and advertising parameters. The buffer pools determine the availability of various sizes of memory blocks for the stack, and you might edit those to optimize performance and RAM usage.

Note: The actual file names may vary in some code examples, but the definitions of the `wiced_bt_cfg_settings` struct and `wiced_bt_cfg_buf_pools` array are required.

8.4.8 Application C file

All starter applications include at least one C source file that starts up the application (from the `application_start()` function) and then implements other application functionality. The actual name of this file varies according to the starter application used to create the application. It is usually the same name as the template but there are exceptions to that rule. For example, in the templates provided for this class this top-level file is always called `app.c`.

Note: MESH examples are implemented with a library called `mesh_app_lib`. The library includes the `application_start()` function in `mesh_application.c` inside the library. This is because the application itself is very regimented, and the developer does not really "own" the way devices interact. The user part of the application is restricted to activity supported by the device's capabilities, such as dimming the light with a PWM or controlling a door lock solenoid.

The application C file begins with various `#include` lines depending on the resources used in your application. These header files can be found in the SDK under `wiced_btsdk/dev-kit/baselib/20819A1/include`. A few examples are shown below. The first 4 are usually required in any project. The "hal" includes are only needed if the specific peripheral is used in the project, e.g. `wiced_hal_i2c.h` is required if you are using I2C.

```
#include "wiced.h"                // Basic formats like stdint, wiced_result_t, WICED_FALSE, WICED_TRUE
#include "wiced_platform.h"        // Platform file for the kit
#include "sparcommon.h"            // Common application definitions
#include "wiced_bt_stack.h"        // Bluetooth Stack
#include "wiced_bt_dev.h"          // Bluetooth Management
#include "wiced_bt_ble.h"          // BLE
#include "wiced_bt_gatt.h"         // BLE GATT database
#include "wiced_bt_uuid.h"         // BLE standard UUIDs
#include "wiced_rtos.h"            // RTOS
#include "wiced_bt_app_common.h"   // Miscellaneous helper functions including wiced_bt_app_init
#include "wiced_transport.h"       // HCI UART drivers
#include "wiced_bt_trace.h"        // Trace message utilities
#include "wiced_timer.h"           // Built-in timer drivers
#include "wiced_hal_i2c.h"         // I2C drivers
#include "wiced_hal_adc.h"         // ADC drivers
#include "wiced_hal_pwm.h"         // PWM drivers
#include "wiced_hal_puart.h"       // PUART drivers
#include "wiced_rtos.h"            // RTOS functions
#include "wiced_hal_nvrnm.h"       // NVRAM drivers
#include "wiced_hal_wdog.h"        // Watchdog
#include "wiced_spar_utils.h"      // Required for stdio functions such as snprintf and sprintf
#include <stdio.h>                  // Stdio C functions such as snprintf and sprintf
```

After the includes list you will find the `application_start()` function, which is the main entry point into the application. That function typically does a minimal amount of initialization, starts the Bluetooth stack and registers a stack callback function by calling `wiced_bt_stack_init()`. Note that the configuration parameters from `app_bt_cfg.c` are provided to the stack here. The callback function is called by the stack whenever it has an event that the user's application might need to know about. It typically controls the rest of the application based on Bluetooth events.

Most application initialization is done once the Bluetooth stack has been enabled. That event is called `BTM_ENABLED_EVT` in the callback function. The full list of events from the Bluetooth stack can be found in the file `wiced_btsdk/dev-kit/baselib/20819A1/include/wiced_bt_dev.h`.

A minimal C file for an application will look something like this:

```
#include "sparcommon.h"
#include "wiced_platform.h"
#include "wiced_bt_dev.h"
#include "wiced_bt_stack.h"
#include "app_bt_cfg.h"

wiced_bt_dev_status_t app_bt_management_callback( wiced_bt_management_evt_t event,
wiced_bt_management_evt_data_t *p_event_data );

void application_start(void)
{
    wiced_bt_stack_init( app_bt_management_callback, &wiced_bt_cfg_settings,
                        wiced_bt_cfg_buf_pools );
}

wiced_result_t app_bt_management_callback( wiced_bt_management_evt_t event,
wiced_bt_management_evt_data_t *p_event_data )
{
    wiced_result_t status = WICED_BT_SUCCESS;

    switch( event )
    {
    case BTM_ENABLED_EVT:           // Bluetooth Controller and Host Stack Enabled

        if( WICED_BT_SUCCESS == p_event_data->enabled.status )
        {
            /* Initialize and start your application here once the BT stack is running */
        }
        break;

    default:
        break;
    }

    return status;
}
```

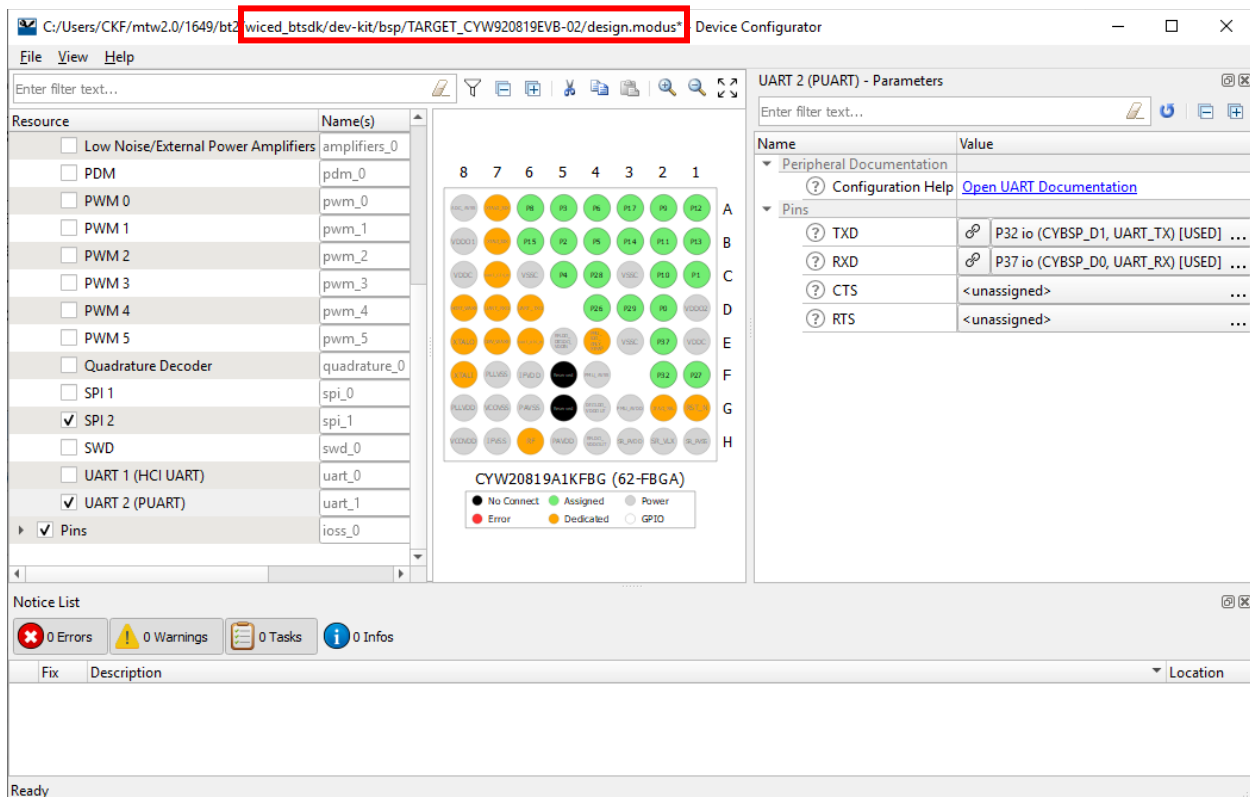
8.5 Device Configuration

The design.modus file is the database of configuration choices for the device. It is the source document for the Device Configurator tool which presents a list of pins and peripherals on the device that you can set up. As mentioned earlier, the design.modus file is shared by all applications in a workspace that use a given BSP. Keep that in mind if you chose to edit the design.modus file.

As described earlier, you can override the design.modus for a single application. This is generally the safest way to modify the device configuration.

To launch the Device Configurator, click on the link in the Quick Panel or double-click on the design.modus file either in the BSP (wiced_btSDK/dev-kit/bsp/<BSP Name>) or in your application if you are using the override method. As you can see there are sections for Peripherals and Pins on the left. When you enable a peripheral or pin, the upper right-hand panel allows you to select configuration options and to open the documentation for that element. In some cases, you can launch a secondary configurator from that window (Bluetooth is one of those cases).

It is a good idea to look at the path at the top of the configurator window to verify you are editing the file from the expected location (either in the BSP or in the application).



Once you save the configurator information (**File> Save**) it creates/updates the Generated Source files in the project BSP or the local copy if you are using the override method.

8.6 Managing Libraries

There is a Library Manager that is intended to add/remove individual libraries from an application. However, for BT applications, the libraries are all currently shared (from wiced_bt_sdk) so the Library Manager isn't very useful (at least not yet).

Instead, the way to add a new library to an application is as follows (you will get to try this out in later chapters):

1. Add the appropriate include in your source code.
2. Add the library name to the COMPONENTS variable in the makefile.
3. Add the path to the library to the SEARCH_LIBS_AND_INCLUDES variable in the makefile.

For example, to add the over-the-air (OTA) firmware upgrade library, you would add:

1. source code:

```
#include "wiced_bt_ota_firmware_upgrade.h"
```

2. makefile:

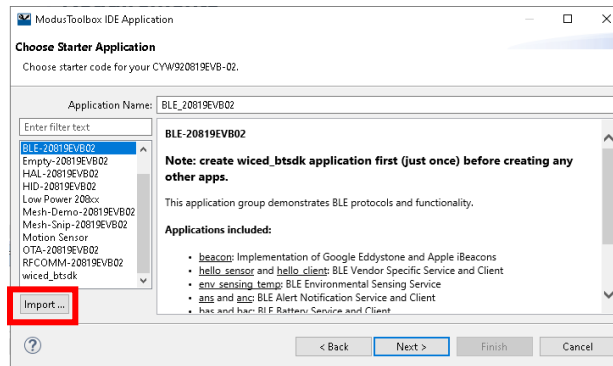
```
COMPONENTS+=fw_upgrade_lib  
.  
.  
.  
SEARCH_LIBS_AND_INCLUDES+=$(CY_SHARED_PATH)/dev-kit/libraries/btsdk-ota
```

Note: For future reference, the required makefile edits are also shown at the bottom of the README.md file in the wiced_bt_sdk project or BT starter applications.

8.7 Renaming / Copying / Sharing Applications

8.7.1 Import Using the New Application Wizard

You can easily create a new application from an existing one using the **Import...** button in the new application wizard after selecting the target hardware.



After clicking on **Import**, just select the application directory to import (the directory containing the makefile).

If you do this for a single application that has additional hierarchy (such as most of the Bluetooth starter application code examples) you need to modify the path in the makefile after importing since the import will not maintain the additional hierarchy. That is, the `CY_SHARED_PATH` will need to be modified to the following:

```
CY_SHARED_PATH=$(CY_APP_PATH)/../wiced_btsdk
```

After updating the makefile, build the project to get the Launches populated in the Quick Panel.

Note that if you provide the top-level directory of an entire set of applications, the hierarchy will be maintained so everything will work as-is.

8.7.2 Rename

You can rename a single project from inside the Eclipse IDE by right clicking on the project in the project explorer and selecting "Rename" (don't rename `wiced_btsdk` though – that will break things). This renames the project as it shows up in Eclipse, but it does NOT change the name of the directories in the file system. The Eclipse project name can be found in the `.project` file inside the project directory.

8.7.3 Copy/Paste

You can copy/paste projects, but you must keep the relative path to the `wiced_btsdk` the same (or else change the path in the makefile to match the new relative path). This is difficult to get just right inside the IDE, so it is easier to copy/paste from the OS (Windows explorer, etc.) and then update the Eclipse project name in the `.project` file. Once you do that, you can import the project into Eclipse using **File > Import > General > Existing Projects into Workspace**.

After either a rename or a copy/paste, the Launches section for the new project in the Quick Panel will be empty. The launches can be recreated from the command line by running the command:

```
make eclipse CY_IDE_PRJNAME=<IDE project name>
```

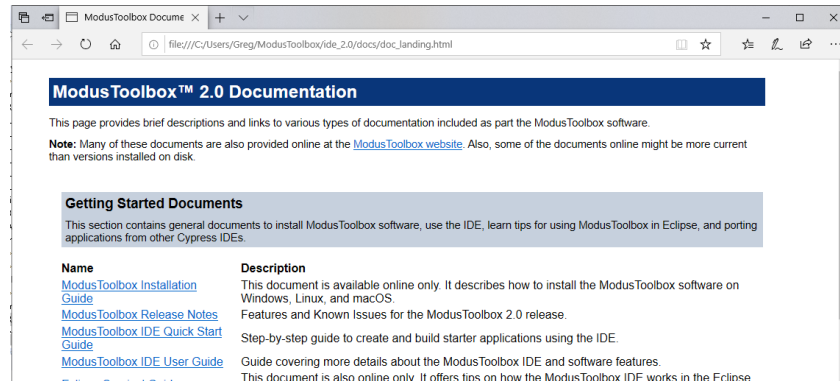
where <IDE project name> is the Eclipse name of the new project (found in the .project file).

8.7.4 Archive Files

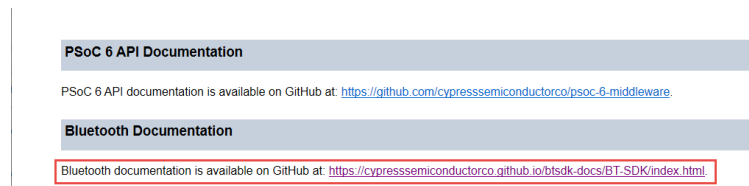
You can use **File > Export > General > Archive File** to create an archive file for one or more projects from a workspace. You should typically uncheck the build directory if it exists to save space (this contains build output files). To import from the archive file into a new workspace, use **File > Import > General > Existing Projects into Workspace**. Using this method, any hierarchy will be flattened so you will again have to modify the path to the wiced_btstack in the makefile if the starter application had hierarchy.

8.8 Documentation

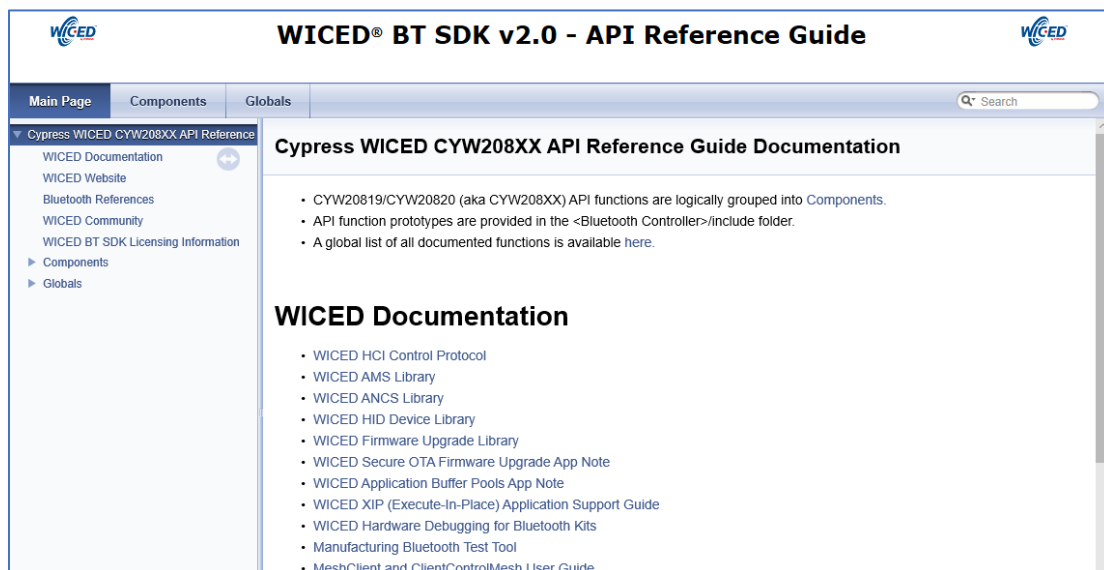
In the ModusToolbox IDE the Bluetooth documentation can be accessed from the menu item **Help > ModusToolbox General Documentation > ModusToolbox Documentation Index**. This will give you a page like this:



At the bottom of that page you will find a link to the BT SDK API guide.



The Bluetooth Documentation link takes you to a page where you select from a list of Bluetooth devices. This page is also available directly from a link in the Quick Panel. Once you select your device, you will get to this page:



The documentation on the right is useful for specific topics while the section on the left is invaluable for understanding the API functions, data types, enumerations, macros, etc.

8.9 Exercises (Part 1)

8.9.1 Turn on an LED

In this exercise, you will install the wiced_btsdk, create a new application, and then build/program it to a kit.



1. Create a new workspace to hold your Bluetooth applications.
 - a. Hint: Use *File->Switch Workspace->Other...* and then enter the new workspace name.
 - b. Hint: You may want to enable *Copy Settings-> Preferences* before creating the new workspace to get your preferences from your existing workspace.



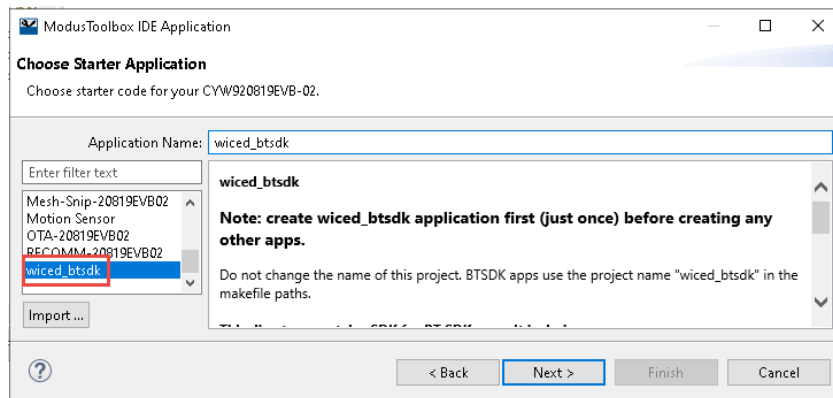
2. In the Quick Panel tab click "New Application".



3. Select the CYW920819EVB-02 kit and click "Next >".



4. Select "wiced_btsdk".



5. Click "Next >"



6. Click "Finish" (it will take a few minutes to download the SDK. Remember you only need this step once per workspace since the SDK is shared).



7. Once that's done, click "New Application" again.



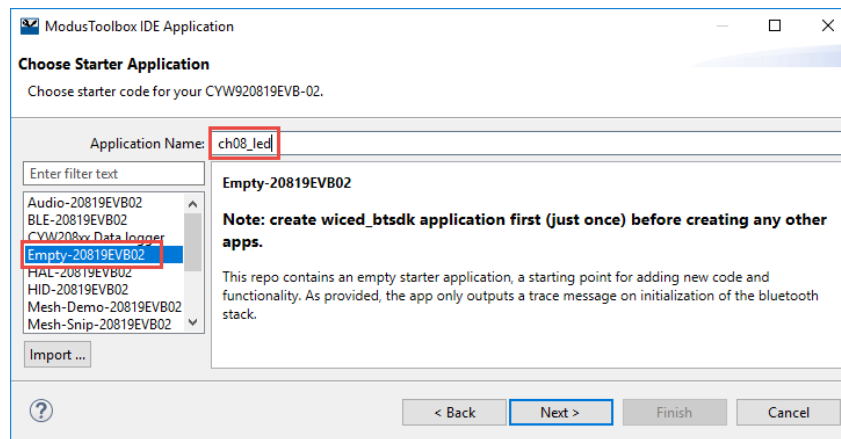
8. Select the CYW920819EVB-02 kit and click "Next >".



9. Select "Empty-20819EVB02"

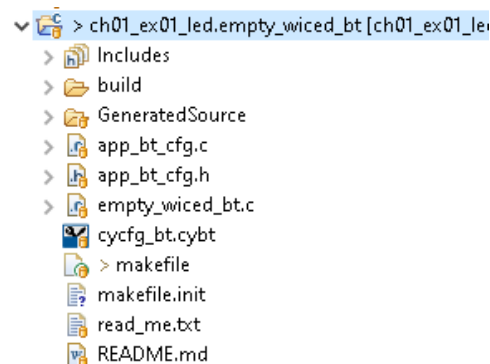


10. Change the application name to **ch08_led**.



11. Click "Next ". Verify the presented device, board, and example, then click "Finish".

When the application has been created, the Project Explorer window in Eclipse should look like the following screenshot (ModusToolbox IDE adds ".empty_wiced_bt" to the project name because, as you learned earlier, some starter applications actually contain a set of applications):



12. Open the top-level C file which in this case is called empty_wiced_bt.c.



13. Add the following line in the application_start function:

```
wiced_hal_gpio_set_pin_output(WICED_GPIO_PIN_LED_1, GPIO_PIN_OUTPUT_LOW);
```



14. Connect your CYW920819EVB-02 kit to a USB port on your computer.



15. In the Quick Panel, look in the "Launches" section and click the "ch08_led.empty_wiced_bt Program" link.



16. Once the build and program operations are done, you should see "Download succeeded" in the Console window and LED1 (the yellow user LED) should be on.

8.9.2 Add a timer to get the LED to blink at 1Hz

In this exercise, instead of just turning the LED on, you will make it blink. Since there is no infinite loop in `application_start`, you will need to initialize and start a 500ms timer and then toggle the LED state in the timer callback.



Make the LED blink. Start with the Empty-20819EVB02 starter application again but call your project **ch08_timer**.

Hints:

- The WICED API documentation is your friend. Look in *Components->Hardware Drivers->Timer Management Services* (init and start timer), and *Components->Hardware Drivers->GPIO*, (set pin output).
- You should initialize and start the timer in the BT management callback function for the `BTM_ENABLED_EVT` case. This will get called once the stack has completed its initialization.
- Include the "wiced_timer.h" header file to get access to the functions. Look in the header file for definitions that you will need such as `wiced_timer_type_e`.
- Create a variable of type `wiced_timer_t` for the handle of the timer instance. You will pass a pointer to this variable to the init and start functions.

8.9.3 Use an RTOS Thread to blink the LED

In this exercise, instead of using a timer create a new thread that controls the LED blinking. Start with the Empty-20819EVB02 starter application again but call your project **ch08_thread**.



Create a new thread.

Hints:

- Look in the documentation under *Components->RTOS*. You should create and initialize the thread in the BT management callback function for the `BTM_ENABLED_EVT` case.
- Include the "wiced_rtos.h" header file.
- Use a stack size of 1024 and a priority of 4 for your thread.
- In the thread, you will need to use some form of delay such as `wiced_rtos_delay_milliseconds`. Use "ALLOW_THREAD_TO_SLEEP" as the second parameter to the delay function.

8.10 Bluetooth Stack Events

In a Bluetooth design, the Stack generates Events based on what is happening in the Bluetooth world. After an event is created, the Stack will call the callback function which you registered when you turned on the Stack. Your callback firmware must look at the event code and the event parameter and take the appropriate action.

In most cases, your application's functionality will be implemented inside various callback events. That is, `application_start` will be mostly empty, and the "guts" of your firmware will be inside the callback functions.

There are two classes of Stack events: Management, and GATT. Each of these has its own callback function.

8.10.1 Essential Bluetooth Management Events

In our template, the callback for management events is called `app_bt_management_callback`. It may have a different name in other code examples, but it is always registered in `application_init`.

The stack will generate more events than are listed below so look at the documentation or in the SDK functions themselves if you need to know about other events. The tables contain the most frequently used Stack events. Only the first 2 events are used if you are not implementing security.

| Event | Description |
|---|--|
| BTM_ENABLED_EVT | When the Stack has everything going. The event data will tell you if it happened with <code>WICED_SUCCESS</code> or <code>!WICED_SUCCESS</code> . |
| BTM_BLE_ADVERT_STATE_CHANGED_EVT | When Advertising is either stopped or started by the Stack. The event parameter will tell you <code>BTM_BLE_ADVERT_OFF</code> or one of the many different levels of active advertising. |
| BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT | At initialization, the BLE stack looks to see if the privacy keys are available. If privacy is not enabled, then this state does not need to be implemented as long as you return a default value of <code>WICED_BT_SUCCESS</code> . |
| BTM_SECURITY_REQUEST_EVT | The occurs when the client requests a secure connection. When this event happens, you need to call <code>wiced_bt_ble_security_grant()</code> to allow a secure connection to be established. |
| BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT | This occurs when the client asks what type of capability your device has that will allow validation of the connection (e.g. screen, keyboard, etc.). You need to set the appropriate values when this event happens. |
| BTM_ENCRYPTION_STATUS_EVT | This occurs when a secure link has been established. |
| BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT | This event is used so that you can store the paired devices keys if you are storing bonding information. If not, then this state does not need to be implemented. |

8.10.2 Essential GATT Events

In our template, the callback for GATT events is called *app_gatt_callback*. It may have a different name in other code examples, but it is usually registered in the BTM_ENABLED_EVT inside the Bluetooth Management callback function.

Again, this is a sub-set of the events that the Stack will generate so look at the documentation or the code if your application needs to deal with additional callback events.

| Event | Description |
|----------------------------|--|
| GATT_CONNECTION_STATUS_EVT | When a connection is made or broken. The event parameter tells you WICED_TRUE if connected. |
| GATT_ATTRIBUTE_REQUEST_EVT | When a GATT Read or Write occurs. The event parameter tells you GATTS_REQ_TYPE_READ or GATTS_REQ_TYPE_WRITE. |

8.10.3 Essential GATT Sub-Events

In addition to the GATT events described above, there are sub-events associated with each of the main events which are also handled in the template.

GATT_CONNECTION_STATUS_EVT

| Event | Description |
|-------------------------|---|
| connected == WICED_TRUE | A GATT connection has been established. |
| connected != WICED_TRUE | A GATT connection has been broken. |

GATT_ATTRIBUTE_REQUEST_EVT

For most BLE peripherals, there are two sub-events for an Attribute Request Event that we care about:

| Event | Description |
|----------------------|--|
| GATTS_REQ_TYPE_READ | A GATT Attribute Read has occurred. The event parameter tells you the request handle and where to save the data. |
| GATTS_REQ_TYPE_WRITE | A GATT Attribute Write has occurred. The event parameter tells you the handle, a pointer to the data and the length of the data. |

8.11 Firmware Architecture

There are four steps to make a basic WICED BLE Peripheral:

- Turn on the Stack
- Start Advertising
- Process Connection Events from the Stack
- Process Read/Write Events from the Stack

We will provide you with a minimal Bluetooth template to use in the exercises that mimics this flow. In addition to `application_start`, it has a Bluetooth Management callback function and a GATT event callback function as described above. The template also contains some helper functions used to simplify reading/writing from/to the GATT database.

8.11.1 Turning on the Stack

When a WICED device turns on, the chip boots, starts the RTOS and then jumps to a function called `application_start` which is where your Application firmware starts. At that point in the proceedings, your Application firmware is responsible for turning on the Stack and making a connection to the WICED radio. This is done with the API call `wiced_bt_stack_init`. One of the key arguments to `wiced_bt_stack_init` is a function pointer to the management callback. The template uses the name `app_bt_management_callback` for the Bluetooth management callback.

In `app_bt_management_callback` it is your job to fill in what the firmware does to processes various events. This is implemented as a switch statement in the callback function where the cases are the Stack events. Some of the necessary actions are provided automatically and others will need to be written by you.

When you start the Stack, it generates the `BTM_ENABLED_EVT` event and calls the `app_bt_management_callback` function which then processes that event.

The `app_bt_management_callback` case for `BTM_ENABLED_EVT` event calls the functions `wiced_bt_gatt_register` and `wiced_bt_gatt_db_init`, which registers a callback function for GATT database events and initializes the GATT database.

The `BTM_ENABLED_EVT` ends by calling the `wiced_bt_start_advertising` function.

8.11.2 Start Advertising

The Stack is triggered to start advertising by a call to `wiced_bt_start_advertising`.

The function `wiced_bt_start_advertising` takes 3 arguments. The first is the advertisement type and has 9 possible values:

```
BTM_BLE_ADVERT_OFF,           /**< Stop advertising */
BTM_BLE_ADVERT_DIRECTED_HIGH, /**< Directed advertisement (high duty cycle) */
BTM_BLE_ADVERT_DIRECTED_LOW,  /**< Directed advertisement (low duty cycle) */
BTM_BLE_ADVERT_UNDIRECTED_HIGH, /**< Undirected advertisement (high duty cycle) */
BTM_BLE_ADVERT_UNDIRECTED_LOW, /**< Undirected advertisement (low duty cycle) */
BTM_BLE_ADVERT_NONCONN_HIGH,  /**< Non-connectable advertisement (high duty cycle) */
BTM_BLE_ADVERT_NONCONN_LOW,   /**< Non-connectable advertisement (low duty cycle) */
BTM_BLE_ADVERT_DISCOVERABLE_HIGH, /**< discoverable advertisement (high duty cycle) */
BTM_BLE_ADVERT_DISCOVERABLE_LOW /**< discoverable advertisement (low duty cycle) */
```

For undirected advertising (which is what we will use in our examples) the 2nd and 3rd arguments can be set to 0 and NULL respectively.

The Stack then generates the `BTM_BLE_ADVERT_STATE_CHANGED_EVT` management event and calls the `app_bt_management_callback`.

The `app_bt_management_callback` case for `BTM_BLE_ADVERT_STATE_CHANGED_EVT` looks at the event parameter to determine if it is a start or end of advertising. In the template code it does not do anything when advertising is started, but you could, for instance, turn on an LED to indicate the advertising state.

8.11.3 Processing Connection Events from the Stack

The getting connected process starts when a Central that is actively Scanning hears your advertising packet and decides to connect. It then sends you a connection request.

The Stack responds to the Central with a connection accepted message.

The Stack then generates a GATT event called `GATT_CONNECTION_STATUS_EVT` which is processed by the `app_gatt_callback` function.

The code for the `GATT_CONNECTION_STATUS_EVT` event uses the event parameter to determine if it is a connection or a disconnection. It then prints a message.

On a connection, the Stack then stops the advertising and calls `app_bt_mangement_callback` with a management event `BTM_BLE_ADVERT_STATE_CHANGED_EVT`.

The `app_bt_management_callback` determines that it is a stop of advertising and just prints out a message. You could add your own code here to, for instance, turn off an LED or restart advertisements.

8.11.4 Processing Client Read Events from the Stack

When the Client wants to read the value of a Characteristic, it sends a read request with the Handle of the Attribute that holds the value of the Characteristic. We will talk about how handles are exchanged between the devices later.

The Stack generates a `GATT_ATTRIBUTE_REQUEST_EVT` and calls `app_gatt_callback`, which determines the event is `GATT_ATTRIBUTE_REQUEST_EVT`. The code for this event looks at the event parameter and determines that it is a `GATTS_REQ_TYPE_READ`, then calls the function `app_gatt_get_value` to find the current value of the Characteristic.

That function looks through that GATT Database to find the Attribute that matches the Handle requested. It then copies the value's bytes out of the GATT Database into the location requested by the Stack.

Finally, the get value function returns a code to indicate what happened - either `WICED_BT_GATT_SUCESS`, or if something bad has happened (like the requested Handle doesn't exist) it returns the appropriate error code such as `WICED_BT_GATT_INVALID_HANDLE`. The list of the return codes is taken from the `wiced_bt_gatt_status_e` enumeration. This enumeration includes (partial list):

```
enum wiced_bt_gatt_status_e
{
    WICED_BT_GATT_SUCCESS                = 0x00, /**< Success */
    WICED_BT_GATT_INVALID_HANDLE         = 0x01, /**< Invalid Handle */
    WICED_BT_GATT_READ_NOT_PERMIT        = 0x02, /**< Read Not Permitted */
    WICED_BT_GATT_WRITE_NOT_PERMIT       = 0x03, /**< Write Not permitted */
    WICED_BT_GATT_INVALID_PDU            = 0x04, /**< Invalid PDU */
    WICED_BT_GATT_INSUF_AUTHENTICATION   = 0x05, /**< Insufficient Authentication */
    WICED_BT_GATT_REQ_NOT_SUPPORTED      = 0x06, /**< Request Not Supported */
    WICED_BT_GATT_INVALID_OFFSET         = 0x07, /**< Invalid Offset */
    WICED_BT_GATT_INSUF_AUTHORIZATION    = 0x08, /**< Insufficient Authorization */
    WICED_BT_GATT_PREPARE_Q_FULL         = 0x09, /**< Prepare Queue Full */
    WICED_BT_GATT_NOT_FOUND              = 0x0a, /**< Not Found */
    WICED_BT_GATT_NOT_LONG               = 0x0b, /**< Not Long Size */
    WICED_BT_GATT_INSUF_KEY_SIZE          = 0x0c, /**< Insufficient Key Size */
    WICED_BT_GATT_INVALID_ATTR_LEN       = 0x0d, /**< Invalid Attribute Length */
    WICED_BT_GATT_ERR_UNLIKELY           = 0x0e, /**< Error Unlikely */
    WICED_BT_GATT_INSUF_ENCRYPTION       = 0x0f, /**< Insufficient Encryption */
    WICED_BT_GATT_UNSUPPORTED_GRP_TYPE   = 0x10, /**< Unsupported Group Type */
    WICED_BT_GATT_INSUF_RESOURCE         = 0x11, /**< Insufficient Resource */
}
```

The status code generated by the get value function is returned up through the function call hierarchy and eventually back to the Stack, which in turn sends it to the Client.

To summarize, the course of events for a read is:

1. Stack calls `app_gatt_callback` with `GATT_ATTRIBUTE_REQUEST_EVT`
2. `app_gatt_callback` detects the `GATTS_REQ_TYPE_READ` request type
3. `app_gatt_callback` calls `app_gatt_get_value`

8.11.5 Processing Client Write Events from the Stack

When the Client wants to write a value to a Characteristic, it sends a write request with the Handle of the Attribute of the Characteristic along with the data.

The Stack generates the GATT event `GATT_ATTRIBUTE_REQUEST_EVT` and calls the function `app_gatt_callback`, which determines the event is `GATT_ATTRIBUTE_REQUEST_EVT`. The code for this event looks at the event parameter and determines that it is a `GATTS_REQ_TYPE_WRITE`, then calls the function `app_gatt_set_value` to update the current value of the Characteristic.

The `app_gatt_set_value` function looks through that GATT Database to find the Attribute that matches the Handle requested. It then copies the value bytes from the Stack generated request into the GATT Database. Finally, the set value function returns a code to indicate what happened just like the Read - either `WICED_BT_GATT_SUCCESS`, or the appropriate error code. The list of the return codes is again taken from the `wiced_bt_gatt_status_e` enumeration.

The status code generated by the set value function is returned up through the function call hierarchy and eventually back to the Stack. One difference here is that if your callback function returns `WICED_BT_GATT_SUCCESS`, the Stack sends a Write response of 0x1E. If your callback returns something other than `WICED_BT_GATT_SUCCESS`, the stack sends an error response with the error code that you chose.

To summarize, function call hierarchy for a write is:

1. Stack calls `app_gatt_callback` with `GATT_ATTRIBUTE_REQUEST_EVT`
2. `app_gatt_callback` detects the `GATTS_REQ_TYPE_WRITE` request type
3. `app_gatt_callback` calls `app_gatt_set_value`

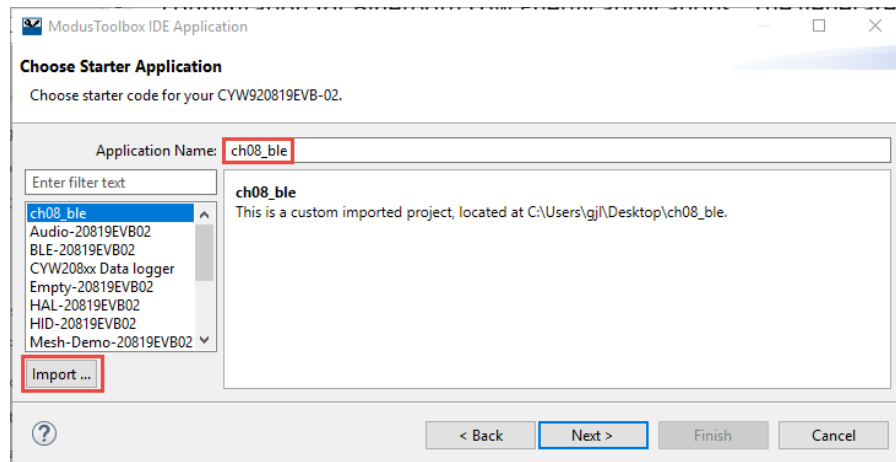
8.12 Simple Peripheral Demo Walkthrough

Now that you know how to start the stack, start advertisements, and process stack events, the next question is how to do you setup and configure the Bluetooth attributes whose values will be sent back and forth across the link? The simple answer is to use the Bluetooth Configurator.

Bluetooth Configurator is a tool that will build a semi-customized GATT database and device configuration for Bluetooth Low Energy applications. The generates two files that you will be using – `cycfg_gatt_db.c` and `cycfg_gatt_db.h`. It also generates a timestamp file called `cycfg_bt.timestamp`.

For this example, I am going to build a BLE application called `ch08_ble` using a template that has one custom service called the “Modus101” Service with one writable characteristic called “LED”. When the Client writes a 0 or 1 (strictly any non-zero value) into that Characteristic, my application firmware will just write that value into the GPIO driving the LED.

First, I'll create a new project for the CYW920819EVB-02 kit but this time I'll use the **Import...** button to import from the **ch08_ble** template.

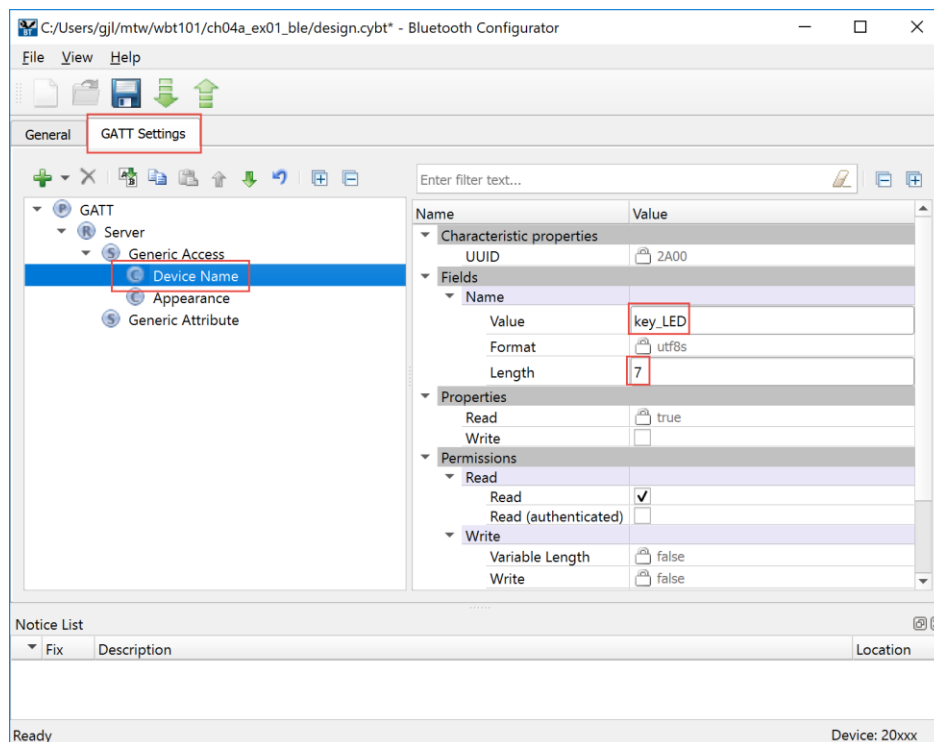


8.12.1 Running the Bluetooth Configurator

You can launch the Bluetooth Configurator from the link in the quick panel. If your project already has a .cybt file, it will be opened. If not, a new file called design.cybt will be created for you. Note that a project can only have one file with the extension .cybt.

This will populate default General and GATT Settings. We will leave the General settings alone, so switch to the GATT Settings tab.

You need to give your device a name and this is done by clicking on the *Device Name* field and typing into the *Value* text box. The name is just a string (format "utf8s" per the BLE spec). You must press Enter to get the tool to calculate the length for you.

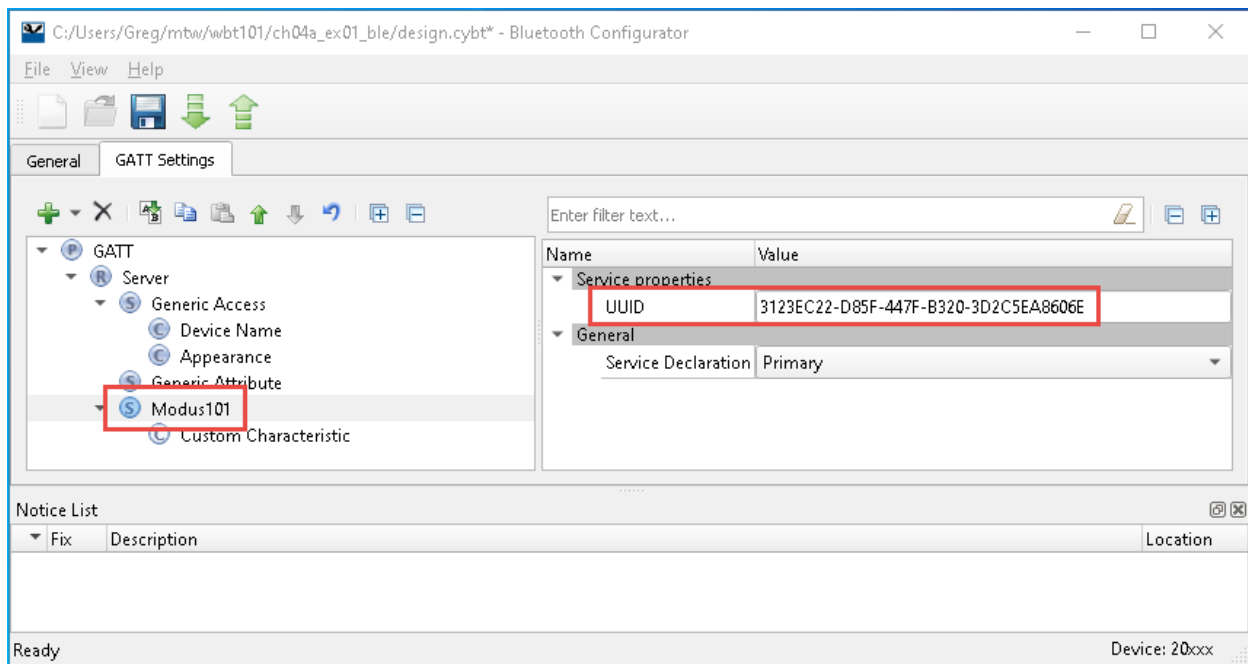


It is important that the name you choose is unique or you will not be able to identify your device when making connections from your cell phone. In this case, I've called the device *key_LED*. When you do this yourself, use a unique device name such as <inits> LED where <inits> is your initials.

Make sure you press the "enter" key after typing in the name. This will calculate the string length and will put it in the Length field.

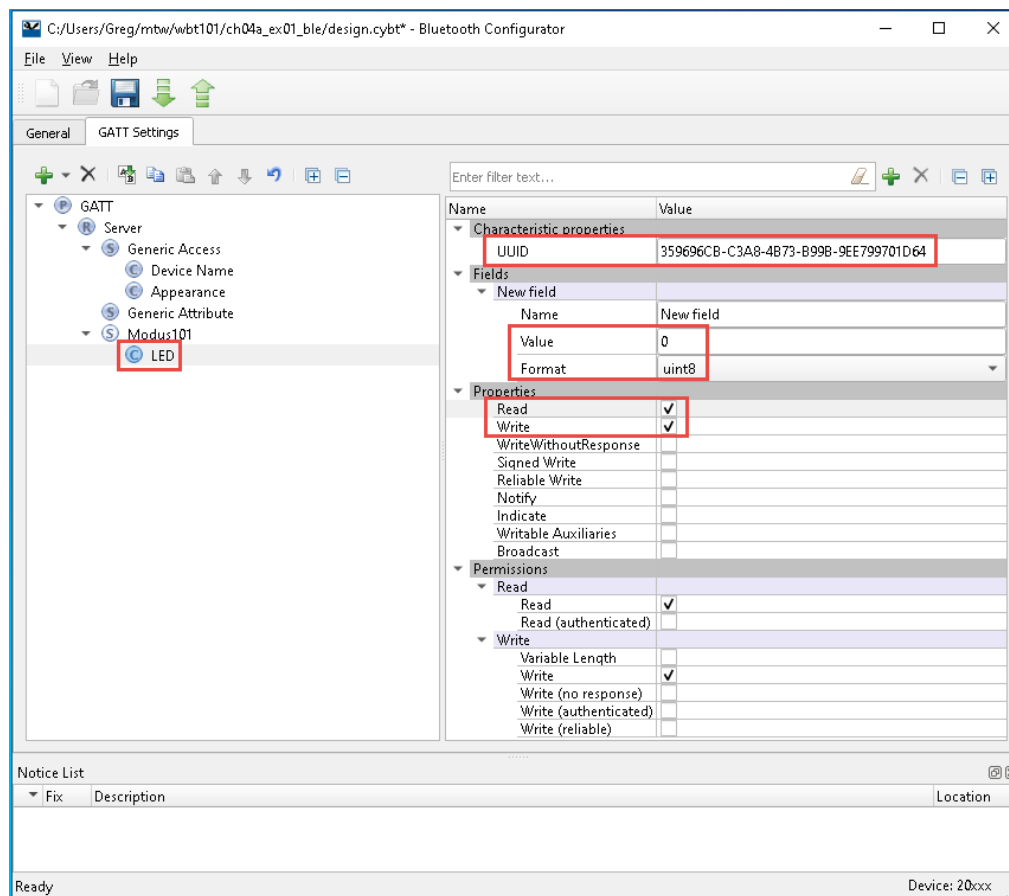
The next step is to set up a Service. To do this:

1. Select *Server* in the GATT database.
2. Right-click and choose *Add Service*, then select *Custom Service* (it is near the bottom of the list). A *Custom Service* entry appears in the GATT database.
3. Right-click on the custom service and select *Rename*. Call the service “Modus”.
4. The tool will choose a random UUID for this Service, but you could specify your own UUID if desired. For this exercise, just keep the random UUID.



The Service includes a Characteristic, which we are going to use to control the LED. To do this you:

1. Right-click on *Custom Characteristic* under the *Modus101* Service and *Rename* to “LED”.
2. Under Fields, optionally provide a name. This name is not used and can be left as-is.
3. Change the format from utf8s (which requires a length) to uint8 (which has a length of 1 by definition).
4. Change the value of the LED characteristic to 0, which we will take to mean “OFF”. This will be the initial value.
5. We want the client to be able to Read and Write this Characteristic, so under *Properties*, enable *Read* and *Write*. Note that the tool makes the corresponding changes to the *Permissions* section for you, so you don't need to set them unless you need an unusual combination of Properties and Permissions.
6. Again, keep the randomly assigned UUID for the Characteristic just like you did for the Service UUID.



7. Click the **Save** button to save the file design.cybt. Note that the name of the file doesn't matter as long as the extension is cybt so you may see different names used in other applications.
8. Saving will create a GeneratedSource directory with the code generated based on your selections. You should not modify the generated code by hand – any changes should be done by re-running the Bluetooth Configurator.

8.12.2 Editing the Firmware

The template includes a little bit of setup code for the BTM_ENABLED_EVT and some very helpful functions, as follows.

- `app_bt_management_callback()` is the callback function that you edited in chapter 2. The BTM_ENABLED_EVT code now prints the Bluetooth Device Address (BDA), sets up the GATT database, and starts advertising for a connection.
- `app_gatt_callback()` handles GATT events such as connect/disconnect and attribute read/write requests.
- `app_set_advertisement_data()` creates the advertising packet that includes the device name you will see in the CySmart app.
- `app_gatt_get_value()` searches the GATT database for the requested characteristic and extracts the value. We use this function to read the state of the LED.
- `app_gatt_set_value()` searches the GATT database for the requested characteristic and updates the value. We use this function to write the state of the LED into the database and, later, notify the central device.

Follow these instructions to control the device behavior. Note that the template sets up the PUART for debugging traces so you can use `WICED_BT_TRACE()` to better understand how the stack is behaving.

1. Start by opening `app.c` and adding the include for the generated database, as follows:

```
#include "cycfg_gatt_db.h"
```

2. Template code for the `BTM_ENABLED_EVT` case in `app_bt_management_callback()` reads and reports the 6-byte Bluetooth Device Address (BDA) in the terminal when the stack gets enabled. Note that this address must be unique to avoid collisions with other devices.

By default, the address format is defined in a file in the SDK. It can be found at:

```
wiced_btstack/dev-kit/baselib/20819A1/platforms/208XX_OCF.btp
```

In this file, there are lots of device specific settings. The one that controls the address is:

```
DLConfigBD_ADDRBase = "20819A1*****"
```

The asterisk characters mean that a value should be chosen for those digits during build. Therefore, the 6-byte address generated for your device will start with 20819A1 with 5 digits after that. By default, the 5 digits are based on the MAC address of your computer. That is, the address generated should always be the same for your computer but will be different for other computers.

There are 2 cases where this may cause a problem: (1) if you are programming more than one kit from a single computer and want them to operate at the same time; or (2) when using a virtual machine, a MAC address may not be found in which case the 5 digits will all be set to 0.

Due to the above potential issues, we will change a setting to get random values for those 5 digits so that there aren't any collisions between students. This means that you will get a different BT address each time you rebuild an application.

To set that up, open the makefile that is in your application and find the line that says:

```
BT_DEVICE_ADDRESS?=default
```

and change it to:

```
BT_DEVICE_ADDRESS?=random
```

Note that in this case "random" only means use random values for the 5 digits with an asterisk in configuration file. The resulting address is still public device address for your device. Don't confuse this with a truly random device address. We will discuss BLE address types in more detail in the privacy section in the next chapter.

- Back in app.c, in the `BTM_ENABLED_EVT` case, add the following lines to set up the GATT database according to your selections in the Configurator:

```
/* Register GATT callback and initialize the GATT database*/
wiced_bt_gatt_register( app_gatt_callback );
wiced_bt_gatt_db_init( gatt_database, gatt_database_len );
```

- Next, I don't want to allow pairing to the device just yet so configure the pairing mode with the parameters set to `WICED_FALSE`:

```
/* Disable pairing */
wiced_bt_set_pairable_mode( WICED_FALSE, WICED_FALSE );
```

The above will allow you to connect to your device and open the GATT database.

The following edits enable the device to respond to GATT read and write requests.

- Add the following case in `app_gatt_get_value()` to print the state of the LED to the UART (the switch is already in the template – you just need to add a new case). This event will occur whenever the Central reads the LED characteristic. Note that the code uses the GATT database value, not the state of the pin itself, and so non-zero implies “on” and zero means “off”. The name of the value array is `app_modus101_led`. It can be found in the GeneratedSource file `cycfg_gatt_db.c` which we will talk about it a minute.

```
// TODO Ex 01: Add code for any action required when this attribute is read
switch ( attr_handle )
{
    case HDLC_MODUS101_LED_VALUE:
        WICED_BT_TRACE( "LED is %s\r\n", app_modus101_led[0] ? "ON" :
            "OFF" );
        break;
}
```

- In `app_gatt_set_value()`, notice how the template function automatically updates the GATT database with a call to `memcpy()`. There is no need to write to the `app_modus101_led` array.

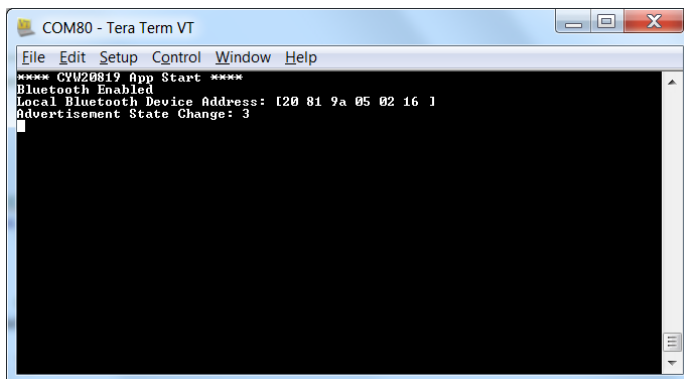
```
// Value fits within the supplied buffer; copy over the value
app_gatt_db_ext_attr_tbl[i].cur_len = len;
memcpy( app_gatt_db_ext_attr_tbl[i].p_data, p_val, len );
res = WICED_BT_GATT_SUCCESS;
```

7. Add the following case in `app_gatt_set_value()` to update the LED and printout the result. Again, the switch statement is in the template – just add the new case. This event will occur whenever the Central writes the LED characteristic. We are going to use LED_2 for this example. Note that the LEDs on the kit are active low so the pin is set to the NOT of the value.

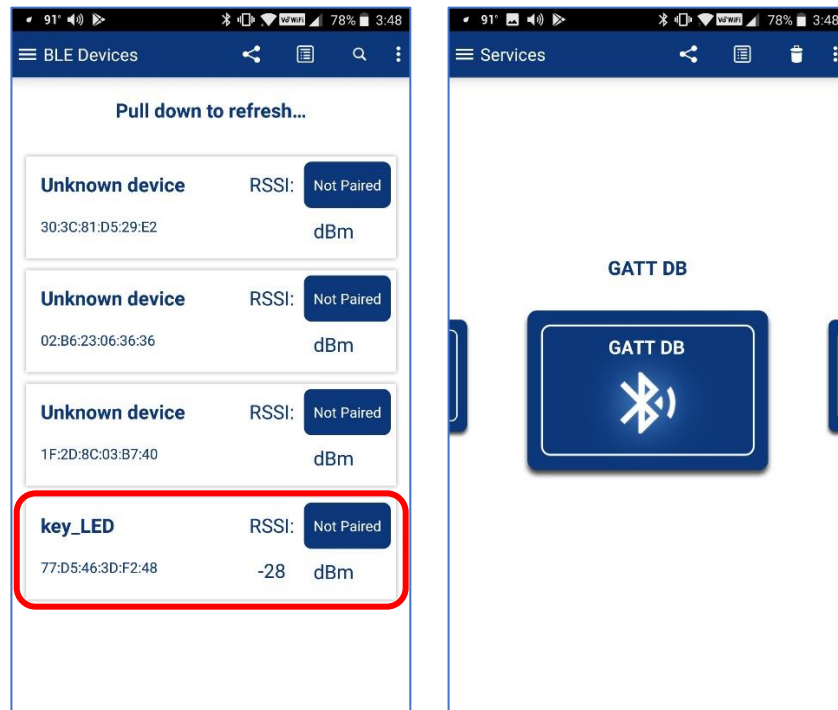
```
// TODO Ex01: Add code for any action required when this attribute is written
// For example, you may need to write the value into NVRAM if it needs to be
// persistent
switch ( attr_handle )
{
    case HDLC_MODUS101_LED_VALUE:
        wiced_hal_gpio_set_pin_output( WICED_GPIO_PIN_LED_2,
            app_modus101_led[0] == 0 );
        WICED_BT_TRACE( "Turn the LED %s\r\n", app_modus101_led[0] ? "ON"
            : "OFF" );
        break;
}
```

8.12.3 Testing the Application

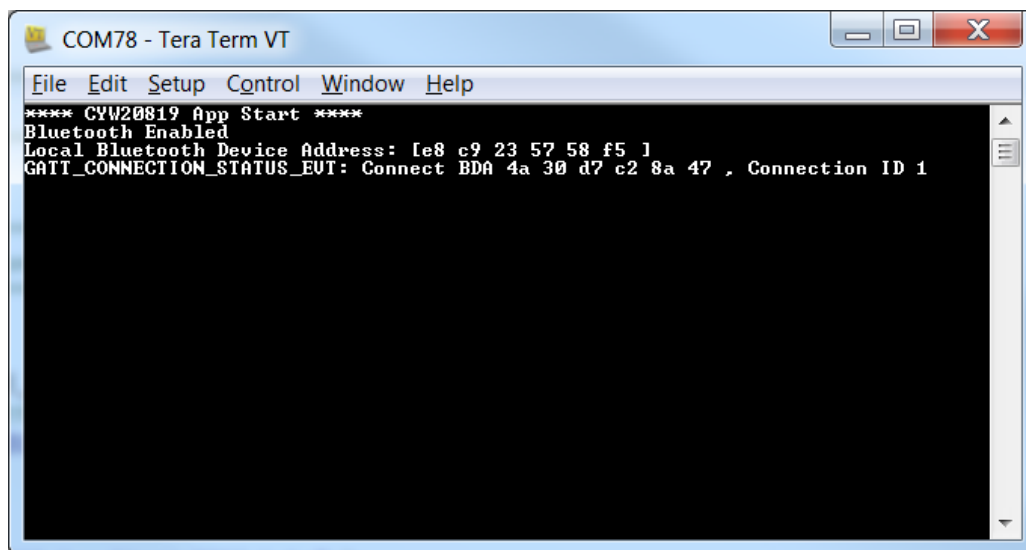
Start up a UART terminal (115200, 8, 1, N), then build and program your kit. When the application firmware starts up you see some messages.



Run CySmart on your phone (more details on CySmart later on). When you see the "<inits>_LED" device, tap on it. CySmart will connect to the device and will show the GATT browser widget.

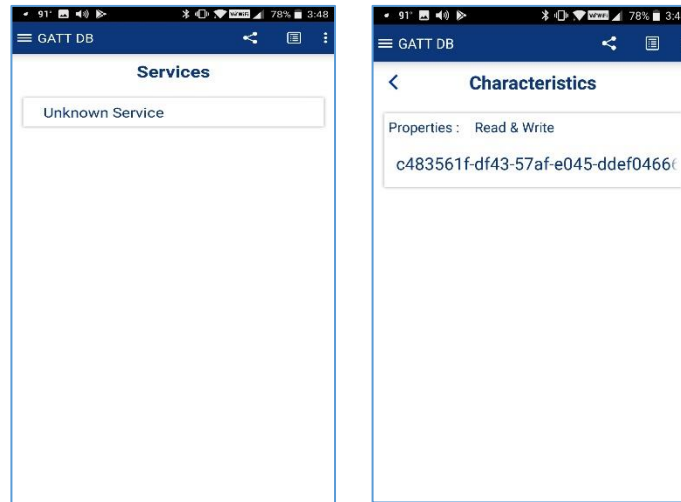


On the terminal window, you will see that there has been a connection and the advertising has stopped.

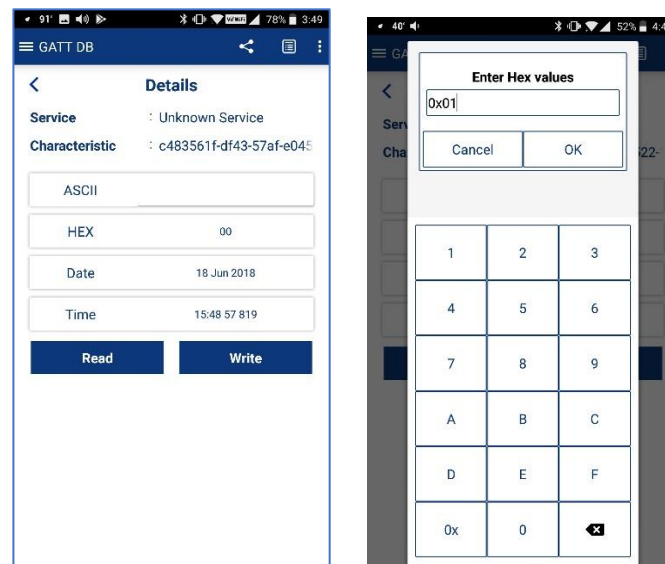


Back in CySmart, tap on the GATT DB widget to open the browser. You will see an Unknown Service (which I know is Modus101). Tap on the Service and CySmart will tell you that there is a Characteristic with the UUID shown (which I know is LED).

Note: In the iOS version of CySmart, the Characteristic UUID will not be shown – it will just say "Unknown Characteristic".



Tap on the Service to see details about it. First, tap the Read button and you will see that the current value is 0. Now you can Write 1s or 0's into the Characteristic and you will find that the LED turns on and off accordingly.



Finally press back until CySmart disconnects. When that happens, you will see the disconnect message in the terminal window.

8.13 WICED GATT Database Implementation

The Bluetooth Configurator automatically creates a GATT Database implementation to serve as a starting point. The database is split between `cycfg_gatt_db.c` and `cycfg_gatt_db.h`.

Even though the Bluetooth Configurator will create all of this for you, some understanding of how it is constructed is worthwhile knowing. The implementation is generic and will work for most situations, however you can make changes to handle custom situations.

When you start the Stack by calling `wiced_bt_stack_init` one of the parameters is a pointer to the GATT DB, meaning that the Stack will directly access your GATT DB for some purposes.

The GATT DB is used by both the Stack and by your application firmware. The Stack will directly access the Handles, UUIDs and Permissions of the Attributes to process some of the Bluetooth Events. Mainly the Stack will verify that a Handle exists and that the Client has Permission to access it before it gives your application a callback.

Your application firmware will use the GATT DB to read and write data in response to WICED BT Events.

The WICED Implementation of the GATT Database is simple generic "C" (obviously) and is composed logically of four parts. The first three are in `cycfg_gatt_db.c` while the last is implemented in the application code (in `app.c` in the template).

- An Array, named `gatt_database`, of `uint8_t` bytes that holds the Handles, Types and Permissions.
- An Array of Structs, named `app_gatt_db_ext_attr_tbl`, which holds Handles, a Maximum and Current Length and a Pointer to the actual Value.
- The Values as arrays of `uint8_t` bytes.
- Functions that serve as the API

8.13.1 `gatt_database[]`

The `gatt_database` is just an array of bytes with special meaning.

To create the bytes representing an Attribute there is a set of C-preprocessor macros that "do the right thing". To create Services, use the macros:

- `PRIMARY_SERVICE_UUID16(handle, service)`
- `PRIMARY_SERVICE_UUID128(handle, service)`
- `SECONDARY_SERVICE_UUID16(handle, service)`
- `SECONDARY_SERVICE_UUID128(handle, service)`
- `INCLUDE_SERVICE_UUID16(handle, service_handle, end_group_handle, service)`
- `INCLUDE_SERVICE_UUID128(handle, service_handle, end_group_handle)`

The handle parameter is just the Service Handle, which is a 16-bit number. The Bluetooth Configurator will automatically create Handles for you that will end up in the `cycfg_gatt_db.h` file. For example:

```
/* Service Generic Access */
#define HDLS_GAP 0x0001u
/* Service Generic Attribute */
#define HDLS_GATT 0x0006u
/* Service Modus101 */
#define HDLS_MODUS101 0x0007u
```

The Service parameter is the UUID of the service, just an array of bytes. The Bluetooth Configurator will create them for you in `cycfg_gatt_db.h`. For example:

```
#define __UUID_SERVICE_MODUS101 0xD5u, 0x8Eu, 0x79u, 0x8Bu, 0x2Cu, 0xDEu, 0x11u,
0x89u, 0x45u, 0x47u, 0x5Au, 0x31u, 0x6Au, 0xA3u, 0xFAu, 0x34u
```

In addition, there are a bunch of predefined UUIDs in `wiced_bt_uuid.h`.

To create Characteristics, use the following C-preprocessor macros which are defined in `wiced_bt_gatt.h`:

- `CHARACTERISTIC_UUID16(handle, handle_value, uuid, properties, permission)`
- `CHARACTERISTIC_UUID128(handle, handle_value, uuid, properties, permission)`
- `CHARACTERISTIC_UUID16_WRITABLE(handle, handle_value, uuid, properties, permission)`
- `CHARACTERISTIC_UUID128_WRITABLE(handle, handle_value, uuid, properties, permission)`

As before, the handle parameter is just the 16-bit number that the Bluetooth Configurator creates for the Characteristics which will be in the form of `#define HDLC_` for example:

```
/* Characteristic LED */
#define HDLC_MODUS101_LED 0x0008u
#define HDLC_MODUS101_LED_VALUE 0x0009u
```

The `_VALUE` parameter is the Handle of the Attribute that will hold the Characteristic's Value. That is, a Characteristic has (at least) two attributes: one to declare the Characteristic and one to hold its value. When you want to read/write the Characteristic, you have to use the handle for the Attribute containing the value, not the declaration.

The UUIDs are 16-bits or 128-bits in an array of bytes. The Bluetooth Configurator will create `#defines` for the UUIDs in the file `cycfg_gatt_db.h`.

Properties is a bit mask which sets the properties (i.e. Read, Write etc.) The bit mask is defined in `wiced_bt_gatt.h`:

```
/* GATT Characteristic Properties */
#define LEGATTDB_CHAR_PROP_BROADCAST (0x1 << 0)
#define LEGATTDB_CHAR_PROP_READ (0x1 << 1)
#define LEGATTDB_CHAR_PROP_WRITE_NO_RESPONSE (0x1 << 2)
#define LEGATTDB_CHAR_PROP_WRITE (0x1 << 3)
#define LEGATTDB_CHAR_PROP_NOTIFY (0x1 << 4)
#define LEGATTDB_CHAR_PROP_INDICATE (0x1 << 5)
#define LEGATTDB_CHAR_PROP_AUTHD_WRITES (0x1 << 6)
#define LEGATTDB_CHAR_PROP_EXTENDED (0x1 << 7)
```

The Permission field is just a bit mask that sets the Permission of an Attribute (remember Permissions are on a per Attribute basis and Properties are on a per Characteristic basis). They are also defined in `wiced_bt_gatt.h`.

```
/* The permission bits (see Vol 3, Part F, 3.3.1.1) */
#define LEGATTDB_PERM_NONE (0x00)
#define LEGATTDB_PERM_VARIABLE_LENGTH (0x1 << 0)
#define LEGATTDB_PERM_READABLE (0x1 << 1)
#define LEGATTDB_PERM_WRITE_CMD (0x1 << 2)
#define LEGATTDB_PERM_WRITE_REQ (0x1 << 3)
#define LEGATTDB_PERM_AUTH_READABLE (0x1 << 4)
#define LEGATTDB_PERM_RELIABLE_WRITE (0x1 << 5)
#define LEGATTDB_PERM_AUTH_WRITABLE (0x1 << 6)

#define LEGATTDB_PERM_WRITABLE (LEGATTDB_PERM_WRITE_CMD | LEGATTDB_PERM_WRITE_REQ |
LEGATTDB_PERM_AUTH_WRITABLE)
#define LEGATTDB_PERM_MASK (0x7f) /* All the
permission bits. */
#define LEGATTDB_PERM_SERVICE_UUID_128 (0x1 << 7)
```

8.13.2 gatt_db_ext_attr_tbl

The `gatt_database` array does not contain the actual values of Attributes. To find the values there is an array of structures of type `gatt_db_lookup_table`. Each structure contains a handle, a max length, actual length and a pointer to the array where the value is stored.

```
// External Lookup Table Entry
typedef struct
{
    uint16_t handle;
    uint16_t max_len;
    uint16_t cur_len;
    uint8_t *p_data;
} gatt_db_lookup_table;
```

Bluetooth Configurator will create this array for you automatically in `cycfg_gatt_db.c`:

```
/* *****
 * GATT Lookup Table
 * ***** */

gatt_db_lookup_table_t app_gatt_db_ext_attr_tbl[] =
{
    /* { attribute handle, maxlen, curlen, attribute data } */
    { HDLC_GAP_DEVICE_NAME_VALUE, 8, 8, app_gap_device_name },
    { HDLC_GAP_APPEARANCE_VALUE, 2, 2, app_gap_appearance },
    { HDLC_MODUS101_LED_VALUE, 1, 1, app_modus101_led },
};
```

The functions `app_gett_get_value` and `app_gatt_set_value` help you search through this array to find the pointer to the value.

8.13.3 uint8_t Arrays for the Values

Bluetooth Configurator will generate arrays of `uint8_t` to hold the values of writable/readable Attributes. You will find these values in a section of the code in `cycfg_gatt_db.c` marked with a comment "GATT Initial Value Arrays". In the example below, you can see there is a Characteristic with the name of the device, a Characteristic with the GAP appearance, and the LED Characteristic. These are the array names that you will use in your firmware to access a GATT database value. In the simple peripheral example, we used `app_modus101_led[0]` when we needed to know the value for the LED characteristic.

```
/* *****  
 * GATT Initial Value Arrays  
 * ***** */  
  
uint8_t app_gap_device_name[] = {'k', 'e', 'y', '_', 'L', 'E', 'D', '\0', };  
uint8_t app_gap_appearance[] = {0x00u, 0x00u, };  
uint8_t app_modus101_led[] = {0x00u, };
```

One thing that you should be aware of is the endianness. Bluetooth uses little endian, which is the same as ARM processors.

8.13.4 Application Programming Interface

There are two functions in our template which make up the interface to the GATT Database: `app_gatt_get_value` and `app_gatt_set_value`. Here are the function prototypes from the template code:

```
wiced_bt_gatt_status_t app_gatt_get_value( wiced_bt_gatt_attribute_request_t *p_attr );  
wiced_bt_gatt_status_t app_gatt_set_value( wiced_bt_gatt_attribute_request_t *p_attr );
```

These functions receive a pointer to the GATT attribute request structure. That structure contains, among other things, the attribute handle, a pointer to the value to be read/written, the length of the value to be written for writes, and a pointer to the length of the value received for reads.

Both functions loop through the GATT Database and look for an attribute handle that matches the input parameter. Then they memcpy the data into the right place, either saving it in the database, or writing into the buffer for the Stack to send back to the Client.

Both functions have a switch where you might put in custom code to do something based on the handle. This place is marked with `//TODO:` in the two functions.

You are supposed to return a `wiced_bt_gatt_status_t` which will tell the Stack what to do next. Assuming things work this function will return `WICED_BT_GATT_SUCCESS`. In the case of a Write this will tell the Stack to send a WRITE Response indicating success to the Client.

8.14 Notifications

In the previous example, I showed you how the GATT Client can Read and Write the GATT Database running on the GATT Server. But, there are cases where you might want the Server to initiate communication. For example, if your Server is a Peripheral device, you might want to send the Client an update each time a button value changes. That leaves us with the obvious questions of how does the Server initiate communication to the Client, and when is it allowed to do so?

The answer to the first question is, the Server can notify the Client that one of the values in the GATT Database has changed by sending a Notification message. That message has the Handle of the Characteristic that has changed and a new value for that Characteristic. Notification messages are not responded to by the Client, and as such are not reliable. If you need a reliable message, you can instead send an Indication which the Client must respond to.

To send a Notification or Indication:

- `wiced_bt_gatt_send_notification (conn_id, handle, length, value)`
- `wiced_bt_gatt_send_indication (conn_id, handle, length, value)`

By convention, the GATT Server will not send Notification or Indication messages unless they are turned on by the Client.

How do you turn on Notifications or Indications? In the last chapter, we talked about the GATT Attribute Database, specifically, the Characteristic. As stated previously, a Characteristic is composed of a minimum of two Attributes:

- Characteristic Declaration
- Characteristic Value

However, information about the Characteristic can be extended by adding more Attributes, which go by the name of Characteristic Descriptors.

For the Client to tell the Server that it wants to have Indications or Notifications, four things need to happen.

First, the Server must add a new Characteristic Descriptor Attribute called the Client Characteristic Configuration Descriptor, often called the CCCD. This Attribute is simply a 16-bit mask field, where bit 0 represents the Notification flag, and bit 1 represents the Indication flag. In other words, the Client can Write a 1 to bit 0 of the CCCD to tell the Server that it wants Notifications.

To add the CCCD to your GATT DB use the following macro (note that Bluetooth Configurator generates this code for you in `cycfg_gatt_db.c`):

```
CHAR_DESCRIPTOR_UUID16_WRITABLE (
    <HANDLE>,
    UUID_DESCRIPTOR_CLIENT_CHARACTERISTIC_CONFIGURATION,
    LEGATTDDB_PERM_READABLE | LEGATTDDB_PERM_WRITE_REQ | LEGATTDDB_PERM_AUTH_WRITABLE ),
```

The permissions above indicate that the CCCD value is readable whenever connected but will only be writable if the connection is authenticated (more on that later). To see the other possible choices, right click on one of them from inside ModusToolbox IDE and select "Open Declaration".

Second, you must change the Properties for the Characteristic to specify that the characteristic allows notifications. That is done by adding LEGATTDB_CHAR_PROP_NOTIFY to the Characteristic's Properties. To see all the available choices, right-click on one of the existing Properties in ModusToolbox IDE and select "Open Declaration".

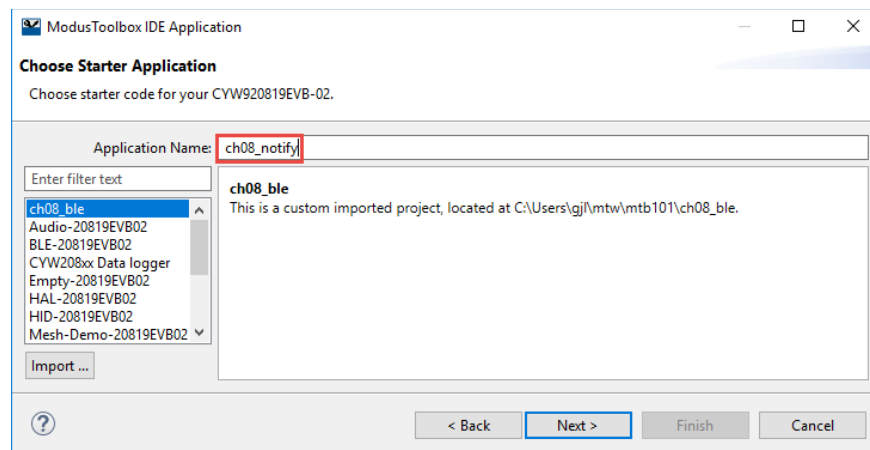
Third, in your GATT Attribute Write Callback you need to save the CCCD value that was written to you (note that this is done automatically in `app_gatt_set_value()` because the CCCD is writable).

Finally, when a value that has Notify and/or Indicate enabled changes in your system, you must send out a new value using the appropriate API.

8.15 Notification Demo Walkthrough

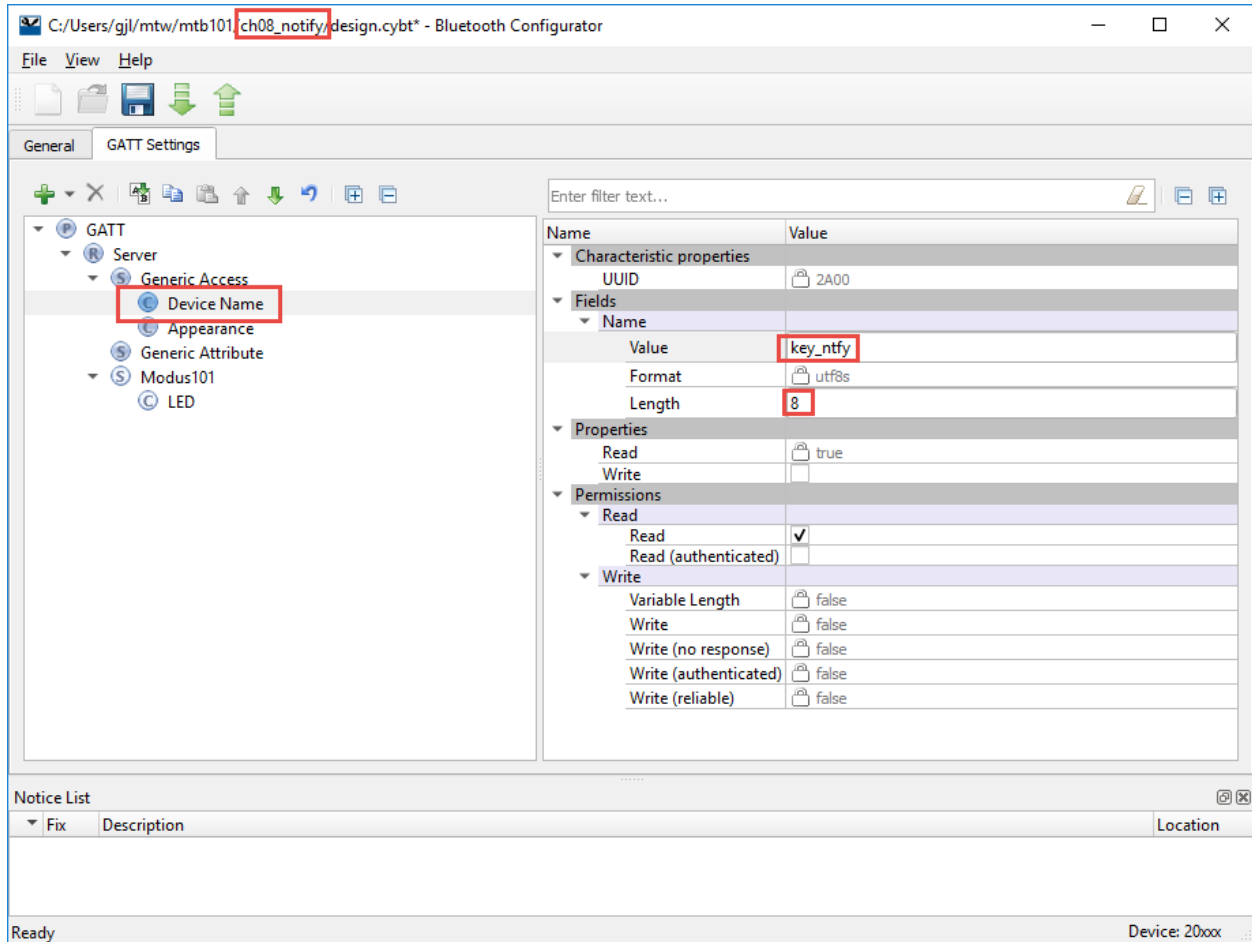
Next, we will add notifications to the previous exercise. We will add a new Characteristic called "Counter" that will count how many times the user button has been pressed since reset. That Characteristic will have Read and Notify properties set so that you can read the value or register to be notified any time the value changes.

The first thing I will do is import the previously completed exercise (not the template) into a new one using the New Application Wizard Import function and I'll call the new application `ch08_notify`.



8.15.1 Running the Bluetooth Configurator

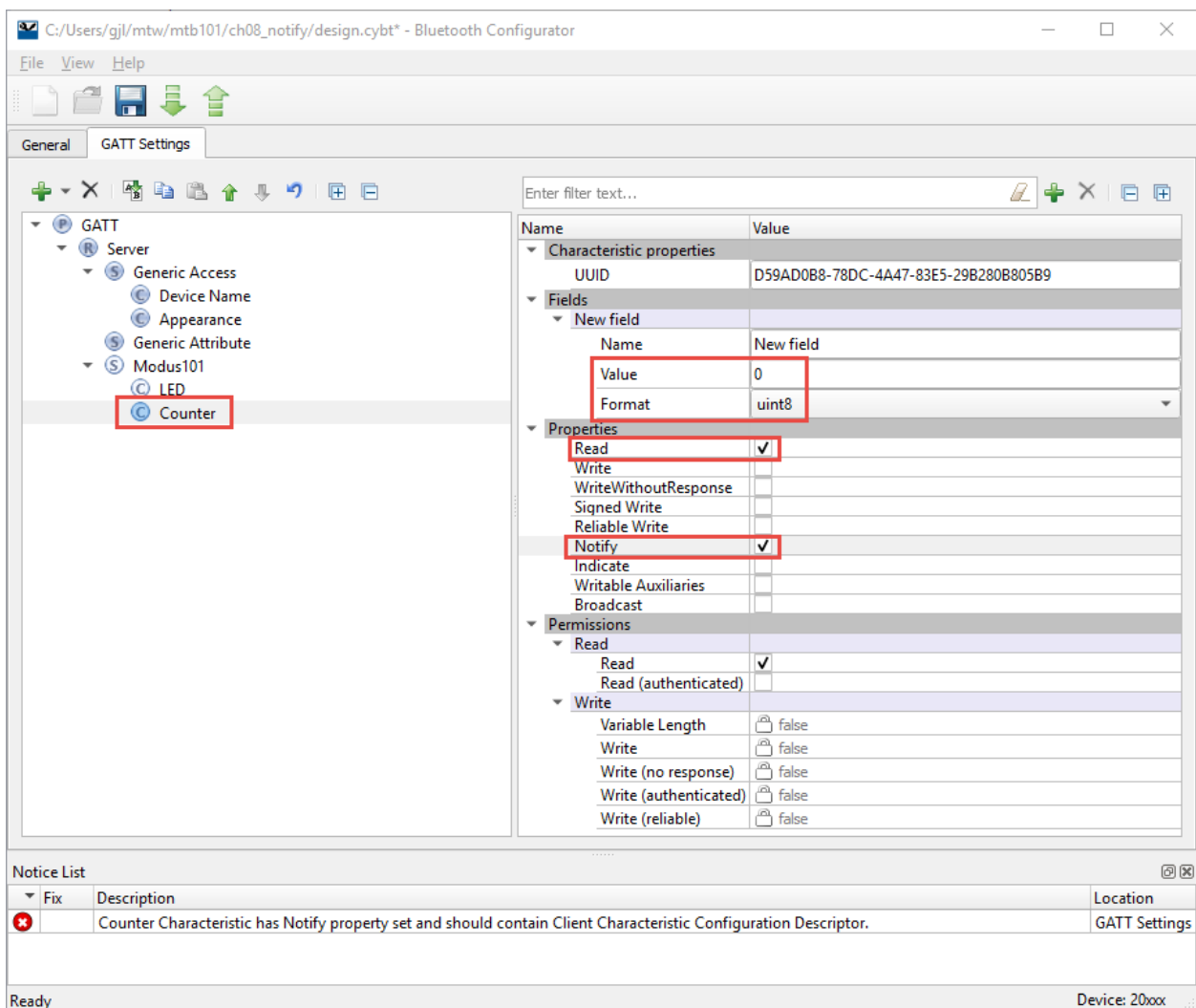
Start the Bluetooth Configurator by clicking on the link in the Quick Panel or by double clicking on design.cybt in the new project. Make sure you open the one from the ch08_notify project. In the Generic Access service change the Device Name. I'll use a device name of "key_ntfy". **When you do this yourself, use a unique name such as <inits>_ntfy where <inits> is your initials.** Otherwise you will have trouble finding your specific device among all the ones that are advertising.



Add a new Custom Characteristic to the "Modus101" Service and rename it to "Counter", give it the type uint8 and initial value of 0.

Under Properties enable Read. Now look in the Permission section. It was set by the tool to Read based on our Properties selections. This means that we will be able to Read the Characteristic value without Pairing first. In real-world applications you would most likely also turn on Read (authenticated) so that Read will require an Authenticated (i.e. Paired) link but we shall handle pairing later.


Back under Properties, enable Notify so that the peripheral will be able to tell us when Counter value changes. Note that enabling notifications generates an error because you have not yet made notifications possible. The message tells you to add a CCCD (Client Characteristic Configuration Descriptor), which will shall do next.



The screenshot shows the Bluetooth Configurator interface with the GATT Settings tab selected. The left pane shows the GATT tree with the 'Counter' characteristic selected under the 'Modus101' service. The right pane displays the properties for the 'Counter' characteristic.

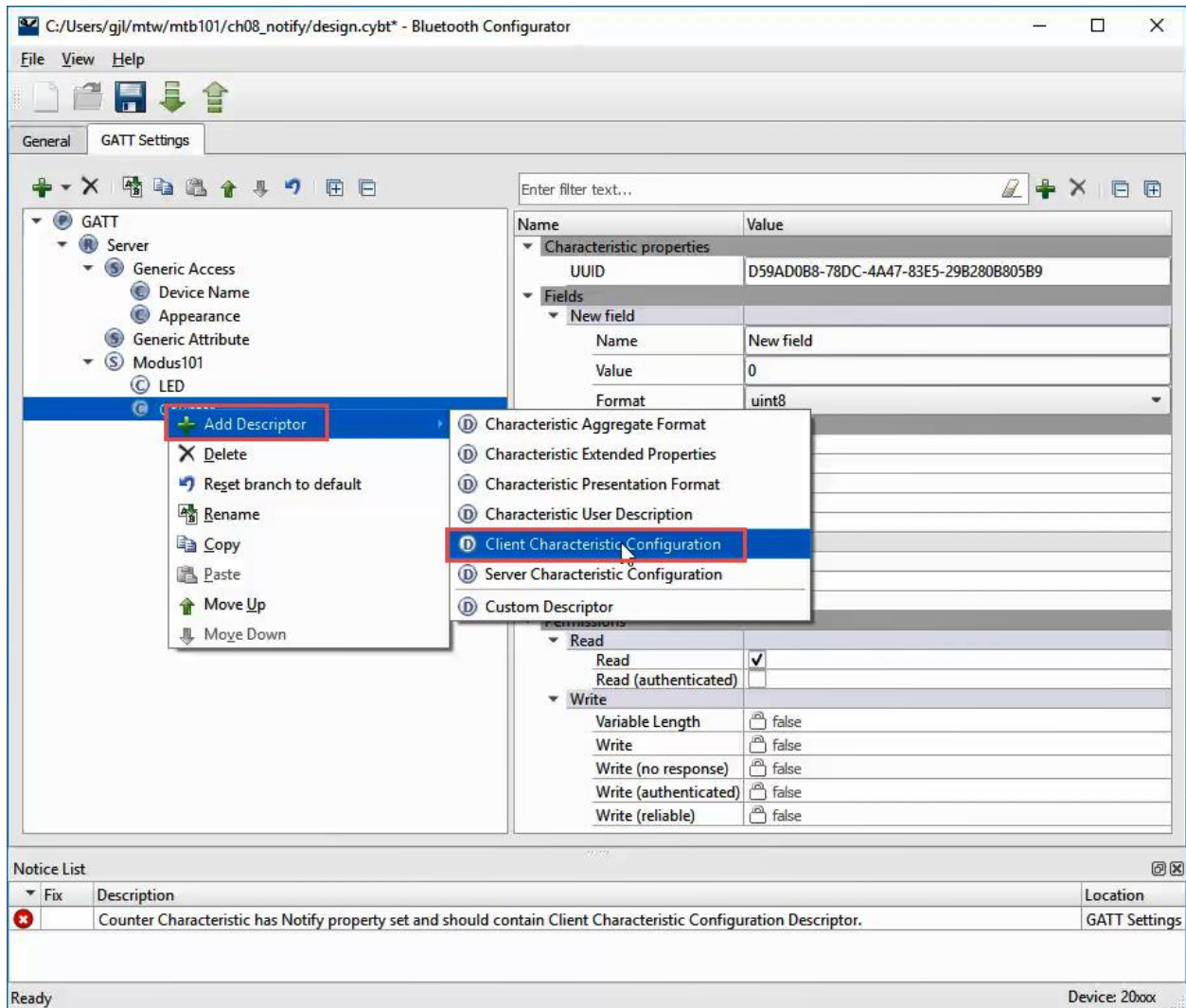
| Name | Value |
|----------------------------------|--------------------------------------|
| Characteristic properties | |
| UUID | D59AD0B8-78DC-4A47-83E5-29B280B805B9 |
| Fields | |
| New field | |
| Name | New field |
| Value | 0 |
| Format | uint8 |
| Properties | |
| Read | <input checked="" type="checkbox"/> |
| Write | <input type="checkbox"/> |
| WriteWithoutResponse | <input type="checkbox"/> |
| Signed Write | <input type="checkbox"/> |
| Reliable Write | <input type="checkbox"/> |
| Notify | <input checked="" type="checkbox"/> |
| Indicate | <input type="checkbox"/> |
| Writable Auxiliaries | <input type="checkbox"/> |
| Broadcast | <input type="checkbox"/> |
| Permissions | |
| Read | |
| Read | <input checked="" type="checkbox"/> |
| Read (authenticated) | <input type="checkbox"/> |
| Write | |
| Variable Length | <input type="checkbox"/> false |
| Write | <input type="checkbox"/> false |
| Write (no response) | <input type="checkbox"/> false |
| Write (authenticated) | <input type="checkbox"/> false |
| Write (reliable) | <input type="checkbox"/> false |

The bottom pane shows a Notice List with the following message:

| Fix | Description | Location |
|---|---|---------------|
|  | Counter Characteristic has Notify property set and should contain Client Characteristic Configuration Descriptor. | GATT Settings |

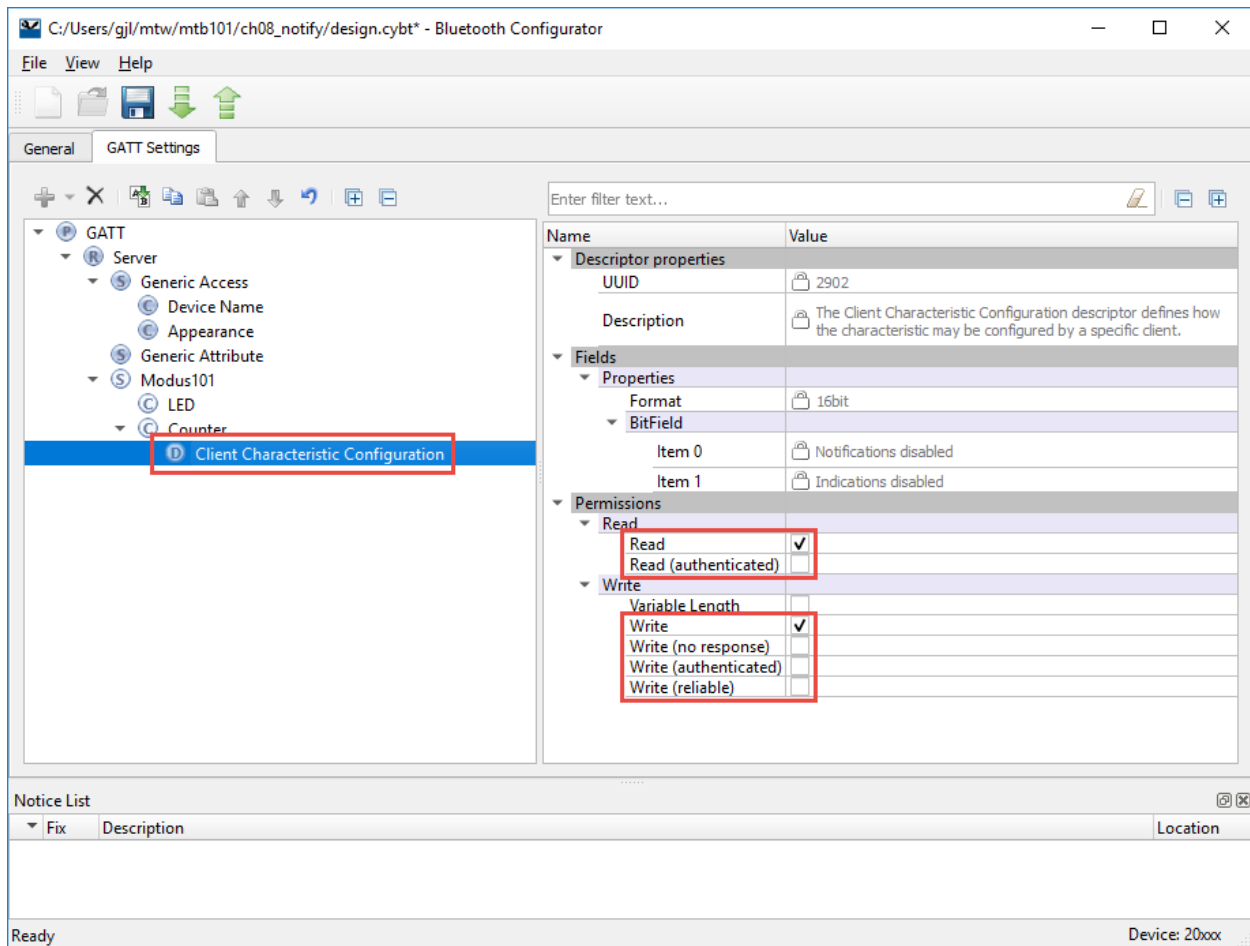
The status bar at the bottom indicates 'Ready' and 'Device: 20xx'.

Add a CCCD by right-clicking on Counter, then Add Descriptor, and choose *Client Characteristic Configuration*.



The new characteristic sets Notifications and Indications to disabled by default (and you cannot change that from the tool – if you want to enable them automatically it is better practice to enable them during pairing). Make sure Write permission is set on the CCCD so that you will be able to set it from CySmart. Note that the error message has gone away.

Make sure the Write (authenticated) permission checkbox is not set because we are not (yet) requiring the devices to pair before enabling notifications.



Finally, save your edits and close the Bluetooth Configurator.

8.15.2 Editing the Firmware

In app.c, we need to make the following changes:

1. Declare a global variable called `connection_id`. Upon a GATT connection (i.e. in `app_gatt_callback`), save the connection ID. Upon a GATT disconnection, reset the connection ID. The ID is needed to send a notification – you need to tell it which connected device to send the notification to. In our case we only allow one connection at a time but there are devices that allow multiple connections.

Global Variable:

```
uint16_t connection_id = 0;
```

GATT Connection:

```
/* Handle the connection */  
connection_id = p_conn->conn_id;
```

GATT Disconnection:

```
/* Handle the disconnection */  
connection_id = 0;
```

2. Configure `BUTTON_1` as a falling edge interrupt under the `BTM_ENABLED_EVT`.

```
/* Configure BUTTON_1 for a falling edge interrupt */  
wiced_hal_gpio_configure_pin(  
    WICED_GPIO_PIN_BUTTON_1,  
    ( GPIO_INPUT_ENABLE | GPIO_PULL_UP | GPIO_EN_INT_FALLING_EDGE ),  
    GPIO_PIN_OUTPUT_HIGH );  
wiced_hal_gpio_register_pin_for_interrupt(  
    WICED_GPIO_PIN_BUTTON_1,  
    button_cback,  
    0 );
```

3. Create the button callback function. In the callback we will increment the Button Characteristic value, and then send a notification if we have a connection and the notification is enabled.

Note that the array `app_modus_counter` was created by the Bluetooth Configurator. It holds the value for our counter characteristic. The name that the configurator uses is of the form: `app_<service_name>_<characteristic_name>`. The button callback function will look like this:

```
/* Button interrupt callback function */
void button_cback( void *data, uint8_t port_pin )
{
    app_modus101_counter[0]++;

    if( connection_id )
    {
        if( app_modus101_counter_client_char_config[0] &
            GATT_CLIENT_CONFIG_NOTIFICATION )
        {
            WICED_BT_TRACE( "Notifying counter change (%d)\r\n",
                app_modus101_counter[0] );
            wiced_bt_gatt_send_notification(
                connection_id,
                HDLC_MODUS101_COUNTER_VALUE,
                app_modus101_counter_len,
                app_modus101_counter );
        }
    }

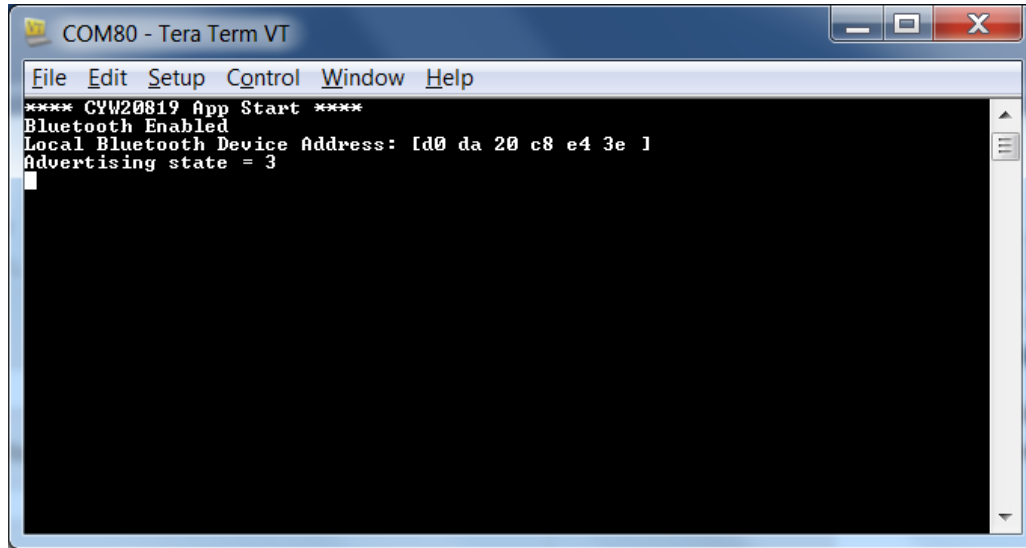
    /* Clear the GPIO interrupt */
    wiced_hal_gpio_clear_pin_interrupt_status( WICED_GPIO_PIN_BUTTON_1 );
}
```

4. Add a debug message to in `app_gatt_set_value()` so you know when notifications get enabled/disabled. Note that the switch statement is already there but you need to add a new case.

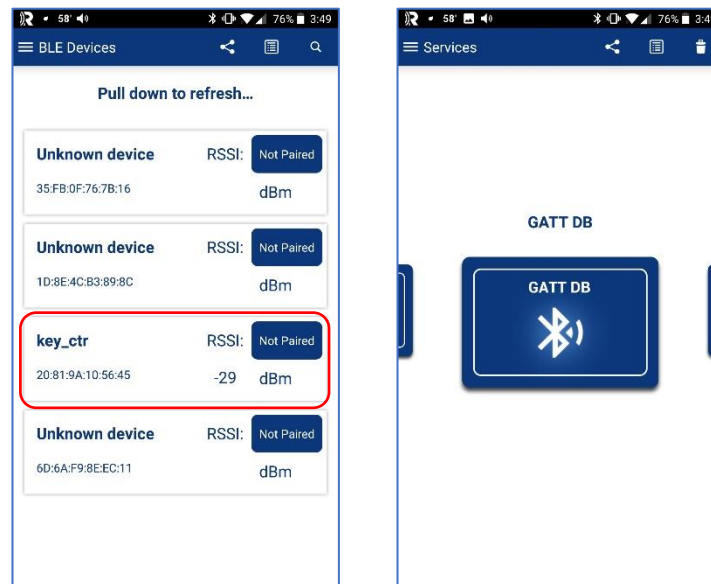
```
switch( attr_handle )
{
    case HDLD_MODUS101_COUNTER_CLIENT_CHAR_CONFIG:
        WICED_BT_TRACE( "Setting notify (0x%02x, 0x%02x)\r\n",
            p_val[0], p_val[1] );
        break;
}
```

8.15.3 Testing the Application

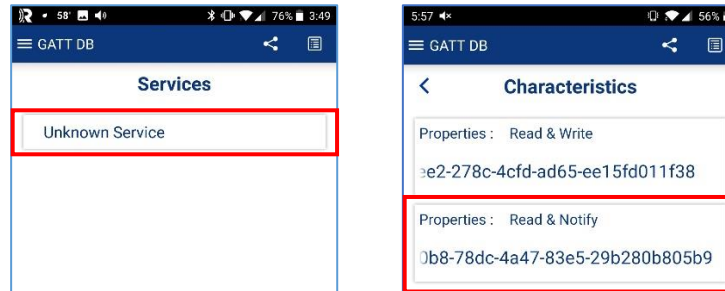
Start up a UART terminal to the WICED Peripheral UART port with a baud of 115200 and then program the kit. When the firmware starts up you will see some messages.



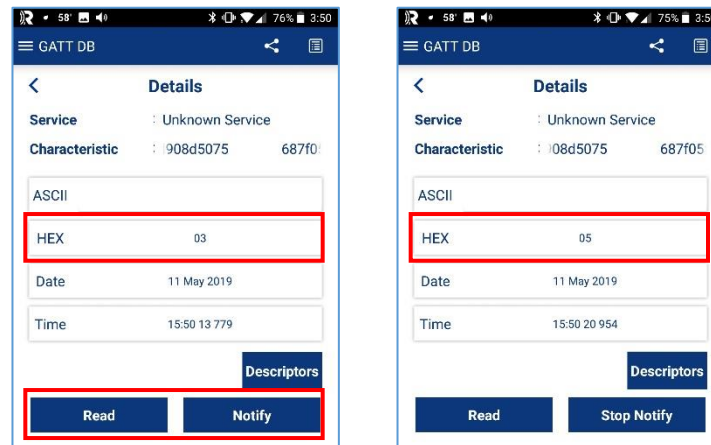
Run CySmart on your phone. When you see the "<inits>_ctr" device, tap on it. CySmart will connect to the device and will show the GATT browser widget.



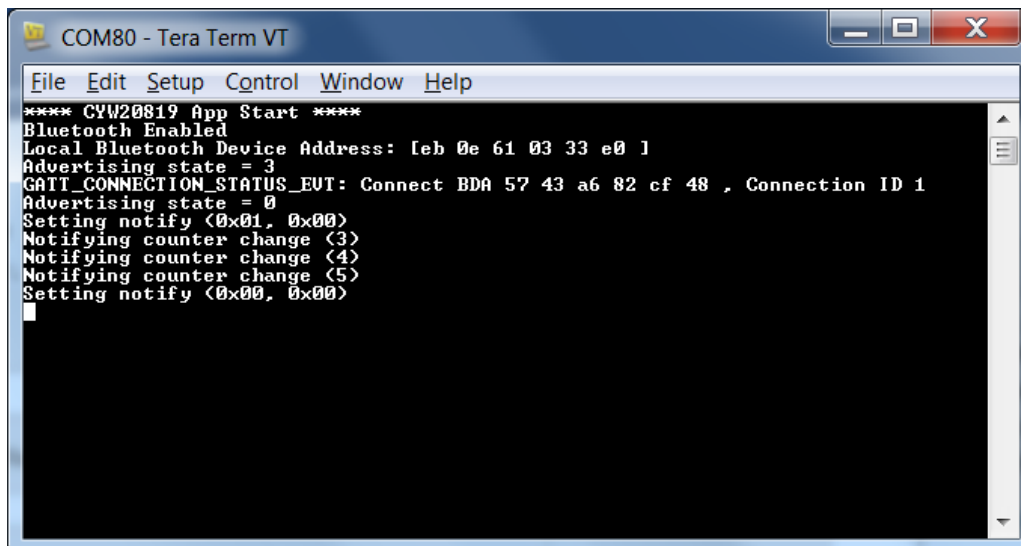
Tap on the GATT DB widget to open the browser. Then tap on the Unknown Service (which we know is Modus101) and then on the Characteristic that has Read and Notify Properties (which we know is Counter).



Tap the Read button to read the value. Press the button on the kit a few times and then Read again to see the incremented value. Then tap the Notify button to enable notifications. Now each time you press the button the value is shown automatically.



While playing with CySmart you will see messages like this in the terminal emulator:



8.16 Security

To securely communicate between two devices, you want to: (1) Authenticate that both sides know who they are talking to; (2) ensure that all access to data is Authorized; (3) Encrypt all message that are transmitted; (4) verify the Integrity of those messages; and (5) ensure that the Identity of each side is hidden from eavesdroppers.

In BLE, this entire security framework is built around AES-128 symmetric key encryption. This type of encryption works by combining a Shared Secret code and the unencrypted data (typically called plain text) to create an encrypted message (typically called cypher text).

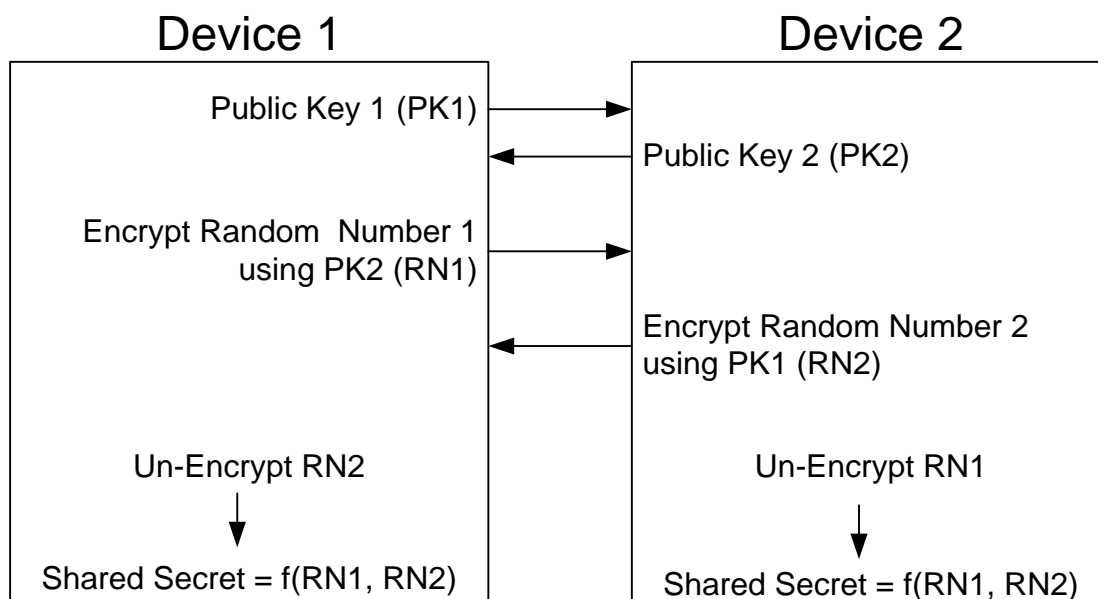
- $CypherText = F(SharedSecret, PlainText)$

There is a bunch of math that goes into AES-128, but for all practical purposes if the Shared Secret code is kept secret, you can assume that it is very unlikely that someone can read the original message.

If this scheme depends on a Shared Secret, the next question is how do two devices that have never been connected get a Shared Secret that no one else can see? In BLE, the process for achieving this state is called Pairing. A device that is Paired is said to be Authenticated.

8.16.1 Pairing

Pairing is the process of arriving at the Shared Secret. The basic problem continues to be how do you send a Shared Secret over the air, unencrypted and still have your Shared Secret be Secret. The answer is that you use public key encryption. Both sides have a public/private key pair that is either embedded in the device or calculated at startup. When you want to authenticate, both sides of the connection exchange public keys. Then both sides exchange encrypted random numbers that form the basis of the shared secret.



But how do you protect against Man-In-The-Middle (MIM)? There are four possible methods.

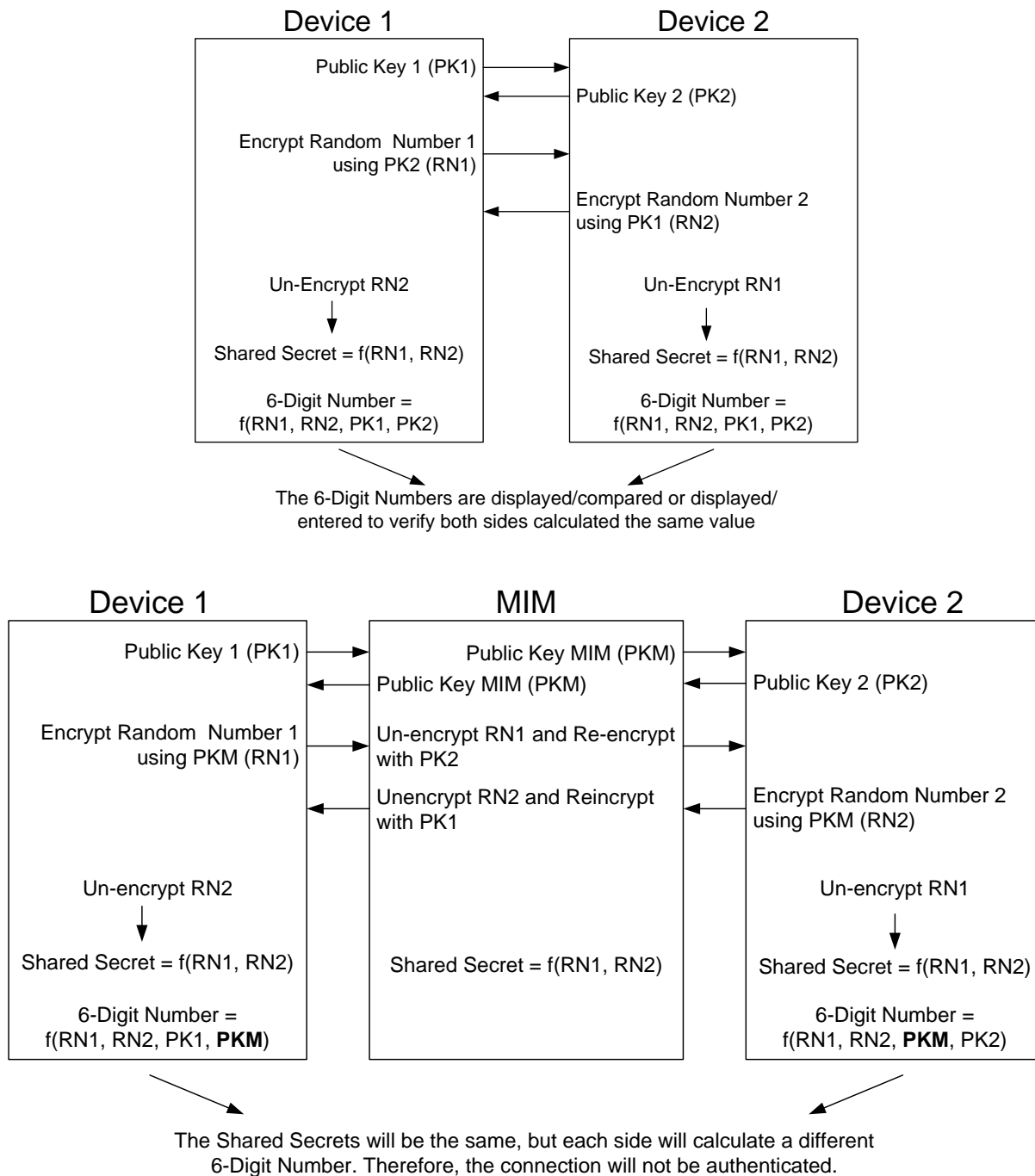
Method 1 is called "Just works". In this mode you have no protection against MIM.

Method 2 is called "Out of Band". Both sides of the connection need to be able to share the PIN via some other connection that is not Bluetooth such as NFC.

Method 3 is called "Numeric Comparison" (V2.PH.7.2.1). In this method, both sides display a 6-digit number that is calculated with a nasty cryptographic function based on the random numbers used to generate the shared key and the public keys of each side. The user observes both devices. If the number is the same on both, then the user confirms on one or both sides. If there is a MITM, then the random numbers on both sides will be different so the 6-digit codes would not match.

Method 4 is called "Passkey Entry" (V2.PH.7.2.3). For this method to work, at least one side needs to be able to enter a 6-digit Passkey. The other side must be able to display the Passkey. One device displays the Passkey and the user is required to enter the Passkey on the other device. Then an exchange and comparison process happens with the Passkeys being divided up, encrypted, exchanged and compared.

Pictorially, the process with no MIM and with MIM is shown below. Note that if there is a man in the middle, the two sides will calculate different numbers because the number is a function of the public keys used to encrypt the random numbers. If both sides used the same two public keys, then there can't be a man in the middle.

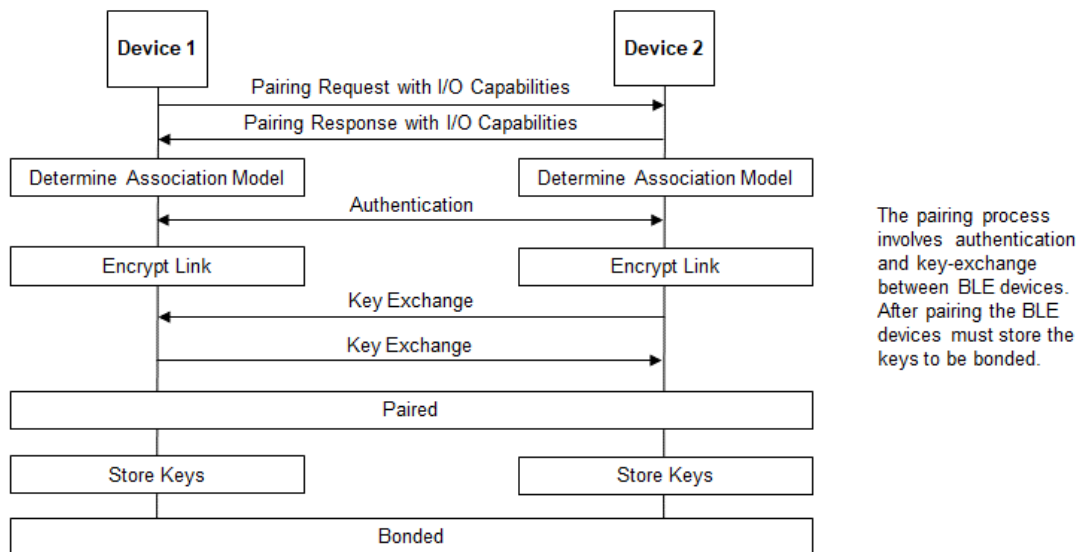


8.16.2 Bonding

The whole process of Pairing is a bit painful and time consuming. It is also the most vulnerable part of establishing security, so it is beneficial to do it only once. Certainly, you don't want to have to repeat it every time two devices connect. This problem is solved by Bonding, which just saves all the relevant information into a non-volatile memory. This allows the next connection to launch without repeating the pairing process.

8.16.3 Pairing & Bonding Process Summary

BLE Pairing And Bonding Procedure



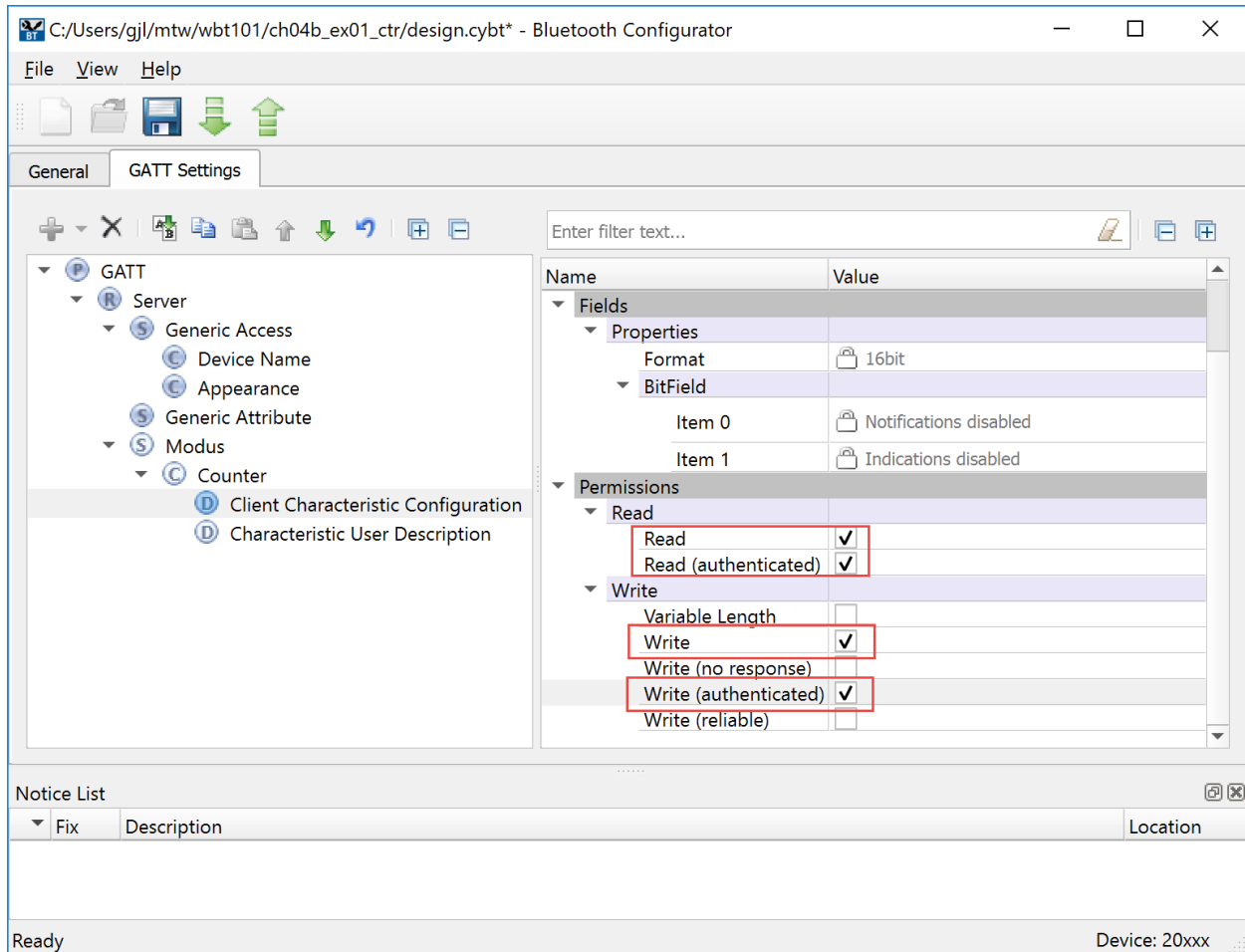
8.16.4 Authentication, Authorization and the GATT DB

In Chapter 4A3.1 we talked about the Attributes and the GATT Database. Each Attribute has a permissions bit field that includes bits for Encryption, Authentication, and Authorization. The WICED Bluetooth Stack will guarantee that you will not be able to access an Attribute that is marked Encryption or Authentication unless the connection is Authenticated and/or Encrypted.

The Authorization flag is not enforced by the WICED Bluetooth Stack. Your Application is responsible for implementing the Authorization semantics. For example, you might not allow someone to turn off/on a switch without entering a password.

8.16.5 Security in the Bluetooth Configurator

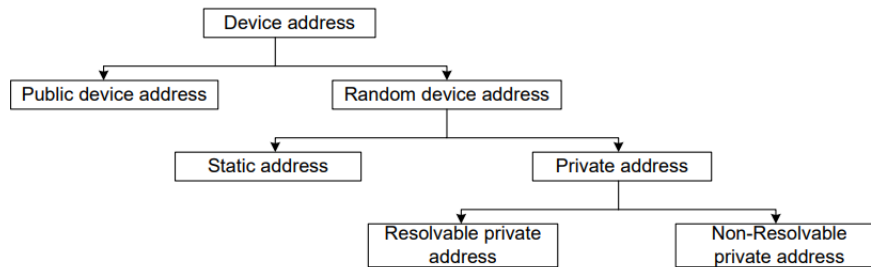
In order to enable security (i.e. to require pairing before allowing the client read or write of a characteristic) you just click the "Read (authenticated)" or "Write (authenticated)" button in the permissions for that characteristic. Note that this is a bitmask setting, so you must still keep "Read" and/or "Write" selected when you enable authentication. If not, reads/writes to that characteristic will fail. Enabling security works the same way for Descriptors such as the CCCD. That is, you can require authentication before allowing reads/writes of the CCCD (thereby preventing the client from turning Notifications on/off without pairing first) from the previous example like this:



8.16.6 Link Layer Privacy

BLE devices are identified using a 48-bit device address. This device address is part of all the packets sent by the device in the advertising channels. A third device which listens on all three advertising channels can easily track the activities of a device by using its device address. Link Layer Privacy is a feature that reduces the ability to track a BLE device by using a private address that is generated and changed at regular intervals. Note that this is different than security (i.e. encrypting of messages).

There are a few different types of address types possible for BLE devices:



The device address can be a Public Device Address or a Random Device Address. The Public Device Addresses are comprised of a 24-bit company ID (an Organizationally Unique Identifier or OUI based on an IEEE standard) and a 24-bit company-assigned number (unique for each device); these addresses do not change over time.

There are two types of Random Addresses: Static Address and Private Address. The Static Address is a 48-bit randomly generated address with the two most significant bits set to 1. Static Addresses are generated on first power up or during manufacturing. A device using a Public Device Address or Static Address can be easily discovered and connected to by a peer device. Private Addresses change at some interval to ensure that the BLE device cannot be tracked. A Non-Resolvable Private Address cannot be resolved by any device so the peer cannot identify who it is connecting to. Resolvable Private Addresses (RPA) can be resolved and are used by Privacy-enabled devices.

Every Privacy-enabled BLE device has a unique address called the Identity Address and an Identity Resolving Key (IRK). The Identity Address is the Public Address or Static Address of the BLE device. The IRK is used by the BLE device to generate its RPA and is used by peer devices to resolve the RPA of the BLE device. Both the Identity Address and the IRK are exchanged during the third stage of the pairing process. Privacy-enabled BLE devices maintain a list that consists of the peer device's Identity Address, the local IRK used by the BLE device to generate its RPA, and the peer device's IRK used to resolve the peer device's RPA. This is called the Resolving List. Only peer devices that have the 128-bit identity resolving key (IRK) of a BLE device can determine the device's address.

A Privacy-enabled BLE device periodically changes its RPA to avoid tracking. The BLE Stack configures the Link Layer with a value called RPA Timeout that specifies the time after which the Link Layer must generate a new RPA. In ModusToolbox, this value is set in `app_bt_cfg.c` and is called `rpa_refresh_timeout`. If the `rpa_refresh_timeout` is set to 0 (i.e. `WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_NEVER_CHANGE`), privacy is disabled, and a public device address will be used.

8.17 Firmware Architecture for Security

The firmware architecture is the same as was described in the previous chapter. The only difference is that there are additional Stack Management events and GATT Database events that occur.

For a typical BLE application that connects using a Paired link but does NOT use privacy, does NOT store bonding information in NVRAM and does NOT require a passkey, the order of callback events will look like this:

| Activity | Callback Event Name (both Stack and GATT) | Reason |
|--------------------------------------|---|---|
| Powerup | BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT | At initialization, the BLE stack looks to see if the privacy keys are available. If privacy is not enabled, then this state does not need to be implemented as long as you return a default value of WICED_BT_SUCCESS. |
| | BTM_ENABLED_EVT | This occurs once the BLE stack has completed initialization. Typically, you will start up the rest of your application here. |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT | This occurs when you enable advertisements. You will see a return value of 3 for fast advertisements. After a timeout, you may see this again with a return value of 4 for slow advertisements. Eventually the state changes to 0 (off) if there have been no connections, giving you a chance to save power. |
| Connect | GATT_CONNECTION_STATUS_EVT | The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required). |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT | Once the connection happens, the stack stops advertisements which will result in this event. You will see a return value of 0 which means advertisements have stopped. |
| Pair (if secure link is required) | BTM_SECURITY_REQUEST_EVT | The occurs when the client requests a secure connection. When this event happens, you need to call wiced_bt_ble_security_grant() to allow a secure connection to be established. |
| | BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT | This occurs when the client asks what type of capability your device has that will allow validation of the connection (e.g. screen, keyboard, etc.). You need to set the appropriate values when this event happens. |
| | BTM_ENCRYPTION_STATUS_EVT | This occurs when the secure link has been established. |
| | BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT | This event is used so that you can store the paired devices keys if you are storing bonding information. If not, then this state does not need to be implemented. |
| | BTM_PAIRING_COMPLETE_EVT | This event indicates that pairing has been completed successfully. |
| Read Values | GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ | The firmware must get the value from the correct location in the GATT database. |
| Write Values | GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE | The firmware must store the provided value in the correct location in the GATT database. |

| Activity | Callback Event Name (both Stack and GATT) | Reason |
|---------------|---|--|
| Notifications | N/A | Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process. |
| Disconnect | GATT_CONNECTION_STATUS_EVT | For a disconnection, the connection ID is reset, all CCCD settings are cleared, and advertisements are restarted. |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT | Upon a disconnect, the firmware will get a GATT event handler callback for the GATT_CONNECTION_STATUS_EVENT (more on this later). At that time, it is the user's responsibility to determine if advertising should be re-started. If it is restarted, then you will get a BLE stack callback once advertisements have restarted with a return value of 3 (fast advertising) or 4 (slow advertising). |

If bonding information is stored to NVRAM, the event sequence will look like the following. The sequence is shown for three cases (each shaded differently):

1. First-time connection before bonding information is saved
2. Connection after bonding information has been saved for disconnect/re-connect without resetting the kit between connections.
3. Connection after bonding information has been saved for disconnect/reset/re-connect.

In the reconnect cases, you can see that the pairing sequence is greatly reduced since keys are already available.

| Activity | Callback Event Name | Reason |
|-------------------------|-------------------------------------|---|
| 1 st Powerup | BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT | When this event occurs, the firmware needs to load the privacy keys from NVRAM. If keys have not been previously saved for the device, then this state must return a value other than WICED_BT_SUCCESS such as WICED_BT_ERROR. The non-success return value causes the stack to generate new privacy keys. |
| | BTM_ENABLED_EVT | This occurs once the BLE stack has completed initialization. Typically, you will start up the rest of your application here. During this event, the firmware needs to load keys (which also includes the BD_ADDR) for a previously bonded device from NVRAM and then call <i>wiced_bt_dev_add_device_to_address_resolution_db()</i> to allow connecting to an bonded device. If a device has not been previously bonded, this will return values of all 0. |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT | This occurs when you enable advertisements. You will see a return value of 3 for fast advertisements. After a timeout, you may see this again with a return value of 4 for slow advertisements. Eventually the state changes to 0 (off) if there have been no connections, giving you a chance to save power. |
| | BTM_LOCAL_IDENTITY_KEYS_UPDATE_EVT | This event is called if reading of the privacy keys from NVRAM failed (i.e. the return value from BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT was not 0). During this event, the privacy keys must be saved to NVRAM. |
| | BTM_LOCAL_IDENTITY_KEYS_UPDATE_EVT | This is called twice to update both the IRK and the ER in two steps. |
| 1 st Connect | GATT_CONNECTION_STATUS_EVT | The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required). |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT | Once the connection happens, the stack stops advertisements which will result in this event. You will see a return value of 0 which means advertisements have stopped. |
| 1 st Pair | BTM_SECURITY_REQUEST_EVT | The occurs when the client requests a secure connection. When this event happens, you need to call <i>wiced_bt_ble_security_grant()</i> to allow a secure connection to be established. |

| Activity | Callback Event Name | Reason |
|---------------|---|--|
| | BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT | This occurs when the client asks what type of capability your device has that will allow validation of the connection (e.g. screen, keyboard, etc.). You need to set the appropriate values when this event happens. |
| | BTM_PASSKEY_NOTIFICATION_EVT | This event only occurs if the IO capabilities are set such that your device has the capability to display a value, such as BTM_IO_CAPABILITIES_DISPLAY_ONLY. In this event, the firmware should display the passkey so that it can be entered on the client to validate the connection. |
| | BTM_USER_CONFIRMATION_REQUEST_EVT | This event only occurs if the IO capabilities are set such that your device has the capability to display a value and accept Yes/No input, such as BTM_IO_CAPABILITIES_DISPLAY_AND_YES_NO_INPUT. In this event, the firmware should display the passkey so that it can be compared with the value displayed on the Client. This state should also provide confirmation to the Stack (either with or without user input first). |
| | BTM_ENCRYPTION_STATUS_EVT | This occurs when the secure link has been established. Previously saved information such as paired device BD_ADDR and notify settings is read. If no device has been previously bonded, this will return all 0's. |
| | BTM_PAIRING_COMPLETE_EVT | During this event, the firmware needs to store the keys of the paired device (including the BD_ADDR) into NVRAM so that they are available for the next time the devices connect. |
| | BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT | This event indicates that pairing has been completed successfully. Information about the paired device such as its BT_ADDR should be saved in NVRAM at this point. You may also initialize other state information to be saved such as notify settings. |
| Read Values | GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ | The firmware must get the value from the correct location in the GATT database. |
| Write Values | GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE | The firmware must store the provided value in the correct location in the GATT database. |
| Notifications | N/A | Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process. |
| Disconnect | BTM_BLE_ADVERT_STATE_CHANGED_EVT | Upon a disconnect, the firmware will get a GATT event handler callback for the GATT_CONNECTION_STATUS_EVENT (more on this later). At that time, it is the user's responsibility to determine if advertising should be re-started. If it is restarted, then you will get a BLE stack callback once advertisements have restarted with a return value of 3 (fast advertising) or 4 (slow advertising). |
| Re-Connect | GATT_CONNECTION_STATUS_EVT | The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required). |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT | Advertising off. |

| Activity | Callback Event Name | Reason |
|---------------|---|---|
| Re-Pair | BTM_ENCRYPTION_STATUS_EVT | <p>In this state, the firmware reads the state of the server from NVRAM. For example, the BD_ADDR of the paired device and the saved state of any notify settings may be read.</p> <p>Since the paired device BD_ADDR and keys were already available, no other steps are needed to complete pairing.</p> |
| Read Values | GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ | The firmware must get the value from the correct location in the GATT database. |
| Write Values | GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE | The firmware must store the provided value in the correct location in the GATT database. |
| Notifications | N/A | Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process. |
| Disconnect | BTM_BLE_ADVERT_STATE_CHANGED_EVT | Advertising on. |
| Reset | BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT | Local keys are loaded from NVRAM. |
| | BTM_ENABLED_EVT | Stack is enabled. Paired device keys (including the BD_ADDR) are loaded from NVRAM and the device is added to the address resolution database. |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT | Advertising on. |
| Re-Connect | GATT_CONNECTION_STATUS_EVT | The callback needs to determine if the event is a connection or a disconnection. For a connection, the connection ID is saved, and pairing is enabled (if a secure link is required). |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT | Advertising off. |
| Re-Pair | BTM_PAIRING_DEVICE_LINK_KEYS_REQUEST_EVT | Since we are connecting to a known device (because it is in the address resolution database), this event is called by the stack so that the firmware can load the paired device's keys from NVRAM. If keys are not available, this state must return WICED_BT_ERROR. That return value causes the stack to generate keys and then it will call the corresponding update event so that the new keys can be saved in NVRAM. |
| | BTM_ENCRYPTION_STATUS_EVT | <p>In this state, the firmware reads the state of the server from NVRAM. For example, the BD_ADDR of the paired device and the saved state of any notify settings may be read.</p> <p>Since the paired device BD_ADDR and keys were already available in NVRAM, no other steps are needed to complete pairing.</p> |
| Read Values | GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_READ | The firmware must get the value from the correct location in the GATT database. |
| Write Values | GATT_ATTRIBUTE_REQUEST_EVT → GATTS_REQ_TYPE_WRITE | The firmware must store the provided value in the correct location in the GATT database. |
| Notifications | N/A | Notifications must be sent whenever an attribute that has notifications set is updated by the firmware. Since the change comes from the local firmware, there is no stack or GATT event that initiates this process. |
| Disconnect | BTM_BLE_ADVERT_STATE_CHANGED_EVT | Advertising on. |

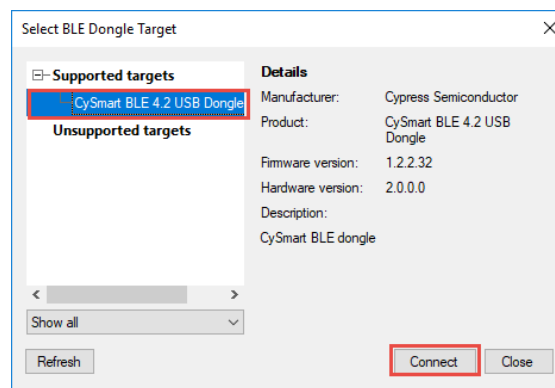
8.18 CySmart

Cypress provides a PC and mobile device application (Android and iOS) called CySmart which can be used to scan, connect, and interact with services, characteristics, and attributes of BLE devices.

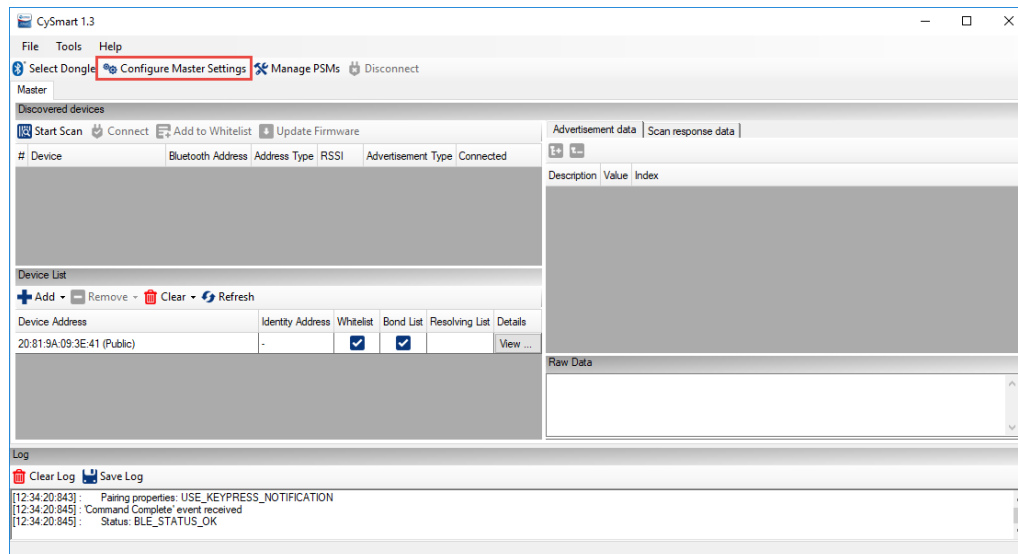
There are other utilities available for iOS and Android (such as Lightblue Explorer) which will also work. Feel free to use one of those if you are more comfortable with it.

8.18.1 CySmart PC Application

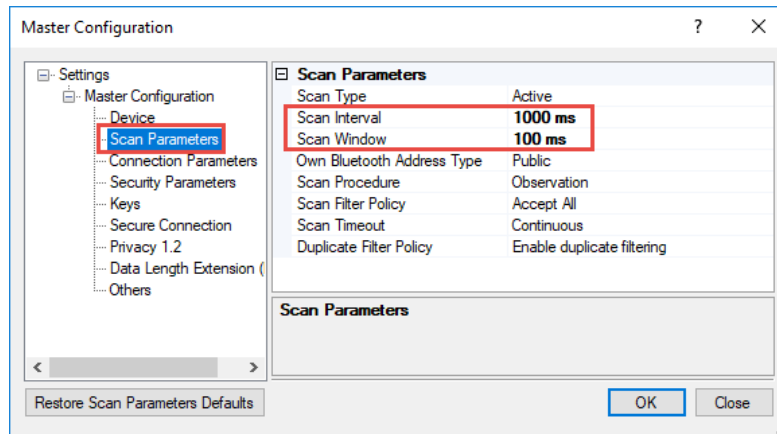
To use the CySmart PC Application, a CY5670 CySmart USB Dongle is required. When CySmart is started, it will search for supported targets and will display the results. Select the dongle that you want to use and click on "Connect".



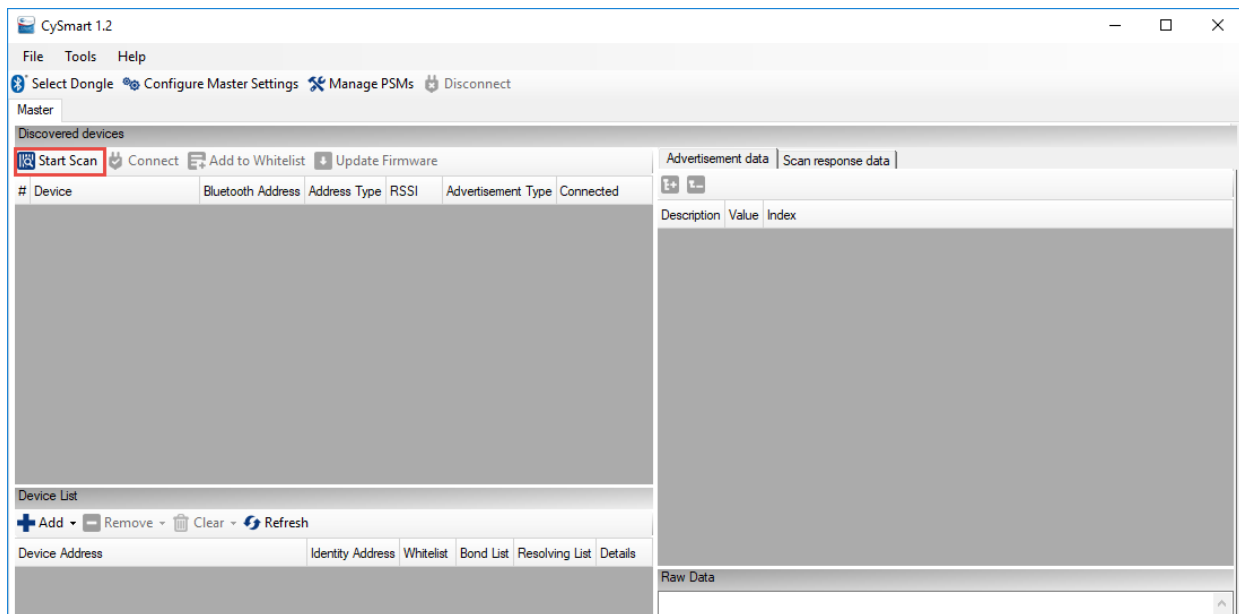
Once a dongle is selected, the main window will open as shown below.



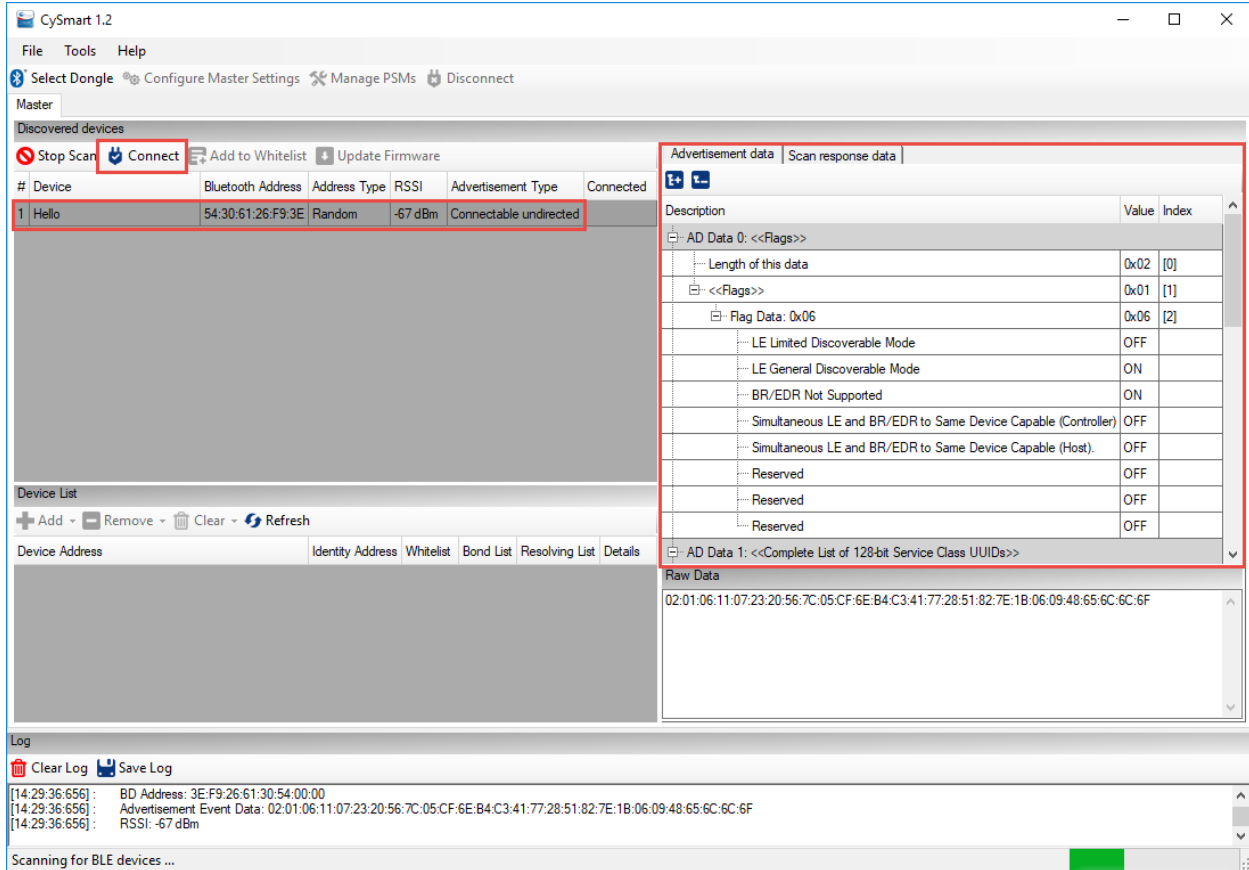
Before starting a scan, it is a good idea to configure the master settings so that scanning is done less frequently. This is especially important in a class environment where there may be many devices advertising at the same time. The tool may act strangely if it is trying to scan too fast. It is recommended to set the Scan Interval to 1000 ms and the Scan Window to 100 ms. Note that these settings are NOT saved when you close CySmart so you will need to set them each time you restart it.



Once you click "OK" to close the Master Configuration window, click on "Start Scan" from the main window to search for advertising BLE devices.



Once the device that you want to connect to appears, click on "Stop Scan" and then click on the device you are interested in. You can then see its Advertisement data and Scan response data in the right-hand window. Click "Connect" to connect to the device.



The screenshot shows the CySmart 1.2 application window. The 'Discovered devices' section lists a device named 'Hello' with a Bluetooth Address of '54:30:61:26:F9:3E', Address Type of 'Random', RSSI of '-67 dBm', and Advertisement Type of 'Connectable undirected'. The 'Connect' button is highlighted. The 'Advertisement data' section is expanded, showing the following data:

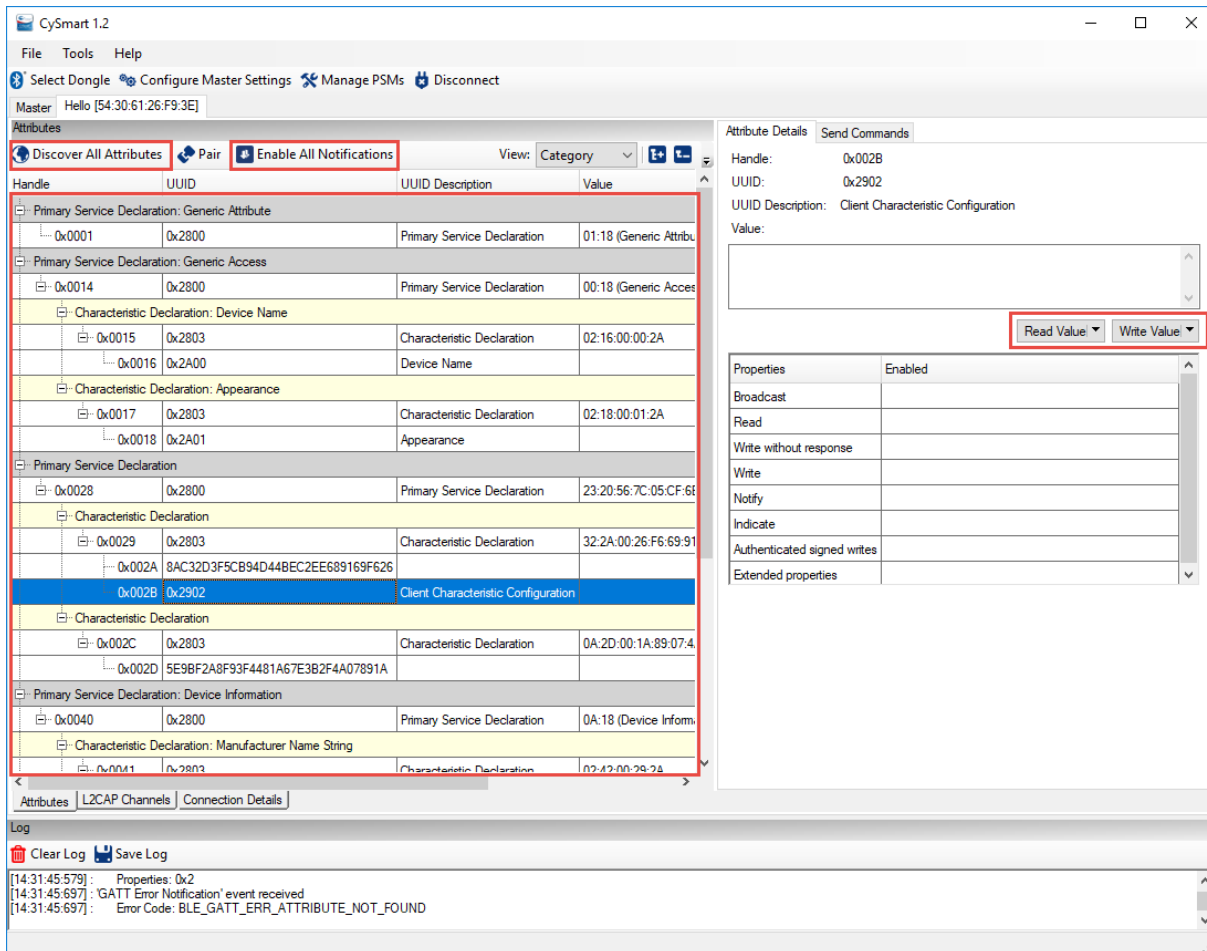
| Description | Value | Index |
|--|--|-------|
| AD Data 0: <<Flags>> | | |
| Length of this data | 0x02 | [0] |
| <<Flags>> | 0x01 | [1] |
| Flag Data: 0x06 | 0x06 | [2] |
| LE Limited Discoverable Mode | OFF | |
| LE General Discoverable Mode | ON | |
| BR/EDR Not Supported | ON | |
| Simultaneous LE and BR/EDR to Same Device Capable (Controller) | OFF | |
| Simultaneous LE and BR/EDR to Same Device Capable (Host) | OFF | |
| Reserved | OFF | |
| Reserved | OFF | |
| Reserved | OFF | |
| AD Data 1: <<Complete List of 128-bit Service Class UUIDs>> | | |
| Raw Data | 02:01:06:11:07:23:20:56:7C:05:CF:6E:B4:C3:41:77:28:51:82:7E:1B:06:09:48:65:6C:6F | |

The 'Log' section at the bottom shows the following messages:

```
[14:29:36.656] : BD Address: 3E:F9:26:61:30:54:00:00
[14:29:36.656] : Advertisement Event Data: 02:01:06:11:07:23:20:56:7C:05:CF:6E:B4:C3:41:77:28:51:82:7E:1B:06:09:48:65:6C:6F
[14:29:36.656] : RSSI: -67 dBm
```

The status bar at the bottom indicates 'Scanning for BLE devices ...' with a green progress bar.

When the device is connected, click on "Pair" and then "Discover All Attributes". Once that is complete, you will see a representation of all Services, Characteristics, and Attributes from the GATT database. You can read and write values by clicking on an attribute and using the buttons in the right-hand window. Click "Enable All Notifications" if you want to see real-time value updates in the left-hand window for characteristics that have notification capability.



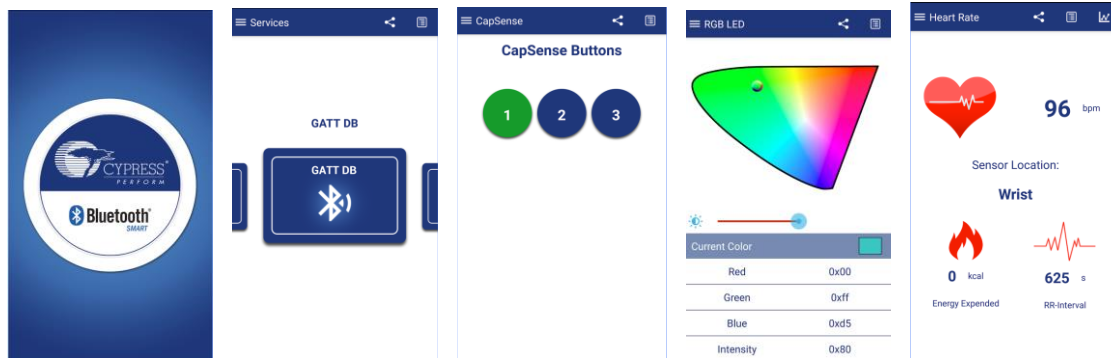
The complete User Guide for the CySmart PC application can be opened in the tool under *Help -> Help Topics*. It can also be found on the CySmart website at:

<http://www.cypress.com/documentation/software-and-drivers/cysmart-bluetooth-le-test-and-debug-tool>

Scroll down to the Related Files section of the page to find the User Guide.

8.18.2 CySmart Mobile Application

The CySmart mobile application is available on the Google Play store and the Apple App store. The app can connect and interact with any connectable BLE device. It supports specialized screens for many of the BLE adopted services and a few Cypress custom services such as CapSense and RGB LED control. In addition, there is a GATT database browser that can be used to read and write attributes for all services even if they are not supported with specialized screens.



Complete documentation and source code can be found on the CySmart Mobile App website at:

<http://www.cypress.com/documentation/software-and-drivers/cysmart-mobile-app>

Documentation of the Cypress custom profiles supported by the tool can be found at:

<http://www.cypress.com/documentation/software-and-drivers/cypress-custom-ble-profiles-and-services>

8.19 Exercises (Part 2)

8.19.1 Basic BLE Peripheral



Follow the steps in section 8.12 to create a basic BLE peripheral that allows you to control the LED on the kit.

8.19.2 Notifications



Follow the steps in section 8.15 to add a button press counter Characteristic that has Read and Notify Properties.

8.19.3 Paring

Add pairing capability to the previous exercise. Change the LED, Counter, and CCCD Attribute permissions so that they require a paired (authenticated) connection before they can be read/written.

Below is a table showing the events that occur during this exercise. Arrows indicate the cause/effect of the stack events. New events introduced in this exercise are highlighted.

| External Event | BLE Stack Event | Action |
|--|---|--|
| Board reset → | BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT → | Not used yet |
| | BTM_ENABLED_EVT → | Initialize application |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_UNDIRECTED_HIGH) | ← Start advertising |
| CySmart will see advertising packets | | |
| Connect to device from CySmart → | GATT_CONNECTION_STATUS_EVT → | Set the connection ID and enable pairing |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_OFF) | |
| Pair → | BTM_SECURITY_REQUEST_EVT → | Grant security |
| | BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT → | Capabilities are set |
| | BTM_ENCRYPTION_STATUS_EVT | Not used yet |
| | BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT | Not used yet |
| | BTM_PAIRING_COMPLETE_EVT | Not used yet |
| Read Button characteristic while pressing button → | GATT_ATTRIBUTE_REQUEST_EVT, GATTS_REQ_TYPE_READ → | Returns button state |
| Read Button CCCD → | GATT_ATTRIBUTE_REQUEST_EVT, GATTS_REQ_TYPE_READ → | Returns button notification setting |
| Write 01:00 to Button CCCD → | GATT_ATTRIBUTE_REQUEST_EVT, GATTS_REQ_TYPE_WRITE → | Enables notifications |
| Press button → | | Send notifications |
| Disconnect → | GATT_CONNECTION_STATUS_EVT → | Clear the connection ID |
| | BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_UNDIRECTED_HIGH) | Re-start advertising |
| Wait for timeout → | BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_UNDIRECTED_LOW) | Stack switches to lower advertising rate to save power |
| Wait for timeout → | BTM_BLE_ADVERT_STATE_CHANGED_EVT (BTM_BLE_ADVERT_OFF) | Stack stops advertising |

Application Creation

☐
☐
☐
☐

1. Import the previously completed notification exercise (not the template) into a new one using the New Application Wizard Import function. Call the new application ch08_pair.

2. Open the Bluetooth Configurator.

- a. Change the Device Name to <init>_pair.

- b. In the Counter characteristic set the Read (authenticated) permission, which will make the peripheral reject read requests unless the devices are paired.

Hint: You MUST leave "Read" checked also. It will not work with just "Read (authenticated)" checked.

☐

- c. Update the Client Characteristic Configuration descriptor to require authenticated read and write.

This will cause the application to require pairing to view or change the notification settings.

Hint: You MUST leave "Read" and "Write" checked also.

☐
☐
☐

- d. Save the edits and close the configurator.

3. In app.c, look for the call to `wiced_bt_set_pairable_mode()` mode and set the first argument to `WICED_TRUE` to allow pairing.

4. In the `BTM_PAIRING_IO_CAPABILITIES_BLE_REQUEST_EVT` management case tell the central that you require MITM protection, but the device has no IO capabilities.

```
p_event_data->pairing_io_capabilities_ble_request.auth_req =
BTM_LE_AUTH_REQ_SC_MITM_BOND;
p_event_data->pairing_io_capabilities_ble_request.init_keys =
BTM_LE_KEY_PENC|BTM_LE_KEY_PID;
p_event_data->pairing_io_capabilities_ble_request.local_io_cap =
BTM_IO_CAPABILITIES_NONE;
```

☐

5. In the `BTM_SECURITY_REQUEST_EVT` management case grant the authorization to the central by using the following code:

```
wiced_bt_ble_security_grant( p_event_data->security_request.bd_addr,
WICED_BT_SUCCESS );
```

Testing

☐
☐
☐
☐
☐
☐
☐
☐

1. Build and program the application to the board.
2. Open the mobile CySmart app.
3. Connect to the device.
4. Open the GATT browser, navigate to the Counter characteristic, press Descriptor and then the Client Characteristic Configuration.
5. If requested, accept the invitation to pair the devices.
6. Return to the Counter, enable notifications and observe the Counter value as you press the button on the kit.
7. Disconnect from the mobile CySmart app.
8. Go to the phone's Bluetooth settings and remove the <init>_pair device from the paired devices list.

This is necessary so that when you re-program the kit the phone won't have stale bonding information stored which could prevent you from re-connecting. In the next exercise we'll store bonding information on the WICED device so that you will be able to leave the devices paired if you desire.

☐

9. Start the PC CySmart app. Scan for your device. Once it appears in the list, stop scanning and connect to it.

Hint: Don't forget to update the scan interval and window to 1000 and 100 ms respectively.

☐
☐
☐
☐
☐
☐
☐

10. Click on "Discover all Attributes" and then on "Enable Notifications". Notice that you will get an authentication error. Click "OK" to close the error dialog.
11. Try reading the Counter Characteristic Value manually. Notice that you again get an authentication error. Click "OK" to close the error window.
12. Click on "Pair" and click "No" if you are asked if you want to add the device to the resolving list since we haven't yet enabled privacy.
13. Click on "Enable All Notifications" again. Now when you press the button you will see the characteristic value change.
14. Click on "Disable All Notifications" and then read the Button Characteristic Value manually. It should now work.
15. Click "Disconnect".
16. From the Device List, select a Device Address and select "Clear -> All" since we have not stored bonding information in the device yet.

8.19.4 Bonding

Introduction

The prior exercise has been modified for you to save and restore bonding information to NVRAM. You will create the application from a template, program it to your kit, experiment with it, and then answer questions about the stack events that occur.

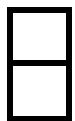
By saving Bonding information on both sides (i.e. the client and the server) future connections between the devices can be established more quickly with fewer steps. This is particularly useful for devices that require a pairing passkey (which will be added in the next exercise) since saving the bonding information means the passkey doesn't have to be entered every time the device connects.

Moreover, since the keys are saved on both devices, they don't need to be exchanged again. This means that after the first connection, there is no possibility of a MITM attack since the keys are not sent out over the air.

The firmware has two "modes": *bonding mode* and *bonded mode*. After programming, the kit will start out in bonding mode. LED1 will blink slowly (1 sec duty cycle) to indicate that the kit is waiting to be Paired/Bonded. Once a Client connects to the kit and pairs with it, the Bonding information will be saved in non-volatile memory. The LED will be ON since the kit is connected. The only Client that will be allowed to pair with the kit is the one that is bonded (the firmware only allows 1 bonded device at a time for now). If the Bonding information is removed from the Client, it will no longer be able to Pair/Bond with the kit without going through the Pairing/Bonding process again.

When you disconnect, LED1 will flash rapidly (200ms duty cycle) to indicate that it is bonded. To remove Bonding information from the kit and return bonding mode, press 'e' in the UART terminal window. This will erase the stored bonding information and put the kit back into Bonding mode. LED1 will now go back to a slow flashing rate. When you reconnect, the key must be entered again to connect. This allows you to Pair/Bond from a Client that has "lost" the bonding information or to Pair/Bond with a new device without having to reprogram the kit.

Application Creation



1. Create a new application called **ch08_bond** using the template provided.
2. Open the Bluetooth Configurator.
 - a. Change the Device Name to <init>_bond.
 - b. Save the edits and close the configurator.



3. Open `app_bt_cfg.c` and change the `rpa_refresh_timeout`:
 - from: `WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_NEVER_CHANGE`
 - to: `WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_CHANGE_TIMEOUT`

This enables privacy.

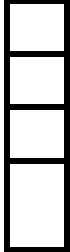


4. The remaining code for this exercise has already been implemented for you in `app.c`



5. Build and program the kit.

Testing



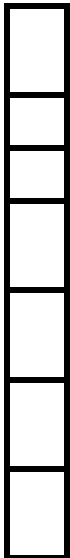
1. Open a UART terminal window to the PUART.
2. Build the application and program it to the board.
3. Open the CySmart PC application and connect to the dongle.
4. Click 'Configure Master Settings' and, under 'Privacy 1.2', change the Address Generation Interval to match the *rpa_refresh_timeout* in *app_bt_cfg.c*.

Also, don't forget to update the scan interval and window to 1000 and 100 ms respectively.



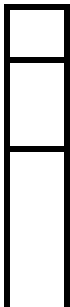
5. If there is anything listed in the "Device List" near the bottom of the screen, click on any device from the list and choose "Clear > All".

This will remove any stored bonding information from CySmart so that it will not conflict with your new firmware. It is necessary to do this each time you re-program the kit so that the old information is not used.



6. Start scanning. Once you see your device in the list stop scanning. Note that your device shows up with a Random Bluetooth address now since privacy is enabled.
7. Connect to your device.
8. Click on "Discover all Attributes".
9. Click on "Pair" and click "Yes" when asked if you want to add the device to the resolving list so that the privacy keys will be remembered by CySmart.
10. Note down the Bluetooth Stack events that occur during pairing. This information is displayed in the UART.
11. Click on "Enable All Notifications". Press the button and observe the characteristic value changes.
12. Click "Disconnect". Do NOT remove the device from the Device List this time – we want bonding information retained.

Hint: You will notice that the LED is flashing rapidly. The firmware was written to do this when it is not connected but has bonding information stored.



13. Start a new scan and stop when your device appears in the list.
14. Notice how the Address is now listed as a Public Identity Address rather than Random in the table of discovered devices.
15. Look at the Device List window at the bottom; both the Device Address and the Public Identity Address are listed. If you click on 'View ...', some Details concerning the device appear including the Identity Resolving Key. The IRK is used to map the Private Random Address to the Public Identity Address.

☐
☐
☐
☐

16. Re-connect to your device.
17. Click on "Discover all Attributes" and "Pair".
18. Once again note down the Bluetooth Stack events that occur during pairing. You will notice that fewer steps are required this time.
19. Press the button and observe that notifications are already enabled since they were enabled when you disconnected. This information was retained in NVRAM.

Note: If you use the mobile CySmart app, notifications will be disabled when you reconnect because the app automatically disables notifications before disconnecting.

☐
☐

20. Disconnect again.
21. Reset or power cycle the board.

Hint: If you power cycle the board, you will need to either reset or re-open the UART terminal window.

☐
☐
☐
☐
☐
☐
☐
☐

22. Start scanning, stop when your device appears, and then connect to your device for a third time.
23. Click on "Discover all Attributes" and "Pair".
24. Note down the Bluetooth Stack events that occur this time during pairing. Compare to the previous two connections.
25. Note that notifications are still enabled.
26. Disconnect again.
27. Clear the Device List at the bottom of the CySmart window (i.e. click on any device listed and do "Clear -> All".
28. Start scanning and stop when your device appears. Notice that it again has a Random address type.
29. Connect to your device, "Discover all Attributes" and then try to "Pair". Note that pairing will not complete because CySmart no longer has the required keys to use.

- Hint: If you look in the UART window you will see a message about the security request being denied.
- Hint: It will take a while before pairing times out.

☐
☐

30. Click on Disconnect and close the Authentication failed message window.
31. Press "e" in the UART window and note that LED1 begins flashing slowly.

This indicates that the bonding information has been cleared from the device and it will now allow a new connection.

☐
☐

32. Connect, Discover Attributes, and Pair again. This time it should work.
33. Note the steps that the firmware goes through this time.



34. Disconnect a final time and clear the Device List so that the saved bonding information won't interfere with the next exercise.

Hint: You should clear the bonding information from CySmart anytime you are going to reprogram the kit or otherwise clear bonding information since the WICED device will no longer have the bonding information on its side.

Overview of Changes

A structure called "hostinfo" is created which holds the BD_ADDR of the bonded device and the value of the Button CCCD. The BD_ADDR is used to determine when we have reconnected to the same device while the CCCD value is saved so that the state of notifications can be retained across connections for bonded devices.

Before initializing the GATT database, existing keys (if any) are loaded from NVRAM. If no keys are available this step will fail so it is necessary to look at the result of the NVRAM read. If the read was successful, then the keys are copied to the address resolution database and the variable called "bonded" is set as TRUE. Otherwise, it stays FALSE, which means the device can accept new pairing requests.

- In the BTM_SECURITY_REQUEST_EVENT look to see if bonded is FALSE. Security is only granted if the device is not bonded.
- In the Bluetooth stack event *BTM_PAIRING_COMPLETE_EVT* if bonding was successful write the information from the hostinfo structure into the NVRAM and set bonded to TRUE.
 - This saves hostinfo upon initial pairing. This event is not called when bonded devices reconnect.
- In the Bluetooth stack event *BTM_ENCRYPTION_STATUS_EVT*, if the device is bonded (i.e. bonded is TRUE), read bonding information from the NVRAM into the hostinfo structure.
 - This reads hostinfo upon a subsequent connection when devices were previously bonded.
- In the Bluetooth stack event *BTM_PAIRING_DEVICE_LINK_KEYS_UPDATE_EVT*, save the keys for the peer device to NVRAM.
- In the Bluetooth stack event *BTM_PAIRING_DEVICE_LINK_KEYS_REQUEST_EVT*, read the keys for the peer device from NVRAM.
- In the Bluetooth stack event *BTM_LOCAL_IDENTITY_KEYS_UPDATE_EVT*, save the keys for the local device to NVRAM.
- In the Bluetooth stack event *BTM_LOCAL_IDENTITY_KEYS_REQUEST_EVT*, read the keys for the local device from NVRAM.
- In the GATT connect callback:
 - For a connection, save the BD_ADDR of the remote device into the hostinfo structure. This will be written to NVRAM in the *BTM_PAIRING_COMPLETE_EVT*.
 - For a disconnection, clear out the BD_ADDR from the hostinfo structure and reset the CCCD to 0.
- In the GATT set value function, save the Button CCCD value to the hostinfo structure whenever it is updated and write the value into NVRAM.

- The UART is configured to accept input. The rx_cback function looks for the key "e". If it has been sent, it sets bonded to FALSE, removes the bonded device from the list of bonded devices, removes the device from the address resolution database, and clears out the bonding information stored in NVRAM.
- The PWM that blinks the LED during advertising has two different rates. The PWM is started during initialization, and then stopped/restarted when a disconnect happens or when bonding information is removed. The blink rate is set depending on the value of bonded.
- Finally, privacy is enabled in wiced_bt_cfg.c by updating the rpa_refresh_timeout to *WICED_BT_CFG_DEFAULT_RANDOM_ADDRESS_CHANGE_TIMEOUT*.

Questions

☐

1. What items are stored in NVRAM?

☐

2. Which event stores each piece of information?

☐

3. Which event retrieves each piece of information?

☐

4. In what event is the privacy info read from NVRAM?

☐

5. Which event is called if privacy information is not retrieved after new keys have been generated by the stack?