Abstract

This Application Note describes how to implement a custom Bluetooth profile on the DA14580 using the sample service, sample128, as a foundation.
Developing a DA14580 Bluetooth profile using sample128

Contents

Contents ................................................................................................................................. 2
Figures .................................................................................................................................. 3
Tables ................................................................................................................................... 4
1 Terms and definitions ........................................................................................................ 4
2 References ......................................................................................................................... 4
3 Introduction ....................................................................................................................... 5
4 Building a project that includes the sample128 service .................................................... 6
   4.1 Creating a new project based on the template_fh sample application .................... 6
   4.2 Add the sample128 service to the project .................................................................... 6
       4.2.1 Adding the sample128 path to the project Include Paths .................................. 6
       4.2.2 Adding the sample128 source code to the project .............................................. 8
       4.2.3 Including sample128 ............................................................................................ 9
5 Interfacing your application with sample128 .................................................................. 11
   5.1 Creating the service database ...................................................................................... 12
   5.2 Enabling the service .................................................................................................... 13
   5.3 Implementing message handlers .................................................................................. 14
   5.4 Trying it out ................................................................................................................ 19
6 Using sample128 ............................................................................................................... 21
   6.1 Implementing a kernel timer ....................................................................................... 22
   6.2 Adding some functionality .......................................................................................... 24
7 Modifying sample128 ......................................................................................................... 27
   7.1 The basics of sample128 .............................................................................................. 27
       7.1.1 The primary service declaration attribute .......................................................... 28
       7.1.2 The characteristic declaration attribute ............................................................. 28
       7.1.3 The characteristic value declaration attribute ..................................................... 28
       7.1.4 The client configuration declaration attribute ..................................................... 28
       7.1.5 Summarizing the components of sample128 ....................................................... 29
   7.2 Modifying the data size of characteristic 1 ................................................................. 29
       7.2.1 Defining our new data type of 8 bytes ................................................................. 29
       7.2.2 Recalculating the size of the database ................................................................. 30
       7.2.3 Modifying the value attribute .............................................................................. 30
       7.2.4 Modifying messages between sample128 and the application ......................... 31
   7.3 Adding a new characteristic to the service................................................................. 34
       7.3.1 Defining our new data type of 10 bytes ............................................................... 34
       7.3.2 Calculating the size of the new database .............................................................. 35
       7.3.3 Building the new database ................................................................................... 36
       7.3.4 Initializing the characteristic value ....................................................................... 40
       7.3.5 Setting the default value of characteristic 3 ......................................................... 42
       7.3.6 Updating the characteristic value from the application ...................................... 42
       7.3.7 Implementing support for GATT notify ............................................................... 45
8 Revision history .................................................................................................................. 50
## Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Options for target</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Opening include paths list</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Adding an include path</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Adding source code</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Finding source files</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Sample128 source files</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>Configuration of the project</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>Including the service</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>Message flow diagram</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>Creating the database</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>Enumerating the service database</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>Creating the service database</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>Enabling the service</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>Sending service enable message</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>Definition of event handlers</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>Event handler prototypes</td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>Database created handler</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>Service disabled handler and Char1 value changed handler</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>GATT discovery using BlueLoupe</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>Sample128 tutorial functionality</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>Adding a message primitive</td>
<td>22</td>
</tr>
<tr>
<td>22</td>
<td>Adding a message handler</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>Timer handler prototype</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>Timer handler implementation</td>
<td>23</td>
</tr>
<tr>
<td>25</td>
<td>Starting our kernel timer</td>
<td>24</td>
</tr>
<tr>
<td>26</td>
<td>Declaring a global variable</td>
<td>24</td>
</tr>
<tr>
<td>27</td>
<td>Timer functionality implementation</td>
<td>25</td>
</tr>
<tr>
<td>28</td>
<td>Write event implementation</td>
<td>25</td>
</tr>
<tr>
<td>29</td>
<td>UUID of sample128 service</td>
<td>28</td>
</tr>
<tr>
<td>30</td>
<td>Characteristic 1 declaration</td>
<td>28</td>
</tr>
<tr>
<td>31</td>
<td>Different sized declaration type IDs</td>
<td>29</td>
</tr>
<tr>
<td>32</td>
<td>Adding service128 to the database</td>
<td>29</td>
</tr>
<tr>
<td>33</td>
<td>Defining a new type</td>
<td>29</td>
</tr>
<tr>
<td>34</td>
<td>Initialization of a global variable</td>
<td>30</td>
</tr>
<tr>
<td>35</td>
<td>Changes to the database size</td>
<td>30</td>
</tr>
<tr>
<td>36</td>
<td>Changing the value attribute</td>
<td>31</td>
</tr>
<tr>
<td>37</td>
<td>The sample128_enable_req structure</td>
<td>31</td>
</tr>
<tr>
<td>38</td>
<td>The sample128_val_ind structure</td>
<td>31</td>
</tr>
<tr>
<td>39</td>
<td>Setting the default value via memcpy</td>
<td>32</td>
</tr>
<tr>
<td>40</td>
<td>Retrieving the value of characteristic 1</td>
<td>32</td>
</tr>
<tr>
<td>41</td>
<td>The sample128_send_val prototype</td>
<td>32</td>
</tr>
<tr>
<td>42</td>
<td>Changes to sample128_send_val</td>
<td>33</td>
</tr>
<tr>
<td>43</td>
<td>Changes to gattc_write_cmd_ind_handler</td>
<td>33</td>
</tr>
<tr>
<td>44</td>
<td>Changes to sample128_enable_req_handler</td>
<td>34</td>
</tr>
<tr>
<td>45</td>
<td>Defining a new data type in sample128.h</td>
<td>34</td>
</tr>
<tr>
<td>46</td>
<td>Database changes</td>
<td>36</td>
</tr>
<tr>
<td>47</td>
<td>Defining attribute values</td>
<td>38</td>
</tr>
<tr>
<td>48</td>
<td>Indexing the 3 new attributes</td>
<td>39</td>
</tr>
<tr>
<td>49</td>
<td>Changes to sample128.h</td>
<td>39</td>
</tr>
<tr>
<td>50</td>
<td>The new characteristic is exposed (BlueLoupe)</td>
<td>40</td>
</tr>
<tr>
<td>51</td>
<td>Initialization of a global variable</td>
<td>40</td>
</tr>
<tr>
<td>52</td>
<td>Modifying the enable structure</td>
<td>41</td>
</tr>
<tr>
<td>53</td>
<td>Initialization of the characteristic value</td>
<td>41</td>
</tr>
<tr>
<td>54</td>
<td>Default value of characteristic 3</td>
<td>42</td>
</tr>
<tr>
<td>55</td>
<td>New characteristic update structure</td>
<td>42</td>
</tr>
<tr>
<td>56</td>
<td>New message primitives</td>
<td>43</td>
</tr>
<tr>
<td>57</td>
<td>Implementing a new handler</td>
<td>43</td>
</tr>
<tr>
<td>58</td>
<td>Change the first byte of characteristic 3</td>
<td>45</td>
</tr>
</tbody>
</table>
Developing a DA14580 Bluetooth profile using sample128

Figure 59: Adding the new characteristic’s client configuration to sample128.h ........................................... 46
Figure 60: Changing the service environment structure .................................................................................. 46
Figure 61: Initializing notification .................................................................................................................. 47
Figure 62: Handling notification subscriptions .............................................................................................. 48

Tables
Table 1: The GATT database of sample128 ........................................................................................................ 27
Table 2: The new GATT table ......................................................................................................................... 35

1 Terms and definitions
BT SIG Bluetooth Special Interest Group
IDE Integrated Development Environment
SDK Software Development Kit
DVK Development Kit (DA14580 Expert, Pro, or Basic)
UUID Universally Unique Identity
GATT Generic Attribute Profile
MDK Microprocessor Development Kit
UUID Universally Unique Identifier

2 References
1. UM-B-003, Software Development Guide
3 Introduction

The Bluetooth Special Interest group has adopted a rich list of profiles and services to cover a wide variety of use cases in which Bluetooth Smart plays a role. The beauty of these already specified profiles and services is that their specifications are very clear and pretty much guarantees interoperability between smartphones and tablets and all kinds of peripheral devices. Before you venture into creating your own service or profile, it is recommended that you visit the BT SIG website www.bluetooth.org to see if a service or profile that meets your requirements has already been adopted.

In some cases, however, it is necessary to implement your own services or profiles. Your application may require some new functionality that does not fit within the already-adopted profiles or services. This document functions as a guide/tutorial on how to implement such custom services on the Dialog Semiconductor SmartBond series DA1458x.

You should already be familiar with the hardware and the Keil µVision MDK IDE, and you should have all the tools and drivers installed and operational. You should also have some basic knowledge of Bluetooth Smart and the concept of peripheral and central devices.

This tutorial will only address custom service implementation on the peripheral device side.
4 Building a project that includes the sample128 service

The Dialog SDK contains a profile named sample128. The ‘128’ part of the name relates to the fact that the sample service uses a 128-bit UUID. The BT SIG adopted services all use a 16-bit short form. Custom services must use the long 128-bit form. In this section, we will demonstrate how to include sample128 in your project.

4.1 Creating a new project based on the template_fh sample application

Clone the template_fh project as described in the Software Development Guide [1]. In the following, we will assume that you have called the project “custom”, but you can use any name you like. Open the newly created project and apply the following steps.

4.2 Add the sample128 service to the project

Adding the sample128 service requires the following steps (detailed instructions will follow):

1. Adding the sample128 path to the project include paths
2. Adding the source code of sample128 to the project
3. Including sample128

4.2.1 Adding the sample128 path to the project Include Paths

Click the “Options for Target” icon:

Figure 1: Options for target
Open the “C/C++” tab and click on the button to the right of the Include Paths field:

![Opening include paths list](image1)

**Figure 2: Opening include paths list**

Click at the bottom of the Include Paths list and type in the path for sample128 (you can also use the button to the right to browse for the path):

![Adding an include path](image2)

**Figure 3: Adding an include path**

Or, for the copy & paste fans out there:

```
:../ip\ble\hl\src\profiles\sample128
```

Click “OK” to close the folder setup window and click “OK” to close the “Options for Target” window.
4.2.2 Adding the sample128 source code to the project

Expand the project folder so it shows the view below:

![Adding source code](Figure 4: Adding source code)

Right-click on the profiles folder and select “Add Existing Files to Group ‘profiles’…” Navigate to the folder “dk_apps/src/ip/ble/hl/src/profiles/sample128”:

![Finding source files](Figure 5: Finding source files)
You should see the following two files:

![Sample128 source files](image)

Figure 6: Sample128 source files

Select the two C source files and click “Add”, then “Close”.

Now, rebuild the project to get the compiler to map the header files (this simplifies navigation).

### 4.2.3 Including sample128

Open the file “da14580_config.h” file (expand the “app” folder and the “app.c” file, locate the file and double-click on it to open it).

Change `#undef CFG_PRF_SAMPLE128` to `#define CFG_PRF_SAMPLE128` as shown below:

```c
/* Accelerometer Profile */
#define CFG_PRF_ACCEL
/* Dialog’s Software Patch Over The Air Profile */
#define CFG_PRF_SOTAR
/* Dialog’s Sample 128 bit UUIDs Profile */
#define CFG_PRF_SAMPLE128

/**********************************************************************************

#define CFG_PRF_SAMPLE128

The file sample128_task header should also be added to the project header file “app_custom_proj.h”:```
You should be able to build the project at this time and only see the two standard warnings.
5 Interfacing your application with sample128

The sample128 service implementation provides two characteristics. The first characteristic is a simple, single-byte-sized characteristic, facilitating client-side read and write permissions. The other characteristic is also a single byte, but this one facilitates read and notify permissions.

The application controls initializing the service, creating the service database and enabling the service upon a remote client connection. The application might also update the value of the second characteristic, which causes a GATT notify to be sent to any connected client that has subscribed to notifications.

The application will receive a confirmation from the service task when the database has been created, and it will receive indications from the service when a remote client writes to a characteristic or a remote client device (a central) causes the service to be disabled. The message flow is illustrated below:

Application  Sample128 Service  Remote Client

App initialized

Create service database request

Service database created confirmation

Remote client connects

Enable service command

Remote client writes to characteristic

Client write indication

Remote client disconnects

Service disabled indication

GATT Notify

Update characteristic 2 value command

Figure 9: Message flow diagram

It is usually recommended to implement a task between the application task and the service task; this is also how the sample applications in the SDK are structured. For this tutorial, in order to minimize the number of files that need to be touched and in order to decrease complexity, we will flatten the structure by implementing all interfaces to the sample service in the application task itself. This approach is acceptable as long as we don’t need a lot of custom services.
5.1 Creating the service database

The database of a service must be created in the “app_db_init_func()” of “app_custom_proj.c”:

```c
bool app_db_init_func(void)
{
    /****************************************************************************
    * Initialize next supported profile's Database.                          *
    * Check if all supported profiles' Databases are initialized and return status. *
    ****************************************************************************/

    // Indicate if more services need to be added in the database
    bool end_db_create = false;

    dbg = APP_PRF_LIST_STOP;

    // Check if another should be added in the database
    if (app_env.next_prf_init < APP_PRF_LIST_STOP)
    {
        switch (app_env.next_prf_init)
        {

            /***************************************************************************
            * #if (BLE_DIS_SERVER)                                              *
            *   case (APP_DIS_TASK):                                             *
            *     {                                                            *
            *       app_dis_create_db_send();                                  *
            *     } break;                                                    *
            *     #endif //BLE_DIS_SERVER                                       *
            /***************************************************************************/

            case (APP_SAMPLE128):
            {
                app_sample128_create_db_send();
            } break;

            default:
            {
                ASSERT_ERR(0);
            } break;

        } // switch
        app_env.next_prf_init++;
    } else
    {
        end_db_create = true;
    }

    return end_db_create;
}
```

Figure 10: Creating the database

APP_SAMPLE128 must be enumerated among all other profiles in “app_api.h”. This allows the application to loop through all the required services and create their databases one by one:
5.2 Enabling the service

The sample128 service must be enabled after a remote client has connected to the device. This can be done in the “app_connection_func()” function of “app_custom_proj.c”:  

```c
void app_sample128_create_db_send(void)
{
    struct sample128_create_db_req *req = KE_MSG_ALLOC(SAMPLE128_CREATE_DB_REQ,
        TASK_APP,
        sample128_create_db_req
    );

    ke_msg_send(req);
}
```

The function simply sends a message to the service task requesting the creation of the service database.
We will implement the "app_sample128_enable()" function in the function definition segment of "app_customProj.c" - just below the definition of "app_sample128_create_db_send()":

```
void app_sample128_enable(void)
{
  // Allocate the message
  struct sample128_enable_req* req = KE_MSG_ALLOC(SAMPLE128_ENABLE_REQ,
                                                   TASK_SAMPLE128,
                                                   TASK_APP,
                                                   sample128_enable_req);
  req->conhdl = app_env.conhdl;
  req->sec_lvl = PERM(SVC, ENABLE);
  req->sample128_1_val = 0x01;  // default value for sample128 characteristic 1
  req->sample128_2_val = 0xff;  // default value for sample128 characteristic 2
  req->feature = 0x00;          // client CFG notify/indicate disabled
  // Send the message
  ke_msg_send(req);
}
```

Figure 14: Sending service enable message

### 5.3 Implementing message handlers

As mentioned earlier, the sample128 service task will send the following kernel messages to the application:

1. A confirmation when the service database has been created
2. An indication when a remote client writes to a characteristic value
3. An indication when the service database is disabled (due to a remote client disconnection)

Three event handlers need to be defined for these events. In “app_task_handlers.h”, add the following code:

```c
#define BLE_SAMPLE128
{SAMPLE128_CREATE_DB_CFM, (ke_msg_func_t)sample128_create_db_cfm_handler},
{SAMPLE128_VAL_IND, (ke_msg_func_t)sample128_val_ind_handler},
{SAMPLE128_DISABLE_IND, (ke_msg_func_t)sample128_disable_ind_handler},
#endif
```

Figure 15: Definition of event handlers

Finally, these handlers must be implemented in the application. In the project header file, “app_custom_proj.h”, add the following prototypes:

```c
/* FUNCTION DECLARATIONS */

/* */

/* $brief Handles sample128 profile database creation confirmation.
* $return If the message was consumed or not.
*/

int sample128_create_db_cfm_handler(ke_msg_id_t msgid,
                  struct sample128_create_db_cfm const *param,
                  ke_task_id_t const dest_id,
                  ke_task_id_t const src_id);

/* */

/* $brief Handles disable indication from the sample128 profile.
* $return If the message was consumed or not.
*/

int sample128_disable_ind_handler(ke_msg_id_t msgid,
                   struct sample128_disable_ind const *param,
                   ke_task_id_t const dest_id,
                   ke_task_id_t const src_id);

/* */

/* $brief Handles write or lat characteristic event indication from sample128 profile
* $return If the message was consumed or not.
*/

int sample128_val_ind_handler(ke_msg_id_t msgid,
                  struct sample128_val_ind const *param,
                  ke_task_id_t const dest_id,
                  ke_task_id_t const src_id);
```

Figure 16: Event handler prototypes
We will implement the tree handlers in the “app_custom_proj.c” file. Handling the “database created confirmation” is implemented as shown below:

```c
/**
 * @brief Handles sample128 profile database creation confirmation.
 * @return If the message was consumed or not.
 */
int sample128_create_db_cfm_handler(ke_msg_id_t const msgid,
        struct sample128_create_db_cfm const *param,
        ke_task_id_t const dest_id,
        ke_task_id_t const src_id);

/**
 * @brief Handles disable indication from the sample128 profile.
 * @return If the message was consumed or not.
 */
int sample128_disable_ind_handler(ke_msg_id_t const msgid,
        struct sample128_disable_ind const *param,
        ke_task_id_t const dest_id,
        ke_task_id_t const src_id);

/**
 * @brief Handles write of 1st characteristic event indication from sample128 profile
 * @return If the message was consumed or not.
 */
int sample128_val_ind_handler(ke_msg_id_t const msgid,
        struct sample128_val_ind const *param,
        ke_task_id_t const dest_id,
        ke_task_id_t const src_id);
```
The handler, as implemented above, sends another message to the application task, indicating that the database has been created.
We will leave the other two handlers empty for now. We will implement them just below the `sample128_create_db_cfm_handler()` function, near the end of "app_custom_proj.c":

```c
int sample128_disable_ind_handler(ke_msg_id_t const msgid,
    struct sample128_disable_ind const *param,
    ke_task_id_t const src_id)
{
    return (KE_MSG_CONSUMED);
}
```

```c
int sample128_val_ind_handler(ke_msg_id_t const msgid,
    struct sample128_val_ind const *param,
    ke_task_id_t const src_id)
{
    return (KE_MSG_CONSUMED);
}
```

Figure 18: Service disabled handler and Char1 value changed handler

Both handlers will simply return the fact that the kernel message has been consumed.
5.4 Trying it out

Using LightBlue for iOS or BlueLoupe for Android (Version 4.3 or later) should allow you to connect to the DVK and confirm that the custom service is provided. You may have to turn Bluetooth on your smart device off and back on to force a fresh service discovery. Both Android and iOS have a tendency to suppress service discovery for devices that they have previously been connected to.

A screenshot from BlueLoupe is shown below. The DVK exposes the custom service and the two characteristics that the service consists of. You can write to the first characteristic and see that the value changes. If you disconnect from the device, the value of the characteristic defaults back to 0x01 as specified in the function “app_sample128_enable()”. No other functionality is enabled at this point. We will add some simple functionality in the following section.
Figure 19: GATT discovery using BlueLouve
6 Using sample128

In the previous section, we implemented all the functionality required to expose the sample128 service. We were able to write to one of the two characteristics using a smartphone or tablet, but none of that was really tied to the application. In this section we will implement some code that allows us to use Bluetooth Notify to monitor when the application changes the value of characteristic 2. We will also make use of the value that a user writes to characteristic 1 via a smartphone or tablet. Here is an overview of what we will implement:

![Diagram showing the flow of operations including Device Idle, Central connects, Device Connected, Timer times out, and Client device writes to Characteristic 1.]

Figure 20: Sample128 tutorial functionality

We will implement and start a timer which will time out after 500 ms. The timer will be started when a central device connects. At timeout, we will increment the value of characteristic 2 and restart the timer. If a user writes to characteristic 1, we will set the value of characteristic 2 to match the new value of characteristic 1 and let the timer function increment it from there.
6.1 Implementing a kernel timer

In order to use the kernel timer from the application, we will need to define a new primitive to reference the timer. The primitive can be defined in the APP_MSG enumeration in “app_api.h”:

```c
#include <bluetooth.h>

static APP_MODULE_INIT_EVT = KE_FIRST_MSG(TASK_APP),

#if (BLE_SAMPLE128)
  APP_SAMPLE128_TIMER,
#endif
#endif
```

Figure 21: Adding a message primitive

We will have to implement a handler function to handle the event of the timer timing out. In “app_task_handlers.h”, add the following code:

```c
#if (BLE_SAMPLE128)
  APP_SAMPLE128_TIMER,
#endif
#endif
```

Figure 22: Adding a message handler

Finally, we will implement the timer handler in “app_custom_proj.c” and a reference to it in “app_custom_proj.h”. First the reference:

```c
#include <bluetooth.h>

app_msg_func_t sample128_timer_handler;
```

Figure 23: Timer handler prototype
And in “app_custom_proj.c” we will implement the handler function:

```c
/**
 * @brief Handles timer timeout
 * @return If the message was consumed or not.
 */
int sample128_timer_handler(ke_msg_id_t const msgid,
                           struct gapm_cmp_evt const *param,
                           ke_task_id_t const dest_id,
                           ke_task_id_t const src_id);
```

Figure 24: Timer handler implementation
We will start the timer whenever a central device connects. In the “app_connection_func()” function of “app_custom_proj.c”, add the following:

```c
void app_connection_func(struct gapp_connection_req_ind const *param)
{
  if (app_env.conidx != GAP_INVALID_CONNIDX)
  {
    Handle connection request event. Enable required profiles
    i.e.
    #if (BLE_DIS_SERVER)
    app_dis_enable_prf(app_env.conhdl);
    #endif
    // Retrieve the connection info from the parameters
    app_env.conhdl = param->conhdl;
    app_sample128_enable();
    ke_timer_set(APP_SAMPLE128_TIMER, TASK_APP, 50);
    ke_state_set(TASK_APP, APP_CONNECTED);
    // Retrieve the connection info from the parameters
    app_env.conhdl = param->conhdl;
  }
}
```

**Figure 25: Starting our kernel timer**

At this time you should be able to successfully build the code. The timer will start as soon as a central connects, but you still need to actually do something useful when the timer times out.

### 6.2 Adding some functionality

We are going to need a placeholder variable with a global scope. In this tutorial, we will simply declare a global variable. This works well as long as we don’t implement deep sleep. If we were to actually use deep sleep, we would need to store the placeholder variable in retention memory in order for it to be retained.

In “app_custom_proj.c”, declare the octet `sample128_placeholder` just above the function definitions:

```c
uint8_t sample128_placeholder = 0;
```

**Figure 26: Declaring a global variable**

When a user writes to characteristic 1 using a smartphone/tablet, we will load the written value into the placeholder variable. Every time the timer times out, we will increment the placeholder value and load it into characteristic 2.
In the timer handler, add the following code:

```c
ke_timer_set(APP_SAMPLE128_TIMER, TASK_APP, 50);
sample128_placeholder++;

struct sample128_upd_char2_req *req = KE_MSGALLOC(
    SAMPLE128_UPD_CHAR2_REQ,
    TASK_SAMPLE128,
    TASK_APP,
    sample128_upd_char2_req);

req->val = sample128_placeholder;
req->conhdl = app_env.conhdl;
ke_msg_send(req);
```

The implementation above restarts the timer, increments the placeholder variable and sends a kernel message to the sample128 service task to update the value of characteristic 2. A Bluetooth Notify will automatically be sent to the smartphone/tablet every time the value is updated if Notify is enabled for characteristic 2 via the smartphone/tablet.

Finally, we wanted to use the value written to characteristic 1 to reload the value of characteristic 2. This is a simple implementation in "sample128-val_ind_handler()" of "app_custom_proj.c":

```c
int sample128_val_ind_handler(ke_msg_id_t const msgid,
    struct sample128_val_ind const *param,
    ke_task_id_t const dest_id,
    ke_task_id_t const src_id)
{
    sample128_placeholder = param->val;
    return (KE_MSG_CONSUMED);
}
```

The above implementation jumps from the timer handler to the event handler. To avoid polluting the interrupt handler with code, we can encapsulate this functionality in a timer task and call it from the event handler instead.
You should be able to build and run the application at this time. Use LightBlue or BlueLoupe to verify that it all works. Set characteristic 2 to Notify in order to see the automatic updates.
7 Modifying sample128

In this section we will dig a little further into the sample128 service to see how it is constructed and to modify parts of it. The application implemented in the previous sections of this document will be used as a foundation upon which all further modifications will be based.

7.1 The basics of sample128

In the Bluetooth domain, a service consists of a collection of attributes or data chunks that are exposed to a connected client. These attributes are arranged in a database or table that is commonly referred to as the GATT database. A client device can explore (or discover) this database and determine the kind of attributes that are available and which methods can be used to interact with the database entries. A device will implement a GATT database which covers all the services that it provides.

The sample128 service contains just two characteristics. The first characteristic is one byte wide and facilitates read and write access to the client. The second characteristic, also one byte wide, facilitates read and notify access. When notification is activated for a characteristic, any change to the value data, will cause a Bluetooth Notify to be sent to the client device.

The database format is defined by the BT SIG. The database of Sample128, consisting of a total of 6 attributes, is structured as shown below:

<table>
<thead>
<tr>
<th>Handle (16-bit)</th>
<th>Attribute Declaration Type</th>
<th>Attribute Declaration Type ID</th>
<th>Size of Declaration Attribute Type ID [Bits]</th>
<th>Data Size [Bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Primary Service Declaration</td>
<td>0x2800</td>
<td>16</td>
<td>0xF0E0D0C0B0A0…</td>
</tr>
<tr>
<td>Start+1</td>
<td>Characteristic Declaration</td>
<td>0x2803</td>
<td>16</td>
<td>0x&lt;RD</td>
</tr>
<tr>
<td>Start+2</td>
<td>Characteristic value declaration</td>
<td>0x1F1E1D…</td>
<td>128</td>
<td>0x00</td>
</tr>
<tr>
<td>Start+3</td>
<td>Characteristic Declaration</td>
<td>0x2803</td>
<td>16</td>
<td>0x&lt;RD</td>
</tr>
<tr>
<td>Start+4</td>
<td>Characteristic value declaration</td>
<td>0x2F2E2D…</td>
<td>128</td>
<td>0x00</td>
</tr>
<tr>
<td>Start+5</td>
<td>Client configuration declaration</td>
<td>0x2902</td>
<td>16</td>
<td>0x0000</td>
</tr>
</tbody>
</table>

As illustrated in the table above, there are a total of six attributes that each are associated with a 16-bit (2 bytes) handle. Sample128 only contains 4 different declaration types (colour coded in the table above):

- One primary service declaration (sample128 service)
- Two characteristic declarations (one for each characteristic)
- Two characteristic value declarations (one for each characteristic)
- One client configuration declaration (enables notifications for the second characteristic)
7.1.1 The primary service declaration attribute

The primary service declaration attribute has a BT SIG assigned declaration type identifier of 0x2800 (16-bit) as shown in Table 1: The GATT database of sample128. The data component of a primary service is the UUID of the service, and because our service is custom (as in not specified by the BT SIG) it is 128-bit (16 bytes) wide. The value is specified in "sample128.c" as follows:

```
35    // sample128_1 Service
36    const struct att_uuid_128 sample128_svc =
37    {
38      {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
39        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F}
```

Figure 29: UUID of sample128 service

This translates to a UUID of 0x0F0E0D0C0B0A09080706050403020100. This UUID was completely randomly selected, hoping that it wouldn’t be used by somebody else. The only way to completely prevent this from happening would be to register a service UUID with the BT SIG – Note that such a UUID would be 16-bit; not 128-bit. There is a charge for BT SIG registration of a UUID.

7.1.2 The characteristic declaration attribute

Each of the two characteristics of sample128 is declared with a characteristic declaration type of 0x2803. The data component of a characteristic declaration consists of three pieces of information:

1. The properties bit field, that specifies how a client can access the characteristic (Read, Write, Notify, Indicate, Write without response etc.). The properties bit field is 1 byte wide.
2. The 2-byte handle to the value declaration of the characteristic. This enables a client device to access the value of a characteristic by referencing the handle directly.
3. The UUID of the characteristic. Any BT SIG assigned characteristic UUID would be 2 bytes wide, a custom UUID is 16 bytes wide.

The total size of a custom characteristic declaration’s data field is therefore 1 + 2 + 16 = 19 bytes.

The declaration of characteristic 1 can be found in "sample128.c":

```
43    struct att_char128_desc sample128_1_char =
44    {
45      { ATT_CHAR_PROP_RD | ATT_CHAR_PROP_WR,
46        {0,0},
47        {0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
48          0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F}
49    };
```

Figure 30: Characteristic 1 declaration

Note: The handle {0,0} is a placeholder that will be populated when the service database is created at runtime.

7.1.3 The characteristic value declaration attribute

Both characteristics of the sample128 service are custom types and therefore use UUIDs of 128 bits. The characteristics were defined as being able to contain data of only one byte each (we will modify the size of one of them later in this section.)

7.1.4 The client configuration declaration attribute

The final type is the client configuration attribute type. It is required only for characteristic 2 because only characteristic 2 enables notifications. A client device will write to the data component of this attribute in order to subscribe to notifications. A client configuration attribute is identified by the type
Developing a DA14580 Bluetooth profile using sample128

number (Attribute type UUID) of 0x2902. The data component is a 2-byte wide bit field (one bit specifies whether notification is active or not)

7.1.5 Summarizing the components of sample128

Table 1 can be used to detail some important information about sample128. We can deduce that there are 6 attributes in total. Two of the attributes use type IDs of 128 bits and the remaining four use only 16-bit type IDs. This information is used in “sample128_task.c”:

```
//Add Service Into Database
76    nb_att_16 = 4; // 4 UUID16 Attribute declaration types
77    nb_att_32 = 0; // 0 UUID32 Attribute declaration types
78    nb_att_128 = 2; // 2 UUID128 Attribute declaration types
```

Figure 31: Different sized declaration type IDs

We can also calculate the total required size of the service database (from “sample128_task.c”):

```
// Total Data portion of GATT database = 58 data bytes:
// + 16 Primary service declaration
// + 19 Declaration of characteristic 1
// + 1 Value declaration of characteristic 1
// + 19 Declaration of characteristic 2
// + 1 Value declaration of characteristic 2
// + 2 Client configuration declaration of characteristic 2
// = 58 Data bytes total
```

Figure 32: Adding service128 to the database

Understanding how we got to the numbers in the above code snippets (Figure 31 and Figure 32) based on the data in Table 1 allows us to start modifying sample128. Without this understanding, you could be in for a rough ride.

7.2 Modifying the data size of characteristic 1

In this section, we will modify the data size of characteristic 1 from one byte to an array of 8 bytes. To do this we will need to do the following:

- Define our new data type of 8-bytes and initialize a variable of this type
- Recalculate the size of the data in the GATT database
- Modify the value attribute to reflect the increased size
- Modify the messages that are sent between the application and sample128 and modify the functions that are involved.

7.2.1 Defining our new data type of 8 bytes

Defining a new variable type for our 8-byte characteristic value allows us later to modify its size in a single step. “sample128.h” is an appropriate place to define this new type

```
#ifndef SAMPLE128_H_
#define SAMPLE128_H_

19 typedef unsigned char my_new_t[8];
```

Figure 33: Defining a new type
We then initialize a new global variable of this type in "app_custom.proj.c", as follows:

```c
typedef unsigned char my_new_t[8];
```

**Figure 34: Initialization of a global variable**

```c
my_new_t sample128_my_new = {0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38};
```

### 7.2.2 Recalculating the size of the database

This is not really a challenge. The data chunk that previously was 1 byte wide is now 8 bytes, so we should simply adjust the total size upwards by 7. We can thus change the size of 58 to the new value of 65:

```c
status = attdb_add_service( &sample128_env.sample128_shdl, 
                         TASK_SAMPLE128, 
                         nb_att_16, 
                         nb_att_32, 
                         nb_att_128, 
                         65); // See calculation below
```

**Figure 35: Changes to the database size**

Just change the numbers in "sample128_task.c" as highlighted above.

### 7.2.3 Modifying the value attribute

This is another simple fix. We need to accommodate 8 bytes or sizeof(my_mew_t) instead of just one byte of data. In "sample128_task.c", make the highlighted changes:

```c
my_new_t sample128_my_new = {0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38};
```
7.2.4 Modifying messages between sample128 and the application

Two different structures, both carrying the value of characteristic 1, are used for sending messages between the application and the sample128 service. Both structures are defined in “sample128-task.h” and must be changed. The first structure, used when the service is enabled, should be changed as shown here:

```c
#define my_new_t  // New Data Type

typedef struct {
    uint8_t conn_hdl;  // Connection handle
    uint8_t sec_lvl;   // Security level
    my_new_t val1;     // Characteristic 1
    my_new_t val2;     // Characteristic 2
    uint8_t feature;   // Feature property status
} sample128_enable_req;
```

**Figure 37: The sample128_enable_req structure**

The other structure, in the same file, is used to indicate to the application when a connected client device is changing the value of characteristic 1. Modify the type of the value as shown below:

```c
typedef struct {
    uint16_t conn_hdl;  // Connection handle
    my_new_t val;       // Value
} sample128_val_ind;
```

**Figure 38: The sample128_val_ind structure**
The value of characteristic 1 is used twice in the “app_custom_proj.c” file. The first time is when the sample128 service is enabled. Make the changes as highlighted below:

```c
void app_sample128_enable(void)
{
    // Allocate the message
    struct sample128_enable_req* req = KE_MSG_ALLOC(
        SAMPLE128_ENABLE_REQ,
        TASK_SAMPLE128,
        TASK_APP,
        sample128_enable_req
    );
    req->conhdl = app_env.conhdl;
    req->sec_lvl = PERM(SVC, ENABLE);
    memcpy(req->sample128_1_val, &sample128_my_new, sizeof(my_new_t)); // default
    req->sample128_2_val = 0xff; // default value for sample128 characteristic 2
    req->feature = 0x00; // client CFG notify/indicate disabled
    // Send the message
    ke_msg_send(req);
}
```

Figure 39: Setting the default value via memcpy

```c
memcpy(req->sample128_1_val, &sample128_my_new, sizeof(my_new_t)); // default
```

The second use of the characteristic 1 value in “app_custom_proj.c” is when we receive an indication that a remote client has changed the value. We load the new value into our global value as shown below:

```c
int sample128_val_ind_handler(ke_msg_id_t const msqid,
                        struct sample128_val_ind const *param,
                        ke_task_id_t const dst_id,
                        ke_task_id_t const src_id)
{
    memcpy(&sample128_my_new, &param->val, sizeof(my_new_t));
    return (KE_MSG_CONSUMED);
}
```

Figure 40: Retrieving the value of characteristic 1

```c
memcpy(&sample128_my_new, &param->val, sizeof(my_new_t));
```

The function “sample128_send_val()” defined in “sample128.c” is responsible for sending the above indication to the application. This function must also be changed. In “sample128.h” we will change the prototype of the function:

```c
/**
 * Brief Send change to application.
 * @param my_new_val Value.
 * @brief Send value change to application.
 */
void sample128_send_val(my_new_t val);
```

Figure 41: The sample128_send_val prototype
And in “sample128.c” we need to make the following changes:

```c
void sample128_send_val(void *param)
{
    // Allocate character 1 change indication
    struct sample128_val_ind *ind = KE_MSG_ALLOC(SAMPLE128_VAL_IND,
        sample128_env.conn_info.appid, TASK_SAMPLE128,
        sample128_val_ind);

    // Fill in the parameter structure
    ind->connid = gapr_get_connid(sample128_env.conn_info.connid);
    memcpay(&ind->val,val,sizeof(my_new_t));

    // Send the message
    ke_msg_send(ind);
}
```

Figure 42: Changes to sample128_send_val

```c
memcpy(&ind->val,val,sizeof(my_new_t));
```

The above function is called by “gattc_write_cmd_ind_handler()” defined in “sample128_task.c”. In this function we will need to make the following change:

```c
static int gattc_write_cmd_ind_handler(ke_msg_id_t const msgid,
    struct gattc_write_cmd_ind const *param,
    ke_task_id_t const dest_id,
    ke_task_id_t const src_id)
{
    uint8_t char_code = SAMPLE128_ERR_CHAR;
    uint8_t status = PRF_AF_COMM_ERROR;

    if (KE_IDK_GET(msgid) == sample128_env.conn_info.connid) {
        if (param->handle == sample128_env.sample128_shdl + SAMPLE128_1_IDK_VAL) {
            char_code = SAMPLE128_1_CHAR;
        }
        if (param->handle == sample128_env.sample128_shdl + SAMPLE128_2_IDK_CFG) {
            char_code = SAMPLE128_2_CFG;
        }
        if (char_code == SAMPLE128_1_CHAR) {
            //Save value in DB
            atmdm_nnt_set_value(param->handle, sizeof(my_new_t), (uint8_t *)&param->value[0]);
            if(!param->last) {
                sample128_send_val((uint8_t *)&param->value[0]);
            }
            status = PRF_ERR_OK;
        }
    }
}
```

Figure 43: Changes to gattc_write_cmd_ind_handler

```c
memcpy(&ind->val,val,sizeof(my_new_t));
```

```c
sample128_send_val((uint8_t *)&param->value[0]);
```

And finally we have to make the same change in the “sample128_enable_req_handler()” function, also defined in “sample128_task.c”
7.3 Adding a new characteristic to the service

In this section, we will add a completely new custom characteristic to the service. We will enable read and notification access to the characteristic and allow it to carry a total of 10 bytes.

The tasks ahead of us are as follows:

- Make another type definition, to make it easier to change the size if we decide to do so at some point.
- Recalculate the number of 128-bit declaration type IDs and recalculate the size of the data in the GAP database.
- Build the new database.
- Implement new functionality in sample128 that allows us to change the value of the new characteristic.

7.3.1 Defining our new data type of 10 bytes

As with our previously created data type, we will place our new type in "sample128.h"

```c
typedef unsigned char my_new_t[8];
typedef unsigned char my_newer_t[10];
```

Figure 45: Defining a new data type in sample128.h
7.3.2 Calculating the size of the new database

The database table must be changed. We are going to need another 3 attributes for our new characteristic. Note that we have also changed the data size for characteristic 2, according to our modifications in the previous section.

Table 2: The new GATT table

<table>
<thead>
<tr>
<th>Handle (16-bit)</th>
<th>Attribute Declaration Type</th>
<th>Attribute Declaration Type ID</th>
<th>Size of Declaration Attribute Type ID [Bits]</th>
<th>Data</th>
<th>Data size [Bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Primary Service Declaration</td>
<td>0x2800</td>
<td>16</td>
<td>0x0F0E0D0C0B0A...</td>
<td>16</td>
</tr>
<tr>
<td>Start+1</td>
<td>Characteristic Declaration</td>
<td>0x2803</td>
<td>16</td>
<td>0x&lt;RD</td>
<td>WR&gt;&lt;start+2&gt;1F1E...</td>
</tr>
<tr>
<td>Start+2</td>
<td>Characteristic value declaration</td>
<td>0x1F1E1D...</td>
<td>128</td>
<td>0x00</td>
<td>8!!!</td>
</tr>
<tr>
<td>Start+3</td>
<td>Characteristic Declaration</td>
<td>0x2803</td>
<td>16</td>
<td>0x&lt;RD</td>
<td>NTFY&gt;&lt;start+4&gt;2F2E...</td>
</tr>
<tr>
<td>Start+4</td>
<td>Characteristic value declaration</td>
<td>0x2F2E2D...</td>
<td>128</td>
<td>0x00</td>
<td>1</td>
</tr>
<tr>
<td>Start+5</td>
<td>Client configuration declaration</td>
<td>0x2902</td>
<td>16</td>
<td>0x0000</td>
<td>2</td>
</tr>
<tr>
<td>Start+6</td>
<td>Characteristic Declaration</td>
<td>0x2803</td>
<td>16</td>
<td>0x&lt;RD</td>
<td>NTFY&gt;&lt;start+7&gt;3F3E...</td>
</tr>
<tr>
<td>Start+7</td>
<td>Characteristic value declaration</td>
<td>0x3F3E3D...</td>
<td>128</td>
<td>0x00</td>
<td>10</td>
</tr>
<tr>
<td>Start+8</td>
<td>Client configuration declaration</td>
<td>0x2902</td>
<td>16</td>
<td>0x0000</td>
<td>2</td>
</tr>
</tbody>
</table>

As can be seen in Table 2, we are adding three attributes. Two of the attributes are referenced using a 16-bit type ID and one is referenced with a 128-bit type ID. We can also see that we are adding 19 + 10 + 2 = 31 data bytes to the database. This information allows us to make the following code changes to “sample128_task.c”: 

```c
typedef unsigned char my_newer_t[10];
```
7.3.3 Building the new database

Adding the three new attributes to the database can be done by copying and slightly modifying the sequence from characteristic 2. In “sample128_task.c”, add the following:
//Characteristic 3:
// Add characteristic declaration attribute to database
status = attmdb_add_attribute( sample128_env.sample128_shdl,
    ATT_UUID_128_LEN + 3, //Data size = 19
    ATT_UUID_16_LEN, //Size of declaration type ID
    (uint8_t*) &att_decl_char, // 0x2803
    PERM(RD, ENABLE), // Permissions
    &(char_hdl) // Handle to the characteristic declaration
);

// Add characteristic value declaration attribute to database
status = attmdb_add_attribute( sample128_env.sample128_shdl,
    sizeof(my_newer_t), //Data size = 10 Bytes
    ATT_UUID_128_LEN, // Size of custom type ID = 128-bit
    &sample128_3_val.uuid, // UUID
    PERM(RD, ENABLE) | PERM(NTF, ENABLE), // Permissions
    &(val_hdl) // Handle to the value attribute
);

// Store the value handle for characteristic 3
memcpy(sample128_3_char.attr_hdl, &val_hdl, sizeof(uint16_t));

// Set initial value of characteristic 3
status = attmdb_att_set_value(char_hdl,
    sizeof(sample128_3_char),
    (uint8_t*)sample128_3_char);

// Add client configuration declaration attribute to database (Facilitates Notify)
status = attmdb_add_attribute( sample128_env.sample128_shdl,
    sizeof(uint16_t), // Data size 2bytes (16-bit)
    ATT_UUID_16_LEN, // Size of client configuration type ID
    &attdecl_cfg, // 0x2902 UUID
    PERM(RD, ENABLE) | PERM(WR, ENABLE), // Permissions
    &(val_hdl) // Handle to value attribute
);
We need to define the attribute values of the new characteristic. This is done in "sample128.c" as follows:

```c
const struct att_uuid_128 sample128_3_val = {{0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F}};

struct att_char128_desc sample128_3_char = {ATT_CHAR_PROP_RD | ATT_CHAR_PROP_NTF, {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F}};
```

Figure 47: Defining attribute values

We need to be able to reference the three new attributes. This is achieved by enumerating them in "sample128.h":

```c
const struct att_uuid_128 sample128_3_val = {{0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F}};

struct att_char128_desc sample128_3_char = {ATT_CHAR_PROP_RD | ATT_CHAR_PROP_NTF, {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F}};
```
In "sample128.h", we make the following changes to accommodate the new characteristic:

```
/*
 * SAMPLE128 PROFILE ATTRIBUTES VALUES DECLARATION
 * ----------------------------------------------
 */

/// sample128 Service
extern const struct att_uuid_128 sample128_svc;
/// sample128_1 - Characteristic
extern struct att_char128_desc sample128_1_char;
/// sample128_1 - Value
extern const struct att_uuid_128 sample128_1_val;
/// sample128_2 - Characteristic
extern struct att_char128_desc sample128_2_char;
/// sample128_2 - Value
extern const struct att_uuid_128 sample128_2_val;
/// sample128_3 - Characteristic
extern struct att_char128_desc sample128_3_char;
/// sample128_3 - Value
extern const struct att_uuid_128 sample128_3_val;
```
You should now be able to build the modified code. You can also load it to your DVK and use Light Blue (iOS) or BlueLoupe (Android) to see that the new characteristic shows up. Note: you may have to turn Bluetooth on your smart device off and back on to see the change:

![BlueLoupe screenshot](image)

**Figure 50: The new characteristic is exposed (BlueLoupe)**

### 7.3.4 Initializing the characteristic value

At this point we have successfully built the new and expanded GATT data base, and it is time to start actually using it. The first thing to do is to initialise the value of the new characteristic. In this tutorial, we will simply define a global variable in "app_custom_proj.c":

```c
my_newer_t sample128_my_newer = {0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A};
```

**Figure 51: Initialization of a global variable**

Note: As mentioned earlier, these global variables will not be retained if deep sleep is enabled. Use retention RAM to store these types of variables if you plan to use deep sleep.

We need to modify the structure used when we enable the service, in order to allow us to initialize the database upon client connection. The structure, defined in "sample128_task.h", must be modified as follows:

```c
my_newer_t sample128_my_newer = {0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A};
```
We are ready to initialize the new characteristic where we enable the service in “sample128_task.c”:

```
static int sample128_enable_req_handler(int req_id, const sample128_enable_req *param,
    const char *req_handler_name)
{
    if (req_id == SAMPLE128_ENABLE_REQ_HANDLERS_ENABLE_REQ) {
        // The connection doesn’t exist, request disallows
        return XPAR_ERR_REQ_DISALLOWED,
        XPARILS_DISABLE_IND, XPARILS_DISABLE_REQ);
    } else {
        // Sample128 service permissions
        atmdrv_set_permision(sample128_env, sample128_shd1, param->svc_lvl);
        // Set characteristic 1 to specified value
        atmdrv_att_set_value(sample128_env, sample128_shd1 + SAMPLE128_1_IDX_VAL,
            static_cast<uint8_t>(my_newer_t feature3[0]), (uint8_t *)param->sample128_1_val);
        // Set characteristic 2 to specified value
        atmdrv_att_set_value(sample128_env, sample128_shd1 + SAMPLE128_2_IDX_VAL,
            static_cast<uint8_t>(my_newer_t feature3[1]), (uint8_t *)param->sample128_2_val);
        // Set characteristic 3 to specified value
        atmdrv_att_set_value(sample128_env, sample128_shd1 + SAMPLE128_3_IDX_VAL,
            static_cast<uint8_t>(my_newer_t feature3[2]), (uint8_t *)param->sample128_3_val);
    }
    return XPAR_ERR_OK;
}
```

Figure 52: Modifying the enable structure

Figure 53: Initialization of the characteristic value
7.3.5 Setting the default value of characteristic 3

We will have to set the default value of the new characteristic when the service is enabled. In “app_custom_proj.c”, add the following line:

```c
void app_sample128_enable(void)
{
    // Allocate the message
    struct sample128_enable_req* req = KE_MSG_ALLOC(SAMPLE128_ENABLE_REQ, TASK_SAMPLE128, TASK_APP, sample128_enable_req);

    req->conhdl = app_env.conhdl;
    req->scvlvl = PERM(SVC, ENABLE);
    memcpy(&req->sample128_3_val, &sample128_my_newer, sizeof(my_newer_t)); // default
    memmove((req->sample128_3_val, &sample128_my_newer, sizeof(my_newer_t))); // default
    req->sample128_2_val = 0xff;  // default value for sample128_characteristic 2
    req->feature = 0x00;  // client CFG notify/indicate disabled
    // Send the message
    ke_msg_send(req);
}
```

Figure 54: Default value of characteristic 3

7.3.6 Updating the characteristic value from the application

We also need a new structure for updating the value of our new characteristic from the application. We will define this structure in “sample128_task.h”:

```c
// Parameters of the @ref SAMPLE128_UPD_CHAR2_REQ message
struct sample128_upd_char2_req
{
    /// Connection handle
    uint16_t conhdl;
    /// Characteristic Value
    uint8_t* t val;
};

// Parameters of the @ref SAMPLE128_UPD_CHAR3_REQ message
struct sample128_upd_char3_req
{
    /// Connection handle
    uint16_t conhdl;
    /// Characteristic Value
    my_newer_t* t val;
};
```

Figure 55: New characteristic update structure
We will need a couple of new message primitives to be able to update the characteristic. In "sample128_task.h" add these two primitives:

```c
// Parameters of the @ref SAMPLE128_UPD_CHAR3_REQ message
struct sample128_upd_char3_req
{
    /// Connection handle
    uint16_t conhdl;
    /// Characteristic Value
    my_newer_t val;
};
```

We also need to implement a new handler for sample128 to manage the value update. The handler must be implemented among the connected state handlers of sample128 defined in "sample128_task.c":

```c
// Update value of characteristic 3
SAMPLE128_UPD_CHAR3_REQ,
/// Confirm the update of value of characteristic 3
SAMPLE128_UPD_CHAR3_CFM,
```

Finally, we must implement the handler function itself. We will just copy the handler function for characteristic 2, and make appropriate adjustments. Place this code in "sample128_task.c" just below the "sample128_upd_char2_req_handler()" function.
Note: We are reusing the “sample128_upd_char2_cfm_send()” function. Our application doesn’t act on the confirmation anyway.

At this time we are ready to change the value of characteristic 3 from the application. We will simply reuse our timer handler and change the first byte of the characteristic value every time the timer times out. Make the following changes to the timer handler function in “app_custom_proj.c”:

```c
static int sample128_upd_char3_req_handler(ke_msg_id_t const msgid, 
    struct sample128_upd_char3_req const *param, 
    ke_task_id_t const dest_id, 
    ke_task_id_t const src_id) 
{
    uint8_t status = PRF_ERR_OK;

    // Check provided values
    if(param->conhdl == gapc_get_conhdl(sample128_env.con_info.conidx))
    {
        // Update value in database
        attmdb_att_set_value(sample128_env.sample128_shdl + SAMPLE128_3_IDX_VAL, 
            sizeof(my_newer_t), (uint8_t *)&param->val);

        if((sample128_env.feature3 & PRF_CLI_START_NTF))
            // Send notification through GATT
            prf_server_send_event((prf_env_struct *)&sample128_env, false, 
                sample128_env.sample128_shdl + SAMPLE128_3_IDX_VAL);
    }
    else
    {
        status = PRF_ERR_INVALID_PARAM;
    }

    if (status != PRF_ERR_OK)
    {
        sample128_upd_char2_cfm_send(status);
    }

    return (KE_MSG_CONSUMED);
}
```
7.3.7 Implementing support for GATT notify

A connected client subscribes to notification of changes to the characteristic value by writing to the client configuration attribute value of the characteristic. We will need a way to distinguish the different write actions from each other. An enumeration is used for this purpose, and we will have to add our new characteristic's client configuration to the enumeration in "sample128.h".

```c
struct sample128_upd_char3_req *req3 = KE_MSG_ALLOC(SAMPLE128_UPD_CHAR3_REQ, TASK_SAMPLE128, TASK_APP, sample128_upd_char3_req);
memcpy(&req3->val, &sample128_my_newer, sizeof(my_newer_t));
memcpy(&req3->val, &sample128_placeholder, 1);
req3->conhdl = app_env.conhdl;
ke_msg_send(req3);
```
In order to keep track of whether notifications are activated for the individual characteristic, we will add a parameter to the environment structure of the service. This must be done in "sample128.h":

```
uint8_t feature3;
```

When we enable the service, we must remember to specify whether notifications are set for the characteristic. Near the bottom of the "sample128_enable_req_handler()" function in "sample128_task.c" add the following:
Finally, we will have to set a flag when a connected client subscribes or unsubscribes to notifications. A client will write a 1 to our client configuration attribute to subscribe and a 0 to unsubscribe. In “sample128_enable_req_handler” of “sample128_task.c” add the following:
And we must change the last “else if” statement in the same function so that it can also handle notification subscriptions to characteristic 3. Replace the entire “else if” block with the following:

```c
if (param->handle == sample128_env.sample128_shdl + SAMPLE128_3_IDX_CFG)
{
  char_code = SAMPLE128_3_CFG;
}
```
else if ( (char_code == SAMPLE128_2_CFG)||(char_code == SAMPLE128_3_CFG))
{
    // Written value
    uint16_t ntf_cfg;

    // Extract value before check
    ntf_cfg = co_read16p(&param->value[0]);

    // Only update configuration if value for stop or notification enable
    if ((ntf_cfg == PRF_CLI_STOP_NTFIND) || (ntf_cfg == PRF_CLI_START_NTF))
    {
        // Save value in DB
        attmdb_att_set_value(param->handle, sizeof(uint16_t), (uint8_t *)&param->value[0]);

        // Conserve information in environment
        if (ntf_cfg == PRF_CLI_START_NTF)
        {
            // Ntf cfg bit set to 1
            if(char_code == SAMPLE128_2_CFG)
                sample128_env.feature |= PRF_CLI_START_NTF;
            else
                sample128_env.feature3 |= PRF_CLI_START_NTF;
        }
        else
        {
            // Ntf cfg bit set to 0
            if(char_code == SAMPLE128_2_CFG)
                sample128_env.feature &= ~PRF_CLI_START_NTF;
            else
                sample128_env.feature3 &= ~PRF_CLI_START_NTF;
        }
    status = PRF_ERR_OK;
    }
}
8 Revision history

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>25-April-2015</td>
<td>Initial version.</td>
</tr>
</tbody>
</table>
Developing a DA14580 Bluetooth profile using sample128

Status definitions

<table>
<thead>
<tr>
<th>Status</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAFT</td>
<td>The content of this document has not been approved for publication.</td>
</tr>
</tbody>
</table>

Disclaimer

Information in this document is believed to be accurate and reliable. However, Dialog Semiconductor does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information. Dialog Semiconductor furthermore takes no responsibility whatsoever for the content in this document if provided by any information source outside of Dialog Semiconductor.

Dialog Semiconductor reserves the right to change without notice the information published in this document, including without limitation the specification and the design of the related semiconductor products, software and applications.

Applications, software, and semiconductor products described in this document are for illustrative purposes only. Dialog Semiconductor makes no representation or warranty that such applications, software and semiconductor products will be suitable for the specified use without further testing or modification. Unless otherwise agreed in writing, such testing or modification is the sole responsibility of the customer and Dialog Semiconductor excludes all liability in this respect.

Customer notes that nothing in this document may be construed as a license for customer to use the Dialog Semiconductor products, software and applications referred to in this document. Such license must be separately sought by customer with Dialog Semiconductor.

All use of Dialog Semiconductor products, software and applications referred to in this document are subject to Dialog Semiconductor’s Standard Terms and Conditions of Sale, unless otherwise stated.

© Dialog Semiconductor GmbH. All rights reserved.

RoHS Compliance

Dialog Semiconductor complies with European Directive 2001/95/EC and from 2 January 2013 onwards to European Directive 2011/65/EU concerning Restriction of Hazardous Substances (RoHS/RoHS2). Dialog Semiconductor’s statement on RoHS can be found on the customer portal https://support.diasemi.com/. RoHS certificates from our suppliers are available on request.

Contacting Dialog Semiconductor

Germany Headquarters
Dialog Semiconductor GmbH
Phone: +49 7021 805-0

United Kingdom
Dialog Semiconductor (UK) Ltd
Phone: +44 1793 757700

The Netherlands
Dialog Semiconductor B.V.
Phone: +31 73 640 8822

Email: enquiry@diasemi.com

Web site: www.dialog-semiconductor.com

North America
Dialog Semiconductor Inc.
Phone: +1 408 845 8500

Japan
Dialog Semiconductor K. K.
Phone: +81 3 5425 4567

Taiwan
Dialog Semiconductor Taiwan
Phone: +886 281 786 222

Email: enq@diasemi.com

Web site: www.dialog-semiconductor.com

Singapore
Dialog Semiconductor Singapore
Phone: +65 64 849929

China
Dialog Semiconductor China
Phone: +86 21 5178 2561

Korea
Dialog Semiconductor Korea
Phone: +82 2 3469 8291

RoHS Compliance

Dialog Semiconductor complies with European Directive 2001/95/EC and from 2 January 2013 onwards to European Directive 2011/65/EU concerning Restriction of Hazardous Substances (RoHS/RoHS2). Dialog Semiconductor’s statement on RoHS can be found on the customer portal https://support.diasemi.com/. RoHS certificates from our suppliers are available on request.

Contacting Dialog Semiconductor

Germany Headquarters
Dialog Semiconductor GmbH
Phone: +49 7021 805-0

United Kingdom
Dialog Semiconductor (UK) Ltd
Phone: +44 1793 757700

The Netherlands
Dialog Semiconductor B.V.
Phone: +31 73 640 8822

Email: enquiry@diasemi.com

Web site: www.dialog-semiconductor.com

RoHS Compliance

Dialog Semiconductor complies with European Directive 2001/95/EC and from 2 January 2013 onwards to European Directive 2011/65/EU concerning Restriction of Hazardous Substances (RoHS/RoHS2). Dialog Semiconductor’s statement on RoHS can be found on the customer portal https://support.diasemi.com/. RoHS certificates from our suppliers are available on request.

Contacting Dialog Semiconductor

Germany Headquarters
Dialog Semiconductor GmbH
Phone: +49 7021 805-0

United Kingdom
Dialog Semiconductor (UK) Ltd
Phone: +44 1793 757700

The Netherlands
Dialog Semiconductor B.V.
Phone: +31 73 640 8822

Email: enquiry@diasemi.com

Web site: www.dialog-semiconductor.com

RoHS Compliance

Dialog Semiconductor complies with European Directive 2001/95/EC and from 2 January 2013 onwards to European Directive 2011/65/EU concerning Restriction of Hazardous Substances (RoHS/RoHS2). Dialog Semiconductor’s statement on RoHS can be found on the customer portal https://support.diasemi.com/. RoHS certificates from our suppliers are available on request.

Contacting Dialog Semiconductor

Germany Headquarters
Dialog Semiconductor GmbH
Phone: +49 7021 805-0

United Kingdom
Dialog Semiconductor (UK) Ltd
Phone: +44 1793 757700

The Netherlands
Dialog Semiconductor B.V.
Phone: +31 73 640 8822

Email: enquiry@diasemi.com

Web site: www.dialog-semiconductor.com

RoHS Compliance

Dialog Semiconductor complies with European Directive 2001/95/EC and from 2 January 2013 onwards to European Directive 2011/65/EU concerning Restriction of Hazardous Substances (RoHS/RoHS2). Dialog Semiconductor’s statement on RoHS can be found on the customer portal https://support.diasemi.com/. RoHS certificates from our suppliers are available on request.