Application Note

Developing a DA14580 Bluetooth Profile Using Sample128

AN-B-029

Abstract

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

Contents

Figures

Tables

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:

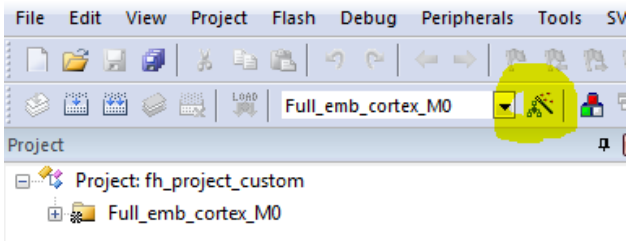

Figure 1: Options for target

Open the “C/C++” tab and click on the button to the right of the Include Paths field:

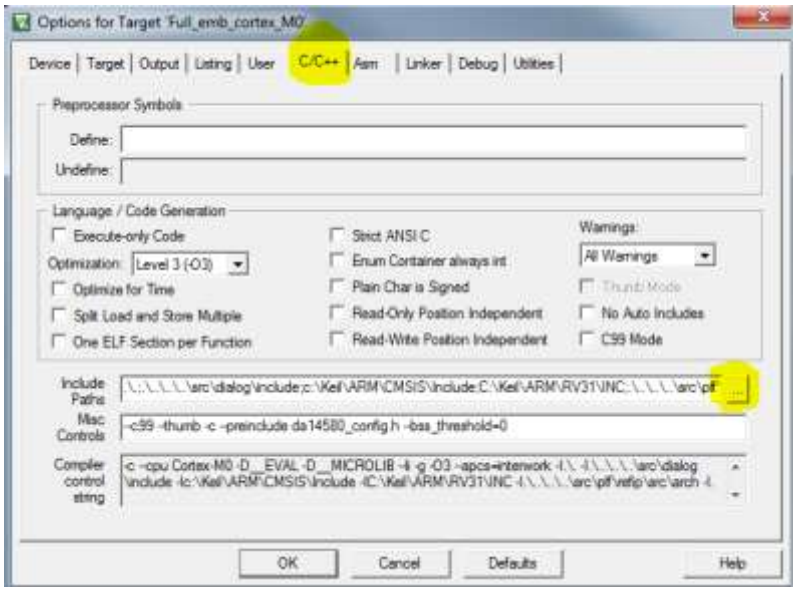

Figure 2: Opening include paths list

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):

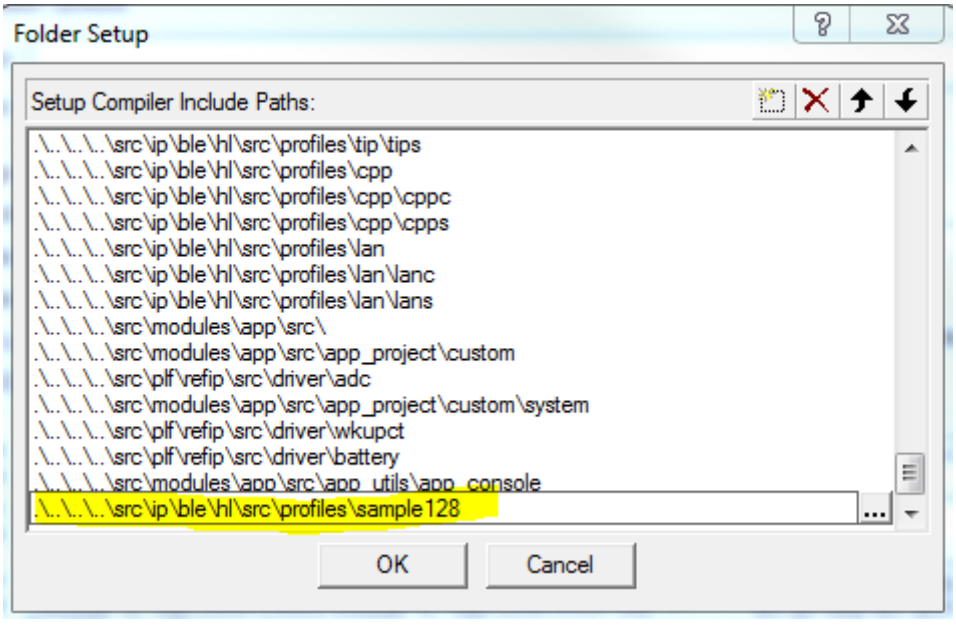

Figure 3: Adding an include path

Or, for the copy & paste fans out there:

```
.\..\..\..\src\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:

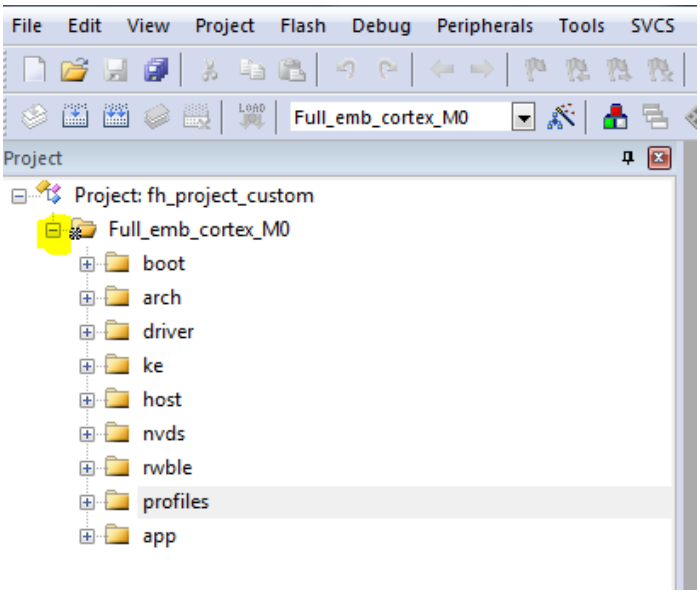

Figure 4: Adding source code

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”:

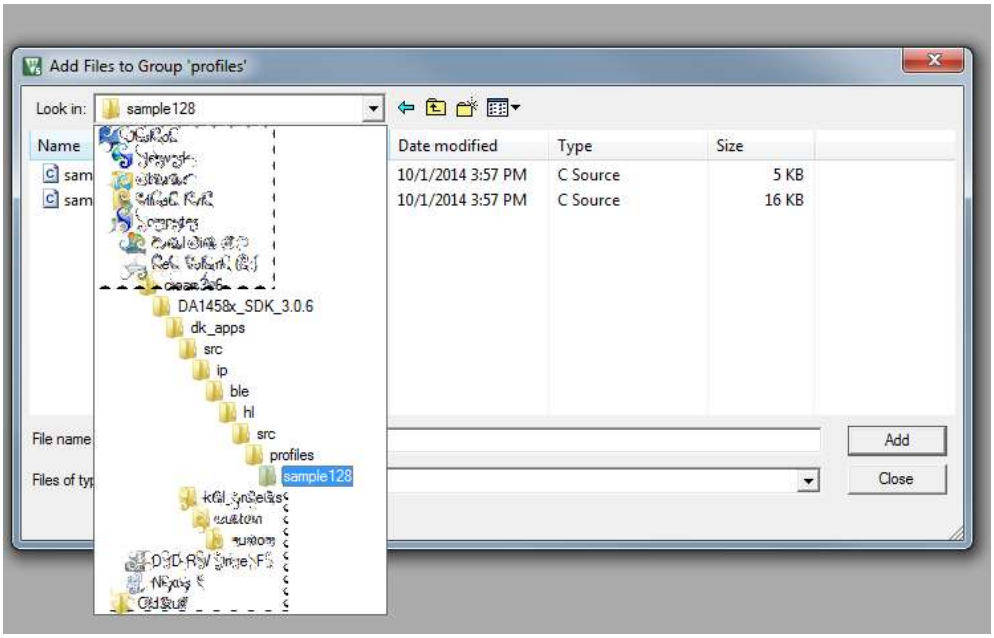

Figure 5: Finding source files

You should see the following two files:

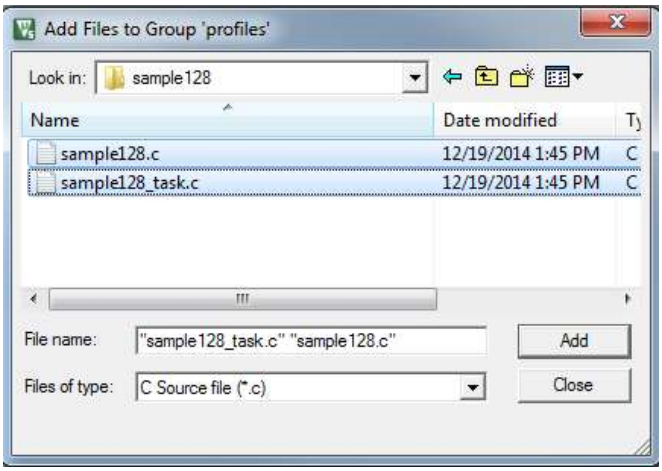

Figure 6: Sample128 source files

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:

```
100 /* Accelerometer Profile */
101 #undef CFG_PRF_ACCEL
102 /* Dialog's Software Patch Over The Air Profile */
103 #undef CFG_PRF_SPOTAR
104 /* Dialog's Sample 128 bit UUIDs Profile */
105 #define CFG_PRF_SAMPLE128
106 ////////////////////////////////////////////////////////
107
```

Figure 7: Configuration of the project

```
#define CFG_PRF_SAMPLE128
```

The file sample128_task header should also be added to the project header file “app_custom_proj.h”:

```
32 /*
33  * INCLUDE FILES
34  ****************************************************************************
35  */
36
37 #include "rwble_config.h"
38 #include "app_task.h"                  // application task
39 #include "gapc_task.h"                 // gap functions and messages
40 #include "gapm_task.h"                 // gap functions and messages
41 #include "app.h"                       // application definitions
42 #include "co_error.h"                  // error code definitions
43 #include "smpc_task.h"                 // error code definitions
44
45 #include "sample128_task.h"
46
```

Figure 8: Including the service

```
#include "sample128_task.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:

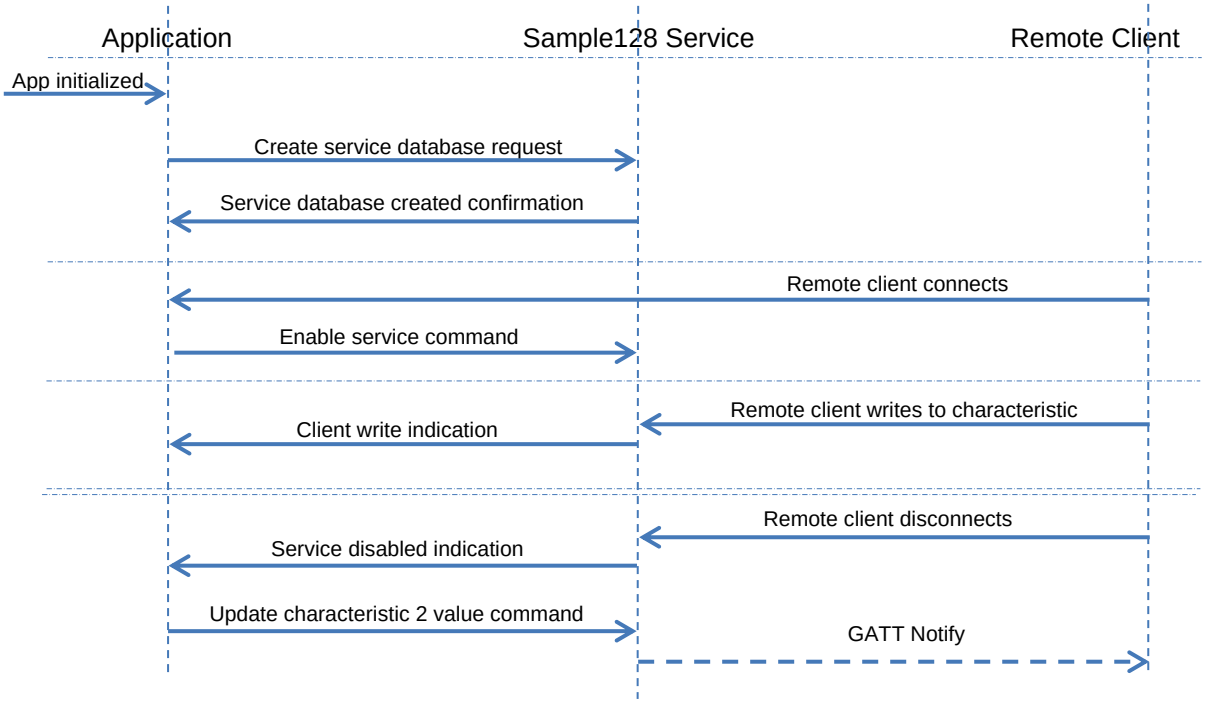

Figure 9: Message flow diagram

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”:

```
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;

        }

        // Select following service to add
        app_env.next_prf_init++;
    }
    else
    {
        end_db_create = true;
    }

    return end_db_create;
}
```

Figure 10: Creating the database

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

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:

```
109 #endif //(BLE_BATT_SERVER)
110 #if (BLE_HID_DEVICE)
111     APP_HOGPD_TASK,
112 #endif //(BLE_APP_KEYBOARD)
113 #if (BLE_SPOTA_RECEIVER)
114     APP_SPOTAR_TASK,
115 #endif //(BLE_SPOTA_RECEIVER)
116
117 #if (BLE_SAMPLE128)
118     APP_SAMPLE128,
119 #endif //(BLE_SAMPLE128)
120
121     APP_PRF_LIST_STOP,
122 };
```

Figure 11: Enumerating the service database

```
#if (BLE_SAMPLE128)
  APP_SAMPLE128,
#endif //(BLE_SAMPLE128)
```

We must also implement the “app_sample128_create_db_send()” function. This can be done in the “app_custom_proj.c” file in the function definitions segment:

```
48 /*
49  * FUNCTION DEFINITIONS
50  ****************************************************************************************
51  */
52
53 void app_sample128_create_db_send(void)
54 {
55     struct sample128_create_db_req *req = KE_MSG_ALLOC(
56                                                        SAMPLE128_CREATE_DB_REQ,
57                                                        TASK_SAMPLE128, TASK_APP,
58                                                        sample128_create_db_req
59                                                       );
60     ke_msg_send(req);
61 }
```

Figure 12: Creating the service database

```
void app_sample128_create_db_send(void)
{
  struct sample128_create_db_req *req = KE_MSG_ALLOC(
                                                     SAMPLE128_CREATE_DB_REQ,
                                                     TASK_SAMPLE128,
                                                     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.

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”:

```
137 void app_connection_func(struct gapc_connection_req_ind const *param)
138 {
139     if (app_env.conidx != GAP_INVALID_CONIDX)
140     {
141
142 /*************************************************
143 Handle connection request event. Enable required profiles
144
145 i.e.
146 #if (BLE_DIS_SERVER)
147     app_dis_enable_prf(app_env.conhdl);
148 #endif
149 *************************************************/
150
151         app_sample128_enable();
152
```

Figure 13: Enabling the service

```
app_sample128_enable();
```

We will implement the “app_sample128_enable()” function in the function definition segment of “app_custom_proj.c” - just below the definition of “app_sample128_create_db_send()”:

```
63 void app_sample128_enable(void)
64 {
65   // Allocate the message
66   struct sample128_enable_req* req = KE_MSG_ALLOC(
67                                                   SAMPLE128_ENABLE_REQ,
68                                                   TASK_SAMPLE128,
69                                                   TASK_APP,
70                                                   sample128_enable_req
71                                                  );
72   req->conhdl = app_env.conhdl;
73   req->sec_lvl = PERM(SVC, ENABLE);
74   req->sample128_1_val = 0x01;  // default value for sample128 characteristic 1
75   req->sample128_2_val = 0xff;  // default value for sample128 characteristic 2
76   req->feature = 0x00;          // client CFG notify/indicate disabled
77   // Send the message
78   ke_msg_send(req);
79 }
```

Figure 14: Sending service enable message

```
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);
}
```

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:

```
#if (BLE_STREAMDATA_DEVICE)
  {STREAMDATAD_CREATE_DB_CFM,              (ke_msg_func_t)stream_create_db_cfm_handler},
    {STREAMDATAD_RCV_DATA_PACKET_IND,        (ke_msg_func_t)stream_rcv_data_packet_ind_handler},
#endif //BLE_STREAMDATA_DEVICE

#if 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

```
#if 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
```

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

```
/*
 * 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 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);
```

Figure 16: Event handler prototypes

```
/**
 ****************************************************************************************
 * @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);
```

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

```
776 #endif //BLE_APP_SEC
777
778
779 /**
780  ****************************************************************************************
781  * @brief Handles Sample128  profile database creation confirmation.
782  *
783  * @param[in] msgid     Id of the message received.
784  * @param[in] param     Pointer to the parameters of the message.
785  * @param[in] dest_id   ID of the receiving task instance .
786  * @param[in] src_id    ID of the sending task instance.
787  *
788  * @return If the message was consumed or not.
789  ****************************************************************************************
790  */
791 int sample128_create_db_cfm_handler(ke_msg_id_t const msgid,
792                                     struct sample128_create_db_cfm const *param,
793                                     ke_task_id_t const dest_id,
794                                     ke_task_id_t const src_id)
795 {
796     // If state is not idle, ignore the message
797     if (ke_state_get(dest_id) == APP_DB_INIT)
798     {
799
800         // Inform the Application Manager
801         struct app_module_init_cmp_evt *cfm = KE_MSG_ALLOC(APP_MODULE_INIT_CMP_EVT,
802                                                            TASK_APP, TASK_APP,
803                                                            app_module_init_cmp_evt);
804
805         cfm->status = param->status;
806
807         ke_msg_send(cfm);
808
809     }
810
811     return (KE_MSG_CONSUMED);
812 }
```

Figure 17: Database created handler

```
/**
 ****************************************************************************************
 * @brief Handles Sample128  profile database creation confirmation.
 *
 * @param[in] msgid     Id of the message received.
 * @param[in] param     Pointer to the parameters of the message.
 * @param[in] dest_id   ID of the receiving task instance .
 * @param[in] src_id    ID of the sending task instance.
 *
 * @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)
{
    // If state is not idle, ignore the message
    if (ke_state_get(dest_id) == APP_DB_INIT)
    {

        // Inform the Application Manager
        struct app_module_init_cmp_evt *cfm = KE_MSG_ALLOC(APP_MODULE_INIT_CMP_EVT,
                                                           TASK_APP, TASK_APP,
                                                           app_module_init_cmp_evt);

        cfm->status = param->status;

        ke_msg_send(cfm);

    }

    return (KE_MSG_CONSUMED);
}
```

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”:

```
/**
 ****************************************************************************************
 * @brief Handles disable indication from Sample128 profile.
 *
 * @param[in] msgid     Id of the message received.
 * @param[in] param     Pointer to the parameters of the message.
 * @param[in] dest_id   ID of the receiving task instance.
 * @param[in] src_id    ID of the sending task instance.
 *
 * @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)
{
  return (KE_MSG_CONSUMED);
}

/**
 ****************************************************************************************
 * @brief Handles write of 1st characteristic event indication from sample128 profile
 *
 * @param[in] msgid     Id of the message received.
 * @param[in] param     Pointer to the parameters of the message.
 * @param[in] dest_id   ID of the receiving task instance (TASK_GAP).
 * @param[in] src_id    ID of the sending task instance.
 *
 * @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)
{
  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.

```
/**
 ****************************************************************************************
 * @brief Handles disable indication from Sample128 profile.
 *
 * @param[in] msgid     Id of the message received.
 * @param[in] param     Pointer to the parameters of the message.
 * @param[in] dest_id   ID of the receiving task instance.
 * @param[in] src_id    ID of the sending task instance.
 *
 * @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)
{
  return (KE_MSG_CONSUMED);
}

/**
 ****************************************************************************************
 * @brief Handles write of 1st characteristic event indication from sample128 profile
 *
 * @param[in] msgid     Id of the message received.
 * @param[in] param     Pointer to the parameters of the message.
 * @param[in] dest_id   ID of the receiving task instance (TASK_GAP).
 * @param[in] src_id    ID of the sending task instance.
 *
 * @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)
{
  return (KE_MSG_CONSUMED);
}
```

By this time, you should be able to build the application and load it onto your DVK.

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 BlueLoupe

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:

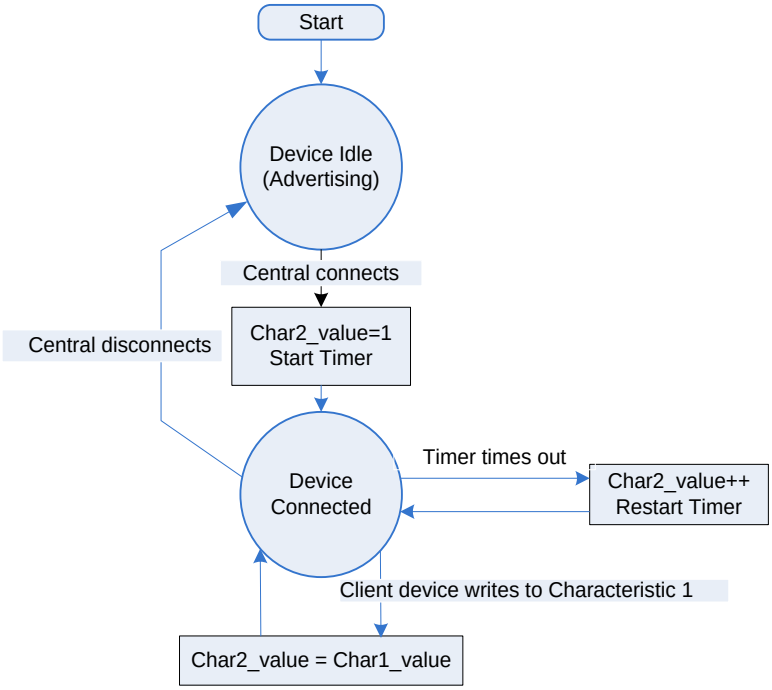

Figure 20: Sample128 tutorial functionality

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”:

```
124  /// APP Task messages
125  enum APP_MSG
126  {
127      APP_MODULE_INIT_CMP_EVT = KE_FIRST_MSG(TASK_APP),
128
129  #if BLE_SAMPLE128
130    APP_SAMPLE128_TIMER,
131  #endif //BLE_SAMPLE128
132
133  #if BLE_ACCEL
```

Figure 21: Adding a message primitive

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

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:

```
184
185  #if BLE_SAMPLE128
186    {SAMPLE128_CREATE_DB_CFM,  (ke_msg_func_t)sample128_create_db_cfm_handler},
187    {SAMPLE128_VAL_IND,        (ke_msg_func_t)sample128_val_ind_handler},
188    {SAMPLE128_DISABLE_IND,    (ke_msg_func_t)sample128_disable_ind_handler},
189    {APP_SAMPLE128_TIMER,      (ke_msg_func_t)sample128_timer_handler},
190  #endif
```

Figure 22: Adding a message handler

```
{APP_SAMPLE128_TIMER,      (ke_msg_func_t)sample128_timer_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:

```
141  /**
142   ****************************************************************************************
143   * @brief Handles timer timeout
144   * @return If the message was consumed or not.
145   ****************************************************************************************
146   */
147
148  int sample128_timer_handler(ke_msg_id_t const msgid,
149                              struct gapm_cmp_event const *param,
150                              ke_task_id_t const dest_id,
151                              ke_task_id_t const src_id);
152
153  /// @} APP
```

Figure 23: Timer handler prototype

```
/**
 ****************************************************************************************
 * @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);
```

And in “app_custom_proj.c” we will implement the handler function:

```
851 /**
852  ****************************************************************************************
853  * @brief Handles timer timeout
854  * @return If the message was consumed or not.
855  ****************************************************************************************
856  */
857
858 int sample128_timer_handler(ke_msg_id_t const msgid,
859                                     struct gapm_cmp_evt const *param,
860                                     ke_task_id_t const dest_id,
861                                     ke_task_id_t const src_id)
862 {
863   return (KE_MSG_CONSUMED);
864 }
865
866
867 #endif  //BLE_APP_PRESENT
```

Figure 24: Timer handler implementation

```
/**
 ****************************************************************************************
 * @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)
{
  return (KE_MSG_CONSUMED);
}
```

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

```
135 void app_connection_func(struct gapc_connection_req_ind const *param)
136 {
137     if (app_env.conidx != GAP_INVALID_CONIDX)
138     {
139
140 /*************************************************
141 Handle connection request event. Enable required profiles
142
143 i.e.
144 #if (BLE_DIS_SERVER)
145     app_dis_enable_prf(app_env.conhdl);
146 #endif
147 *************************************************/
148
149         app_sample128_enable();
150
151         ke_timer_set(APP_SAMPLE128_TIMER,TASK_APP,50);
152
153         ke_state_set(TASK_APP, APP_CONNECTED);
154
155         // Retrieve the connection info from the parameters
156         app_env.conhdl = param->conhdl;
157
```

Figure 25: Starting our kernel timer

```
ke_timer_set(APP_SAMPLE128_TIMER,TASK_APP,50);
```

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:

```
46 uint8_t sample128_placeholder = 0;
47
48 /*
49  * FUNCTION DEFINITIONS
```

Figure 26: Declaring a global variable

```
uint8_t sample128_placeholder = 0;
```

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:

```
/**
 ****************************************************************************************
 * @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)
{
    ke_timer_set(APP_SAMPLE128_TIMER,TASK_APP,50);
    sample128_placeholder++;

    struct sample128_upd_char2_req *req = KE_MSG_ALLOC(
                                                       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);

    return (KE_MSG_CONSUMED);
}
```

Figure 27: Timer functionality implementation

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

struct sample128_upd_char2_req *req = KE_MSG_ALLOC(
                                                   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”:

```
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);
}
```

Figure 28: Write event implementation

```
sample128_placeholder = param->val;
```

 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 1: The GATT database of sample128

Handle (16-bit)	**Attribute Declaration Type**	**Attribute Declaration Type ID**	**Size of Declaration Attribute Type ID** [Bits]	**Data**	**Data size** [Bytes]
Start	Primary Service Declaration	0x2800	16	0x0F0E0D0C0B0A0…	16
Start+1	Characteristic Declaration	0x2803	16	0x<RD|WR><start+2>1F1E1D…	19
Start+2	Characteristic value declaration	0x1F1E1D…	128	0x00	1
Start+3	Characteristic Declaration	0x2803	16	0x<RD|NTFY><start+4>2F2E…	19
Start+4	Characteristic value declaration	0x2F2E2D…	128	0x00	1
Start+5	Client configuration declaration	0x2902	16	0x0000	2

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    {{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
38      0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F
39    }};
```

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

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 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”:

```
76          //Add Service Into Database
77
78          nb_att_16  = 4;    // 4 UUID16 Attribute declaration types
79          nb_att_32  = 0;    // 0 UUID32 Attribute declaration types
80          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”):

```
90          //  Total Data portion of GATT database = 58 data bytes:
91          //    16  Primary service declaration
92          //  + 19  Declaration of characteristic 1
93          //  +  1  Value declaration of characteristic 1
94          //  + 19  Declaration of characteristic 2
95          //  +  1  Value declaration of characteristic 2
96          //  +  2  Client configuration declaration of characteristic 2
97          //  = 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_

typedef unsigned          char my_new_t[8];

```

Figure 33: Defining a new type

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

We then initialize a new global variable of this type in “app_custom.proj.c”, as follows:

```
my_new_t sample128_my_new = {0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38};
uint8_t sample128_placeholder = 0;

/*
 * FUNCTION DEFINITIONS
 ****************************************************************************
```

Figure 34: Initialization of a global variable

```
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:

```
        status = attmdb_add_service(  &(sample128_env.sample128_shdl),
                                      TASK_SAMPLE128,
                                      nb_att_16,
                                      nb_att_32,
                                      nb_att_128,
                                      65 // See calcualtion below
                                   );

        //   Total Data portion of GATT database = 58 data bytes:
        //    16  Primary service declaration
        //   + 19  Declaration of characteristic 1
        //   +  8  Value declaration of characteristic 1
        //   + 19  Declaration of characteristic 2
        //   +  1  Value declaration of characteristic 2
        //   +  2  Client configuration declaration of characteristic 2
        //   = 65  Data bytes total
```

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:

```
// Characteristic 1: ////////////////////////////////////////////////////////////////////////////

// Add characteristic declaration attribute to database
status = attmdb_add_attribute(  sample128_env.sample128_shdl,
                                ATT_UUID_128_LEN + 3, // Data size = 19 (ATT_UUID_128_LEN + 3)
                                ATT_UUID_16_LEN,  // Size of declaration type ID
                                (uint8_t*) &att_decl_char, // 0x2803 for a characteristic declaration
                                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_new_t), // Data size = 8 Bytes
                                ATT_UUID_128_LEN,// Size of custom declaration type = 128bit
                                (uint8_t*)&sample128_1_val.uuid, // UUID of the characteristic value
                                PERM(RD, ENABLE) | PERM(WR, ENABLE),// Permissions
                                &(val_hdl) // handle to the value attribute
                              );
```

Figure 36: Changing the value attribute

```
sizeof(my_new_t), // Data size = 8 Bytes
```

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:

```
/// Parameters of the @ref SAMPLE128_ENABLE_REQ message
struct sample128_enable_req
{
    /// Connection Handle
    uint16_t conhdl;
    /// Security level
    uint8_t sec_lvl;

    /// characteristic 1 value
    my_new_t sample128_1_val;

    /// characteristic 2 value
    uint8_t sample128_2_val;

    /// char 2 Ntf property status
    uint8_t feature;
};
```

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:

```
/// Parameters of the @ref SAMPLE128_VAL_IND message
struct sample128_val_ind
{
    /// Connection handle
    uint16_t conhdl;
    /// Value
    my_new_t val;

};
```

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:

```
65  void app_sample128_enable(void)
66  {
67    // Allocate the message
68    struct sample128_enable_req* req = KE_MSG_ALLOC(
69                                                    SAMPLE128_ENABLE_REQ,
70                                                    TASK_SAMPLE128,
71                                                    TASK_APP,
72                                                    sample128_enable_req
73                                                   );
74    req->conhdl = app_env.conhdl;
75    req->sec_lvl = PERM(SVC, ENABLE);
76    memcpy(&req->sample128_1_val,&sample128_my_new,sizeof(my_new_t)); // default
77    req->sample128_2_val = 0xff;  // default value for sample128 characteristic 2
78    req->feature = 0x00;     // client CFG notify/indicate disabled
79    // Send the message
80    ke_msg_send(req);
81  }
```

Figure 39: Setting the default value via memcpy

```
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:

```
843  int sample128_val_ind_handler(ke_msg_id_t const msgid,
844                                 struct sample128_val_ind const *param,
845                                 ke_task_id_t const dest_id,
846                                 ke_task_id_t const src_id)
847  {
848    memcpy(&sample128_my_new,&param->val,sizeof(my_new_t));
849    return (KE_MSG_CONSUMED);
850  }
```

Figure 40: Retrieving the value of characteristic 1

```
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:

```
/**
 ****************************************************
 * @brief Send value change to application.
 * @param val Value.
 ****************************************************
 */

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:

```
void sample128_send_val(my_new_t val)
{
    // Allocate character 1 change indication
    struct sample128_val_ind *ind = KE_MSG_ALLOC(SAMPLE128_VAL_IND,
                                                 sample128_env.con_info.appid, TASK_SAMPLE128,
                                                 sample128_val_ind);
    // Fill in the parameter structure
    ind->conhdl = gapc_get_conhdl(sample128_env.con_info.conidx);

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

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

Figure 42: Changes to sample128_send_val

```
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:

```
318 static int gattc_write_cmd_ind_handler(ke_msg_id_t const msgid,
319                                         struct gattc_write_cmd_ind const *param,
320                                         ke_task_id_t const dest_id,
321                                         ke_task_id_t const src_id)
322 {
323     uint8_t char_code = SAMPLE128_ERR_CHAR;
324     uint8_t status = PRF_APP_ERROR;
325
326     if (KE_IDX_GET(src_id) == sample128_env.con_info.conidx)
327     {
328         if (param->handle == sample128_env.sample128_shdl + SAMPLE128_1_IDX_VAL)
329         {
330             char_code = SAMPLE128_1_CHAR;
331         }
332
333         if (param->handle == sample128_env.sample128_shdl + SAMPLE128_2_IDX_CFG)
334         {
335             char_code = SAMPLE128_2_CFG;
336         }
337
338         if (char_code == SAMPLE128_1_CHAR)
339         {
340
341             //Save value in DB
342             attmdb_att_set_value(param->handle, sizeof(my_new_t), (uint8_t *)&param->value[0]);
343
344             if(param->last)
345             {
346                 sample128_send_val((uint8_t *)&param->value[0]);
347             }
348
349             status = PRF_ERR_OK;
350
351         }
```

Figure 43: Changes to gattc_write_cmd_ind_handler

```
sizeof(my_new_t);
```

```
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”

```
212 static int sample128_enable_req_handler(ke_msg_id_t const msgid,
213                                        struct sample128_enable_req const *param,
214                                        ke_task_id_t const dest_id,
215                                        ke_task_id_t const src_id)
216 {
217
218     uint16_t temp = 1;
219
220     // Keep source of message, to respond to it further on
221     sample128_env.con_info.appid = src_id;
222     // Store the connection handle for which this profile is enabled
223     sample128_env.con_info.conidx = gapc_get_conidx(param->conhdl);
224
225     // Check if the provided connection exist
226     if (sample128_env.con_info.conidx == GAP_INVALID_CONIDX)
227     {
228         // The connection doesn't exist, request disallowed
229         prf_server_error_ind_send((prf_env_struct *)&sample128_env, PRF_ERR_REQ_DISALLOWED,
230                 SAMPLE128_ERROR_IND, SAMPLE128_ENABLE_REQ);
231     }
232     else
233     {
234         // Sample128 service permissions
235         attmdb_svc_set_permission(sample128_env.sample128_shdl, param->sec_lvl);
236
237         // Set characteristic 1 to specified value
238         attmdb_att_set_value(sample128_env.sample128_shdl + SAMPLE128_1_IDX_VAL,
239                              sizeof(my_new_t), (uint8_t *)&param->sample128_1_val);
240
241         // Set characteristic 2 to specified value
242         attmdb_att_set_value(sample128_env.sample128_shdl + SAMPLE128_2_IDX_VAL,
243                              sizeof(uint8_t), (uint8_t *)&param->sample128_2_val);
244
```

Figure 44: Changes to sample128_enable_req_handler

You should now be able to build and download the modified code.

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”

```
16 #ifndef SAMPLE128_H_
17 #define SAMPLE128_H_
18
19 typedef unsigned          char my_new_t[8];
20 typedef unsigned          char my_newer_t[10];
21
```

Figure 45: Defining a new data type in sample128.h

```
typedef unsigned          char my_newer_t[10];
```

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

Handle (16-bit)	**Attribute Declaration Type**	**Attribute Declaration Type ID**	**Size of Declaration Attribute Type ID** [Bits]	**Data**	**Data size** [Bytes]
Start	Primary Service Declaration	0x2800	16	0x0F0E0D0C0B0A0…	16
Start+1	Characteristic Declaration	0x2803	16	0x<RD|WR><start+2>1F1E1D…	19
Start+2	Characteristic value declaration	0x1F1E1D…	128	0x00	8!!!
Start+3	Characteristic Declaration	0x2803	16	0x<RD|NTFY><start+4>2F2E…	19
Start+4	Characteristic value declaration	0x2F2E2D…	128	0x00	1
Start+5	Client configuration declaration	0x2902	16	0x0000	2
Start+6	Characteristic Declaration	0x2803	16	0x<RD|NTFY><start+7>3F3E…	19
Start+7	Characteristic value declaration	0x3F3E3D…	128	0x00	10
Start+8	Client configuration declaration	0x2902	16	0x0000	2

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”:

```
72      /*---------------------------------------------------*
73       * Sample128 Service Creation
74       *---------------------------------------------------*/
75
76          //Add Service Into Database
77
78          nb_att_16  = 6;    // 6 UUID16 Attribute declaration types
79          nb_att_32  = 0;    // 0 UUID32 Attribute declaration types
80          nb_att_128 = 3;    // 3 UUID128 Attribute declaration types
81
82          status = attmdb_add_service(  &(sample128_env.sample128_shdl),
83                                        TASK_SAMPLE128,
84                                        nb_att_16,
85                                        nb_att_32,
86                                        nb_att_128,
87                                        96 // See calculation below
88                                     );
89
90          //   Total Data portion of GATT database = 96 data bytes:
91          //     16  Primary service declaration
92          //   + 19  Declaration of characteristic 1
93          //   +  8  Value declaration of characteristic 1
94          //   + 19  Declaration of characteristic 2
95          //   +  1  Value declaration of characteristic 2
96          //   +  2  Client configuration declaration of characteristic 2
97          //   + 19  Declaration of characteristic 3
98          //   + 10  Value declaration of characteristic 3
99          //   +  2  Client configuration declaration of characteristic 3
100
101         //   = 96  Data bytes total
```

Figure 46: Database changes

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
                                (uint8_t*)&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
                                (uint8_t*) &att_decl_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:

```
31   *  SAMPLE128 PROFILE ATTRIBUTES VALUES DEFINTION
32   ****************************************************************************************
33   */
34
35  /// sample128_1 Service
36 const struct att_uuid_128 sample128_svc     = {{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
37                                                  0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F}};
38
39
40  /// sample128_1 value attribute UUID
41 const struct att_uuid_128 sample128_1_val     = {{0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
42                                                    0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F}};
43
44 struct att_char128_desc sample128_1_char = {ATT_CHAR_PROP_RD | ATT_CHAR_PROP_WR,
45                                             {0,0},
46                                             {0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
47                                              0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F}};
48
49 const struct att_uuid_128 sample128_2_val     = {{0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
50                                                    0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F}};
51
52 struct att_char128_desc sample128_2_char = {ATT_CHAR_PROP_RD | ATT_CHAR_PROP_NTF,
53                                             {0,0},
54                                             {0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
55                                              0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F}};
56
57 const struct att_uuid_128 sample128_3_val     = {{0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
58                                                    0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F}};
59
60 struct att_char128_desc sample128_3_char = {ATT_CHAR_PROP_RD | ATT_CHAR_PROP_NTF,
61                                             {0,0},
62                                             {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
63                                              0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F}};
64
```

Figure 47: Defining attribute values

```
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,
                                            {0,0},
                                            {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36,
0x37,
                                             0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E,
0x3F}};
```

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

```
/*
 * ENUMERATIONS
 ****************************
 */

/// Handles offsets
enum
{
    SAMPLE128_1_IDX_SVC,

    SAMPLE128_1_IDX_CHAR,
    SAMPLE128_1_IDX_VAL,

    SAMPLE128_2_IDX_CHAR,
    SAMPLE128_2_IDX_VAL,
    SAMPLE128_2_IDX_CFG,

    SAMPLE128_3_IDX_CHAR,
    SAMPLE128_3_IDX_VAL,
    SAMPLE128_3_IDX_CFG,

    SAMPLE128_1_IDX_NB,
};
```

Figure 48: Indexing the 3 new attributes

```
    SAMPLE128_3_IDX_CHAR,
    SAMPLE128_3_IDX_VAL,
    SAMPLE128_3_IDX_CFG,
```

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;

```

Figure 49: Changes to sample128.h

```
// 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:

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”:

```
45
46  my_new_t sample128_my_new = {0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38};
47  my_newer_t sample128_my_newer = {0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4A};
48
49  uint8_t sample128_placeholder = 0;
```

Figure 51: Initialization of a global variable

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

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:

```
/*
 * API MESSAGES STRUCTURES
 ****************************************************
 */

/// Parameters of the @ref SAMPLE128_ENABLE_REQ message
struct sample128_enable_req
{
    /// Connection Handle
    uint16_t conhdl;
    /// Security level
    uint8_t sec_lvl;

    /// characteristic 1 value
    my_new_t sample128_1_val;

    /// characteristic 2 value
    uint8_t sample128_2_val;

    /// char 2 Ntf property status
    uint8_t feature;

    /// characteristic 3 value
    my_newer_t sample128_3_val;

    /// char 3 Ntf property status
    uint8_t feature3;

};
```

Figure 52: Modifying the enable structure

```
/// characteristic 3 value
my_newer_t sample128_3_val;

/// char 3 Ntf property status
uint8_t feature3;
```

We are ready to initialize the new characteristic where we enable the service in “sample128_task.c”:

```
251 static int sample128_enable_req_handler(ke_msg_id_t const msgid,
252                                        struct sample128_enable_req const *param,
253                                        ke_task_id_t const dest_id,
254                                        ke_task_id_t const src_id)
255 {
256
257     uint16_t temp = 1;
258
259     // Keep source of message, to respond to it further on
260     sample128_env.con_info.appid = src_id;
261     // Store the connection handle for which this profile is enabled
262     sample128_env.con_info.conidx = gapc_get_conidx(param->conhdl);
263
264     // Check if the provided connection exist
265     if (sample128_env.con_info.conidx == GAP_INVALID_CONIDX)
266     {
267         // The connection doesn't exist, request disallowed
268         prf_server_error_ind_send((prf_env_struct *)&sample128_env, PRF_ERR_REQ_DISALLOWED,
269                 SAMPLE128_ERROR_IND, SAMPLE128_ENABLE_REQ);
270     }
271     else
272     {
273         // Sample128 service permissions
274         attmdb_svc_set_permission(sample128_env.sample128_shdl, param->sec_lvl);
275
276         // Set characteristic 1 to specified value
277         attmdb_att_set_value(sample128_env.sample128_shdl + SAMPLE128_1_IDX_VAL,
278                              sizeof(my_new_t), (uint8_t *)&param->sample128_1_val);
279
280         // Set characteristic 2 to specified value
281         attmdb_att_set_value(sample128_env.sample128_shdl + SAMPLE128_2_IDX_VAL,
282                              sizeof(uint8_t), (uint8_t *)&param->sample128_2_val);
283
284         // Set characteristic 3 to specified value
285         attmdb_att_set_value(sample128_env.sample128_shdl + SAMPLE128_3_IDX_VAL,
286                              sizeof(my_newer_t), (uint8_t *)&param->sample128_3_val);
287
```

Figure 53: Initialization of the characteristic value

```
// Set characteristic 3 to specified value
attmdb_att_set_value(sample128_env.sample128_shdl + SAMPLE128_3_IDX_VAL,
                             sizeof(my_newer_t), (uint8_t *)&param->sample128_3_val);
```

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:

```
66  void app_sample128_enable(void)
67  {
68    // Allocate the message
69    struct sample128_enable_req* req = KE_MSG_ALLOC(
70                                                    SAMPLE128_ENABLE_REQ,
71                                                    TASK_SAMPLE128,
72                                                    TASK_APP,
73                                                    sample128_enable_req
74                                                   );
75    req->conhdl = app_env.conhdl;
76    req->sec_lvl = PERM(SVC, ENABLE);
77    memcpy(&req->sample128_1_val,&sample128_my_new,sizeof(my_new_t)); // default
78    memcpy(&req->sample128_3_val,&sample128_my_newer,sizeof(my_newer_t)); // default
79    req->sample128_2_val = 0xff;  // default value for sample128 characteristic 2
80    req->feature = 0x00;          // client CFG notify/indicate disabled
81    // Send the message
82    ke_msg_send(req);
83  }
```

Figure 54: Default value of characteristic 3

```
memcpy(&req->sample128_3_val,&sample128_my_newer,sizeof(my_newer_t));     // default
```

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”:

```
147 /// Parameters of the @ref SAMPLE128_UPD_CHAR2_REQ message
148 struct sample128_upd_char2_req
149 {
150     /// Connection handle
151     uint16_t conhdl;
152     /// Characteristic Value
153     uint8_t val;
154 };
155
156 /// Parameters of the @ref SAMPLE128_UPD_CHAR3_REQ message
157 struct sample128_upd_char3_req
158 {
159     /// Connection handle
160     uint16_t conhdl;
161     /// Characteristic Value
162     my_newer_t val;
163 };
```

Figure 55: New characteristic update structure

```
/// 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 will need a couple of new message primitives to be able to update the characteristic. In “sample128_task.h” add these two primitives:

```
59 /// Messages for Sample128
60 enum
61 {
62     /// Start sample128. Device connection
63     SAMPLE128_ENABLE_REQ = KE_FIRST_MSG(TASK_SAMPLE128),
64
65     /// Disable confirm.
66     SAMPLE128_DISABLE_IND,
67
68     /// Att Value change indication
69     SAMPLE128_VAL_IND,
70
71     ///Create DataBase
72     SAMPLE128_CREATE_DB_REQ,
73     ///Inform APP of database creation status
74     SAMPLE128_CREATE_DB_CFM,
75
76     ///Update value of characteristic 2
77     SAMPLE128_UPD_CHAR2_REQ,
78     ///Confirm the update of value of characteristic 2
79     SAMPLE128_UPD_CHAR2_CFM,
80
81     ///Update value of characteristic 3
82     SAMPLE128_UPD_CHAR3_REQ,
83     ///Confirm the update of value of characteristic 3
84     SAMPLE128_UPD_CHAR3_CFM,
85
86     /// Error Indication
87     SAMPLE128_ERROR_IND,
88 };
```

Figure 56: New message primitives

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

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”:

```
478 /// Connected State handler definition.
479 const struct ke_msg_handler sample128_connected[] =
480 {
481     {GATTC_WRITE_CMD_IND,       (ke_msg_func_t) gattc_write_cmd_ind_handler},
482     {SAMPLE128_UPD_CHAR2_REQ,   (ke_msg_func_t) sample128_upd_char2_req_handler},
483     {SAMPLE128_UPD_CHAR3_REQ,   (ke_msg_func_t) sample128_upd_char3_req_handler},
484 };
```

Figure 57: Implementing a new handler

```
{SAMPLE128_UPD_CHAR3_REQ,   (ke_msg_func_t) sample128_upd_char3_req_handler},
```

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.

```
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);
}
```

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 simple 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”:

```
870 int sample128_timer_handler(ke_msg_id_t const msgid,
871                                     struct gapm_cmp_evt const *param,
872                                     ke_task_id_t const dest_id,
873                                     ke_task_id_t const src_id)
874 {
875   ke_timer_set(APP_SAMPLE128_TIMER,TASK_APP,50);
876   sample128_placeholder++;
877
878
879   struct sample128_upd_char2_req *req = KE_MSG_ALLOC(
880                                                       SAMPLE128_UPD_CHAR2_REQ,
881                                                       TASK_SAMPLE128,
882                                                       TASK_APP,
883                                                       sample128_upd_char2_req
884                                                     );
885   req->val = sample128_placeholder;
886   req->conhdl = app_env.conhdl;
887
888   ke_msg_send(req);
889
890   struct sample128_upd_char3_req *req3 = KE_MSG_ALLOC(
891                                                       SAMPLE128_UPD_CHAR3_REQ,
892                                                       TASK_SAMPLE128,
893                                                       TASK_APP,
894                                                       sample128_upd_char3_req
895                                                     );
896   memcpy(&req3->val,&sample128_my_newer,sizeof(my_newer_t));
897   memcpy(&req3->val,&sample128_placeholder,1);
898   req3->conhdl = app_env.conhdl;
899
900   ke_msg_send(req3);
901
902   return (KE_MSG_CONSUMED);
903 }
904
```

Figure 58: Change the first byte of characteristic 3

```
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);
```

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".

```
57  ///Characteristics Code for Write Indications
58  enum
59  {
60      SAMPLE128_ERR_CHAR,
61      SAMPLE128_1_CHAR,
62      SAMPLE128_2_CFG,
63      SAMPLE128_3_CFG,
64  };
```

Figure 59: Adding the new characteristic's client configuration to sample128.h

```
✂
SAMPLE128_3_CFG,
```

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":

```
71  /// sample128 environment variable
72  struct sample128_env_tag
73  {
74      /// Connection Information
75      struct prf_con_info con_info;
76
77      /// Sample128  svc Start Handle
78      uint16_t sample128_shdl;
79
80      //Notification property status
81      uint8_t feature;
82
83      //Notification property status characteristic 3
84      uint8_t feature3;
85  };
```

Figure 60: Changing the service environment structure

```
✂
    // Notification property status characteristic 3
    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:

```
293         sample128_env.feature = param->feature;
294
295         if (!sample128_env.feature)
296         {
297                 temp = 0;
298         }
299
300         attmdb_att_set_value(sample128_env.sample128_shdl + SAMPLE128_2_IDX_CFG,
301                              sizeof(uint16_t), (uint8_t *)&temp);
302
303         sample128_env.feature3 = param->feature3;
304
305         if (!sample128_env.feature3)
306         {
307                 temp = 0;
308         }
309         else temp=1;
310
311         attmdb_att_set_value(sample128_env.sample128_shdl + SAMPLE128_3_IDX_CFG,
312                              sizeof(uint16_t), (uint8_t *)&temp);
313
314         // Go to Connected state
315         ke_state_set(TASK_SAMPLE128, SAMPLE128_CONNECTED);
316     }
317
318     return (KE_MSG_CONSUMED);
319 }
320
```

Figure 61: Initializing notification

```
sample128_env.feature3 = param->feature3;
if (!sample128_env.feature3)
{
  temp = 0;
}
else temp=1;

attmdb_att_set_value(sample128_env.sample128_shdl + SAMPLE128_3_IDX_CFG,
                     sizeof(uint16_t), (uint8_t *)&temp);
```

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:

```
420 static int gattc_write_cmd_ind_handler(ke_msg_id_t const msgid,
421                                          struct gattc_write_cmd_ind const *param,
422                                          ke_task_id_t const dest_id,
423                                          ke_task_id_t const src_id)
424 {
425     uint8_t char_code = SAMPLE128_ERR_CHAR;
426     uint8_t status = PRF_APP_ERROR;
427
428     if (KE_IDX_GET(src_id) == sample128_env.con_info.conidx)
429     {
430         if (param->handle == sample128_env.sample128_shdl + SAMPLE128_1_IDX_VAL)
431         {
432             char_code = SAMPLE128_1_CHAR;
433         }
434
435         if (param->handle == sample128_env.sample128_shdl + SAMPLE128_2_IDX_CFG)
436         {
437             char_code = SAMPLE128_2_CFG;
438         }
439
440         if (param->handle == sample128_env.sample128_shdl + SAMPLE128_3_IDX_CFG)
441         {
442             char_code = SAMPLE128_3_CFG;
443         }
444
```

Figure 62: Handling notification subscriptions

```
if (param->handle == sample128_env.sample128_shdl + SAMPLE128_3_IDX_CFG)
{
  char_code = SAMPLE128_3_CFG;
}
```

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:

```
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;
  }
}
```

You should be able to build the project, download it to your DVK and run it. Using LightBlue or BlueLoupe should allow you to see the new characteristic, read the value and set up notifications to receive an updated value every 500ms.

8 Revision history

Revision	Date	Description
1.1	19-Jan-2022	Updated logo, disclaimer, copyright.
1.0	25-April-2015	Initial version.

Status Definitions

Status	Definition
DRAFT	The content of this document is under review and subject to formal approval, which may result in modifications or additions.
APPROVED or unmarked	The content of this document has been approved for publication.

RoHS Compliance

Dialog Semiconductor's suppliers certify that its products are in compliance with the requirements of Directive 2011/65/EU of the European Parliament on the restriction of the use of certain hazardous substances in electrical and electronic equipment. RoHS certificates from our suppliers are available on request.