# RSL10 Sample Code User's Guide

M-20840-012
December 2019

ON Semiconductor®

# Table of Contents

Page

1. Introduction . . . 3
    1.1 Purpose . . . 3
    1.2 Intended Audience . . . 3
    1.3 Conventions . . . 3
    1.4 Further Reading . . . 3

2. Using Sample Applications . . . 4
    2.1 Requirements . . . 4
    2.2 Setup . . . 4
    2.3 Accessing Sample Code Files . . . 4

3. The peripheral_server Application . . . 5
    3.1 Purpose . . . 5
        3.1.1 What Does The Peripheral-Server Application Do? . . . 5
        3.1.2 What peripheral_server Demonstrates . . . 5
    3.2 File Structure . . . 5
        3.2.1 Types of Files and Stacks . . . 5
        3.2.2 How Files Relate . . . 6
    3.3 Walkthrough of the peripheral_server Application . . . 6
        3.3.1 System Initialization . . . 6
        3.3.2 GAP and GATT Services -- Establishing Communication . . . 6
        3.3.3 The Kernel Scheduler . . . 6
        3.3.4 Message Handlers . . . 7
        3.3.5 Standard Profiles/Services . . . 7
        3.3.6 Custom Profiles/Services. . . . 7
        3.3.7 The Main Loop—peripheral_server Event Kernel Execution . . . 7
    3.4 Walkthrough of the Message Sequence Chart (MSC) . . . 8
    3.5 How to Run peripheral_server with the Evaluation and Development Board . . . 8
    3.6 Adding New Profiles/Services . . . 9
        3.6.1 How to Add a New Standard Service/Profile . . . 9
        3.6.2 How to Add a New Custom Service/Profile . . . 9
        3.6.3 More About Working with Custom Services . . . 10

4. Available Sample Applications . . . 12

A. Message Sequence Chart for Peripheral Device with Server Sample Application . . . 20

# CHAPTER 1

# Introduction

## 1.1 Purpose

*RSL10 Sample Code User's Guide* explains how to use the sample applications provided with the RSL10 software development tools. As you follow this guide, you learn about setting up your system, accessing code files, and how the sample application Peripheral Device with Server (*peripheral_server*) works. In addition, this manual shows you what this sample code demonstrates, and how the principles can be used in the RSL10 applications you develop. The guide also points you to more information about using RSL10.

## 1.2 Intended Audience

This manual is for people who intend to develop applications for RSL10. It assumes that you are familiar with software development activities and the use of Bluetooth low energy technology.

## 1.3 Conventions

The following conventions are used in this manual to signify particular types of information:

| | |
|---|---|
| `monospace` | Commands and their options, file and path names, error messages, code samples and code snippets. |
| **`mono bold`** | A placeholder for the specified information. For example, replace **`filename`** with the actual name of the file. |
| **bold** | Graphical user interface labels, such as those for menus, menu items and buttons. |
| *italics* | File names and path names, or any portion of them. |

## 1.4 Further Reading

For more information, refer to the following documents:

- *RSL10 Getting Started Guide*
- *RSL10 Hardware Reference*
- *RSL10 Firmware Reference*
- *RSL10 Evaluation and Development Board Manual*

# CHAPTER 2

# Using Sample Applications

## 2.1 REQUIREMENTS

To use any sample application, such as the *peripheral_server* sample application, you must have the installed RSL10 software development tools, along with the RSL10 Evaluation and Development Board, a computer running Windows®, and a cable to connect the two. You also need a device to serve as the central client, such as another RSL10 Evaluation and Development Board, a control panel, or a device of your choice that complies with the Bluetooth low energy standard, and fills the GAP central and GATT client roles.

## 2.2 SETUP

Refer to the *RSL10 Getting Started Guide* for information on how to connect the Evaluation and Development Board to your computer.

## 2.3 ACCESSING SAMPLE CODE FILES

This guide assumes that you have installed the RSL10 software development tools and that you know how to import sample applications into your workspace. If not, see the *RSL10 Getting Started Guide* for basic instructions. For alternate ways of accessing the sample applications, such as copying the files or linking to them from a project, see the documentation for your IDE.

# CHAPTER 3

# The peripheral_server Application

## 3.1 PURPOSE

For the purpose of understanding the sample code as a whole, this chapter provides a deeper explanation of the Peripheral Device with Server (*peripheral_server*) sample application. This application is a basic Bluetooth® low energy technology based sample application that demonstrates many of the fundamentals of an end user device, and this guide answers the following questions:

- What does the *peripheral_server* application do?
- What are components of the *peripheral_server* application?
- How does the *peripheral_server* application works?

### 3.1.1 What Does The Peripheral-Server Application Do?

In this program, RSL10 fills two Bluetooth technology roles:

- It is a GATT server because it provides data in the form of GATT characteristics that can be read by remote GATT clients.
- It acts as a GAP peripheral because it advertises its presence and willingness to connect with a GAP central device.

The *peripheral_server* program implements a battery service and a custom service, then starts undirected, connectable advertising. Your central device scans the advertising and a connection is established. During advertising, LED DIO#6 on the Evaluation and Development Board blinks. It turns steadily on once the link is established. The application then reads the device's battery level every 200 milliseconds, as dictated by a kernel timer event. It calculates the average for 16 reads of the battery level (3200 ms). If the average value of the level changes, the program sets a flag to send a battery level update to the stack. If notification of battery level is enabled, then the stack sends a battery level notification to the peer device. It also demonstrates a custom service in which two characteristics are defined. For one of them, a notification with an incremented value is sent every 30 kernel timer events (6 seconds), if the notification is not disabled by the client device.

### 3.1.2 What peripheral_server Demonstrates

The application is an example of both standard service and custom service use. The code shows how to connect RSL10 (which takes the role of a server) with a central client device, send requests to the Bluetooth low energy stack, receive and handle responses, perform calculations, use kernel timer events, and send notifications. All of these functionalities - generating a service, connecting, measuring, timing, and notifying - are some of the basic building blocks of the Bluetooth low energy applications you can develop for RSL10.

## 3.2 FILE STRUCTURE

This application uses various kinds of files to communicate and perform functions with Bluetooth low energy client devices.

### 3.2.1 Types of Files and Stacks

The *peripheral_server* application resides in two folders and one file. The folders are called *code* and *include*.

The files in the *code* folder (*.c* files) contain functions, which are responsible for different functionalities in the application. For instance, the main program loop is one of the functions found in the file called *app.c*; the initialization functions are in the*app_init.c* file; and so on.

The *include* folder contains application-related public header files (*.h* files) which allow the code to access libraries.

The sample application *peripheral_server* uses the Bluetooth low energy stack.

### 3.2.2 How Files Relate

Some of the functions in the *code* files call other functions during the operation of the application; they also communicate with the Bluetooth low energy stack. The *include* files serve as access points for libraries, which are not part of the project itself but are required by application functionality.

## 3.3 Walkthrough of the peripheral_server Application

This walkthrough shows the details of *peripheral_server*’s functionalities, so that you can understand and use them in your RSL10 application development.

### 3.3.1 System Initialization

When the main function begins, it calls the function `App_Initialize()`. This function, in turn, calls other functions to initialize everything, including Bluetooth low energy, the kernel, power supplies, clock dividers, interrupts, and any environments used by the application.

### 3.3.2 GAP and GATT Services -- Establishing Communication

The Generic Access Profile (GAP) contains guidelines for the broadcasting and connecting mechanisms by which a Bluetooth low energy device can communicate with the outside world. The GAP is divided into GAPM (GAP Manager) and GAPC (GAP Controller) services.

The Generic Attribute Profile (GATT) contains rules for how attributes (data) are formatted, packaged, and sent between Bluetooth low energy devices once they have a dedicated connection. The GATT is divided into GATTM (GATT Manager) and GATTC (GATT Controller) services.

### 3.3.3 The Kernel Scheduler

After initialization, an infinite While loop begins running the kernel scheduler.

The kernel scheduler is responsible for constantly checking messages communicated by different tasks. In this application there is only one task, but in any application there can be different instances of the same task. Different application tasks control different Bluetooth low energy functionalities. An application sometimes needs to communicate with different tasks, such as GAPM, GAPC, GATTM, GATTC, and different standard and custom profiles (services). The kernel scheduler is responsible for handling these messages.

When the application needs to send a message, it allocates memory in the kernel buffer, fills in the message with any parameter that it wants to send, and then fills in the source address and destination address as the message identifier. Then the application calls a function from the kernel asking to send this message to the destination. The kernel scheduler checks that buffer. If it is filled, then the scheduler sends the message to the destination task by automatically calling a pre-defined message handler. The kernel scheduler also handles all timers used by the stack, services, or application. When a timer expires, the kernel scheduler automatically calls a function or message handler allocated to the timer’s identifier.

### 3.3.4 Message Handlers

Another important location in the code is for message handlers. For example, any message, request or command sent from the application to the GAPM has an associated `GAPM_COMPLETE` event. This event message is sent from the stack to the application. Based on its message identifier, this message is called automatically by the kernel. The definition for this event message is in `BLE_MESSAGE_HANDLER_LIST`, found in *ble_standard.h*. Any message that you want to add, and that you want a message handler for, needs to be listed in `BLE_MESSAGE_HANDLER_LIST`. You might read the CEVA documentation for GATT and GAP and decide to add your own message handler for some specific API or message. That also must be listed here.

To add a message handler, first, you need to add `DEFINE_MESSAGE_HANDLER` in the corresponding *.h* file, and define a function in the corresponding file, which can be *ble_standard.h*, *application.h*, or *standard_profile.h*.

### 3.3.5 Standard Profiles/Services

Standard services are pre-defined methods by which Bluetooth low energy devices can communicate specific types of data to one another. You can think of a profile as a group of one or more services.

The attributes of standard services are already recognized by the Bluetooth low energy stack, so you do not need to list the attributes when adding standard services to an application. Each standard service has a 16-bit UUID (unique numeric identifier).

### 3.3.6 Custom Profiles/Services

Custom services, like standard services, are methods by which Bluetooth low energy devices can communicate, and work with your Bluetooth low energy applications to add functionalities. Unlike standard services, custom services are not pre-defined. You can control what they do, creating them to your own specifications. You also need to list all the attributes you want for a custom service because the stack cannot automatically tell which attributes to add. Each custom service needs a 128-bit UUID.

### 3.3.7 The Main Loop—peripheral_server Event Kernel Execution

The primary functionality of *peripheral_server* is performed by the program’s main loop, located in *app.c*. This loop checks to see whether the new value for a battery has changed, and sets a flag called `send_batt_ntf`. Next, the loop calls a function, `Batt_LevelUpdateSend`, to send the value of the battery level to the Battery Service. If notification is enabled for this service, the Bluetooth stack notifies the peer device (client device) in communication with RSL10 of the updated battery value.

There is also an additional flag which is set when the value of one of the custom attributes is changed. The flag is called `tx_value_exchange`. Based on this flag, the function `custom_service_send_notification` is called to send a new value to the stack, which is then responsible for sending the message over the air to the peer device. In this example we are communicating some parameters from the peripheral, RSL10, to the client device/central device. To show that the link is established and communicating, we send the value of `tx_attribute` every six seconds.

On the central side, you can use any central client device—another RSL10 Evaluation and Development Board, a Bluetooth Low Energy Explorer on an RSL10 dongle, or any central application that can connect to the peripheral device and show the service attribute values. Using your central device, you can verify that the new battery data is measured and read correctly every six seconds. If the battery level changes, the central device is notified and can show the new value, provided the corresponding notification is enabled. Every six seconds, the central device is also notified of a new value from one of the custom characteristics.

## 3.4 WALKTHROUGH OF THE MESSAGE SEQUENCE CHART (MSC)

The MSC is a visual representation (as shown in Figure 1 on page 21) of the messages communicated between the application and the Bluetooth low energy stack in establishing a connection so that data can be exchanged. All stages of this process make changes to a variable called `BLE_STATE`, which shows the state of the Bluetooth low energy application manager. At first it is initialized to `APPM_INIT`, when the application is initializing and is configured in an idle state.

After initializing the Bluetooth stack, the kernel, and any environments that the application works with, the code sends a reset to the Bluetooth low energy stack. If this is successful, the stack sends back a `GAPM_COMP_EVENT`, which contains the name of the operation, and its status. The application receives it and checks that the operation is `GAPM_RESET`; it also checks whether the status is successful. If these conditions are met, the `GAPM_SET_DEV_CONFIG_CMD` command is sent to the stack. Again, the stack sends back a `GAPM_COMP_EVENT`. The status is checked for errors because if the parameter is set incorrectly, the stack might not accept the application's message, and will not properly configure itself. If no errors occur, the application receives a successful `GAPM_COMP_EVENT`. The application is now set to the `APPM_CREATE_DB` (create database) state, beginning to add databases to the Bluetooth low energy stack, and configure GAP.

The program now sends a `GAPM_PROFILE_TASK_ADD_CMD` command for adding a standard service to the stack, and receives an indicator if the service has been added successfully. Next, the program sends a `GATTM_ADD_SVC_REQ` request to the stack, to request the addition of a custom service. `GATTM_ADD_SVC_RSP` is the response from the stack, informing the program whether the custom service can be added.

After adding services, the application is in the `APPM_READY` state. Now the code runs a function that checks whether any further service is waiting to be added. Once it sees that no further service is to be added, it sends a `GAPM_START_ADVERTISING` command to begin advertising. If the parameter is set correctly, the code has no need to wait for a `GAPM_COMP_EVENT` response from the stack because advertising begins as soon as the command is sent. The application is now in the `APPM_ADVERTISING` state.

Once advertising starts, the central device can scan the peripheral device (RSL10). Once the central device observes that the peripheral is advertising, it can establish a connection. The application receives a `GAPC_CONNECTION_REQ_IND` indication, which informs it which Bluetooth address it is connected to. The application then sends a `GAPC_CONNECTION_CONFIRMED` notification to the stack, indicating that the program has received the stack's message, and that the message has provided the necessary parameters. The application is now in the `APPM_CONNECTED` state.

When the link is established, the program sends a `BASS_ENABLE` request to enable battery service in the stack. After receiving the response to this message, `BASS_SUPPORT_ENV.ENABLE` is set to indicate to the application that battery service is enabled. From this point it can send custom service notification, receive a read request from the central device for a custom service characteristic value, or perform any other sort of communication with the central device.

## 3.5 HOW TO RUN PERIPHERAL_SERVER WITH THE EVALUATION AND DEVELOPMENT BOARD

Refer to *RSL10 Getting Started Guide* for information on how to connect the RSL10 Evaluation and Development Board to your computer, and a demonstration of how to import, build, and run sample code. The example used in *RSL10 Getting Started Guide* is a program called *blinky*. Import *peripheral_server* instead, and work with it according to the information presented for *blinky*.

## 3.6 Adding New Profiles/Services

### 3.6.1 How to Add a New Standard Service/Profile

You can add standard services to the Bluetooth low energy stack by sending a `GAPM_ProfileTaskAdd` command in your program, with the message stating which standard service is being added. In this sample program, you are adding *ble_bass* – the battery service server. You can add more standard profiles the same way. After sending a `GAPM_ProfileTaskAdd` command, your program needs to wait until the `GAPM_ProfileAddedInd` indication is received to make sure that this service is successfully added. Incorrect parameter values can cause the stack to reject the command. This is also true of the GATTM services.

Here are step-by-step instructions for adding a new standard profile/service:

1. Find the document related to the desired profile/service API, located in the *documentation/ceva* folder.
2. Create two files, a *.c* file and a *.h* file, and name them based on the service that you have selected. For example, if you want to add the ANPS service, you would create the files *ble_anps.c* and *ble_anps.h*, for consistency with the code structure.
3. In Eclipse, go to the **Project** setting, in **Build Configuration** for **Cross Arm C Compiler**, and add the symbol `CFG_PRF_ANPS`. Adjust `CFG_NB_PRF` based on the maximum the number of profiles that are added to the project.
4. In the **Project** setting, at **Cross Arm C Link** under **Libraries**, add “anps” under the **-l** section.
5. In the *ble_anps.h* file, define all message handlers that are required for this profile according to the CEVA documentation, and create corresponding functions for them. For example, for BASS we have:

```
#define BASS_MESSAGE_HANDLER_LIST \
   DEFINE_MESSAGE_HANDLER(BASS_ENABLE_RSP, Batt_EnableRsp_Server), \
DEFINE_MESSAGE_HANDLER(BASS_BATT_LEVEL_NTF_CFG_IND, Batt_LevelNtfCfgInd)
```

6. Define the anps variable environment as `struct anp_support_env_tag`, with at least the `bool enable` field, plus any other fields that are required for this profile.
7. All message handler functions and any other functions must have their prototype declarations in *ble_anps*.h.
8. Declare `anps_supprt_env` in the *ble_anps.c* file, and then initialize it with the `Anps_Env_Initialize` function. This function should be added to `App_Env_Initialize()`.
9. In the function `BLE_SetServiceState()`, this example for BASS has an `If` condition. When the condition is true, add `Anp_ServiceEnable_Server(ble_env.conidx)`. If the `else` is true instead, add `anps_support_env.enable = false`.
10. In *app.h*, write a service add function for anps in this way:

```
#define SERVICE_ADD_FUNCTION_LIST \
   DEFINE_SERVICE_ADD_FUNCTION(Batt_ServiceAdd_Server), \
DEFINE_SERVICE_ADD_FUNCTION(Anp_ServiceAdd_Server), \
   DEFINE_SERVICE_ADD_FUNCTION(CustomService_ServiceAdd)
```

In *app.c* or *app_process*, you can call any function that you have created for anps, such as to send something if `ble_env.state` is in `APPM_CONNECTED` and `anps_support_env.enable==true`.

### 3.6.2 How to Add a New Custom Service/Profile

After adding standard services, you can add custom services.

Custom services are added by using the message `gattm_add_svc_req`. In it you list all the characteristics (attributes) you want for this custom service. First you define the service UUID, and then any characteristics that you want to add, such as characteristic UUID, client configuration characteristics (CCCD), and optional user descriptors which function as names for characteristics. Your application sends a `GATTM_ADD_SVC_REQ` message. When a successful response is received, the stack sends a start handle value to the application in the `GATTM_ADD_SVC_RSP` message. This start handle indicates where in the stack your requested service has been added, and serves as this service's reference in the stack. The service's characteristics that you add to the stack also have handle values, but they are offset from this start handle value. For example, if the start handle value is 10, the handle value for the first characteristic added to the service in the stack is 11.

Here is an example of adding a new custom service. The function is called `CustomService_ServiceAdd`, and is part of the RSL10 software development tools.

First, send the `GATTM_ADD_SVC_REQ` message. Include a service UUID in it, and then a list of attributes (characteristics) that you want to add. In this `CustomService_ServiceAdd` example, two groups of attributes, `Tx` and `Rx`, are added. These two values make it possible for the peripheral and central devices to send values to each other in two-way communication.

The `Tx` characteristic declaration is added first. It is a standard characteristic, but it needs to be added in this way because a custom service requires a 128-bit UUID. (You need a standard UUID if you are adding a characteristic declaration, but when adding value characteristics you need to use a custom UUID.) After adding the characteristic declaration, you need to add the characteristic value, the client configuration characteristic (CCCD), and the user description characteristic, so that you can add a name and observe it in the central device, rather than just seeing the identification of the service as a number. Macros such as `add_declaration_characteristic`, `add_declaration_characteristic_UUID`, `add_declaration_characteristic_ccc`, and `add_declaration_characteristic_user_description` make this process easier.

You then need to define the permissions and properties of every characteristic you add. For example, one characteristic in the `Tx` group is `Tx_value`. This characteristic can be modified, indicated, read from the central device, and written by receiving either a write request or a write command. Any of these properties can be set here, along with whether the characteristic will respond to read or write requests, whether it needs authentication, encryption, permissions, or security – these properties get set here. (This example does not use security, permissions or authentication.)

There are macros available for setting up permissions. For example, `per_rd_command_enable` sets permissions to enable read commands. For the extra permission characteristic in this example, there is a flag in `attribute_declaration_characteristic_UUID_128`, called `per_ri_enable`. This flag indicates that the value of this attribute is not located in the stack, and should be handled in the application. Maximum size is another characteristic: for custom length values, such as this example illustrates, you need to set the maximum length of this attribute in bytes.

When all the attributes are listed, the application can send `GATTM_SERVICE_ADD`. To add another custom service, the application needs to send another message with its own service UUID and characteristic definitions.

### 3.6.3 More About Working with Custom Services

When the application receives a `GATTM_ADD_SVC_RSP` response from the stack, it saves the start handle in the custom service environment. For each custom service attribute, there is an index, because these attributes are listed in the *custom_service.h* file. For example, if the RSL10 application receives a read request from a peer device, that request will be sent by its handle. In effect, the request will say something like, "I am asking to read the value of handle number 11." (Or another handle number, depending on the request.) To map the value of the handle onto the attribute index, the application needs to find the offset between the start handle in the GATT database, minus one. The first index number, 0, is reserved for the service UUID, and the attribute in this example starts from number 1. So if the start handle is located at number 10, the first attribute is located at start handle number 11. Therefore, index number 0 in this example points to handle number 11. The attribute index is calculated this way based on the attribute number in the function; using a switch-case, the application can easily access any attribute that a peer device request is asking about.

Based on that attribute, the application can receive the value. Sometimes permissions do not allow responses to requests about attributes. For example, if permission for the peer device to change the user description is not desired, then write permission can be disabled.

To change the Rx value of this custom service using a central device, a write command or a write request can be sent from the client device, and the application received `GATTC_WRITE_REQ_IND` when this message is received by the stack.

To send a custom service notification, your program needs to use `CUSTOM_SERVICE_SEND_NOTIFICATION`. For this you must indicate the link or connection for which the request is being sent, the attribute index this notification is being sent for, what the value is, and the length of this value.

# CHAPTER 4

# Available Sample Applications

Table 1 lists all the sample applications that are available with the RSL10 software development tools, and explains what each one is intended to demonstrate.

If a Keil version of the sample exists, when you install the Keil CMSIS-Pack, you will find the sample application in a folder of the same name, but in the *IDE_Configuration/Keil* subfolder.

**Table 1. Sample Code**

| Name | Folder | Description | Keil Version | IAR Version |
|---|---|---|---|---|
| ADC with UART | ADC_UART | A sample project that demonstrates the ADC and battery monitor block. The program measures the voltage of the supply voltage provided at the battery input and reports it on the UART port. Detecting under voltage conditions triggers a battery monitor alarm. This sample project demonstrates the usage of ADC and BATMON interrupts. | Yes | Yes |
| AES-128 | aes128 | A complete Arm Cortex-M3 processor implementation for AES-128 ECB mode encryption/ decryption in C according to the NIST recommendations. The encryption demonstrates encryption using the hardware engine of the baseband block, or using a pure software implementation. The decryption side is performed only in software. | | |
| Android Audio Streaming for Hearing Aid Profile with Server | ble_android_asha | Implements an Android ASHA server device. The ASHA profile supports streaming either the left or right audio channels from a BLE 5.0 Android phone. It only supports a single client at a time.<br>NOTE: Android ASHA is an experimental feature in Android and is not enabled in factory builds. A custom Android version with ASHA enabled is required.<br>This sample application also implements a battery service and the device information service. It supports pairing and bonding. The application stores the bond information in the RSL10 NVR2 flash and has the ability to perform Private Address Resolution. | | |
| Scanning with Central Device | ble_central_client_scan | This sample application demonstrates a Bluetooth scanning procedure. It scans for advertisements and prints a list of scanned devices over UART. You can use a UART terminal application (like TeraTerm) with a baud rate of 115200 to see scanned devices and connect up to four peripheral peer devices. This application makes use of the RSL10 UART CMSIS-Driver to manage the interface communication. | Yes | Yes |

**Table 1. Sample Code (Continued)**

| Name | Folder | Description | Keil Version | IAR Version |
|---|---|---|---|---|
| Pairing and Bonding with Peripheral Device | ble_peripheral_server_bond | This sample project generates a battery service and a custom service. It then begins undirected connectable advertising. Up to four simultaneous connections are supported. Any central device can scan, connect, pair/bond, perform service discovery and read the battery/custom services characteristics. The application sends periodic notifications of the battery level and a custom service characteristic to the connected clients. The application stores the bond information in the RSL10 NVR2 flash and has the ability to perform Private Address Resolution. Additionally, it demonstrates how to use the BATMON alarm hardware of RSL10 to trigger an interrupt when the battery level falls below a configured threshold. | Yes | Yes |
| Heart Rate Peripheral FOTA | ble_peripheral_server_hrp_fota | This application is similar to the Heart Rate Bluetooth Low Energy application (*ble_peripheral_server_hrp*) with added features to support Firmware Over-The-Air (FOTA) updates. See the *Firmware Over-The-Air User's Guide* for more information. | | Yes |
| Heart Rate Peripheral Device with Server | ble_peripheral_server_hrp | This sample code extends the *ble_peripheral_server_bond* application to support the Heart Rate Profile (HRP) and the required paring/bonding sequences. The Heart Rate Profile is used to enable a data collection device to obtain data from a Heart Rate Sensor that exposes the Heart Rate Service (intended for fitness applications). This sample demonstrates how to implement the Heart Rate Sensor portion, including advertising, pairing, bonding and whitelisting; you must provide the Collector that receives the data. Data includes battery level notifications and the measured battery voltage. | Yes | |
| Private Resolvable Address with Peripheral Device | ble_peripheral_server_PRA | This application is similar to *ble_peripheral_server_bond*. In addition, it generates and changes the application Private Resolvable Address every 150 seconds. | | |
| Pairing and Bonding with Client Device | ble_central_client_bond | This sample project generates a battery service and a custom service client. It then connects up to four peripheral peer devices in direct connectable mode with known Bluetooth peer addresses. Once the connection is established with any peer device, the central device attempts to pair/bond and start encryption. It also starts battery/custom service discovery for that device. If the services are discovered, the application periodically sends read requests for the battery level and custom service attributes, using a kernel timer. The application stores the bond information in the RSL10 NVR2 flash, and can perform Private Address Resolution. Upon reconnection, the application starts encryption using the saved bond information. | Yes | Yes |
| SAI CMSIS-Driver | sai_cmsis_driver | This sample project demonstrates how to use the SAI CMSIS-Driver to transfer data in both master and slave modes between two RSL10 Evaluation and Development Boards. The project introduces you to using the SAI CMSIS-Driver with RSL10. | | |

**Table 1. Sample Code (Continued)**

| Name | Folder | Description | Keil Version | IAR Version |
|---|---|---|---|---|
| Central and Peripheral Device | ble_central_peripheral | This sample project fills both the central and the peripheral GAP roles, preparing the battery service and custom service clients for the central role of the application, and the battery service and custom service servers for the peripheral role. When the peripheral role establishes a connection with a central device (or fails to form a connection after advertising for 10 seconds), the application switches to the central role, and sends start connection commands to put the device in automatic connectable mode. This also applies to the central role of the application. This code demonstrates how a single application can switch between central and peripheral roles under conditions when this might be advantageous. | | |
| Blinky—Simple GPIO I/O | blinky | A simple sample project that causes the LED on the Evaluation and Development Board to flash. It is a gentle introduction to programming the RSL10 SoC, and demonstrates successful communication with the board, use of an interrupt, and GPIO functionality. | Yes | Yes |
| Bootloader | bootloader | This sample project provides a simple UART-based bootloader application. This allows the user to load their own software over the UART interface rather than using the JTAG/SWD debug port. The bootloader provides basic CRC verification of the loaded software to ensure that invalid applications are not executed on the device. If an invalid application is detected, the device remains in the bootloader state until new software is loaded. A helper application for the PC is also provided to aid in the loading process; it is written in Python for portability. | | |
| Central Device with Client UART | central_client_uart | This sample project generates a battery service client and a client for a bi-directional UART custom service. After initialization, this application uses the device in direct-connectable scan mode, looking to connect to a device with a known Bluetooth peer address. Once connected, this application queries the available services, and if the expected UART custom service is found, it uses this service to transmit data received on UART to the peer peripheral device, and to receive data from the peer device to be transmitted on the local UART interface. | | |

Table 1. Sample Code (Continued)

| Name | Folder | Description | Keil Version | IAR Version |
|---|---|---|---|---|
| Low Latency Audio Sample Application with Custom Protocol | custom_protocol_trx | This sample code demonstrates a complete audio path using the low-latency custom protocol as the wireless carrier of data.<br>On the transmitter side, data is received over the SPI interface. It is processed with the asynchronous sample rate converter (ASRC) to synchronize the audio sample rate between the Ezairo® 7100 Digital Signal Processor (DSP) system and RSL10. Data is then encoded using the G.772 codec on the LPDSP32 DSP, and sent wirelessly to the receiver using the custom low-latency protocol. The receiver decodes the data with a G.722 decoder implemented on LPDSP32 DSP, resynchronizes the data to local timing using the ASRC, and then sends the data via SPI to Ezairo 7100.<br>This code provides an example that shows how to use the low-latency custom protocol to implement low-latency audio streaming, which can be used as the basis for binaural processing applications.<br>NOTE: Ezairo 7100 components of this sample application set are provided in the *RSL10_Utility_Apps.zip* file; you need the Ezairo 7100 EDK to work with these components. | | |
| DMIC and OD | DMIC_OD | This sample project creates an audio pass-through from the DMIC interface to the output driver (OD). It shows the usage of the DMIC and OD interrupts. By pressing the button, the user can switch between four modes: DMIC0, DMIC1, DMIC0+1 mixed, and beam forming mode using fractional delay on the DMIC1 input. | | |
| Default System Initialization Function | default_MANU_INFO_INIT | This sample project:<br>• Erases the NVR3 area of flash memory, saving and restoring the Bluetooth device address and IP protection configuration<br>• Writes the MANU_INFO_INIT area of non-volatile record #3 (NVR3) with:<br>a. A length variable indicating the length of the default system initialization function<br>b. A default system initialization function<br>c. A CRC-CCITT value calculated over the length variable and the initialization function's code<br><br>The default system initialization function loaded is an example of how to load calibration settings that have been calculated for each part during device manufacturing to the various trimming bit-fields and registers for the system. | Yes | Yes |
| Host Controller Interface | hci_app | The Direct Test Mode (DTM) application implements a UART interface to the Bluetooth Host Controller Interface (HCI) wrapper that can be used to perform DTM and other HCI-based testing of the RSL10 device. This application configures voltage regulators, RF power supplies, and UART flow control signals to ensure that testing is performed correctly. This code demonstrates how to use the HCI interface, UART drivers, and different RF power supply configurations when executing HCI commands in a test environment. | Yes | Yes |

**Table 1. Sample Code (Continued)**

| Name | Folder | Description | Keil Version | IAR Version |
|---|---|---|---|---|
| Flash Copier and CRC | flash_copier_and_crc | This sample code project copies 4 KB of the flash into PRAM and computes the CRC of the same data with the flash copier block. It demonstrates usage of the flash copier interrupt. | | |
| DMA Driver | dma_driver | This sample project demonstrates how to use DMA to copy data between two arrays, and shows how to verify data correctness at the completion of the transfer. The project introduces you to the use of the DMA driver in RSL10 applications. | | |
| GPIO Driver | gpio_driver | This sample project demonstrates an application that uses a GPIO to toggle an LED on the RSL10 Evaluation and Development Board, and to disable the LED toggling. It is an introduction to using a GPIO driver in RSL10 applications. | | |
| I2C CMSIS-Driver | i2c_cmsis_driver | This sample project demonstrates how to operate the I$^2$C interface using the I$^2$C CMSIS-Driver abstraction. It shows how to transmit and receive messages between two RSL10 Evaluation and Development boards in both master and slave modes. | Yes | Yes |
| Kernel Timer | kernel_timer | A sample project that shows the usage of the kernel timer. The timer is set to two seconds; the kernel calls the timer's handler routine when the timer expires. | Yes | Yes |
| Measure 32 kHz RC Oscillator | measure_rc_osc | A sample project that measures the 32 kHz RC oscillator frequency every second by using the Audio Sink Clock Counters block, and reports the value through the UART interface to a terminal application on the host PC. It shows the usage of the Audio Sink Period interrupt. | Yes | Yes |
| Peripheral Device with Server | peripheral_server | This sample project generates a battery service and a custom service. After initialization, this application advertises its availability so that any central device can scan, connect, perform service discovery, receive battery value notifications, and read the battery level from this application. This code is an example of communication (reading, writing, and message-handling) between central and peripheral devices, and kernel timer use.<br>The central device has the ability to read and write the custom attributes provided by this sample application’s custom service. The RSL10 ADC is used to read the battery level. Reading the battery level happens every 200 ms when there is a kernel timer event. The average for 16 reads is calculated. If the average value changes, a flag is set to send the battery level notification. | | |
| Peripheral Device with Sleep Mode | peripheral_server_sleep | This sample code incorporates a Sleep Mode into the *peripheral_server* sample code project. This application places the peripheral device into Sleep Mode whenever possible to save power. On waking up, the device retains its Bluetooth low energy connection with the central device and resumes all normal operations of its application. This code serves as a template for the use of Sleep Mode for power-saving in server applications used for peripheral devices. | | |

Table 1. Sample Code (Continued)

| Name | Folder | Description | Keil Version | IAR Version |
|---|---|---|---|---|
| Peripheral Device with Standby Mode | peripheral_server_standby | This sample code incorporates a Standby Mode into the peripheral device with a server sample code project. This application places the device into Standby Mode whenever possible to save power. On waking up, the device retains its Bluetooth low energy connection with the central device and resumes all normal operations of its application. This code serves as a template for the use of Standby Mode for power-saving in server applications used for peripheral devices. | Yes | Yes |
| Peripheral Device with UART Server | peripheral_server_uart | This sample project generates a battery service and a bi-directional UART custom service. After initialization, this application advertises, so that any central device can scan, connect, perform service discovery, receive battery value notifications, and read the battery level. If connected, the application can use this service to notify the connected client of data received on the UART interface to the peer central device, and to receive data from the peer device to be transmitted on the local UART interface. | | |
| Print NVR Information Using Semi-Hosting | print_nvr_info | This sample project prints out the contents of the NVR sectors via semi-hosting. The project has been created so that the developer can easily see what data is stored in the NVR sectors of RSL10 components mounted on any Evaluation and Development Board. No external connection is required.<br>The program runs from PRAM. Executing it does not change the contents of the flash. | | |
| PWM Driver | pwm_driver | This sample application uses pwm[0] to control the brightness of the LED diode with use of the duty cycle, and uses the push button (DIO5) input to change the brightness of the LED (DIO6) by changing the Pulse With Modulator (PWM) duty cycle. | | |
| Mono Audio Stream Broadcast Receiver Custom Protocol Coexistence | remote_mic_rx_coex | This sample code shows how the Audio Stream Broadcast Custom Protocol can be used by a remote microphone to receive audio in RSL10 while coexisting with a Bluetooth low energy connection. The code is configurable for raw or encoded data streams.<br>The Bluetooth low energy application used in this case is based on the *peripheral_server* sample application, defining a custom service which allows reception of custom protocol parameters from a central device. The custom service also configures the process for starting and stopping audio reception. | | |
| Stereo Audio Stream Broadcast Transmitter Custom Protocol Coexistence | remote_mic_tx_coex | This RSL10 sample application demonstrates switching a remote microphone from actively-connected Bluetooth low energy mode, to stereo audio stream transmission in RSL10 through the Audio Stream Broadcast Custom Protocol.<br>Once the transmitter has established a Bluetooth low energy connection with the receiver, the transmitter exchanges the correct parameters to begin the audio stream, then cancels its Bluetooth low energy connection and streams the audio through the remote microphone custom protocol. The code is configurable for encoded or raw data streams. This sample code can be compiled and loaded onto the RSL10 transmitter. Transmission of stereo audio is supported. | | |

**Table 1. Sample Code (Continued)**

| Name | Folder | Description | Keil Version | IAR Version |
|---|---|---|---|---|
| Stereo Raw Audio Stream Broadcast Transmitter Custom Protocol | remote_mic_tx_raw | This RSL10 sample code shows a complete audio path for a remote microphone in the transmitter side, with raw stereo input audio stream, using the Audio Stream Broadcast Custom Protocol.<br>The SPI, DMIC, or PCM interface receives the broadcast data stream, and synchronizes the sampling rate to work with RSL10 using an asynchronous sample rate converter. The stereo audio stream then passes to the LPDSP32 DSP for encoding, and is broadcast using the remote microphone custom protocol. In addition, this sample application represents a standard LPDSP32 framework for other codec implementations. | | |
| Mono Raw Audio Stream Broadcast Receiver Custom Protocol | remote_mic_rx_raw | This RSL10 sample code shows a complete audio path for a remote microphone in the receiver side, with raw output audio stream, using the Audio Stream Broadcast Custom Protocol.<br>A remote microphone receives the broadcast data stream, and passes it to the LPDSP32 DSP for decoding. An asynchronous sample rate converter then synchronizes the sampling rate to work with RSL10. After this, transmission of data takes place through an SPI or OD interface. | | |
| Coded Audio Stream Broadcast Transmitter/ Receiver Custom Protocol Sample Application | remote_mic_trx_coded | This RSL10 sample application shows a complete audio path in a remote microphone use case, for both stereo transmitter and mono receiver, using the Audio Stream Broadcast Custom Protocol. No encoding or decoding takes place in RSL10; the paired Ezairo 7100 performs these processes,<br>The compiled application can be loaded onto both the RSL10 transmitter and the receiver. On the transmitter side, an SPI interface is used to receive the data, which is then sent to the radio side for transmission. | | |
| UART CMSIS-Driver | uart_cmsis_driver | This sample project demonstrates how to operate the UART peripheral using the UART CMSIS-Driver abstraction. It shows how to transmit and receive messages between two RSL10 Evaluation and Development boards. | Yes | Yes |
| Sleep and Wakeup | sleep_RAM_retention | This sample project demonstrates how to blink an LED, switch to Sleep Mode (with or without memory retention) and wake up from sleep via the RTC alarm or the WAKEUP pad on a rising edge. Using the RTC as is demonstrates a 16 second sleep with wakeup on RTC alarm. This code shows how to use the Sleep Power Mode to save power, with different wakeup methods. Additional options within this application allow configuration to retain one or two memory instances during Sleep Mode. | | |
| SPI CMSIS-Driver | spi_cmsis_driver | This sample project demonstrates how to simultaneously operate both SPI0 and SPI1 interfaces using the SPI CMSIS-Driver abstraction. It shows how to transmit and receive messages between two RSL10 Evaluation and Development boards in both master and slave modes. | Yes | Yes |
| Standby Power Mode | standby_power_mode | This application briefly blinks an LED, switches to Standby Mode, configured to wake up on the RTC alarm and restart execution. This code demonstrates using Standby Mode with an alarm to save power. | | |

**Table 1. Sample Code (Continued)**

| Name | Folder | Description | Keil Version | IAR Version |
|---|---|---|---|---|
| Supplemental Calibration | supplemental_calibrate | This sample project:<br>• Calibrates VDDRF to 1.13 V (targeting a 1 dBm TX power), VDDPA to 1.45 V (targeting a 4 dBm TX power), and the RC start oscillator to 10.24 MHz (targeting an even multiple of a 16 kHz sampling frequency)<br>• Erases the NVR3 area of flash memory, saving and restoring the Bluetooth device address and IP protection configuration, and storing the supplemental calibration information.<br>• Writes the MANU_INFO_INIT area of non-volatile record #3 (NVR3) with:<br>  • A length variable indicating the length of the default system initialization function<br>  • A custom system initialization function that loads some the three supplemental calibrated targets, some standard but non-default calibration entries, and some default calibration entries. This provides an example implementation for a use case when the default configuration is not what a user application needs or will use.<br>  • A CRC-CCITT value calculated over the length variable and the initialization function's code<br><br>The custom system initialization function is an example of how to load calibration settings that have been calculated for each part during device manufacturing, supplemented during customer manufacturing, to the various trimming bit-fields and registers for the system. | Yes | Yes |
| Timer Driver | timer_driver | This application demonstrates the use of three timers running in three different modes:<br>• Timer [0] gives the interval between each blink of the LED; timer [0] works in free run mode.<br>• Timer [1] changes the timer[0] interval and starts timer [2]; timer[1] works in single shot mode.<br>• Timer[2] gradually reduces the timer[0] interval four times; timer[2] works in multi shot mode. | | |
| SysTick Timer Using Reference Clock | systick_ref_clk | This sample project demonstrates the usage of the SysTick timer. Pressing a push button causes the change of frequency of the SysTick timer expiration by changing its load value, which is indicated by an LED blinking on the Evaluation and Development board. | | |
| Timer Free Run | timer_free_run | This sample project demonstrates the usage of a timer in free running mode. Pressing a push button causes the change of frequency which is indicated by an LED blinking on the Evaluation and Development board. It shows the usage of the timer interrupt as well. | | |
| Timer Multi-Shot | timer_multi_shot | This sample project demonstrates the usage of a timer in free running mode and another timer in multi-shot mode. Pressing a push button changes the number of LED blinks on the Evaluation and Development board. It also shows the usage of timer interrupts. | | |

# APPENDIX A

# Message Sequence Chart for Peripheral Device with Server Sample Application

The messages communicated between the *peripheral_server* application and the Bluetooth low energy stack in establishing a connection for data to be exchanged are illustrated in Figure 1.

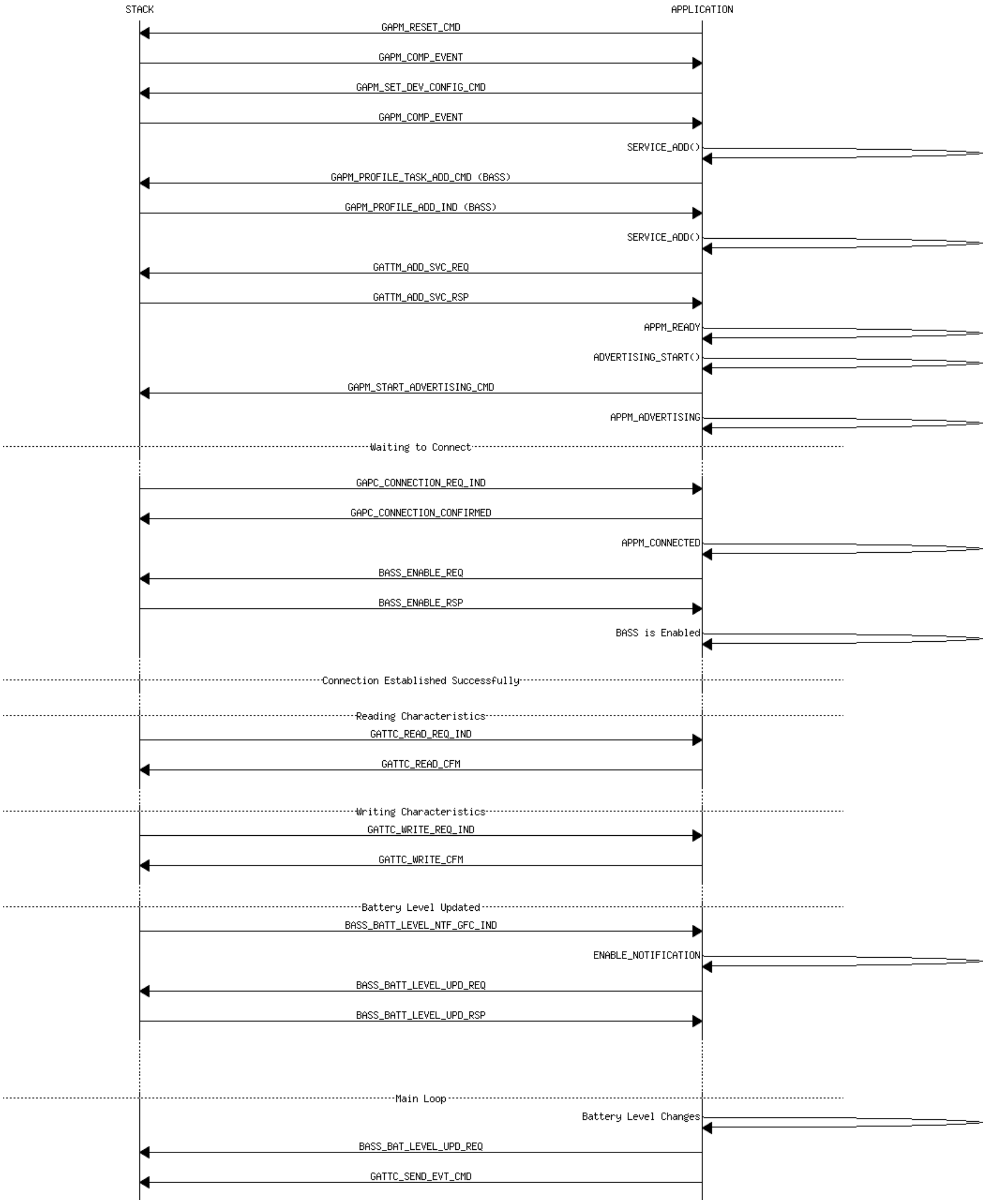

**Figure 1. MSC for Peripheral Device with Server Sample Code**

**PUBLICATION ORDERING INFORMATION**

**LITERATURE FULFILLMENT:**
Literature Distribution Center for ON Semiconductor
19521 E. 32nd Pkwy, Aurora, Colorado 80011 USA
**Phone**: 303-675-2175 or 800-344-3860 Toll Free USA/Canada
**Fax**: 303-675-2176 or 800-344-3867 Toll Free USA/Canada
**Email**: orderlit@onsemi.com

**N. American Technical Support**: 800-282-9855 Toll Free USA/Canada
**Europe, Middle East and Africa Technical Support**: Phone: 421 33 790 2910

**ON Semiconductor Website: www.onsemi.com**

**Order Literature**: http://www.onsemi.com/orderlit

For additional information, please contact your local Sales Representative