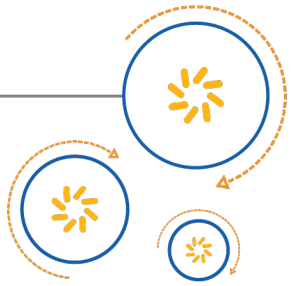
Qualcomm Technologies International, Ltd.



# Writing Qualcomm BlueCore Applications

## User Guide

80-CT402-1 Rev. AH

October 18, 2017



Qualcomm Technologies International, Ltd. (formerly known as Cambridge Silicon Radio Limited) is a company registered in England and Wales with a registered office at: Churchill House, Cambridge Business Park, Cowley Road, Cambridge, CB4 0WZ, United Kingdom.
Registered Number: 3665875 | VAT number: GB787433096

# Revision history

| Revision | Date | Description |
|---|---|---|
| 1 | JUL 2010 | Initial release. Alternative document number CS-00207482-UG. |
| 2 | OCT 2010 | Updated code fragments to reflect example applications. |
| 3 | JUL 2011 | New Non Profile message bases added. Updated to latest style. |
| 4 | JAN 2012 | Updated to latest style. |
| 5 | DEC 2013 | Updated to new CSR™ style. |
| 6 | SEP 2016 | Updated to conform to QTI standards; No technical content was changed in this document revision |
| 7 | APR 2017 | Updated for ADK 4.2. Added to the Content Management System |
| AH | OCT 2017 | Document Reference Number updated to use the Agile number. No change to technical content. |

# Contents

# Figures

# 1 How to write BlueCore applications - Overview

Qualcomm® BlueCore™ technology SDKs provide developers with a series of libraries that support the implementation of Bluetooth applications that run on BlueCore ICs.

The use of the supplied libraries allows engineers to develop BlueCore applications that implement Bluetooth Profiles without becoming distracted by the complexity of the underlying Bluetooth Protocols.

This approach has the additional benefit of allowing the underlying BlueCore firmware to be considered as a pre-qualified component of the final product.

# 2 Tasks, messages and message handlers in BlueCore applications

It is important to understand the concept of tasks, messages and message handling as they are integral to the way BlueCore applications are implemented.

**NOTE** The use of function pointers (with the exception of the task handler) is generally not recommended and should be avoided as:

- xIDE is unable to automatically calculate the stack size
- It generates excessive code instructions which degrade the performance in Native VM

### Tasks

Tasks are the basic architectural building blocks used to construct an application and to provide an interface with the BlueCore firmware.

Tasks provide functionality to the application, that is, an instance of the appropriate task must be initialized by the code in order for the functionality it supports to be available to the application.

A task is basically a message handling function and a structure containing the task's current state. All tasks are run as a single thread, that is tasks do not execute concurrently.

The tasks that make up the application include a top level task known as the application task. The application task responds to messages and controls the behaviour of the application.

**NOTE** More complex applications may define more than one application task.

### Messages

Messages pass information between the tasks and are constructed in the form:

```
Task t, MessageId id, Message payload
```

The elements of a message are:

- `Task t`: Identifies the destination of the message, that is, it is a pointer to the recipient task, for example, `&AppTask`.
- `MessageId id`: is the message label that allows the task receiving the message to identify the appropriate function within the handler code.
  The `MessageId` id also implies the fields that can be expected in the payload. This allows the required information to be extracted by casting pointers to the payload fields.

**NOTE** By convention, a message's structure is always enumerated as the Message Id with a suffix of `_T`. For example, the structure of the message `CL_INIT_CFM` is enumerated as Type `CL_INIT_CFM_T`.

- `Message payload`: The payload should contain any state data required by the handler function to correctly react to the message. The payload is freed after the message has been delivered. To maximize the efficient use of memory the data passed in the payload should be kept to the minimum required by the message handling function to implement the appropriate response. In some cases, the message id alone may be sufficient. Therefore, the payload field is optional and can be `NULL`.

### Message ID numbering

The base number for `MessageIds` adhere to the following conventions:

- Messages send by a task to itself start at `0x00`.
- System messages start at `0x8000`, see BlueCore library message bases.
- Messages sent to the library tasks have been assigned specific base values based on its associated library task. These base values are defined in `library.h`. See.

### Sending messages

A number of functions are provided to facilitate the sending of messages, see `message.h` in the on-line help *VM and Native Reference Guide*.

## 2.1 How messages are handled in BlueCore applications

The message handling facility for each task must be able to successfully handle all the `MessageIds` and payloads that can be passed to it.

In practice the developer's main responsibility is to provide the message handling code for the messages that will be received by the application task.

**NOTE** The message handlers for profile and support tasks initialized from SDK libraries need not concern developers. The handlers for these tasks are implemented by the libraries. These tasks are initialized during the connection establishment in some profile libraries that implement the resource efficient method of task creation. Other profile libraries and support libraries initializes the tasks as a result of calling the appropriate Init function.

The application must handle messages received from the libraries that it has initialized.

### What does the MessageLoop function do

The `MessageLoop` function controls the main scheduler loop and handles the delivery of task messages.

Messages are added to the queue in the order they are due to be delivered. The scheduler checks the first message in the queue and if due delivers the message to the appropriate task.

**NOTE** The `MessageLoop` function never returns and therefore is the last function called before the return statement in main.

Tasks and messages allow the application developer to split the application up into cooperating modules that is, tasks) that handle messages from the firmware and each other.

# 3 Working examples of tasks and messages

### Example 1: Single task application flashing an LED

The first example looks at a single task application that flashes an LED on a PIO pin with a regular on/off period of 500 milliseconds.

**NOTE** The application has been chosen for simplicity, as it is one of the few examples that can be run on BlueCore chips without using the connection library.

To do this the code:

- Defines a state for the single task required.
- Defines a handling routine for messages (that will read the state of a PIO pin and switch it at regular intervals).
- Sets the PIO as output.
- Declares an instance of the task and initializes it.
- Sends an initial message that invokes the handler code.
- Calls the `MessageLoop` function.

### The code

```
#include <message.h>
#include <pio.h>
#define LED_1 (1<<6) /* set PIO pin no for LED_1 */
typedef struct
{
    TaskData task;
    uint16 change;
} ToggleTask;
static void MyHandler(Task t, MessageId id, Message payload)
{
    /* id and payload unused and both 0 */
    uint16 change = ((ToggleTask *) t)->change;
    PioSet32(change, PioGet32() ^ change);
    MessageSendLater(t, 0, 0, 500);
}
static ToggleTask toggle = { { MyHandler }, LED_1 };
int main(void)
```

```
{
    PioSetDir32 (LED_1, ~0); /* Set LED_1 PIO pin as output */
    MessageSend (&toggle.task, 0, 0);
    MessageLoop(); /* never returns */
    return 0;
}
```

### Analysis of code

1. The code employs functions declared in the `message.h` and `pio.h` header files so the first step is to `#include` these files:

   ```
   #include <message.h>
   #include <pio.h>
   ```

2. The next line of code defines a variable used to set a mask that identifies the PIO pin number that the LED is attached to (in this example PIO 6):

   ```
   #define LED_1 (1<<6) /* set PIO pin no for LED_1 */
   ```

3. The next step is to define a structure for the task state. This is done using the `typedef` specifier:

   ```
   typedef struct
   {
       TaskData task;
       uint16 change;
   } ToggleTask;
   ```

   This defines a structure for the task and gives it the type ToggleTask.
   The first element of the structure, that is task, has the type TaskData defined as:

   ```
   { void (*handler)(Task, MessageId, Message); }
   ```

   This consists of a pointer to a function that takes the three standard message fields as arguments, that is a handler function.

4. The next block of code defines the message handler function `MyHandler`:

   ```
   static void MyHandler(Task t, MessageId id, Message payload)
   {
       /* id and payload unused and both 0 */
       uint16 change = ((ToggleTask *) t)->change;
       PioSet32(change, PioGet32() ^ change);
       MessageSendLater(t, 0, 0, 500);
   }
   ```

   This code accepts a message as arguments then:

   - Variable change is assigned the value `t->change` where task `t` is a pointer of type `ToggleTask`.
   - Calls the `PioSet32` function, the arguments passed to it have the effect of flipping the LED state, that is, if the LED is on it is switched to off and vice versa.
   - The final statement sends a message after a delay of 500 milliseconds. The message scheduler delivers the message to `MyHandler` and the process is repeated, resulting in the LED flashing on/off every 500 milliseconds.

5. The next line of code initializes the task:

```
static ToggleTask toggle = { { MyHandler }, LED_1 };
```

   It defines a task, toggle with parameters that point to the handler function and set the bit mask for the PIO connected to the LED, that is, the handler will set PIO 6 (see `#define` statement).

6. The final block of code is the main function:

```
int main(void)
{
    PioSetDir32 (LED_1, ~0); /* Set LED_1 PIO pin as output */
    MessageSend (&toggle.task, 0, 0);
    MessageLoop(); /* never returns */
    return 0;
}
```

7. Looking at these statements individually:
   - The first statement sets the PIO pin to which the LED is connected to output.
   - The second sends a message to the handler, this kicks-off the LED sequence.
   - The third statement calls the `MessageLoop()` function for message delivery.
   - The final line complies with the ANSI C requirement to return a value.

### Example 2: Single tsk application flashing two LEDs

Example 2 expands on example 1 by adding a second flashing LED. This is achieved by adding a second task (in this example called flash).

### The code

```
#include <message.h>
#include <pio.h>
#define LED_1 (1<<6) /* set PIO pin no for LED_1 */
#define LED_2 (1<<7) /* set PIO pin no for LED_2 */
typedef struct
{
    TaskData task;
    uint16 change;
} ToggleTask;
static void MyHandler(Task t, MessageId id, Message payload)
{
    /* id and payload unused and both 0 */
    uint16 change = ((ToggleTask *) t)->change;
    PioSet32(change, PioGet32() ^ change);
    MessageSendLater(t, 0, 0, 500);
}
static ToggleTask toggle = { { MyHandler }, LED_1 };
static ToggleTask flash = { { MyHandler }, LED_2 };
int main(void)
```

```
{
    PioSetDir32(LED_1|LED_2, ~0); /* LED_1 and LED_2 PIOs as output */
            MessageSend (&toggle.task, 0, 0);
MessageSendLater (&flash.task, 0, 0, 250 );
    MessageLoop(); /* never returns */
    return 0;
}
```

### Analysis of code

This section considers how the code has been adjusted to add the second LED. Essentially this involves just 4 lines of code:

```
#define LED_2 (1<<7) /* set PIO pin no for LED_2 */
```

Defines a label for the PIO pin to which the LED is connected (in this case PIO 7).

```
static ToggleTask flash = { { MyHandler }, LED_2 };
```

Initializes a second task flash with parameters that point to the handler function and set the bit mask for the PIO connected to a second LED, that is, the handler will set PIO 7 (see `#define` statement).

```
PioSetDir32 (LED_1|LED_2, ~0); /* LED_1 and LED_2 PIOs as output */
```

Sets the PIOs connected to the LEDs direction to output.

```
MessageSendLater (&flash.task, 0, 0, 250);
```

Finally, the second LED is started after the first LED by the `MessageSendLater` function, the final parameter of which dictates the delay (in milliseconds) before the message is sent.

### Example 3: Alternative method of controling flashing LEDs

Example 3 examines an alternative method of controling the flashing LEDs. The salient point in this example is the addition of a field to the Task structure used to control the timing of the LEDs flashing sequence.

### The code

```
#include <message.h>
#include <pio.h>
typedef struct
{
    TaskData task;
    uint16 change;
    Delay delay;
} ToggleTask;
static void MyHandler(Task t, MessageId, Message payload)
{
    /* id and payload unused and both 0 */
    ToggleTask *tt = (ToggleTask *) t;
    uint16 change = tt->change;
    PioSet32(change, PioGet32() ^ tt->change);
    MessageSendLater(t, 0, 0, tt->delay);
```

```
}
static ToggleTask toggle1 = { { MyHandler }, 1<<1, 500 };
static ToggleTask toggle2 = { { MyHandler }, 1<<2, 250 };
int main(void)
{
    PioSetDir32(1<<1|1<<2, ~0);        /* PIO1 and PIO2 as output */
    MessageSend(&toggle1.task, 0, 0);
    MessageSend(&toggle2.task, 0, 0);
    MessageLoop();                    /* never returns */
    return 0;
}
```

### Analysis of code

The first point to note is the addition of a new field to the `Task` structure:

```
typedef struct

{

TaskData task;

uint16 change;

Delay delay;

} ToggleTask;
```

The code adds a new field to `ToggleTask` with the pre-declared type Delay. This new field delay is used later in the code to control the rate at which the LEDs will flash.

```
static ToggleTask toggle1 = { { MyHandler }, 1<<1, 500 };
static ToggleTask toggle2 = { { MyHandler }, 1<<2, 250 };
```

The task definition now uses the delay parameter to set the rate at which messages will be sent by the handler function, that is, controlling the flash rate of the LEDs.

The tutorial applications `blinking_lights`, buttons and `message_handler` demonstrates more examples on handling the tasks and messages.

# 4 BlueCore application architecture

A typical BlueCore application uses tasks and messages to support Bluetooth profiles.

The application architecture can be considered in three parts:

- The application task.
- The profile and support tasks (including the connection task).

**NOTE** These are library tasks provided by QTIL to implement common functions, such as Bluetooth profiles.

- The firmware.

The architecture of the application is best illustrated graphically:

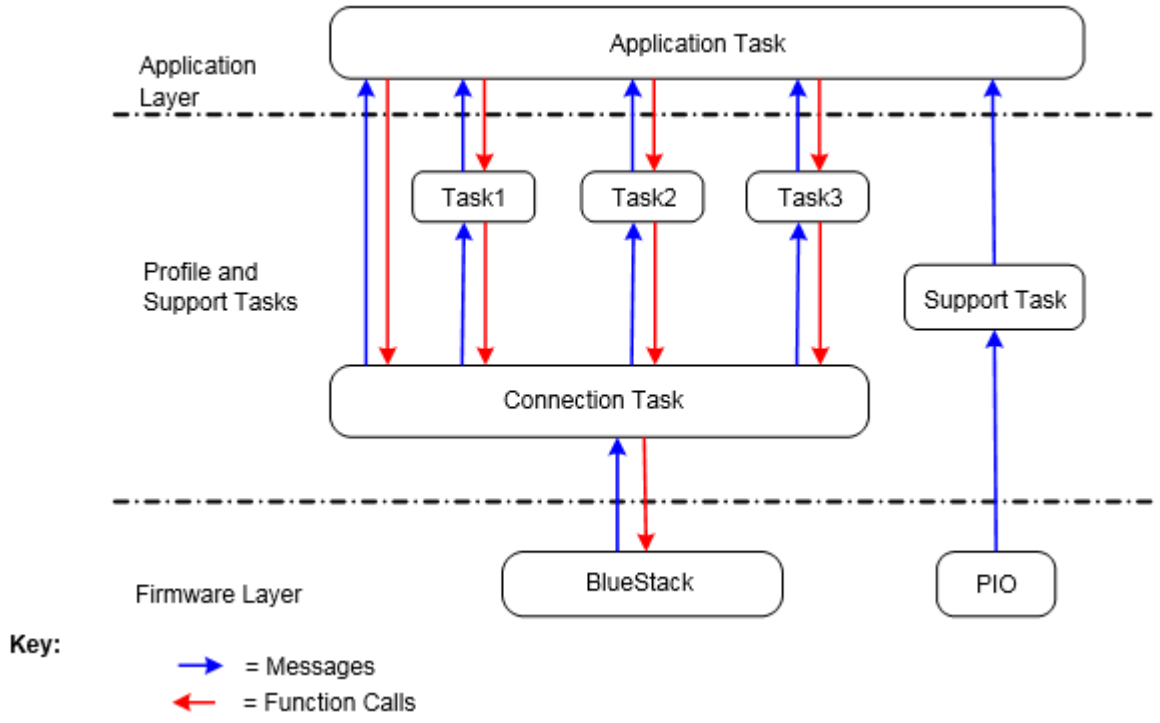



**Figure 4-1 Typical application architecture**

### The application task

The application itself must be initialized as a task, and is generally known as the `AppTask`.

Defining a structure for the `AppTask` and coding the message handler represents the minimum requirement when developing a BlueCore application.

The data structure for the `AppTask` is determined by the specific application's requirements and should include any data required to identify the state of the other tasks used by the application.

### Profile and support tasks

BlueCore SDKs provides libraries that implement commonly used functionality, including Bluetooth profiles, to support the development of BlueCore applications.

To use a profile or support library in an application the developer must #include the appropriate library header file and call the appropriate initialization function to initialize the profile or support library task. If an Initialization function is provided by the library, it must be called only once from the application.

The resultant task can then be regarded as a black box that handles any messages it receives and responds with a message to the application task where appropriate.

```
#include <connection.h>
int main(void)
{
    /* Init the Connection Manager */
    ConnectionInit(&theSppApp.task);
}
```

The code initializes the `connection` library. The connection task created can support multiple connections and should only be called once.

The application task receives an `<XXXX>_INIT_CFM` message for an `<Xxxx>Init()` on initialising the profile or support task and registering the application task. The `message->status` indicates whether the task was initialized successfully or not.

As long as the developer's application message handler code correctly handles messages received from an instance of a task, the task will provide the expected functionality.

**NOTE** It is important to initialize the connection library before any other profile libraries because the connection library task is required during the creation of profile tasks.

Some profile libraries capable of handling multiple connections implement the dynamic task creation. These profile libraries may not have an `Init()` function, but instead creates the task and registers the `AppTask` during the connection establishment and releases the task during the profile disconnection. See the API documentation of the profile libraries for more details

### Firmware

The connection library provides an interface between SDK applications and the BlueCore firmware that implements the Bluetooth features.

When the connection task is initialized, the developer does not need to be concerned with how the connection task implements underlying functionality supported by the firmware.

The developer can concentrate on handling the limited number of messages that the application task can potentially receive from the connection task.

**NOTE** Details of the messages generated by a connection task can be found in the xIDE on-line *VM/Native Library Reference Guide*.

### Message scheduler

BlueCore SDKs provides a function to implement message scheduling. This function handles the delivery of task messages.

```
/* Start the message scheduler loop */
MessageLoop();
```

The `MessageLoop` function acts on queued messages owned by the tasks. Messages are added to the queue in delivery due time order. The scheduler looks at the first message in the queue and if due, delivers it as arguments to the appropriate task handler.

After a message has been passed to the task handler the `MessageLoop` function frees the original message payload before handling the next message in the queue or waiting until another message is added to the queue.

**NOTE** The message scheduler is not pre-emptive. It is therefore important that all handler functions run to completion and do not loop forever.

# 5 Planning and coding a BlueCore application

Where operations need to be run before the initialization of tasks and even before the main loop of the application, the `Init` function should be used. The `Init` function can be used to initialize time critical functionality, such as registering USB descriptors. It is called before the debugger gets control of the application and therefore cannot be debugged. As a result, `Init` should only be used where absolutely required.

## 5.1 Planning the development of BlueCore application

The first step when developing an application is to determine the tasks required to provide the required functionality.

In practical terms Bluetooth applications always require:

- An application task
- A connection task

The other tasks depend on the type of device being developed and it is the developer's responsibility to identify the Profile and Support tasks that are required by their application.

When all the tasks that are required to build the application have been identified, the developer can begin to consider the code required to support them.

### Coding overview

In simplistic terms the finished application code needs to:

1. Set up and initialize the required tasks.
2. Handle the messages received by the application task.

These two basic requirements constitute the essential elements of all BlueCore application code. The actual way the code handles these requirements is largely at the discretion of the developer.

The code required to implement an actual application can become complex.

## 5.2 Initializing tasks

The code required to initialize a task depends on whether the task is developed by the software engineer, for example, the application task, or a task supported by a SDK library, for example a profile task.

### Initialising an application task

Setting up and initialising an application task involves three distinct steps:

- Typedef a data structure for the application task.
- Declare a variable for the application task.
- Initialize the application task states.

When initialising an engineer defined task the developer is responsible for including code to handle each of the above steps.

### Defining task state structures

The structure of the application task data type is at the discretion of the developer. It should allow for any data required to identify the state of other tasks and any other data the developer may want to use within the application.

It is the developer's responsibility to define a suitable structure for the application task.

The figure below indicates a typical `typedef` for a simple Serial Port Profile (SPP) application task. A simple example implementation for the SPP client and server are provided in the example applications `spp_client` and `spp_server`.

**NOTE** SPP has been selected as it is provides a simple example.

```
typedef enum
{
clientInitialising = 0x00,
clientUnconnected = 0xC0,
clientConnected = 0xE0
        } client_state;

        typedef struct
{
TaskData task;
bdaddr bd_addr;
PioState pio;
client_state state;
SPP *spp;
        } APP_SPPC_T;
```

**Figure 5-1 Defining a task data structure**

When declaring a structure for a task the `TaskData` element is mandatory and must be the first element of the structure.

**NOTE** TaskData is defined as:

```
{ void (*handler)(Task, MessageId, Message); }
```

### Declaring the application task variable

When the structure for the application task has been defined, the application task can be declared, for example:

```
static APP_SPPC_T theApp;
```

This declaration defines the application task as a static variable called `theApp` with a data type as defined by `APP_SPPC_T`.

### Initializing the application task

Initialize the application task by setting task's message handler and its initial states:

```
/* Set up the task handler. */
theApp.task.handler = sppc_handler;

/* Set the initial client state */
PioSetDir32(LED_MASK,LED_MASK);
setClientState(clientInitialising);
```

**Figure 5-2 Initializing an application task**

In most cases it is also desirable to set up initial states for any variables associated with the task data.

### Initializing a profile task

Before calling any profile library APIs, the application must initialize the Connection library:

```
/* Start the connection library */
ConnectionInit (&theApp.task);
```

**Figure 5-3 Initializing the connection library**

**NOTE** Some profile libraries also provide Init functions similar to `ConnectionInit` to initialize a task. If this is the case the application must call the appropriate Init function before calling any other APIs provided by the profile library. See the on-line Support Documentation for further information.

Profiles that do not provide an `Init` function, create a task instance when a connection is requested (1), for example:

```
SppConnectRequest (&theApp.task, &theApp.bd_addr, 0, 0);
```

### Requesting a connection

**NOTE** The created task is automatically destroyed and the memory released when the connection is disconnected.

When the connection has been established the task created sends an `SPP_CLIENT_CONNECT_CFM` message to the Application, for example:

```
static void sppc_handler(Task task, MessageId id, Message message)
{
    switch(id)
    {
         case SPP_CLIENT_CONNECT_CFM:
              {
              SPP_CLIENT_CONNECT_CFM_T *sccc =
         (SPP_CLIENT_CONNECT_CFM_T*)message;
              PRINT(("SPP_CLIENT_CONNECT_CFM :-"));
              if (spp_connect_pending == sccc->status)
              {
                PRINT(("Pending\n"));
                theApp.spp = sccc->spp;
}
else if (spp_connect_success == sccc->status)
                              {
                              Source uartSource = (Source) PanicZero(
                                                    StreamUartSource() );
                              Sink uartSink = (Sink) PanicZero(
                                         StreamUartSink() );
                              PRINT(("Success\n"));
                              setClientState(clientConnected);

                    ..................
                    ..................

                              }
              default:
                    break;
         }
    }
```

**Figure 5-4 Receiving a connection confirmation**

The SPP library destroys the task instance when disconnecting the associated SPP connection. In order to ensure that the clean up process is complete the App task calls `MessageTaskFlush` to clear any queued messages before it is destroyed. This frees all the memory associated with the connection.

## 5.3 Dynamic tasks

Some profiles implement a resource efficient method of task creation. These tasks are dynamically created during connection establishment and destroyed during disconnection. The creation and destruction of these tasks is completely handled within the profiles without affecting the application interface.

**Profile initialization**

Initialization of profile includes:

- Registration of SDP record
- Registration of L2CAP channel
- Allocation of RFCOMM channels, and so on.

In most cases, the client role of the profiles does not require an SDP record. In these scenarios profile initialization can be ignored.

If profile initialization is required it must provide an `Init` function to do that. The profile creates a temporary task during the initialization and deletes the task at the end of initialization.

**Incoming connections**

On an incoming connection the task instance needs to be created dynamically by the profile library when it first receives the incoming connection indication from the connection library.

At this point the profile task instance has been created and partially initialized and the incoming connection indication can be sent to the application task along with the profile task instance. The application must use this task instance as a connection handle for each profile connection.

**Outgoing connections**

On an outgoing connection the profile library creates the profile instance and using it in the connect request to the `connection` library. All libraries follow the same convention and the API function for initiating a connection creates an internal message to send to itself.

The profile instance must be created and initialized before this internal message is sent so that, by the time the message is delivered, the task is ready to be used.

**Destroying a dynamic task**

Tasks that are created when required must be destroyed when they are no longer needed. In most cases this will be when the profile level connection is disconnected. On a disconnect indication, the profile library is responsible for notifying any clients of the disconnect and then cleaning up its state and destroying the profile task instance.

When a client has received a disconnect indication it must not attempt to use the profile task instance (identified in the indication message) as that task has already been destroyed.

**NOTE** A disconnect is not the only case where a task will need to be destroyed. Other cases include a connect fail or the rejection of an incoming connection by the application.

When the task needs to clean itself up, there may be outstanding messages for it in the queue. `MessageTaskFlush` must be called to ensure any such messages are removed. This is done by the profile instance as part of the clean up process.

## 5.4 Message handling

The application task message handler must be able to successfully handle all the message states that can be passed to it by the other tasks that provide the application functionality.

Typically, the handler consists of switch statements designed to take the appropriate action in response to the message id and the task state as indicated by a message passed to the handler function. The example shows a simplified example indicating typical message handler code.

Example:

```
/* Task handler function */
static void foo_app_handler(Task task, MessageId id, Message message)
{
    /* Save the application's state */
    fooAppState state = fooApp.state;
    switch(id)
    {
    case FOO_MESSAGE_1:
          printf(("FOO_MESSAGE_1\n"));
          switch(state)
          {
          case FOO_UNINITIALISED:
                printf("foo not initialised!\n");
                break;

          case FOO_READY:
                fooPerformMessage1Processing();
                break;
          }
          break;

    case FOO_MESSAGE_2:
          printf(("FOO_MESSAGE_2\n"));
          switch(state)
          {
          case FOO_UNINITIALISED:
                printf("foo not initialised!\n");
                break;

          case FOO_READY:
          fooPerformMessage2Processing();
          break;
          }

          break;
    }
}
```

**Figure 5-5 Message handling code**

It is the developer's responsibility to ensure that the application task handles the messages it receives, to achieve the results expected/required by the application.

**NOTE** By convention, the message structure is always enumerated as the name of the message (that is, the Message Id) with a suffix of `_T`. For example, if the message name is `CL_INIT_CFM` then the message type will be `CL_INIT_CFM_T`.

# 6 Using the Bluetooth radio, working example

In this example to set up a simple ACL connection between two Bluetooth devices, the connection is used for L2CAP and employs Bluetooth 2.1 Simple Pairing *Just Works* for authentication of the link.

The example requires two separate communicating applications, a client application on one device and a server application on the other device.

Example applications `l2cap_echo_client` and `l2cap_echo_server` demonstrate the L2CAP connection establishment between two Bluetooth devices.

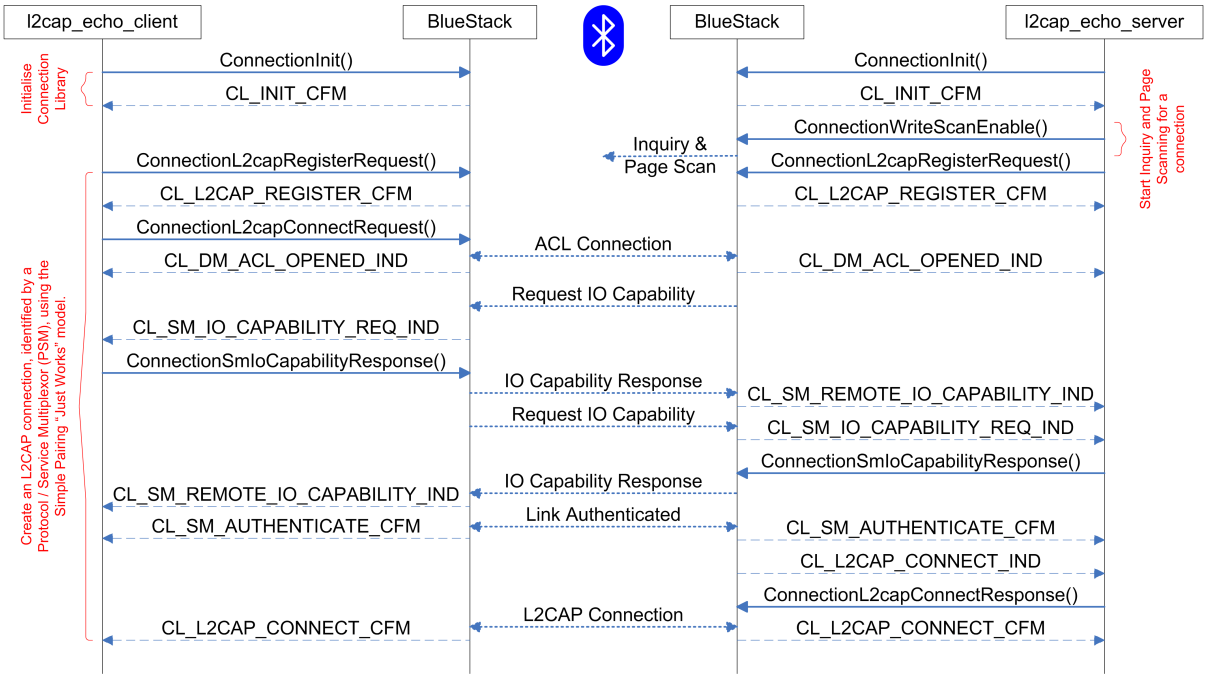



**Figure 6-1 L2CAP Connection message sequence chart**

The figure shows the basic message sequence chart for the applications on two Bluetooth devices, `l2cap_echo_client` and `l2cap_echo_server`. In the example, `l2cap_echo_client` is the master and initiates the connection to `l2cap_echo_server` which is Inquiry and Page scanning.

**NOTE** The protocol stack radio interactions are simplified, as that is not the focus of this example.

The applications in this example share much of their functionality, therefore the description of the code samples indicates whether it is specific to the master application, specific to the slave application or common to both:

- `common`: Common to both applications
- `l2cap_echo_client`: Specific to the master application
- `l2cap_echo_server`: Specific to the slave application

The initialization of the `connection` library, used to provide the functionality and API for creating the L2CAP connection, is covered in Planning and coding a BlueCore application.

The client application has been hard coded with the Bluetooth address of the server device so that no device inquiry and discovery phase is required.

### Initialization (common)

The slave server application is configured to enable scans for pages, although in the above example it has been configured to respond to inquiries as well.

### Authorize the L2CAP connection (l2cap_echo_server)

After Simple Pairing is complete, the `CL_SM_AUTHORISE_IND` message indicates to the server device that the client device is trying to establish a connection to it. The server application authorizes the connection on the indicated L2CAP channel.

## 6.1 Initialization (common)

```
#if defined(ALWAYS_PAIR)
    /* Delete previously authenticated devices to force simple
     * pairing to be used
     */
    ConnectionSmDeleteAllAuthDevices(0);
#endif
```

**Figure 6-2 Common initialization**

When a connection has been established and bonded between two devices, the `connection` library stores the Bluetooth address of a bonded device and its link keys. The example demonstrates Simple Pairing when making a connection. To ensure this is carried out, all previously authenticated device information is deleted as part of the application initialization.

### Enable inquiry and page scan (l2cap_echo_server)

```
/* Make this device discoverable for inquiries and pages */
ConnectionWriteScanEnable(hci_scan_enable_inq_and_page);
```

**Figure 6-3 Make discoverable**

### Register the L2CAP PSM (common)

```
/* Register a Protocol/Service Multiplexor (PSM) that will be
* used for this application. The same PSM is used at both
* ends.
*/
ConnectionL2capRegisterRequest(task, PSM, 0);
```

**Figure 6-4 Register the L2CAP PSM**

Both applications must register the L2CAP PSM value that is used to identify the connection, before attempting to establish the connection. The PSM must be an odd number and is a 16-bit value. In the example, a PSM of `0x1001` is used, see *Bluetooth Core Specification for L2CAP*.

The Protocol Stack firmware responds to the PSM registration request with a `CL_L2CAP_REGISTER_CFM` message, indicating success or failure.

### Initiate an L2CAP connection (l2cap_echo_client)

```
uint16 l2cap_conftab[] =
{
/* Configuration Table must start with a separator. */
/* 00 */ L2CAP_AUTOPT_SEPARATOR,
/* Flow & Error Control Mode. */
/* 01 */ L2CAP_AUTOPT_FLOW_MODE,
/* Set to Basic mode with no fallback mode */
/* 02 */ BKV_16_FLOW_MODE( FLOW_MODE_BASIC, 0 ),
/* Local MTU exact value (incoming). */
/* 03 */ L2CAP_AUTOPT_MTU_IN,
/* Exact MTU for this L2CAP connection - 672. */
/* 04 */ 0x2a0,
/* Remote MTU Minumum value (outgoing). */
/* 06 */ L2CAP_AUTOPT_MTU_OUT,
/* Minimum MTU accepted from the Remote device. */
/* 07 */ 0x30,
/* Configuration Table must end with a terminator. */
/* 08 */ L2CAP_AUTOPT_TERMINATOR
};

uint16 *p_conftab = PanicUnlessMalloc(sizeof(l2cap_conftab));
memmove(p_conftab, &l2cap_conftab, sizeof(l2cap_conftab));
ConnectionL2capConnectRequest(
        task,                   /* The client task. */
        &theApp.echo_server,    /* Address of the remote device. */
        PSM,                    /* Local PSM. */
        PSM,                    /* Remote PSM. */
        sizeof(l2cap_conftab),  /* Length of configuration table. */
        p_conftab               /* The configuration table. */
                    );
```

**Figure 6-5 Initiate the L2CAP connection**

`p_conftab` is a BlueStack Key/Value pair table used for configuring the L2CAP channels.

Triggered by the successful registration of the PSM, the client application initiates an L2CAP connection request. The Bluetooth address of the device to connect to (`&theApp.echo_server`) and the PSM of the L2CAP connection to connect from and to are passed as parameters. In this example, it is the same PSM value.

Further L2CAP configuration parameters can be set but in this example, the default configuration is used by passing `0` or `NULL` for the configuration parameter block.

## 6.2 Simple pairing (common)

```
ConnectionSmIoCapabilityResponse(
      &tpaddr,                              /* Remote device address. */
      cl_sm_io_cap_no_input_no_output,
      mitm_not_required,                    /* Don't force MITM. */
      TRUE,                                 /* Remote device bonded. */
      0,                                    /* No LE key distribution. */
      oob_data_none,                        /* No OOB data. */
      0,                                    /* No hash C for OOB. */
      0                                     /* No rand R for OOB. */
      );
```

**Figure 6-6 Setup simple pairing**

The example code above is taken from the echo client application, although the code is common to both example applications. The echo server device requests the IO Capability of the client device first, and then after the client device has responded, it requests the IO Capability of the server device.

The application must interact with Protocol Stack during the Simple Pairing scenario to indicate the hardware capability of the device which is being paired. In this example, both the client and server devices indicate that they have no capability for input or output, and no Out Of Band data is used.

Simple Pairing then proceeds using the *Just Works* model. Although more secure than the authentication previous to Bluetooth v2.1, it is still susceptible to a Man In The Middle (MITM) attack. See *Simple Pairing in the Bluetooth Core Specification*.

When the other device, receives the IO Capability Response, it indicates this to the application by a `CL_SM_REMOTE_IO_CAPABILITY_IND` message.

## 6.3 Authorize the L2CAP connection (l2cap_echo_server)

```
case CL_SM_AUTHORISE_IND:
        {
             CL_SM_AUTHORISE_IND_T *csai = (CL_SM_AUTHORISE_IND_T *)message

             printf("CL_SM_AUTHORISE_IND :- \n");
             printf("\t  protocol_id: %d\n", csai->protocol_id);
             printf("\t  channel: %ld\n", csai->channel);
             printf("\t  incoming: %d\n", csai->incoming);

             ConnectionSmAuthoriseResponse(
                  &csai->bd_addr,
                  csai->protocol_id,
                  csai->channel,
                  csai->incoming,
                  TRUE
             );
        }
        break;
```

**Figure 6-7 Authorizing the L2CAP connection**

```
case CL_L2CAP_CONNECT_IND:
        {
        CL_L2CAP_CONNECT_IND_T *clci = (CL_L2CAP_CONNECT_IND_T*)message;
        printf("CL_L2CAP_CONNECT_IND :-\n");

        /* Send a response accepting the connection. */
        ConnectionL2capConnectResponse(
             task,                   /* The Client task. */
             TRUE,                   /* Except the connection. */
             PSM,                    /* The local PSM. */
             clci->connection_id,    /* The L2CAP Connection Id.*/
             clci->identifier,       /* The L2CAP signal identifier. */
             0,                 /* 0 length Configuration table */
             0                  /* 0 indicated default conftab. */
        );
}
break;
```

**Figure 6-8 Accept the L2CAP connection**

```
case CL_L2CAP_CONNECT_CFM:
        {
          CL_L2CAP_CONNECT_CFM_T *clci = (CL_L2CAP_CONNECT_CFM_T*)message;
          printf("CL_L2CAP_CONNECT_CFM :-");
          if (l2cap_connect_pending == clci->status)
          {

             ......

          }
```

**Figure 6-9 Confirmation of the L2CAP connection**

After the L2CAP channel connection has been authorized, the L2CAP Connection is indicated to the server application. The application then sends a response indicating that the L2CAP connection for the PSM is accepted. See the online API documentation.

### L2CAP connection confirmation (common)

Both applications receive a `CL_L2CAP_CONNECT_CFM` message to indicate that the L2CAP connection has been successfully set-up and is ready to use. The applications can start to transfer data across the radio using the L2CAP connection.

### Key points to note from this example

- Several helper macros have been defined in `connection.h`, for example:

  `BKV_UINT32R(most_preferred,least_preferred),`
  `BKV_UINT16R(most_preferred,least_preferred), and so on.`

- These should be used when specifying `uint32` and `uint16` ranges in a configuration table (that is, `conftab`).
- When defining a range using either the `BKV_UINT16R` or `BKV_UINT32R` macro, the first value in the range is the most preferred value during parameter negotiation, and the second value is the least preferred value. For example, defining a range 1 to 10 where 10 is the preferred value should be defined as `BKV_UINT16R(10, 1)`. If 1 is the preferred value then it should be defined as `BKV_UINT16R(1, 10)`.
- The L2CAP Flow and Error Control value (`L2CAP_AUTOPT_FLOW_MODE`) combines the preferred mode and fallback mode mask into a single `uint16`, the first 8-bits being the preferred mode and the last 8-bits being the fallback mode mask.
- The local MTU (`L2CAP_AUTOPT_MTU_IN`) defines the exact local MTU.
- The remote MTU (`L2CAP_AUTOPT_MTU_OUT`) defines the minimum MTU that can be accepted for the L2CAP link from the peer device. The profile using the L2CAP connection may specify this value. The default minimum value is 48.
- The Remote Flush Timeout (`L2CA_AUTOPT_FLUSH_OUT`) defines a range for this parameter. The default value is for an infinite flush timeout (`DEFAULT_L2CAP_FLUSH_TIMEOUT = 0xffffffff`). Setting the Least Preferred value of the Remote Flush Timeout to `0` indicates that a finite flush timeout will be accepted during parameter negotiation. If the infinite timeout is only

accepted for this parameter, then both the least and most preferred value should be set to the same value (`DEFAULT_L2CAP_FLUSH_TIMEOUT = 0xffffffff`).

- The L2CAP interface file `l2cap_prim.h` defines all the configuration parameters that can be set using the `conftab`. If the application mandates any specific configuration values for an L2CAP connection, it can be set like the example in Initialization (common) .
- The `p_conftab` pointer is converted to a firmware handle at which time the VM memory slot is freed. The handle is freed by the firmware when the configuration data has been read. There is no need to free the allocated memory in the VM application.

# 7 Technical support

Further information on all QTIL products can be found on (createpoint).

# A BlueCore system messages

This section describes BlueCore's system messages. They provide a means for a VM application to find out about asynchronous events. Effectively every possible asynchronous event is mapped onto a message from the system. A single task within the application can register an interest in particular messages. When a task has registered an interest any appropriate message types are routed to that task. If no task registers an interest in a particular message then that message is silently discarded.

**NOTE** Registering an interest in any Protocol Stack primitive registers all Protocol Stack primitives which will then be sent to the registering task.

| Message | Payload Type | Task | Granularity | Description |
|---|---|---|---|---|
| BLUESTACK_LC_PRIM | uint16* | MessageBlueStackTask | System wide | The corresponding BlueStack primitives |
| BLUESTACK_LM_PRIM | | | | |
| BLUESTACK_HCI_PRIM | | | | |
| BLUESTACK_DM_PRIM | | | | |
| BLUESTACK_L2CAP_PRIM | | | | |
| BLUESTACK_RFCOMM_PRIM | | | | |
| BLUESTACK_SDP_PRIM | | | | |
| BLUESTACK_BCSP_LM_PRIM | | | | |
| BLUESTACK_BCSP_HQ_PRIM | | | | |
| BLUESTACK_BCSP_BCCMD_PRIM | | | | |
| BLUESTACK_CALLBACK_PRIM | | | | |
| BLUESTACK_TCS_PRIM | | | | |
| BLUESTACK_BNEP_PRIM | | | | |
| BLUESTACK_TCP_PRIM | | | | |
| BLUESTACK_UDP_PRIM | | | | |
| BLUESTACK_FB_PRIM | | | | |

| Message | Payload Type | Task | Granularity | Description |
|---|---|---|---|---|
| FROM_HOST | Bluestack primitive* | MessageHostCommsTask | System wide | Word-oriented messages passed over BCSP#13 |
| MORE_DATA | MessageMoreData* | MessageSinkTask | Per sink | Data may have arrived on the source corresponding to the registered sink. |
| MORE_SPACE | MessageMoreSpace* | MessageSinkTask | Per sink | Data may have left the registered sink. |
| PIO_CHANGED | MessagePioChanged* | MessagePioTask | System wide | PioDebounce (or PioDebounce32) must have been called to enable the messages |
| FROM_KALIMBA | MessageFromKalimba* | MessageKalimbaTask | System wide | DSP application must be running |
| ADC_RESULT | MessageAdcResult* | AdcRequest | Per request | Task registered per-request using AdcRequest |
| STREAM_DISCONNECT | MessageStreamDisconnect* | MessageSinkTask | Per sink | Message is sent to both tasks registered for source and sink of the connection. |
| ENERGY_CHANGED | MessageEnergyChanged* | MessageSinkTask | Per sink and request | The message is sent to the task associated with the sink identifying the SCO stream. Estimation must be enabled (on a single-shot basis) using EnergyEstimationOn or EnergyEstimationSetBounds. |
| STATUS_CHANGED | NONE | MessageStatusTask | Per request | The message is sent (an a one-off basis) if any of the fields specified in the MessageStatusTask change. Details can then be found using StatusQuery. |
| SOURCE_EMPTY | MessageSourceEmpty* | MessageSinkTask | Per sink | Sent when the source corresponding to the sink becomes empty (and cannot possibly become full again.) |
| FROM_KALIMBA_LONG | LongMessageFromKalimba* | MessageLongKalimbaTask | System wide | A long message (longer than four words) is received from Kalimba. |
| USB_ENUMERATED | NONE | MessageSystemTask | System wide | - |

| Message | Payload Type | Task | Granularity | Description |
|---|---|---|---|---|
| USB_SUSPENDED | MessageUsbSuspended* | MessageSystemTask | System wide | - |
| CHARGER_CHANGED | MessageChargerChanged* | MessageChargerTask | System wide | - |
| PSFL_FAULT | NONE | MessageSystemTask | System wide | - |
| USB_DECONFIGURED | NONE | MessageSystemTask | System wide | - |
| USB_ALT_INTERFACE | MessageUsbAltInterface | MessageSystemTask | - | - |
| MESSAGE_USB_ATTACHED | NONE | MessageSystemTask | - | - |
| MESSAGE_USB_DETACHED | NONE | MessageSystemTask | - | - |
| MESSAGE_KALIMBA_WATCHDOG_EVENT | NONE | MessageKalimbaTask | - | - |

# B BlueCore library message bases

This appendix shows the hex value bases for library messages.

| Library | Message Base | Library | Message Base |
|---|---|---|---|
| Connection | 0x5000 | AVRCP | 0x5e00 |
| SPP | 0x5100 | HDP | 0x6000 |
| DUN Library | 0x5200 | HID | 0x6100 |
| GOEP | 0x5300 | HIDKP | 0x6150 |
| OBEX | 0x5350 | Non Profile Library Message Bases | |
| FTP Client | 0x5400 | Audio Downstream | 0x7000 |
| FTP Server | 0x5450 | Battery | 0x7100 |
| OPP Client | 0x5500 | Codec | 0x7200 |
| OPP Server | 0x5550 | Audio Upstream | 0x7300 |
| PBAP Client | 0x5600 | Debongle | 0x7400 |
| PBAP Server | 0x5650 | GAIA | 0x7480 |
| MAP Client | 0x5700 | USB Device Class | 0x7500 |
| MAP Server | 0x5750 | Display Upstream | 0x7600 |
| HFP | 0x5a00 | Display Downstream | 0x7700 |
| AGHFP | 0x5b00 | Power | 0x7800 |
| WBS | 0x5c00 | CSR Speech Recognition | 0x7900 |
| A2DP | 0x5d00 | - | - |

**NOTE** Not all the libraries listed in the table are included in every SDK, that is,. only the libraries appropriate to the application are included in individual ADKs.

# Document references

| Document | Reference |
|---|---|
| *Qualcomm Bluetooth SDK Environment* | 80-CT436-1/CS-00207480-UG |
| *xIDE User Guide* | 80-CT405-1/CS-00101500-UG |
| *Guide to Qualcomm Bluetooth SDK Libraries* | 80-CT435-1/CS-00207478-UG |
| *Implementing Streams in Qualcomm BlueCore Applications* | 80-CT402-1/CS-00207482-UG |
| *VM Memory Mapping and Memory Usage* | 80-CT415-1/CS-00110364-AN |

# Terms and definitions

| Term | Definition |
|---|---|
| ADK | Audio/Application Development Kit |
| API | Application Programming Interface |
| BlueCore | Group term for the range of Qualcomm Bluetooth wireless technology ICs |
| Bluetooth | Set of technologies providing audio and data transfer over short-range radio connections |
| IC | Integrated Circuit |
| L2CAP | Logical Link Control and Adaptation Protocol |
| MITM | Man-In-The-Middle |
| PSM | Protocol Service Multiplexer |
| QTIL | Qualcomm Technologies International, Ltd. |
| SPP | Serial Port Profile |
| VM | Virtual Machine; environment in the BlueCore firmware for running application-specific code. |
| xIDE | The QTIL Integrated Development Environment |